2023

Angular & Wordpress

Learn how to quickly build cutting-edge web applications using Angular and Wordpress in 2023

John Manners

ISBN 978-1-3999-4589-9

DESIGN & PRODUCTION John Manners
PRINTING & BINDING Trade Digital Print, Dublin, Ireland

Published January 2023 by Dubhlinn Nua Publishing
(www.dnuapublishing.com) in Dublin, Ireland.

Angular & Wordpress

2023

John Manners

CONTENTS

INTRODUCTION

This book will teach you how to combine the forces of Angular and Wordpress to create cutting-edge web applications and websites. It is intended for developers who know JavaScript and PHP and have a little prior knowledge of Angular and Wordpress (but it also might be of use to developers who are completely new to those frameworks - the appendix details the installation process for both). It will teach you how to quickly put these two platforms working together, without getting bogged down in detail.

Statistics repeatedly reveal Wordpress to be one of the most popular (if not the most popular) web framework and content management systems in the world of the internet. And with good reason. Starting out as a platform which allowed non-coders quickly set up website blogs, it has evolved into a powerful CMS or content management system that is unrivalled for its flexibility and unrivalled for the sheer size of the community of developers that support it. But caveat emptor. Let the buyer beware. Technology has marched on and left Wordpress a little behind. One aspect of it has become a little outdated. It produces multipage applications. The single page application (SPA) is the new kid on the block. With good reason. It offers a better and more logical user experience. In a nutshell, when you navigate through a Wordpress website, you go from page to page and the browser refreshes each time to retrieve new content from the server, and it blinks each time it does this. Whereas a single page application is just that. It is one page or one screen, and you can navigate through the pages of the website without the browser refreshing and without that blink. So the content of the page is much more solid and the experience for the user is smoother and much more seamless.

This is where Angular comes in. Developed by Google, it is one of the major players in the single page application sphere. Other competitors are Vue or Facebook's React. But to my mind, due to the great flexibility and sheer diversity that it has, Angular is out in front. But that flexibility and diversity comes at a price. Angular is harder to learn, no doubt about

it. But it pays off in the end. Important for any developer is the official documentation and API reference. Here Angular also has the edge over its competitors. The official developer guides available at Angular have Google's customary attention to detail and clarity.

And that brings me to the main point of this introduction. The Wordpress official documentation is not good. And that primarily is the reason why I am providing this course. I suppose given the nature of Wordpress - which has been developed by a community of developers and not by a single company or organisation - it's not surprising that the official docs are a little haphazard. On the other hand, Google is a mega corporation driven by profit, and thus it makes sure that the Angular docs are well put together. The Wordpress documentation has started to get an overhaul in the last couple of years, however. But not quick enough for me, and not quick enough for you. And that's probably why you're here. At least one of the reasons. In this course I'm going to show you what I've learnt - information that I had to find out for myself, because there was no course, tutorial or e-book available to me to show me how to quickly join up Angular and Wordpress. I lie. I came across one online course, but it was very expensive, and I could not be sure if it was any good, so I decided I'd spend the time seeking out the information myself. In this course, I present that information to you, in a no-nonsense, clear and concise fashion.

In a nutshell, we will create a web app or website and Angular will be our frontend and Wordpress will be our backend. Angular is of course a frontend framework, and we will use its full capabilities. We will, however, only use a section of the Wordpress ecosystem, namely its database. We don't need any of its templates or frontend functionality. Angular has got us covered there. So Angular will be sitting on the browser - the client. And it will occasionally reach out to the server i.e. the Wordpress database. A very simple application may not need to store information on a server database. Everything could be on the client, the browser, in one initial download. But the vast majority of applications need to store info on a database. And so it is with the application we will build in this book. It will be a website for a fictitious athletics club, with e-commerce functionality. This will involve integrating a third-party app. This will be the popular payments framework, Stripe.

The application you will create with me will have you covered for a lot of the scenarios you will find yourself in as a web developer.

ABOUT THE AUTHOR

John Manners is a web developer of many years experience. In another life he spent five years as a secondary school teacher in Ireland, the equivalent of high school in other countries. He taught mathematics and the Irish language. Both extremely complicated topics. But he was successful because he learned how to impart information clearly and concisely. No waffle. And he brings that wealth of experience to this course on Angular and Wordpress.

There are many books and online courses currently in existence on web development which are overly bloated. The author pads out his or her lectures in order to make the course look long and thus comprehensive. Information is unnecessarily repeated. Not so here. John will go straight to the point. He won't waste your time or his time. And that's why this course appears quite short compared to typical web development courses. There will be no waffle. He will give you the information you need, clearly and efficiently, and nothing else.

So join him on this unique course where he will unleash the combined power of two of the most popular frameworks on the web in 2023, Angular and Wordpress.

CHAPTER 1

INITIAL SETUP

For this course I will be using Microsoft's free Visual Studio Code. But you are free to use any modern IDE (integrated development environment) software. This course assumes that you, the developer, have a basic grasp of the fundamentals of Wordpress and Angular, and therefore, in turn, PHP and JavaScript/TypeScript. Having said that, if you are new to Angular and Wordpress, and you need a quick solution, this course may be quite useful to you. Instructions will be given on how to set up these two frameworks.

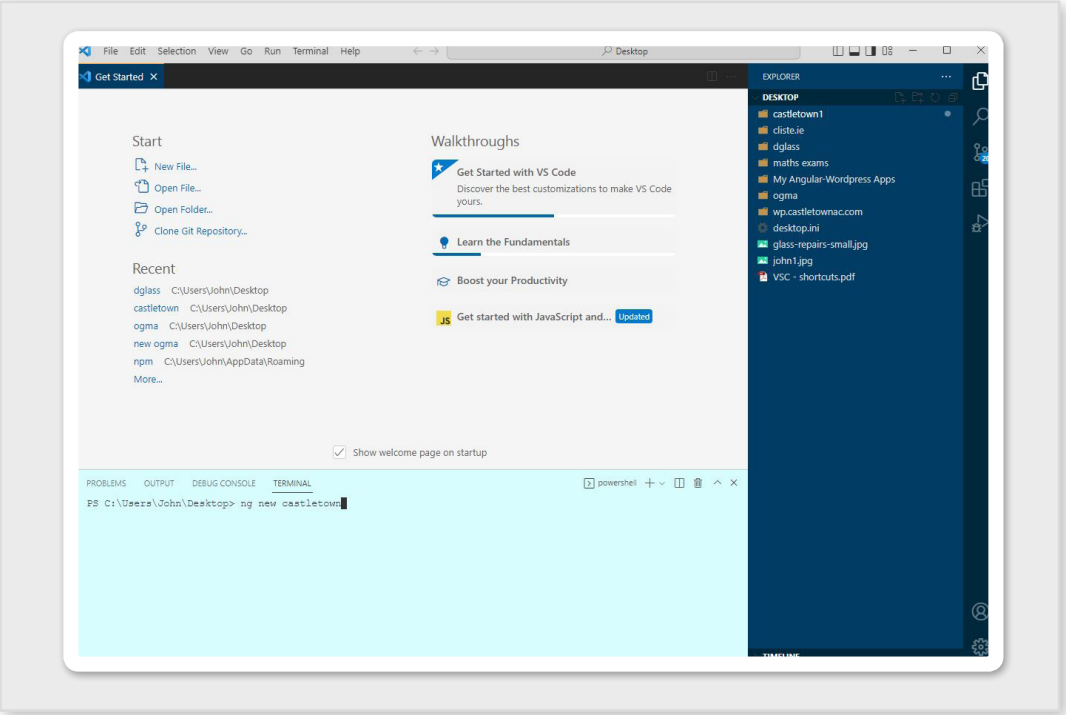
We will turn our attention first of all to setting up our project app in Angular. The app will be a website for a fictitious athletics club called Castletown Athletics Club. Firstly you will have Node.js and NPM installed on your machine. Secondly, you will have the latest version of the Angular CLI (command line interface), via NPM, installed on your machine (as of January 2023, I'm using Angular CLI 14.2.3 and Angular 14.2.0). If you don't have that or NPM, please see the appendix at the back of this book for instructions. Thirdly, please create a new app on the CLI as follows.

1. Open up your IDE and open up your desktop folder within your IDE.
2. Open up a terminal window.
3. Run this command in the terminal:

```
ng new castletown --routing --style css
```

This is an Angular CLI command which will create a new Angular (ng) app **castletown**. The routing flag will create the necessary routing files, which we'll explain later. The style flag indicates we wish to use CSS, instead of Sass etc. The command will create a folder **castletown** into which all necessary files will be inserted - the node modules along with the new files of the new Angular app. This will take a minute or two - be patient! Once complete, you should now see that folder on your desktop.

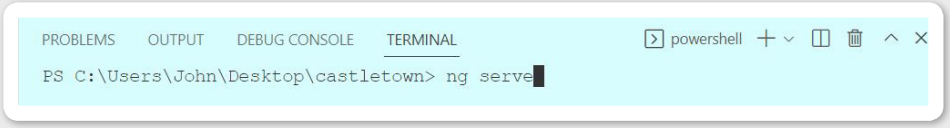
A terminal window opened up inside Visual Studio Code, within the desktop folder, with the ng command ready to be executed.



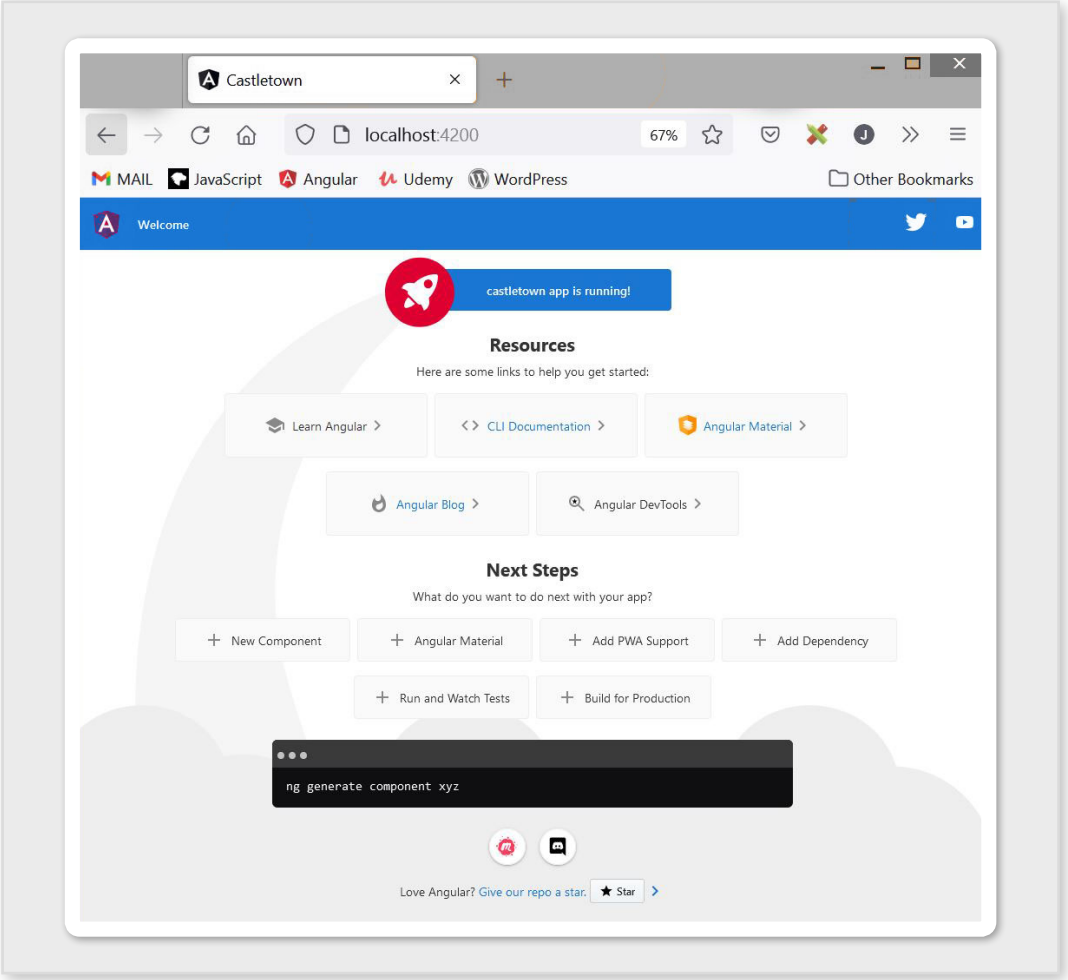
Now, let's serve up the **castletown** app in this basic state, to see that everything is functioning correctly. Open up the **castletown** folder in your IDE. Run the following command in the terminal within that folder:

```
ng serve
```

The ng serve command ready to be executed within the castletown app folder.



The result of the above command, all going well.



You should see this in your browser at http://localhost:4200

What you see in the browser is mainly the result of the creation of a component called **app.component.ts** - you will see it in the src/app folder of the **castletown** folder:

castletown/src/app

- app-routing.module.ts
- app.component.css
- app.component.html
- app.component.spec.ts
- **app.component.ts**
- app.module.ts

All the new files we create throughout this course will go into that app folder and into the assets folder. Before we go any further, we need to make a few minor changes to the **angular.json** file and the **tsconfig.json** file. The former needs to know that we aren't interested in creating testing files every time we create a component, nor do we want a style file:

castletown/angular.json

```
"projects": {
  "castletown": {
   "projectType": "application",
   "schematics": {
    "@schematics/angular:component": {
     "skipTests": true,
     "inlineStyle": true
    }
   },
   "root": "",
```

Change the 'schematics' property such that it looks like the above. This will prevent the **spec.ts** file from being created, along with the **style.css** file, everytime we create a new component. Now let's change the **tsconfig.json** file, which configures the TypeScript options:

castletown/tsconfig.json

```
"lib": [
    "es2020",
    "dom"
  ],
  "noImplicitAny": false,
  "strictPropertyInitialization": false
```

After the 'lib' property, add in the two properties as above to the end of the 'compilerOptions' property. This will remove some of the newer strict TypeScript rules and make writing the code a little quicker.

Now go into the **app** folder and delete 2 files: **app.component.spec.ts** and **app.component.css**. Then open up the **app.component.html** file and delete all the html content within that file - and save. Go and open up **app.component.ts** and delete the **styleUrls** property and its key. Save.

castletown/src/app/app.component.ts

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
```

The **@Component** decorator should look like the above. If you now have a look at your browser you will see a blank page - Angular's default content is gone. We are now ready to create our first new component. This project will see the creation of ten components:

- about.component.ts
- contact.component.ts
- news.component.ts
- homepage.component.ts
- payments.component.ts
- registration.component.ts
- sub.component.ts
- training.component.ts
- header.component.ts
- footer.component.ts

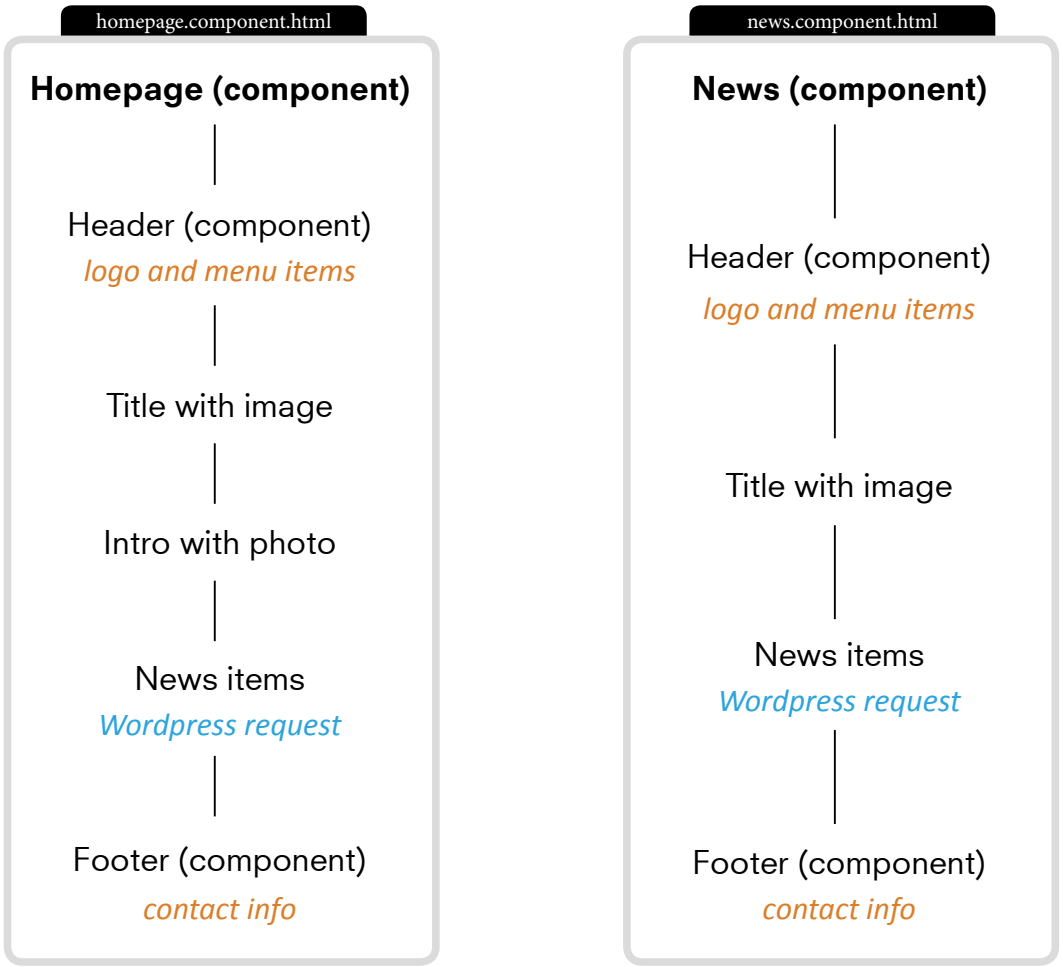
The above components (except the last two) represent the individual pages of the Castletown AC website. The last two, **header** and **footer**, are child components within these pages. To see the website in action, go to www.castletownac.com.

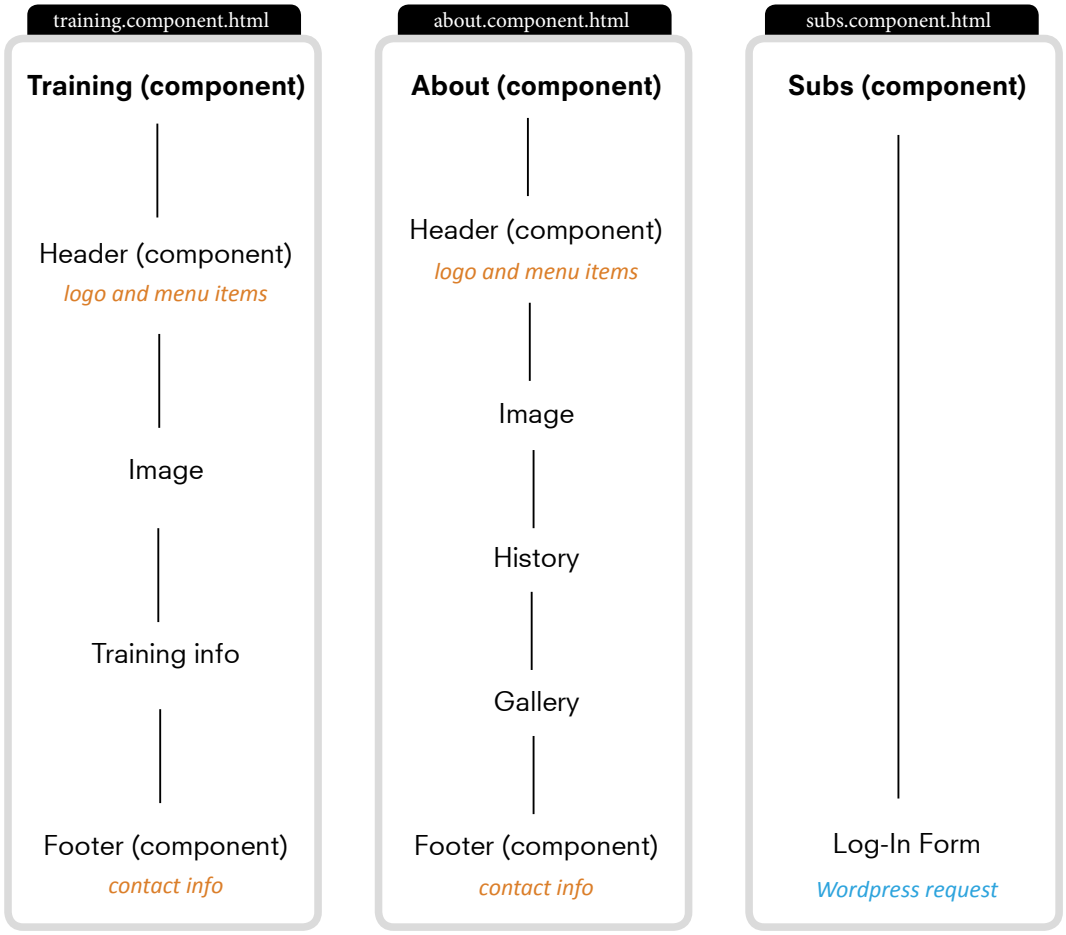
Read on for a detailed description of the website's structure.

CHAPTER 2

THE PROJECT STRUCTURE

Essentially we are building the website of a club society with e-commerce functionality i.e. the ability to take subscriptions from members. For this we will use a third-party API: the Stripe payments platform available at www.stripe.com. But more of that anon. Our website will have eight pages: **Homepage**, **News**, **Training**, **About, Subs**, **Contact**, **Payments** and **Registration**. All of these represent components. In addition, we will have two further components i.e. **Header** and **Footer** - so called 'child' components because they will be used within other components - they will be reused on several pages.





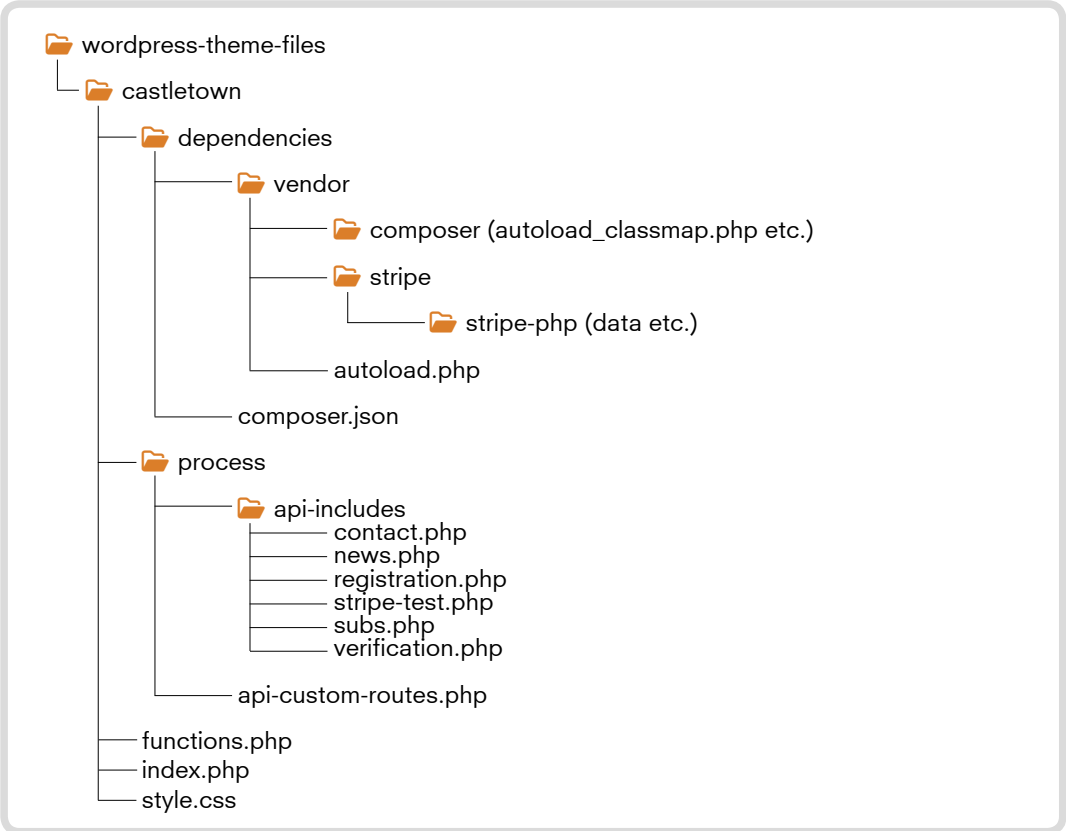
An outline structure of the app's html templates showing where contact is made with the Wordpress platform.

- Homepage: a brief introduction to the club with photo, followed by two of the latest news items (WP posts), retrieved from Wordpress.
- News page: all the news items retrieved from Wordpress.
- Training page: Castletown AC's weekly training schedule.
- About page: the history of Castletown AC, along with a slide gallery of photos.
- Subs page: the log-in form, verified on Wordpress, which takes registered members to the payments page.
- Contact page: contact form for general queries.
- Payments page: credit/debit card payment form provided by Stripe allowing registered members to pay their subscription.
- Registration page: form for non-registered members, which will send verification email with link. The clicked link will bring the member to the payments page.

The files of the completed project are available to download from this link: www.angular-wordpress.com/source-files

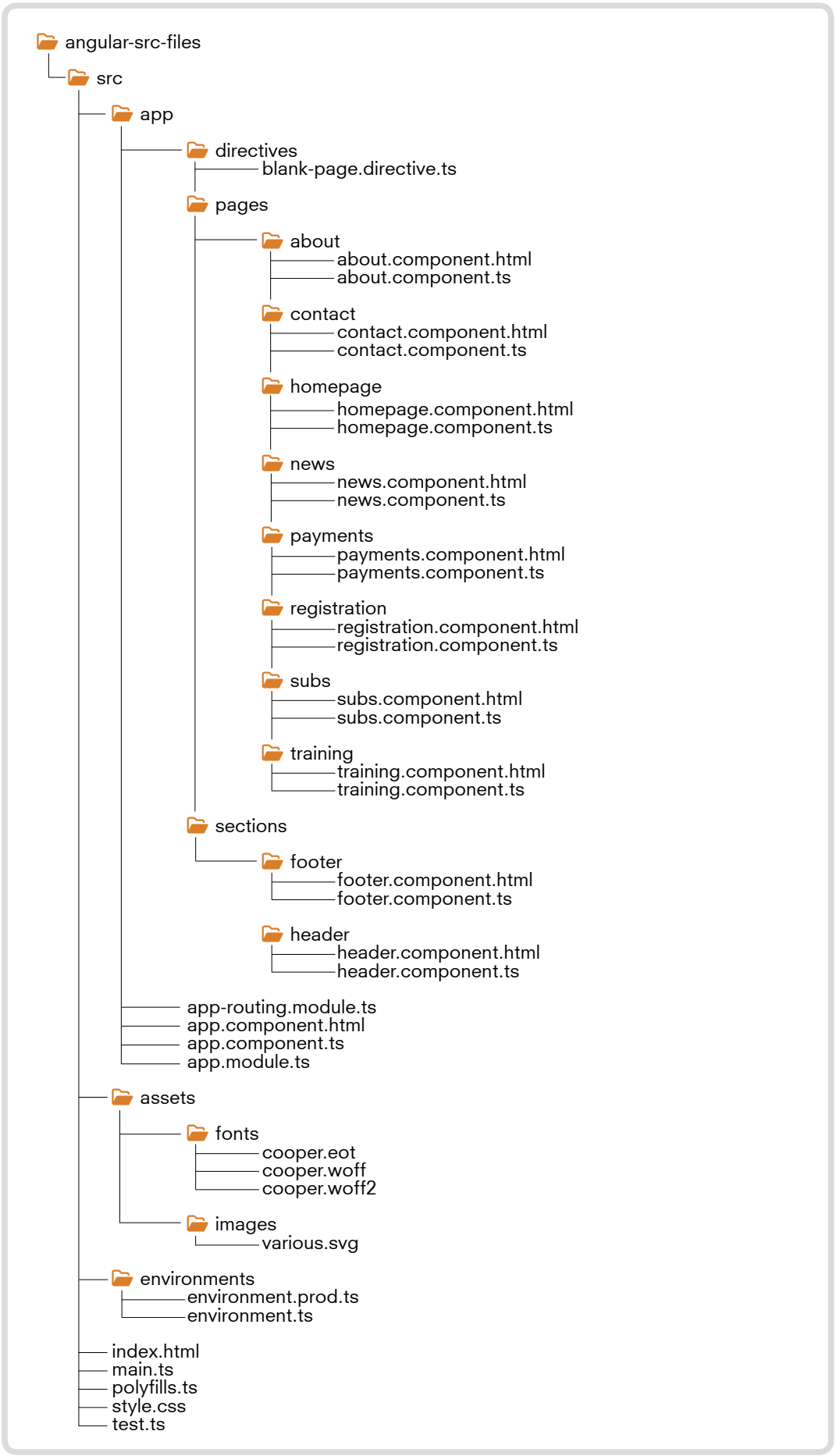
The Wordpress files are structured thus:

The file tree of the Wordpress theme.



I recommend that you don't copy and paste these folders into your own project. You will learn much more if you follow the course in this book step by step. The above and below file tree diagrams are purely for your reference. It might of course be of some benefit to see the app up and running correctly, so that's why I am providing these files for download.

The Angular files are structured thus:



The file tree of the Angular src folder.

CHAPTER 3

FIRST STEPS

We are now ready to start creating our first new components, using the Angular CLI. As mentioned, it is important that you follow each step of this course one by one, rather than copying and pasting the source files. You will learn a lot quicker if you do so. Remember to refer to the file tree diagrams of the previous chapter to help you visualize what direction this course is going in. So without further ado, here are the first steps you need to take.

Step 1. Create the first new component using the CLI and using the following command:

```
ng g c pages/homepage
```

We are using shorthand here. **g** is for generate. **c** is for component. We are instructing Angular to create a component **homepage** within a new folder **pages** - we don't have to create this folder - Angular will do it automatically, and it will also generate a folder called **homepage**. You will create all subsequent components in this manner. So if you look at your app folder, you will now see 2 new files within 2 new folders:

```
castletown/src/app/pages/homepage
                        ├── homepage.component.ts
                        └── homepage.component.html
```

In addition, the CLI will have added **HomepageComponent** (the automatically generated JavaScript class name) to the declarations array in the app's route module, **app.module.ts**. Again, this will also happen when you use the CLI to generate all subsequent components. Open up

header.component.html and insert the following code, removing whatever is already there:

castletown/src/app/pages/homepage.component.html

```
<div class="jumbotron gretta-jumbotron gretta-blue-wrapper">
  <div class="container gretta-banner-container">
    <div class="row gretta-banner-row">
      <div class="col-lg-5 col-lg-push-7 gretta-banner-col1">
        <div class="gretta-banner-image">
          <img src="../../assets/images/homepage-01.svg" title="Casltetown AC" alt="Castletown AC"
itemprop="image"/>
        </div>
      </div>
      <div class="col-lg-7 col-lg-pull-5 gretta-banner-col2">
        <div class="gretta-banner-content-wrap">
          <h1 class="gretta-banner-heading" itemprop="name">Train With Us!</h1>
          <p><span class="gretta-description" itemprop="description">Welcome to Castletown Athlec-
tics Club website, one of Ireland's largest and most successful athletics and running clubs!</span></
p>
          <div class="gretta-mailing-list"><a routerLink="/training" role="button" class="btn btn-lg
btn-primary gretta-dglass-orange mb-5">Training Times</a><a routerLink="/contact" role="button"
class="btn btn-lg btn-primary mb-5 ml-5 gretta-dglass-blue">Contact us</a></div>
        </div>
      </div>
    </div>
  </div>
</div>

<div id="gretta-category-list2" class="gretta-home-category-wrapper mt-5">
  <div class="container gretta-home-category-container mt-0 pt-0 pb-0">
    <div class="row">
      <div class="col text-center gretta-content">
        <h2 class="gretta-content-header">Castletown AC, Dublin City</h2>
        <p><img src="../../assets/images/home.jpg" class="figure-img img-fluid rounded" alt="Castle-
town AC History" width="700"></p>
        <p>Situated in Castletown Stadium, just minutes from Dublin city centre, we have a membership
of about 600 athletes, and we welcome all ages, from juvenile, through junior and senior, to masters.
We also cater for all standards, from beginners and recreational runners, to elite athletes, Olympians
and everything between.</p>
      </div>
    </div>
  </div>
</div>

<div id="gretta-category-list1" class="gretta-home-category-wrapper gretta-blue-container mt-5">
  <div class="container gretta-home-category-container pb-5">
    <div class="row">
      <div class="col text-center gretta-content">
      <h1 class="gretta-story-header gretta-banner-heading text-center">Get running now!</h1>
        <p>A club dedicated to excellence in fitness and health.</p>
        <a routerLink="/contact" role="button" class="btn btn-lg btn-primary gretta-dglass-orange
mb-5 mt-3">Join now</a>
      </div>
    </div>
  </div>
</div>
```

Step 2. Use the CLI to create two further components, the header and footer. But put them into a folder **sections** instead of **pages.**

```
ng g c sections/header
```

Repeat the step above for the footer component. Open up the header component TypeScript file, **header.component.ts** - you will see that the selector property has this value: **app-header**. These values mimic the nomenclature for CSS selectors. For instance, **app-header** points to this custom HTML element:

```
<app-header class="my-css-class">
```

Whereas [app-header] would point to any HTML element were app header is an attribute:

```
<div app-header class="my-css-class">
```

We will soon be using the **app-header** custom element. Take note that we could use any name for this element i.e. "my-header". But we are sticking with the convention of prefixing everything with "app". You will also notice that the class implements the **OnInit** interface, which requires the insertion of the **ngOnInit** method into the class - and that runs immediately after the component class is instantiated. We will be using **ngOnInit** regularly in this app. The CLI sets up all component classes like this. Ok, close that file - it doesn't need any further alterations i.e. the class **HeaderComponent** requires no logic.

Step 3. Open up the corresponding header template file, **header.component.html** and insert the following code:

castletown/src/app/sections/header.component.html

```
<nav class="navbar navbar-expand-lg navbar-light bg-faded gretta-grey-nav">
  <div class="container gretta-header-container">
    <a class="navbar-brand" routerLink="/">
      <img class="gretta-logo" src="../../assets/images/castletown-logo-01.svg" title="Castletown
AC" alt="Castletown AC" />
    </a>
    <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbar-
NavDropdown" aria-controls="navbarNavDropdown" aria-expanded="false" aria-label="Toggle
navigation">
    <span class="navbar-toggler-icon"></span>
    </button>
    <div id="navbarNavDropdown" class="navbar-collapse collapse">
```

```
    <ul class="navbar-nav ml-auto">
      <li class="nav-item">
        <a class="nav-link gretta-nav-items" routerLink="/news" routerLinkActive="active-url"
style="padding-right:2rem;">News</a>
      </li>
      <li class="nav-item">
        <a class="nav-link gretta-nav-items" routerLink="/training" routerLinkActive="active-url"
style="padding-right:2rem;">Training</a>
      </li>
      <li class="nav-item">
        <a class="nav-link gretta-nav-items" routerLink="/about" routerLinkActive="active-url"
style="padding-right:2rem;">About</a>
      </li>
       <li class="nav-item">
        <a class="nav-link gretta-nav-items" routerLink="/subs" style="padding-right:2rem;">Sub-
scriptions</a>
      </li>
      <li class="nav-item">
        <a class="nav-link gretta-nav-items" routerLink="/contact">Contact</a>
      </li>
    </ul>
   </div>
  </div>
</nav>
```

You will notice two directives on the anchor elements: **routerLink** and **routerLinkActive**. These are provided by the router module, **app-routing.module.ts**. **routerLink** will allow the user to navigate to the different pages while **routerLinkActive** will allow us to add CSS classes to indicate to the user what page (URL) is active. **active-url** is the class we've chosen. (On the subject of CSS classes, on the very first line you will see "navbar" as a class. That's a Bootstrap class. That is one of many in this HTML template. We will come to Bootstrap shortly). Of course, we need to tell Angular what components to bring into the view once those anchor elements are clicked. We will get to that later. For the moment, those links won't do anything.

Step 4. Let's alter the footer component template. Open up **footer.component.html** and insert code into it as follows:

castletown/src/app/sections/footer.component.html

```
<div class="container-fluid gretta-orange-no-hover gretta-grey-nav">
  <div class="gretta-footer-container">
    <div class="row text-center pt-3 pb-3 align-items-center gretta-black-link">
      <div class="col-md-2 mb-3 mt-3">
        <a routerLink="/subs">Subscriptions</a>
      </div>
      <div class="col-md-2">
```

```
        <a routerLink="/about">About</a>
      </div>
      <div class="col-md-4">
        <a class="navbar-brand" routerLink="/"><img class="gretta-logo-small" src="../../assets/imag-
es/castletown-logo-01.svg" title="Castletown AC" alt="Castletown AC Logo" /></a>
      </div>
      <div class="col-md-2 mb-3 mt-3">
        <a routerLink="/training">Training</a>
      </div>
      <div class="col-md-2">
        <a routerLink="/contact">Contact</a>
      </div>
    </div>
  </div>
</div>
<div class="container-fluid gretta-dark-blue gretta-tiny-text">
  <div class="row text-center">
    <div class="col">
      <ul class="list-inline mt-4 mb-4 gretta-black-link">
        <li class="list-inline-item ml-2 mr-2"><a title="Email" target="_blank" href="mailto:info@castle-
townac.eu">info@castletownac.eu</a> | <a title="Phone" target="_blank" href="tel:015240900">Tel:
01 5240900</a></li>
        <li class="list-inline-item ml-2 mr-2">© 2022 Castletown AC</li>
      </ul>
    </div>
  </div>
</div>
```

Step 5. Alter **app-routing.module.ts** such that it looks like this:

castletown/src/app/app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HomepageComponent } from './pages/homepage/homepage.component';

const routes: Routes = [
  {
    path: '', component: HomepageComponent
  }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

This is perhaps the most significant file in our whole app. In the **Routes** array we have indicated that we want the root URL of the website (the empty string) to point to the homepage component. Now open up **app.component.html** (to which our whole app bootstraps) and insert as follows:

castletown/src/app/app.component.html

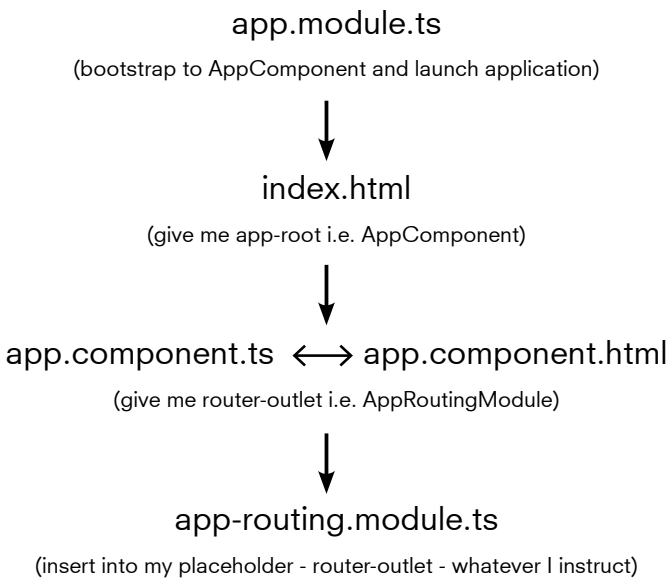
```
<router-outlet></router-outlet>
```

This HTML element acts as a placeholder for whatever the current router state requests. We will not place anything else into this template. So that means that whatever view Angular inserts into our app will be entirely based on the router state. If you look at **index.html** you will see the selector for the app component, <app-root></app-root>.

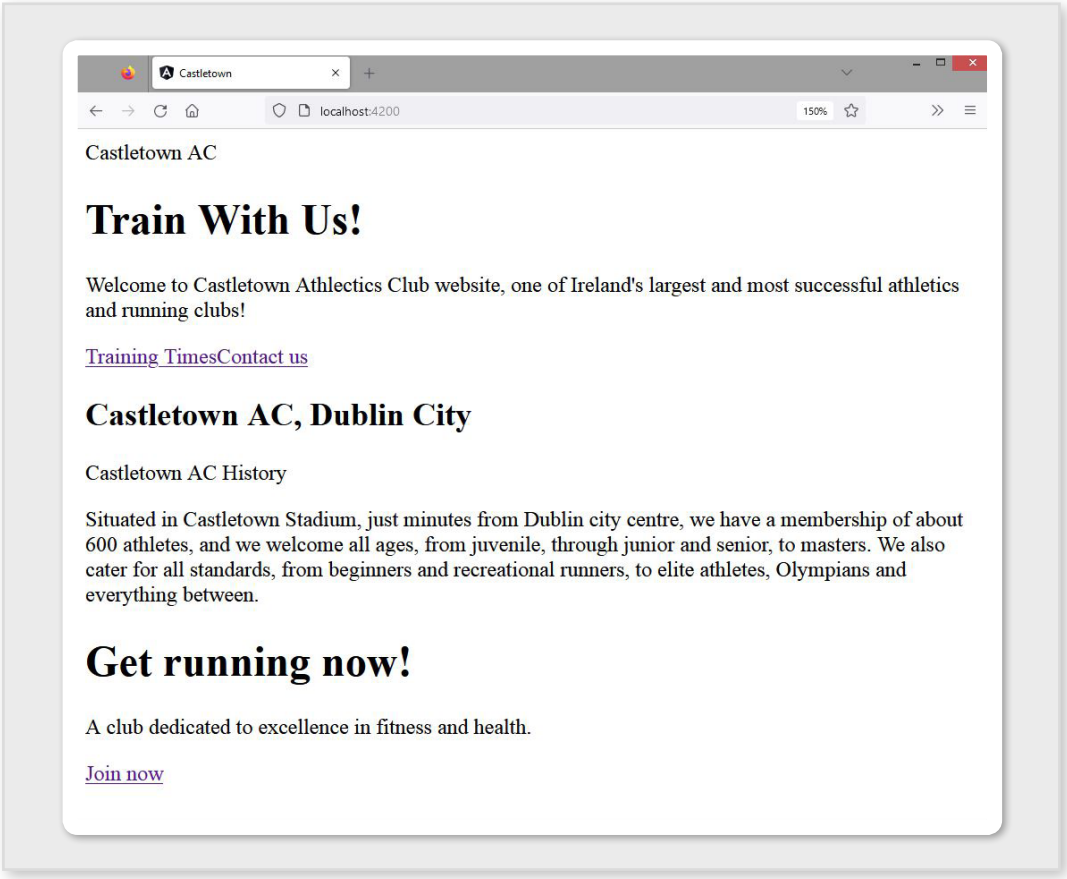
castletown/src/index.html

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Castletown</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

So the hierarchy looks something like this:



So we are now ready to see what the app looks like after all those changes. Make sure to save all the files! Go ahead and run the command **ng serve** (if you haven't already done so from Chapter 1) and open your browser. This is what you should see:



No work of art for the moment! There are a few issues. Firstly, where are our header and footer components? Let's fix that.

Step 6. Insert the header and footer selectors at the very top and very bottom of the homepage component template as shown below:

castletown/src/app/pages/homepage.component.html

```
<app-header></app-header>
<div class="jumbotron gretta-jumbotron gretta-blue-wrapper">
        .....
</div>
<app-footer></app-footer>
```

Save and now look at your browser. You will see the header and footer content. But we still have problems. The links don't work. The styling is non-existent. Let's fix the latter by indicating to Angular to use the Bootstrap library. Open up a terminal window, make sure you are in the Castletown folder, and type the following command:

```
npm install bootstrap@v4.2.1
```

That will install bootstrap version 4 locally in our project. Now open up the **angular.json** file in the root folder. There you will see a **styles** property and a **scripts** property, within the **build** property. Alter those properties such that they look like this:

castletown/angular.json

```
"styles": [
  "node_modules/bootstrap/dist/css/bootstrap.min.css",
  "src/styles.css"
],
"scripts": [
  "node_modules/bootstrap/dist/js/bootstrap.min.js"
]
```

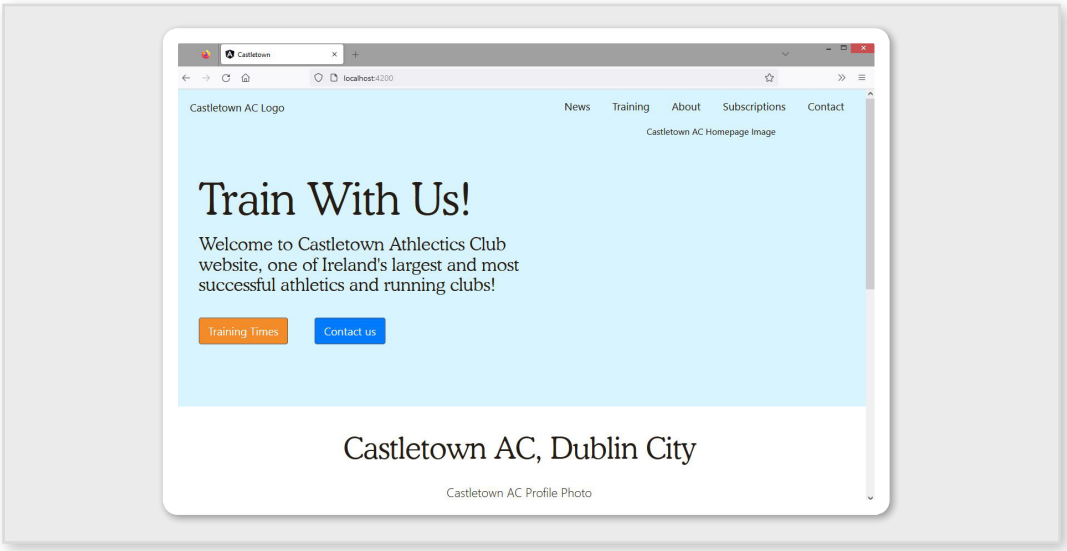
You might want to check that these are actually the correct paths for the Bootstrap files by navigating to them yourself in the folder panel of your IDE. Be aware that the order of the CSS files listed above is important. We are granting precedence to our own **styles.css** file over Bootstrap. Next, download the app's fonts from here:

www.angular-wordpress.com/source-files/fonts.zip

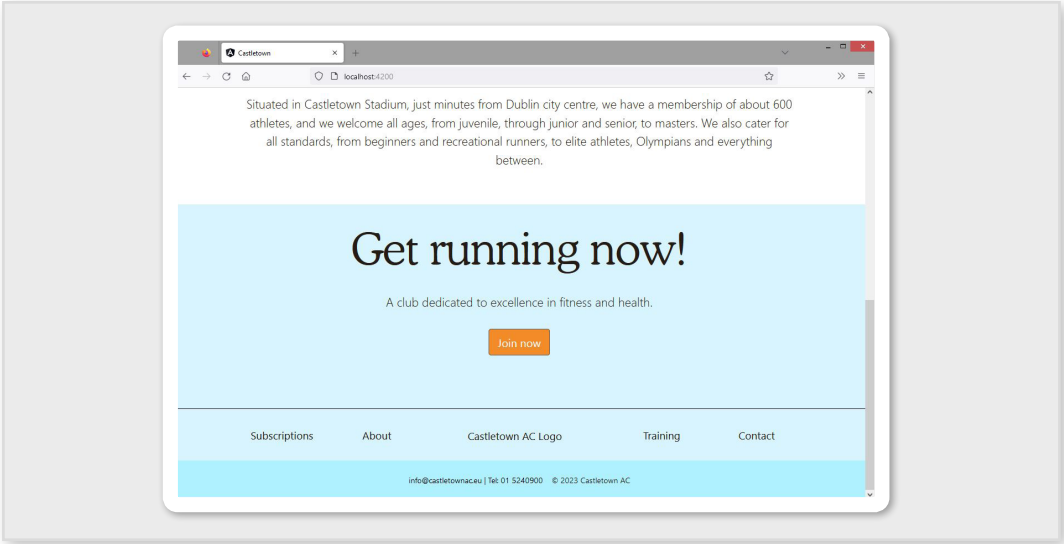
Unzip and place within the **assets** folder. Now download the project CSS styles file from here:

www.angular-wordpress.com/source-files/styles.css

Replace the existing **styles.css** with this one, in the **src** folder. Finally let's have a look at our work. Shut down the server currently running by hitting "Ctrl C" (or "Cmd C") in your terminal window. Now run **ng serve** once more. The app should now have Bootstrap available to it, in addition to our custom styles. The top half of the homepage will now look as follows:



The bottom half should look like this:



But we're missing our imagery. You will have noticed the alt text. Let's sort that out. Download the project images from here:

www.angular-wordpress.com/source-files/images.zip

Unzip and place in the **assets** folder. Refresh your browser, and the images will appear.

All right, all is going well so far! I mentioned the links not working. But we'll come back to that in a later chapter. So we've got the foundations of our app up and running. Well done! Let's move on to Chapter 4.

CHAPTER 4

HELLO WORDPRESS

Here in this chapter we will make our first communication with Wordpress, specifically the Wordpress database. This is done via the Wordpress REST API. Our app is essentially a client-side JavaScript application using the Wordpress framework's database and server functionality. But more of that anon. Firstly, I want you to create a new component called **news**, using the CLI as normal. This is a page, just like **homepage**, so put it into the **pages** folder.

```
ng g c pages/news
```

The CLI will have done its job and declared this new component in the app's root module, **app.module.ts**. Open up the newly generated **news** component. There are some important additions to make in this file, so let's tread carefully. Import a new class from the Angular repository, **HttpClient**:

castletown/src/app/pages/news/news.component.ts (excerpt)

```
import { Component, OnInit } from '@angular/core';
import { HttpClient } from '@angular/common/http';
```

HttpClient will allow us to make HTTP requests. This class is a service that is injectable. So inject it into the class using the constructor as follows and so an instance of **HttpClient** will be available to us within **NewsComponent**:

castletown/src/app/pages/news/news.component.ts (excerpt)

```
export class NewsComponent implements OnInit {

  constructor(private http:HttpClient) {}
```

Note that TypeScript will now declare and initialize this parameter as a property because we have prefixed it with 'private' which is an accessibiltiy modifier keyword. Now let's use the **ngOnInit** method. We mentioned this in Chapter 3. This method is called when the instance of the **news** component is first created. So put any logic in here that should be executed immediately. Why not do it in the constructor? Angular created this method for this very purpose - so avoid using the constructor. What do we put into **ngOnInit**? Have a look at the completed file:

castletown/src/app/pages/news/news.component.ts

```
import { Component, OnInit } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Component({
  selector: 'app-news',
  templateUrl: './news.component.html',
  styles: [
  ]
})
export class NewsComponent implements OnInit {

  constructor(private http:HttpClient) {}

  posts: any;

  ngOnInit(): void {

    this.http.get('https://wp.castletownac.com/wp-json/castletown/v1/news')
      .subscribe(response => {
        this.posts = response;
      });

  }

}
```

We've just made our first HTTP request to Wordpress! Fairly simple, no? **HttpClient** has a get method, which takes a string i.e. your request URL. It returns an observable (an RXJS feature which is integrated into Angular) to which you can subscribe with a callback, and so you can access the response.

But what is that URL? I have a Wordpress installation at

https://wp.castletownac.com

which is live on the web and which we will use for our project. It simply has 4 posts. The rest of the URL (wp-json etc.) is a custom route (which can be anything) facilitated by the Wordpress REST API. We will go into this in more depth later, but suffice it is to say for the moment that I have

a script at that address that queries my Wordpress database and returns my Wordpress posts. Specifically I am returning an array of Wordpress post objects to Angular. As I said, we'll get to the Wordpress side of the transaction shortly. For now, we need to set up a route for this 'news' page. Go to the routing module and alter it such that it looks like this:

castletown/src/app/app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HomepageComponent } from './pages/homepage/homepage.component';
import { NewsComponent } from './pages/news/news.component';

const routes: Routes = [
  {
    path: '', component: HomepageComponent
  },
  {
    path: 'news', component: NewsComponent
  }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

We have simply added in a new object into the **Routes** array, indicating what component we want www.castletownac.com/news to point to. Note that we also need to import that component. We will do this for all subsequent components i.e. pages. Now open up the news HTML template and alter as follows:

castletown/src/app/app-routing.module.ts

```
<app-header></app-header>
<div class="jumbotron gretta-jumbotron gretta-blue-wrapper">
  <div class="container gretta-banner-container">
    <div class="row gretta-banner-row">
      <div class="col-lg-5 col-lg-push-7 gretta-banner-col1">
        <div class="gretta-banner-image">
          <img src="../../assets/images/payments-01.svg" title="Casltetown AC" alt="Castletown
AC"/>
        </div>
      </div>
      <div class="col-lg-7 col-lg-pull-5 gretta-banner-col2">
        <div class="gretta-banner-content-wrap">
          <h1 class="gretta-banner-heading">Latest News</h1>
          <p><span class="gretta-description">Read all about what's happening in Castletown Athletics
Club. Stay informed and keep volunteering!</span></p>
          <div class="gretta-mailing-list"><a routerLink="/training" role="button" class="btn btn-lg
btn-primary gretta-dglass-orange mb-5">Training Times</a><a routerLink="/contact" role="button"
```

```
class="btn btn-lg btn-primary mb-5 ml-5 gretta-dglass-blue">Contact us</a></div>
      </div>
     </div>
    </div>
  </div>
</div>

<div class="jumbotron gretta-jumbotron gretta-white-wrapper">
  <div class="container gretta-banner-container">
    <div class="row gretta-banner-row">
      <div class="col-7">
        <div class="gretta-banner-content-wrap" *ngFor="let post of posts">
          <h2 class="gretta-content-header gretta-posts-header">{{post.post_title}}</h2>
          <div class="gretta-content">
            <div class="gretta-posts-author"><em>by {{post.display_name}}</em></div>
            <p class="text-justify"><img src="{{post.img1}}" class="figure-img img-fluid rounded" alt="-
Castletown AC Blog Posts"><br>
            {{post.post_content}}</p>
          </div>
        </div>
      </div>
    </div>
  </div>
</div>
<app-footer></app-footer>
```

Now, before we head over to our Wordpress installation, one piece in this particular jigsaw remains. Open up our root app module file and alter it such that it looks as follows:

castletown/src/app/app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { HomepageComponent } from './pages/homepage/homepage.component';
import { FooterComponent } from './sections/footer/footer.component';
import { HeaderComponent } from './sections/header/header.component';
import { NewsComponent } from './pages/news/news.component';

@NgModule({
  declarations: [
    AppComponent,
    HomepageComponent,
    FooterComponent,
    HeaderComponent,
    NewsComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    HttpClientModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

This is the first time we have looked at this file. Up to this, Angular has filled it out for us automatically, as we have previously mentioned, whenever we create a component. Also, when first creating the app, **NgModule**, **BrowserModule**, **AppRoutingModule** were all added, as was **AppComponent**. Now that we are introducing a new feature i.e. the HTTP module, we need to add it in ourselves, importing it as shown and then adding it to the **imports** array. You can also see where the app has been instructed to bootstrap to **AppComponent**.

Now, return to your browser, click on the **news** link and you should be able to see our newly created news page up and running, and collecting data from Wordpress successfully i.e. 4 regular Wordpress posts. In our component's template, displaying these posts is a fairly straightfoward process. We use Angular's **NgFor** structural directive - a 'for of' loop essentially, looping through the array of posts. Firstly we have stored this array in a **posts** property in the class, and then we access the properties of each post object using Angular's text interpolation with the double curly brackets. We can put any JavaScript expression in here, so long as it resolves to a string e.g.

```
<h1>{{post.post_title}}</h1>
```

And just for your information, if you console.log (you can do this in **ngOnInit**) the response from Wordpress, it will look as follows:

```
Array(4) [ {…}, {…}, {…}, {…} ]
0: Object { ID: 28, post_author: "1", ...}
1: Object { ID: 9, post_author: "1", ...}
2: Object { ID: 7, post_author: "1", ...}
3: Object { ID: 1, post_author: "1", ...}
```

Finally, I am just going to add in a little extra code to our files, in order to showcase Angular's **Pipe** - this is a convenient way to return readable values in our templates:

castletown/src/app/pages/news/news.component.ts (excerpt)

```
this.http.get('https://wp.castletownac.com/wp-json/castletown/v1/news')
    .subscribe(response => {
      this.posts = response;
      for (const post of this.posts) {
        post.post_date_object = new Date(post.post_date);
      }
    });
```

castletown/src/app/pages/news/news.component.html (excerpt)

```
<div class="gretta-content">
        <div class="gretta-posts-author"><em>by {{post.display_name}}, {{post.post_date_object |
date: 'd MMM YYYY'}}</em></div>
        <p class="text-justify"><img src="{{post.img1}}" class="figure-img img-fluid rounded" alt="-
Castletown AC Blog Posts"><br>
        {{post.post_content}}</p>
</div>
```

Wordpress will give us the MySQL date format (ISO 8601), so we want to change that to our own liking. We create a JavaScript **Date** object, and then we can put that through Angular's date pipe as indicated above, using 'l' as the pipe syntax. Save those files and have a look at the nicely formatted date on the news page.

Okay, good stuff. We are pulling info from Wordpress successfully. Let's now look at it more closely.

CHAPTER 5

THE WORDPRESS REST API: CUSTOM ROUTES

The Wordpress REST API provides default routes or urls (with customisable query parameters) that will allow you, the developer, to pull whatever information you want from the Wordpress database. The REST API will supply public data to any client anonymously, as well as private data once authentication has taken place. It uses JSON for requests and repsonses. For instance,

https://wp.castletownac.com/wp-json/wp/v2/posts?per_page=2

will return the two latest posts to us. Try the above in Firefox for a nicely formatted JSON response. However, whilst there are myriad query parameters we can use for the various Wordpress resources (posts, users, taxonomies etc.) – and this will suffice for a lot of our needs – we still can't run our own functions in addition to those database queries e.g. send an email. So I think it's better just to use this API's custom route/endpoint functionality in all cases. And that's what we are doing in this project. We do this by the following means.

Firstly, refer back to the diagram of the Wordpress theme file structure on page 17. You need to create a folder **process** within your theme. Within that folder, create a file **api-custom-routes.php** and insert the following code:

themes/castletown/process/api-custom-routes.php

```
require('api-includes/news.php');

function register_my_routes() {
  register_rest_route( 'castletown/v1', '/news', array(
    'methods'  => WP_REST_Server::READABLE,
    'callback' => 'news',
  ));
}

add_action( 'rest_api_init', 'register_my_routes' );
```

Let's explain what's going on here. The all important **register_rest_route** function has three parameters. I'll discuss the first two in a moment. The third is an array of options - we are just using the bare basics here - we specify the method we want, and then the callback for the endpoint. As we don't want to make any changes on the server for this particular endpoint, we ask for **READABLE**. We will use **CREATABLE** later in this course when we do want to make changes. We could also use a permissions callback, here in this third parameter, for restricting access to the endpoint, but we don't need it in this case. Finally we hook into **rest_api_init** so that we don't use any resources unnecessarily.

So the first and second parameters are strings and will make up our route or URL. The first ('casltetown/v1') is the namespace, so that we can group our routes more logically. The second ('/news') is the resource path i.e. the function we want to fire. Note that these 2 parameters can be any string of your choosing. The complete URL, using these parameters, will look like this:

https://wp.castletownac.com/wp-json/castletown/v1/news

So now let's create that function. Within the **process** folder, create another folder **api-includes**, where we will store our endpoint functions. Inside that, create **news.php** and insert the following code:

themes/castletown/process/api-includes/news.php

```
function news() {

  $new_posts = array();
  $posts = get_posts(array('cat' => 1, 'author' => 1, 'post_status' => 'publish' ));

  foreach ($posts as $post) {
    $user = get_user_by('id',$post->post_author);
    $post->{'display_name'} = $user->display_name;
    $img1 = get_the_post_thumbnail_url($post->ID);
    $post->{'img1'} = $img1;
    $img2 = get_the_post_thumbnail_url($post->ID,'square');
    $post->{'img2'} = $img2;
    $new_posts[] = $post;
  }

  return rest_ensure_response($new_posts);

}
```

You can see that we are altering the **get_posts** query result so that we can add in the URLs for the featured images. Wordpress will then convert our response **new_posts** to JSON for us when we use the **rest_ensure_reponse** function. Angular converts this JSON back into an array for us.

Now in **functions.php**, add the following:

themes/castletown/functions.php

```
add_theme_support( 'post-thumbnails' );
add_image_size( 'square', 451, 451, true );

function load_api_files() {
        require(get_template_directory().'/process/api-custom-routes.php');
}
add_action('init', 'load_api_files');
```

Here we are asking Wordpress to load the custom routes/endpoints file when Wordpress starts up. We are also adding support for featured images, and adding an extra size - the reason for this will become obvious later. And that's it! Now that we have the Castletown Athletics news page functioning and understand how we pulled posts from Wordpress, we are ready to actually send data to Wordpress rather than just pull data. But that's for Chapter 7.

CHAPTER 6

INTERNAL PAGES AND ROUTING

Before we start sending data (**POST** requests) to Wordpress, let us complete very quickly these two pages on the Castletown website, "Training" and "About". Also let's make the necessary changes to our routing. But first, before we go any further, we need to add another dependency to our project, jQuery. If you look at your browser with the app running, and scale down its width, you will see the menu collapse to a button with three horizontal lines. This button doesn't work. It should open up the menu. Bootstrap is relying on jQuery here. Let's fix that.

```
npm install jquery
```

Run the above command, making sure you are in the **castletown** folder! Then open up **angular.json** and include jQuery in the **scripts** property:

castletown/angular.json (excerpt)

```
"scripts": [
  "node_modules/jquery/dist/jquery.min.js",
  "node_modules/bootstrap/dist/js/bootstrap.min.js"
]
```

To see that take effect, you will need to shut down the app (ctrl c/cmd c) if it's running in your browser, and serve it again via **ng serve**. Now that menu button should be working. Needless to say, you will have noticed that this website scales down well and will fit any device size.

Moving swiftly on, create 2 new components, 'training' and 'about', using the CLI as you did before for the homepage and news components. Open up **training.component.html** and alter it as follows:

castletown/src/app/pages/training/training.component.html

```
<app-header></app-header>
<div class="jumbotron gretta-jumbotron gretta-white-wrapper">
  <div class="container gretta-banner-container">
    <div class="row gretta-banner-row">
      <div class="col-lg-5 col-lg-push-7 gretta-banner-col1">
        <div class="gretta-banner-image">
        <p class="gretta-mobile-vanish"> </p>
        <figure class="figure">
          <img src="../../assets/images/training-01.svg" class="figure-img img-fluid rounded" alt="Train-
ing Schedule" height="451" width="451">
        </figure>
        </div>
      </div>
      <div class="col-lg-7 col-lg-pull-5 gretta-banner-col2">
        <div class="gretta-banner-content-wrap">
          <h2 class="gretta-content-header">Training Schedule</h2>
          <div class="gretta-content">
            <p class="text-left">
              <b>Mondays and Wednesdays</b><br>
              Juniors - 6pm<br>
              <em>Castletown AC Stadium</em>
              <br><br>
              <b>Tuesdays and Thursdays</b><br>
              Seniors - 7pm<br>
              <em>Castletown AC Stadium</em>
              <br><br>
              <b>Saturdays</b><br>
              Juniors & Seniors - 11am<br>
              <em>Castletown AC Stadium</em>
              <br><br>
              <b>Sundays</b><br>
              Juniors & Seniors - 11am<br>
              <em>Castletown AC Stadium</em>
            </p>
          </div>
        </div>
      </div>
    </div>
  </div>
</div>
<app-footer></app-footer>
```

Open up **about.component.html** and alter as follows:

castletown/src/app/pages/about/about.component.html

```
<app-header></app-header>
<div class="jumbotron gretta-jumbotron gretta-white-wrapper">
  <div class="container gretta-banner-container">
    <div class="row gretta-banner-row">
      <div class="col-lg-5 col-lg-push-7 gretta-banner-col1">
        <div class="gretta-banner-image">
        <p class="gretta-mobile-vanish"> </p>
```

```
      <figure class="figure">
        <img src="../../assets/images/about-01.svg" class="figure-img img-fluid rounded" alt="Train-
ing Schedule Image" height="451" width="451">
      </figure>
      </div>
    </div>
    <div class="col-lg-7 col-lg-pull-5 gretta-banner-col2">
      <div class="gretta-banner-content-wrap">
        <h2 class="gretta-content-header">Castletown AC History</h2>
        <div class="gretta-content">
          <p class="text-left">
            <b>The Early Years</b><br>
            Castletown Athletics Club was founded on a misty morning in the winter of 1972, when Kevin
Murphy was out running in the Dublin mountains, making a lonesome trek along the Kilmashogue
forest path. He saw another runner in the distance - Dave O'Grady - soon caught up with him - they
had a brief chat - and there and then decided that they would form a club. They both saw the need for
an athletics club in the area that would priortise track and field events. From a shed they rented on the
GAA playing pitches in Castletown, the newly born athletics club took its first tentative steps into the
world of running.<br><br>
            <b>Today</b><br>
            The club now boasts a membership of 550 people, mostly active runners, but some enthusi-
atic supporters on the sidelines also. In 2010, the club completed its multi-million athletics stadium,
with a 400 metre state-of-the-art track, indoor training facilities, and spectator stands. Both juveniles
and seniors have had great success in the last 5 years, both in national and international competi-
tions. Castletown AC's success is down to the dedication of its members, and the first class training
facilities the club provides to all its members, regardless of age or ability.<br><br>
            <b>The Future</b><br>
            We're hoping to see some Olympic medals proudly displayed in our awards cabinet in the
next few years! More generally, we hope to continue to increase club membership, in particular
amongst leisure runners who are over 30 years of age.
            <br><br>
          </p>
        </div>
      </div>
      <div class="gretta-banner-content-wrap mb-5">
        <h2 class="gretta-content-header">Gallery</h2>
        <div class="gretta-content">
          <div id="carouselExampleControls" class="carousel slide" data-ride="carousel">
            <div class="carousel-inner">
              <div class="carousel-item active">
                <img src="../../assets/images/gallery1.jpg" class="d-block w-100" alt="...">
              </div>
              <div class="carousel-item">
                <img src="../../assets/images/gallery2.jpg" class="d-block w-100" alt="...">
              </div>
              <div class="carousel-item">
                <img src="../../assets/images/gallery3.jpg" class="d-block w-100" alt="...">
              </div>
              <div class="carousel-item">
                <img src="../../assets/images/gallery4.jpg" class="d-block w-100" alt="...">
              </div>
            </div>
            <a class="carousel-control-prev" href="#carouselExampleControls" role="button" da-
ta-slide="prev">
              <span class="carousel-control-prev-icon" aria-hidden="true"></span>
              <span class="sr-only">Previous</span>
            </a>
            <a class="carousel-control-next" href="#carouselExampleControls" role="button" da-
```

```
ta-slide="next">
            <span class="carousel-control-next-icon" aria-hidden="true"></span>
            <span class="sr-only">Next</span>
          </a>
        </div>
      </div>
    </div>
  </div>
 </div>
</div>
<app-footer></app-footer>
```

We won't make any changes to the corresponding TypeScript files for these components. Nothing is needed. Above in the template files, you might normally be pulling this information (if it changes regularly i.e. training schedules or gallery images) from Wordpress in a **GET** request, as performed for the news component. However, to save time here, we will just hardcode it in. So finally, in order that we can insert these components into the view, we need to alter the routing module:

castletown/src/app/app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HomepageComponent } from './pages/homepage/homepage.component';
import { NewsComponent } from './pages/news/news.component';
import { TrainingComponent } from './pages/training/training.component';
import { AboutComponent } from './pages/about/about.component';

const routes: Routes = [
  {
    path: '', component: HomepageComponent
  },
  {
    path: 'news', component: NewsComponent
  },
  {
    path: 'training', component: TrainingComponent
  },
  {
    path: 'about', component: AboutComponent
  }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Now save the files and check your browser. Compare what you see to the corresponding pages on www.castletownac.com

Now that we have the routing module open, it is a good time to add in some extra features i.e. some basic SEO features - HTML title and meta tags. Angular provides us with a way of doing this, via its **Title** and **Meta** class services. Firstly, we will add that info into the objects of the **routes** array, as a **data** property:

castletown/src/app/app-routing.module.ts (excerpt)

```
const routes: Routes = [
 {
  path: '', component: HomepageComponent, data: {
   title: 'Home | Castletown AC',
   description: "Welcome to Castletown Athlectics Club website, one of Ireland's largest and most successful athletics and running clubs!",
   robots: 'noindex'
  }
 },
 {
  path: 'news', component: NewsComponent, data: {
   title: 'News | Castletown AC',
   description: 'Read about the latest news from Castletown Athletics Club.',
   robots: 'noindex'
  }
 },
 {
  path: 'training', component: TrainingComponent, data: {
   title: 'Training | Castletown AC',
   description: 'Find out what the training schedule is of Castletown AC.',
   robots: 'noindex'
  }
 },
 {
  path: 'about', component: AboutComponent, data: {
   title: 'About | Castletown AC',
   description: 'Read about the history of Castletown Athletics Club, from inception to the present day.',
   robots: 'noindex'
  }
 }
];
```

As I am creating a fictitious website, I don't want any pages indexed. You also, for your app, may have some resources that you don't want indexed. Ok, let's now open up our root component:

castletown/src/app/app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Meta, Title } from '@angular/platform-browser';
import { ActivatedRoute, Router, NavigationEnd } from '@angular/router';
import { filter } from 'rxjs/operators';

@Component({
 selector: 'app-root',
```

```
  templateUrl: './app.component.html'
})
export class AppComponent implements OnInit {

  constructor(private activatedRoute: ActivatedRoute, private titleService: Title, private metaService:
Meta, private router: Router) { }

  ngOnInit() {
    this.router.events.pipe(
      filter(event => event instanceof NavigationEnd),
    )
      .subscribe(() => {
        const route = this.getChild(this.activatedRoute);
        route.data.subscribe(data => {
          this.titleService.setTitle(data['title']);
          this.metaService.updateTag({ name: 'description', content: data['description'] })
          if (data['robots']) {
            this.metaService.updateTag({ name: 'robots', content: data['robots'] })
          } else {
            this.metaService.updateTag({ name: 'robots', content: "follow,index" })
          }
        })
      })
  }

  getChild(activatedRoute: ActivatedRoute) {
    if (activatedRoute.firstChild) {
      return this.getChild(activatedRoute.firstChild);
    } else {
      return activatedRoute;
    }
  }

}
```

Make the changes as you see above. It is beyond the scope of this book to explain the intricacies of what Angular is doing above, but I will summarize. We are injecting an instance of the **Router** service class into **AppComponent** so that we can **pipe**, **filter** and **subscribe** to the data stream provided by **Router**. Those last three terms are all RxJS features, which is built into Angular. You can see from above that if you don't provide robots data, it will automatically be filled in for you.

If you now open your browser, and look in the source code, you will see these title and meta tags have been added to every page that we have created so far. There is one final feature I want to add to that routing module:

castletown/src/app/app-routing.module.ts (excerpt)

```
import { ExtraOptions } from '@angular/router';

const routerOptions: ExtraOptions = {
  scrollPositionRestoration: 'enabled',
};
```

Insert that code just above the **routes** array. And now alter the **NgModule** decorator as follows:

castletown/src/app/app-routing.module.ts (excerpt)

```
@NgModule({
  imports: [RouterModule.forRoot(routes, routerOptions)],
  exports: [RouterModule]
})
```

This is a good option to apply i.e. the scroll position will go to the top of the new page you navigate to, rather than maintaining the current scroll position. It also enables anchor scrolling. This will become the default in future versions of Angular.

CHAPTER 7

SENDING DATA TO WORDPRESS

For our 'contact' page, we will perform a **POST** request to our Wordpress app, using the **HttpClient** injectable service class as before. Also, we will get a glimpse in this chapter into the power of Angular's form functionality - an important facitliy to have for the development of any sort of application. Angular provides two ways of doing this i.e. what the Angular devleopers call 'reactive' forms or 'template-driven' forms. We will use 'reactive' - a little more complex than 'template-driven', but ultimately more controllable and robust.

So firstly, please register the reactive forms module in your app:

castletown/src/app/app.module.ts (excerpt)

```
import { ReactiveFormsModule } from '@angular/forms';

@NgModule({
  imports: [
    // other imports ...
    ReactiveFormsModule
  ],
})
export class AppModule { }
```

Next, please create a new page component entitled 'contact', in the same manner we created all previous component pages. When that's done, alter the component file such that it looks like this:

castletown/src/app/pages/contact/contact.component.ts

```
import { HttpClient } from '@angular/common/http';
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';

@Component({
  selector: 'app-contact',
  templateUrl: './contact.component.html',
  styles: [
```

```
  ]
})
export class ContactComponent implements OnInit {

  constructor(private fb: FormBuilder, private http: HttpClient) { }

  contactForm: FormGroup = this.fb.group({
    firstName: ['', Validators.required],
    lastName: ['', Validators.required],
    email: ['', [Validators.required,Validators.email]],
    message: ['', Validators.required]
  });
  statusMessage: string = 'details unsubmitted';
  disableSubmitButton = false;

  ngOnInit(): void {
  }

onSubmit(){
    if (this.contactForm.controls['message'].errors?.['required']) {
      this.statusMessage = 'please enter your message';
    }
    if (this.contactForm.controls['email'].errors?.['email']) {
      this.statusMessage = "that's not an email!";
    }
    if (this.contactForm.controls['email'].errors?.['required']) {
      this.statusMessage = 'please enter your email';
    }
    if (this.contactForm.controls['lastName'].errors?.['required']) {
      this.statusMessage = 'please enter your last name';
    }
    if (this.contactForm.controls['firstName'].errors?.['required']) {
      this.statusMessage = 'please enter your first name';
    }
    if (this.contactForm.valid) {

      this.statusMessage = "submitting...";
      this.disableSubmitButton = true;

      const formData = new FormData();
      formData.append('firstName',this.contactForm.value.firstName);
      formData.append('lastName',this.contactForm.value.lastName);
      formData.append('email',this.contactForm.value.email);
      formData.append('message',this.contactForm.value.message);

      this.http
        .post('https://wp.castletownac.com/wp-json/castletown/v1/contact',formData)
        .subscribe(message => {
          if (message === 'success') {
            this.statusMessage = 'success!'
          }
          else {
            this.statusMessage = 'a problem occurred!'
            this.disableSubmitButton = false;
          }
        });
    }
  }

}
```

Two new classes (that we haven't seen before) are imported and injected i.e. **FormBuilder** and **Validators**. A third associated class, **FormGroup**, we also import. The names of these are, to some extent, self-explanatory, and we won't spend time going into too much detail. Suffice it is to say that **FormBuilder** has a method **group** which returns an instance of **FormGroup**. You will see shortly how we link this object (via the **FormGroup** directive) with the actual HTML form in our template. For now you can see how we have created properties (**firstName** etc.) which will correspond to the input elements' name attributes, and how we have used some basic validation. When we hit the submit button on our form, we will fire a method that we have created. You can call it anything. We call it 'onSubmit'. And then we introduce our **HttpClient** object **http** which you are familiar with, but this time we use the **post** method. The second argument, as you can see, is the data from the form. What we are doing with the returned data will become clearer in our endpoint **contact.php** file.

Open up the contact component's template, and have it look like this:

castletown/src/app/pages/contact/contact.component.html

```
<div class="gretta-blank-wrapper">
  <div class="gretta-blank-content">
    <form class="form-signin" [formGroup]="contactForm" (ngSubmit)="onSubmit()">
      <div class="text-center mb-4">
        <a routerLink="/"><img class="mb-4" src="../../assets/images/castletown-logo-01.svg" alt="-
Castletown AC Logo" width="300"
            height="72"></a>
        <h3 class="mb-3">Want to talk?</h3>
        <p>Please feel free to drop us a line below, to give us a call on <a href="tel:015240100">01
5240100</a> or to email us (<a href="mailto:info@castletownac.ie">info@castletownac.ie</a>) with
any questions you have.</p>
      </div>
      <div class="form-group">
        <input formControlName="firstName" type="text" class="form-control form-control-lg gret-
ta-form-control" placeholder="First Name" autofocus="">
      </div>
      <div class="form-group">
        <input formControlName="lastName" type="text" class="form-control form-control-lg gret-
ta-form-control" placeholder="Last Name">
      </div>
      <div class="form-group">
        <input formControlName="email" type="text" class="form-control form-control-lg gret-
ta-form-control" placeholder="Email">
      </div>
      <div class="form-group">
        <textarea formControlName="message" class="form-control form-control-lg gretta-form-con-
trol" placeholder="Your message" required rows="3"></textarea>
      </div>
      <div id="gretta-status-message"><b>Status:</b> {{statusMessage}}</div>
      <p class="robotic" id="pot">
        <label>If you're human leave this blank:</label>
```

```
      <input name="robotest" type="text" id="robotest" class="robotest">
    </p>
    <div id="gretta-replace" class="text-center">
     <button class="btn btn-lg btn-primary gretta-dglass-orange" id="btn-submit" type="submit"
[disabled]="disableSubmitButton">Submit</button>
    </div>
    <p class="mt-5 mb-3 text-muted text-center small"><i class="fa fa-envelope"></i>  Castle-
town Athlectics Club, 1 Castletown Drive, Castltown<br>Dublin D4E7Y3</p>
   </form>
  </div>
</div>
```

Firstly, we are binding our **contactForm** object to **formGroup**, which is a property of **FormGroupDirective**. Then we bind the individual properties of that object (**firstName** etc.) to **formControlName**, which is a property of the **FormControlName** directive. Without going too deeply into how it all works, **formGroup** allows us to bring all the form controls into a coherent whole. The form controls are themselves instances of **FormControl**.

Secondly, when the HTML **submit** event fires, the **ngSubmit** event fires. To this we bind our own **onSubmit** method, and so the code within that method is in turn invoked (as a side note, we could have captured the event data by passing in **$event** as an argument of **onSubmit**).

Now, let's have a look at the contact page in the browser. But before you do that, remember that you must add this new page to the routing module, just as we have done with previous pages. When that's done, make sure all files are saved, and then look at the browser. Compare your contact page to www.castletownac.com/contact - do you notice anything different? You should. The form is not centralized on the page. Let's fix that. Perform the following command:

```
ng g d directives/blank-page --skip-tests
```

Here we are creating our first ordinary directive. But note that components are actually directives also - they just have templates associated with them. As before, the above will declare this new directive in the app's root module. It won't create the testing file. Now open up this new file:

castletown/src/app/directives/blank-page.directive.ts

```
import { Directive, OnInit, Renderer2, Inject, OnDestroy } from '@angular/core';
import { DOCUMENT } from '@angular/common';

@Directive({
```

```
  selector: '[appBlankPage]'
})
export class BlankPageDirective implements OnInit, OnDestroy {

  constructor(@Inject(DOCUMENT) private document:any, private rend: Renderer2) { }

  ngOnInit(): void {
    this.rend.addClass(this.document.body, 'gretta-body-blank');
    this.rend.addClass(this.document.documentElement, 'gretta-html-blank'); //html element
  }

  ngOnDestroy(): void {
    this.rend.removeClass(this.document.body, 'gretta-body-blank');
    this.rend.removeClass(this.document.documentElement, 'gretta-html-blank');
  }

}
```

Make the changes as you see above. So let's explain this. To make the contact form centralized on the contact page, we need to add CSS classes to the **body** and **html** elements. But they are outside the scope of the app component. The app component's HTML element **<app-root>** is, you will recall, included in the index.html file, but it is inside the **body** and **html** elements. Therefore, we don't have control over these elements, in the traditional manner. We therefore access the whole of the **DOM** as you see above, using **Renderer2** and its associated methods to add our custom classes. The final piece in the jigsaw is to add this directive's selector to our contact template, as a HTML attribute:

castletown/src/app/pages/contact/contact.component.ts (excerpt)

```
<div class="gretta-blank-wrapper" appBlankPage>
  <div class="gretta-blank-content">
  //html code
  </div>
</div>
```

Now go back and have a look at your browser. The form should now be nicely centralized on the page.

We will now go the Wordpress side of the transaction. In the form, we are asking for the visitor's name, email address, and message. This info is then sent by email to Castletown AC's secretary. See how we do it:

themes/castletown/process/api-custom-routes.php

```
<?php
require('api-includes/news.php');
require('api-includes/contact.php');
```

```
function register_my_routes() {

 register_rest_route( 'castletown/v1', '/news', array(
   'methods'  => WP_REST_Server::READABLE,
   'callback' => 'news',
   ));

 register_rest_route( 'castletown/v1', '/contact', array(
   'methods'  => WP_REST_Server::CREATABLE,
   'callback' => 'contact',
   ));

}

add_action( 'rest_api_init', 'register_my_routes' );
```

We register the new route and endpoint as above. As we are now sending data to Wordpress, the method is now **CREATABLE**.

themes/castletown/process/api-includes/contact.php

```
<?php
function contact($request) {

#----------------------------------------------------------------
# Get request from Angular
#----------------------------------------------------------------
$params = $request->get_params();
$first_name = $params['firstName'];
$last_name = $params['lastName'];
$email = $params['email'];
$message = $params['message'];

#----------------------------------------------------------------
# Configure email
#----------------------------------------------------------------
$message = "Dear Secretary,<br> The contact form has been submitted with these de-
tails:<br>Name: $first_name $last_name<br>Email: $email<br>Message: $message";

// To send HTML mail, the Content-type header must be set
$headers[] = 'MIME-Version: 1.0';
$headers[] = 'Content-type: text/html; charset=iso-8859-1';

// Additional headers
$headers[] = "From: $first_name $last_name <$email>";
$headers[] = "Reply-To: $email";

mail('info@castletownac.com','Contact Form Submitted',$message, implode("\r\n", $headers));

#----------------------------------------------------------------
# Send response to Angular
#----------------------------------------------------------------
return rest_ensure_response('success');

}
```

Create the **contact** function file as you see above. The Wordpress REST API has provided a convenient way to access the request from Angular. You see **$request** and what we do with it. The original JSON data from Angular has been converted into a PHP object which has a **get_params** method. This returns an associative array of that form data, and you can see how we access it. We send the email and then we let Angular know that the function has executed successfully by sending back a simple string, "success".

CHAPTER 8

PREPARING FOR STRIPE PAYMENTS

Castletown AC's members pay a subscription every year of €100. The best way to do this is have them register on the website i.e. as WP users. Therefore there will be a log-in page for members who are already registered. If the credentials given are valid, they will be logged in to the payments page. If a member isn't registered, they can go to the registration page to register. They submit an email address and password of their choosing. To validate that email address, we will send them an email. That email will contain a link, which, when clicked, will take the member to the payments page. But the link URL will also include a token which Wordpress will generate and which will be valid for 24 hours only.

Once on the payments page, the credit card payment will be handled by Stripe in the background, without the user having to leave the page. Without further ado, let's create the log-in page. We will call it 'subs', as taking subscriptions is the only reason for having a log-in page in the first place. So create a page component **subs** as normal and alter the component file as follows:

castletown/src/app/pages/subs/subs.component.ts

```
import { HttpClient } from '@angular/common/http';
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
import { Router, ActivatedRoute, NavigationExtras } from '@angular/router';

@Component({
  selector: 'app-subs',
  templateUrl: './subs.component.html',
  styles: [
  ]
})
export class SubsComponent implements OnInit {

  contactForm: FormGroup = this.fb.group({
    email: ['', [Validators.required,Validators.email]],
```

```
    password: ['', Validators.required],
  });
  statusMessage: string = 'details unsubmitted';
  disableSubmitButton = false;

  constructor(private fb: FormBuilder, private http:HttpClient, private router:Router, private route:Acti-
vatedRoute) { }

  ngOnInit(): void {
  }

  onSubmit(){
    if (this.contactForm.controls['password'].errors?.['required']) {
      this.statusMessage = 'please enter your password';
    }
    if (this.contactForm.controls['email'].errors?.['email']) {
      this.statusMessage = "that's not an email!";
    }
    if (this.contactForm.controls['email'].errors?.['required']) {
      this.statusMessage = 'please enter your email';
    }
    if (this.contactForm.valid) {

      this.statusMessage = "submitting...";
      this.disableSubmitButton = true;

      const formData = new FormData();
      formData.append('email',this.contactForm.value.email);
      formData.append('password',this.contactForm.value.password);

      this.http
        .post('https://wp.castletownac.com/wp-json/castletown/v1/subs', formData)
        .subscribe(objResponse => {

          const NavigationExtras:NavigationExtras = { queryParams: {userId: objResponse['user_id'] }}
          console.log(objResponse)
          if (objResponse['message'] === 'success') {
            this.statusMessage = 'success!'
            this.router.navigate(['/payments'], NavigationExtras);
          }
          else {
            this.statusMessage = 'a problem occurred!'
            this.disableSubmitButton = false;
          }
        });
    }
  }

}
```

The subs page is very much like the contact page, with a form and information being sent to Wordpress.

Note that the members' Wordpress username is their email address. If Wordpress validates the username and password, then the member is brought to the payment page. Again, if they are incorrect, Wordpress will

send back an error and a member will stay stuck on the subs page.We will look at our WP code shortly.

But how is the member (if all validation checks are successful) brought to the payments page? You will notice that we have injected an instance of **Router**. This has a **navigate** method which will change the URL (and hence the view associated with it) if we allow it. We also add in a query paramater to the new URL, to help us identify the WP user on the payments page.

Now change the template for the subs component so that it looks like this:

castletown/src/app/pages/subs/subs.component.html

```
<div class="gretta-blank-wrapper" appBlankPage>
 <div class="gretta-blank-content">
  <form class="form-signin" [formGroup]="contactForm" (ngSubmit)="onSubmit()">
   <div class="text-center mb-4">
    <a routerLink="/"><img class="mb-4" src="../../assets/images/castletown-logo-01.svg" alt="-
Castletown AC Logo" width="300"
      height="72"></a>
    <h3 class="mb-3">Log in</h3>
    <p>Please log in to make your subscription payment.<br>Not registered? Click on <a href='/reg-
istration'>this link</a> to register.</p>
   </div>
   <div class="form-group">
    <input formControlName="email" type="text" class="form-control form-control-lg gret-
ta-form-control" placeholder="email" autofocus="">
   </div>
   <div class="form-group">
    <input formControlName="password" type="password" class="form-control form-control-lg
gretta-form-control" placeholder="password" autofocus="">
   </div>
   <div id="gretta-status-message"><b>Status:</b> {{statusMessage}}</div>
   <p class="robotic" id="pot">
    <label>If you're human leave this blank:</label>
    <input name="robotest" type="text" id="robotest" class="robotest">
   </p>
   <div id="gretta-replace" class="text-center">
    <button class="btn btn-lg btn-primary gretta-dglass-orange" id="btn-submit" type="submit"
[disabled]="disableSubmitButton">Submit</button>
   </div>
   <p class="mt-5 mb-3 text-muted text-center small"><i class="fa fa-envelope"></i>  Castle-
town Athlectics Club, 1 Castletown Drive, Castletown<br>Dublin D4E7Y3</p>
  </form>
 </div>
</div>
```

Again, this is very similar to our contact page template. Now update the routing module appropriately as before. Check your browser. Navigate to the subs page. All should be in order. You can compare it to the corresponding page on my Castletown website.

Ok, it's time to go over to Wordpress. Create the subs endpoint function file. But before you do that, register it in **api-custom-routes.php** as before, just like the contact custom route and endpoint. Make sure the method you choose is **CREATABLE**. Insert as follows into the subs function file:

themes/castletown/process/api-includes/subs.php

```
<?php
function subs($request) {

#--------------------------------------------------------------
# Retrieve info from Angular HTTP POST request
#--------------------------------------------------------------
$params = $request->get_params();
$userName = $params['email'];
$userPass = $params['password'];

#--------------------------------------------------------------
# Check username and password
#--------------------------------------------------------------
if ( is_wp_error( wp_authenticate( $userName, $userPass ) ) ) {
  return rest_ensure_response( array( 'message' => 'invalid' ) );
  exit;
}

$user = get_user_by('email', $userName);

#--------------------------------------------------------------
# Send success response back to Angular
#--------------------------------------------------------------
return rest_ensure_response( array( 'message' => 'success', 'url' => 'payments', 'user_id' => $us-
er->ID ) );

}
```

As before, we used the **get_params** method to get the data. Wordpress has two handy and simple functions which we use to validate: **wp_authenticate** and **is_wp_error**. If validation is successful, we tell Angular to proceed with navigating to the payments page. Otherwise we send back an error message.

The next chapter will deal with the payments page and setting up Stripe. So now we will complete this part of the process by creating our registration page (you will have noticed a link on the subs page pointing to the registration page - for members who aren't registered WP users).

Please create a 'registration' page component as usual. Insert the necessary routing information into the routing module. Alter the component file such that it looks like this:

castletown/src/app/pages/registration/registration.component.ts

```
import { HttpClient } from '@angular/common/http';
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';

@Component({
  selector: 'app-registration',
  templateUrl: './registration.component.html',
  styles: [
  ]
})
export class RegistrationComponent implements OnInit {

  constructor(private fb: FormBuilder, private http:HttpClient) { }

  registrationForm: FormGroup = this.fb.group({
    email: ['', [Validators.required,Validators.email]],
    password: ['', Validators.required],
    firstName: ['', Validators.required],
    lastName: ['', Validators.required]
  });
  statusMessage: string = 'details unsubmitted';
  disableSubmitButton = false;

  ngOnInit(): void {
  }

  onSubmit(){
    if (this.registrationForm.controls['password'].errors?.['required']) {
      this.statusMessage = 'please enter your password';
    }
    if (this.registrationForm.controls['password'].errors?.['minlength']) {
      this.statusMessage = 'your password is not long enough!';
    }
    if (this.registrationForm.controls['password'].errors?.['pattern']) {
      this.statusMessage = 'you must include at least one number and one letter!';
    }
    if (this.registrationForm.controls['email'].errors?.['email']) {
      this.statusMessage = "that's not an email!";
    }
    if (this.registrationForm.controls['email'].errors?.['required']) {
      this.statusMessage = 'please enter your email';
    }
    if (this.registrationForm.controls['lastName'].errors?.['required']) {
      this.statusMessage = 'please enter your last name';
    }
    if (this.registrationForm.controls['firstName'].errors?.['required']) {
      this.statusMessage = 'please enter your first name';
    }
    if (this.registrationForm.valid) {

      this.statusMessage = "submitting...";
      this.disableSubmitButton = true;

      const formData = new FormData();
      formData.append('email',this.registrationForm.value.email);
      formData.append('password',this.registrationForm.value.password);
```

```
    formData.append('firstName',this.registrationForm.value.firstName);
    formData.append('lastName',this.registrationForm.value.lastName);

    this.http
      .post('https://wp.castletownac.com/wp-json/castletown/v1/registration', formData)
      .subscribe(message => {
        if (message === 'success') {
          this.statusMessage = 'Please check your inbox!'
        }
        else {
          this.statusMessage = 'That email is already registered!'
          this.disableSubmitButton = false;
        }
      });
  }
 }

}
```

This is very similar to our contact and subs components. We just have some more validation to do because of the password creation. Now, update the template as follows:

castletown/src/app/pages/registration/registration.component.html

```
<div class="gretta-blank-wrapper" appBlankPage>
 <div class="gretta-blank-content">
  <form class="form-signin" [formGroup]="registrationForm" (ngSubmit)="onSubmit()">
   <div class="text-center mb-4">
    <a routerLink="/"><img class="mb-4" src="../../assets/images/castletown-logo-01.svg" alt="-
Castletown AC Logo" width="300"
      height="72"></a>
    <h3 class="mb-3">Registration Form</h3>
    <p>Please register to make your subscription payment.<br> Enter your email address as a user-
name and choose a password.</p>
   </div>
   <div class="form-group">
    <input formControlName="firstName" type="test" class="form-control form-control-lg gret-
ta-form-control" placeholder="First name" autofocus>
   </div>
   <div class="form-group">
    <input formControlName="lastName" type="test" class="form-control form-control-lg gret-
ta-form-control" placeholder="Last name">
   </div>
   <div class="form-group">
    <input formControlName="email" type="email" class="form-control form-control-lg gret-
ta-form-control" placeholder="email">
   </div>
   <div class="form-group">
    <input formControlName="password" type="password" class="form-control form-control-lg
gretta-form-control" placeholder="password" pattern="\S*(\S*([a-zA-Z]\S*[0-9])|([0-9]\S*[a-zA-
Z]))\S*" minlength="8">
    <div class="text-center"><small id="emailHelp" class="form-text text-muted">At least 8 charac-
ters (which must include a number)</small></div>
   </div>
```

```
    <div id="gretta-status-message" class="mt-4"><b>Status:</b> {{statusMessage}}</div>
    <p class="robotic" id="pot">
      <label>If you're human leave this blank:</label>
      <input name="robotest" type="text" id="robotest" class="robotest">
    </p>
    <div id="gretta-replace" class="text-center">
      <button class="btn btn-lg btn-primary gretta-dglass-orange" id="btn-submit" type="submit"
[disabled]="disableSubmitButton">Submit</button>
    </div>
    <p class="mt-5 mb-3 text-muted text-center small"><i class="fa fa-envelope"></i>  Castle-
town Athlectics Club, 1 Castletown Drive, Castletown<br>Dublin D4E7Y3</p>
   </form>
  </div>
</div>
```

Before we head over to Wordpress, please update the routing module with the new routing information for this component. Now check your browser and see that everything is functioning like it should. In your Wordpress installation, register a new custom route and endpoint for 'registration' in **api-custom-routes.php** just like before. Create the registration function file and insert as follows:

themes/castletown/process/api-includes/registration.php

```
<?php
function registration($request) {

#---------------------------------------------------------------
# Get request from Angular
#---------------------------------------------------------------
$params = $request->get_params();
$first_name = $params['firstName'];
$last_name = $params['lastName'];
$email = $params['email'];
$password = $params['password'];

#---------------------------------------------------------------
# Check if username exists. If not, create user and key
#---------------------------------------------------------------
//check if username exists
if ( username_exists($email) || email_exists($email) ) {
  return rest_ensure_response( 'email_exists' );
  exit;
}

//create user
$userdata = array(
  'user_login' =>  $email,
  'user_pass'  =>  $password,
  'first_name'  =>  $first_name,
  'last_name'  =>  $last_name,
  'user_email' => $email,
  'role' => 'subscriber'
);
```

```
$user_id = wp_insert_user( $userdata ) ;
$user = new WP_User($user_id);
$key = get_password_reset_key($user);

#--------------------------------------------------------------------
# Configure email
#--------------------------------------------------------------------
$message = "Dear Member,<br>Please click on the link below to verify your email address. You
will then be taken to Castletown AC's subscription payment page.<br><br><a href='https://www.
castletownac.com/payments?userKey=$key&userLogin=$email&userId=$user_id'>Verify Email Link</
a><br><br>Regards,<br>The Management@Castletown AC";

$headers[] = 'MIME-Version: 1.0';
$headers[] = 'Content-type: text/html; charset=iso-8859-1';
$headers[] = "From: Castletown AC <info@castletownac.com>";
$headers[] = "Reply-To: info@castletownac.com";

mail($email,'Email Verification',$message, implode("\r\n", $headers));

#--------------------------------------------------------------------
# Send response to Angular
#--------------------------------------------------------------------
return rest_ensure_response('success');

}
```

If the email isn't already registered as a username or an email, we create a new WP user with "subscriber" privileges. We then create a token (technically a password reset key) which expires in 24 hours against the new user. Next we send an email to the user, to check that the email exists and that what we're dealing with is a human. So the email has a link with "userLogin" (the email address in this case) and the token associated with it, as query parameters in the URL of that link. The URL is the payments page – and full access will be blocked if the username and token don't match (or if the token has expired). The URL I'm using above will of course need to be changed to match the base URL of your own web app. Of course, you could change that to **localhost:4200** when in development. You may also want to change the email address in the **$headers** array.

CHAPTER 9

MAKING STRIPE PAYMENTS

Stripe is an impressive payments platform and we will use it here to take subscription payments from the Castletown Athletics Club members. If you are new to Stripe, head over to stripe.com and set up an account for yourself. This can be done in a matter of minutes and you can then start testing the API. Of course, everything we do in this chapter will be in "test mode" on Stripe. We won't go into the intricacies of the Stripe API - that is beyond the scope of this book. I will provide you with the code provided by Stripe for a PHP & JavaScript app, but modified by me for Angular and our purposes - I won't go into too much detail, however.

There are a number of things we need to do to set up our app for taking payments. As you can appreciate, this isn't a simple matter. But Stripe does make it a smooth process, having said that. Firstly, open up the **index.html** file and change it such that it looks like this:

castletown/src/index.html

```
<!doctype html>
<html lang="en" class="gretta-standard">
<head>
  <script src="https://js.stripe.com/v3/"></script>
  <script src="https://polyfill.io/v3/polyfill.min.js?version=3.52.1&features=fetch"></script>
  <meta charset="utf-8">
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="assets/images/fav.ico">
</head>
<body class="gretta-grey-wrap">
  <app-root></app-root>
</body>
</html>
```

We have inserted the two necessary Stripe scripts. While we are in this file, I've also changed the default **fav.ico** to our own, and added in a few classes. To see these changes take effect, you will need to shut down the

app (ctrl c/cmd c) if it's running in your browser, and serve it again via **ng serve**. Ok, go ahead and create the payments page component.

```
ng g c pages/payments --s false
```

Use this command, because this time we actually need an associated CSS file. This is where the styles for the Stripe payment form go. Download that CSS styles file from here:

www.angular-wordpress.com/source-files/payments.component.css

Replace the existing file with this one. Next, open up the component file and change as follows:

castletown/src/app/pages/payments/payments.component.ts

```
import { Component, OnInit, ViewChild } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Renderer2, ElementRef } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
declare const Stripe: any;

@Component({
  selector: 'app-payments',
  templateUrl: './payments.component.html',
  styleUrls: [ './payments.component.css' ]
})
export class PaymentsComponent implements OnInit {

  constructor(private rend: Renderer2, private http: HttpClient, private route: ActivatedRoute ) {}

  @ViewChild('submit', {static: true}) submit: ElementRef;
  @ViewChild('cardError', {static: true}) cardError: ElementRef;
  @ViewChild('paymentForm', {static: true}) paymentForm: ElementRef;
  @ViewChild('spinner', {static: true}) spinner: ElementRef;
  @ViewChild('buttonText', {static: true}) buttonText: ElementRef;
  @ViewChild('resultMessage', {static: true}) resultMessage: ElementRef;

  userId: number; //WP user id retrieved from WP via the subs form
  userKey: string;
  userLogin: string;
  verificationSuccess = false;
  verificationFailure = false;

  ngOnInit(): void {
    this.userId = this.route.snapshot.queryParams['userId'];
    this.userKey = this.route.snapshot.queryParams['userKey'];
    this.userLogin = this.route.snapshot.queryParams['userLogin'];

    if (this.userKey && this.userLogin) {
      this.http
        .post('https://wp.castletownac.com/wp-json/castletown/v1/verification', {userKey: this.userKey,
```

```
userLogin: this.userLogin})
      .subscribe(response => {
        if (response === 'verification success') {
          this.verificationSuccess = true;
        }
        if (response === 'verification error') {
          this.verificationFailure = true;
          this.rend.setAttribute(this.submit.nativeElement, 'disabled', 'true');
        }
      });
  }

  // This is your test publishable API key.
  const stripe = Stripe("pk_test_xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx");

  this.http
    .post('https://wp.castletownac.com/wp-json/castletown/v1/stripe-test', {user_id: this.userId})
    .subscribe((response) => {
      console.log(response);
      const elements = stripe.elements();
      const style = {
        base: {
          color: "#000",
          backgroundColor: '#fff',
          fontFamily: 'Arial, sans-serif',
          fontSmoothing: "antialiased",
          fontSize: "16px",
          "::placeholder": {
            color: "#32325d"
          }
        },
        invalid: {
          fontFamily: 'Arial, sans-serif',
          color: "#fa755a",
          iconColor: "#fa755a"
        }
      };

      const card = elements.create("card", { style: style });
      // Stripe injects an iframe into the DOM
      card.mount("#card-element");

      card.on("change", (event) => {
        // Disable the Pay button if there are no card details in the Element
        if (event?.empty) { this.rend.setAttribute(this.submit.nativeElement, 'disabled', 'true'); }
        if (event?.complete) { this.rend.removeAttribute(this.submit.nativeElement, 'disabled'); }
        this.rend.setProperty(this.cardError.nativeElement, 'textContent', event?.error ? event.error.
message : "");
      });

      this.rend.listen(this.paymentForm.nativeElement, 'submit', (event) => {
        event.preventDefault();
        // Complete payment when the submit button is clicked
        payWithCard(stripe, card, response['clientSecret']);
      });

    });
```

```
  // Calls stripe.confirmCardPayment
  // If the card requires authentication Stripe shows a pop-up modal to
  // prompt the user to enter authentication details without leaving your page.
  function payWithCard(stripe, card, clientSecret) {
    loading(true);
    stripe
      .confirmCardPayment(clientSecret, {
        payment_method: { card: card }
      })
      .then(function (result) {
        if (result.error) {
          // Show error to your customer
          showError(result.error.message);
        } else {
          // The payment succeeded!
          orderComplete(result.paymentIntent.id);
        }
      });
  };

  /* ------- UI helpers ------- */

  // Shows a success message when the payment is complete
  const orderComplete = (paymentIntentId) => {
    loading(false);
    this.rend.removeClass(this.resultMessage.nativeElement, 'hidden');
    this.rend.setAttribute(this.submit.nativeElement, 'disabled', 'true');
  };

  // Show the customer the error from Stripe if their card fails to charge
  const showError = (errorMsgText) => {
    loading(false);
    this.rend.setProperty(this.cardError.nativeElement, 'textContent', errorMsgText);

    setTimeout(() => {
      this.rend.setProperty(this.cardError.nativeElement, 'textContent', '');
    }, 4000);

  };

  // Show a spinner on payment submission
  const loading = (isLoading) => {
    if (isLoading) {
      // Disable the button and show a spinner
      this.rend.setAttribute(this.submit.nativeElement, 'disabled', 'true');
      this.rend.removeClass(this.spinner.nativeElement, 'hidden');
      this.rend.addClass(this.buttonText.nativeElement, 'hidden');
    } else {
      this.rend.removeAttribute(this.submit.nativeElement, 'disabled');
      this.rend.addClass(this.spinner.nativeElement, 'hidden');
      this.rend.removeClass(this.buttonText.nativeElement, 'hidden');
    }
  };

 }

}
```

Quite a long file! Without going into the Stripe side of things in any great detail, a few things need explaining. Because Stripe injects an iframe into the DOM (the payment form) at runtime, we don't have access to it now. So we must use the **Renderer2** class and the **@ViewChild** decorator to access these HTML elements, in order to make changes to them. We are also using **ActivatedRoute** for the first time, to access the query parameters of the URL. Our first POST request to Wordpress relates to this - the 'verification' endpoint. If the parameters in the email link (which the unregistered member uses to access this page) are not valid, Wordpress will notify us, and we will thus bar access to the Stripe form, and indicate to the user that there has been a problem. The **userId** parameter will only be present if the user has come from the subs page, in which case they have already been cleared to access the payments page and form. And so we store this in the **userId** property, so that we know who the user is.

Next we indicate what our Stripe 'test publishable API key' is. You must change this to your own. Beneath that, you see our second POST request - note that these requests happen within **ngOnInit**. That's the endpoint where we do some of the initial Stripe processing i.e. set up a 'PaymentIntent' object. If all is in order, Stripe will set up the form on our frontend payments page. Stripe has a few different ways of processing payments - for our app, we are using the 'Card Element' option, and that's what Stripe injects into the DOM for us. Stripe will look after the validation in this form i.e. valid credit information, and if all is in order, these details will be submitted to Stripe, along with the 'client secret' key which our endpoint supplies in its response. (We also give Stripe some information about our member in our endpoint, but we will get to that shortly). Stripe then attempts to take payment, and will display the result, success or failure, in the UI. Indeed, as you can see, a lot of the code above concerns UI elements - altering the UI based on validation etc.

Let us now alter the component template:

castletown/src/app/pages/payments/payments.component.html

```
<app-header></app-header>
<div class="jumbotron gretta-jumbotron gretta-white-wrapper">
  <div class="container gretta-banner-container">
    <div class="row gretta-banner-row">
      <div class="col-lg-5 col-lg-push-7 gretta-banner-col1">
        <div class="gretta-banner-image">
          <p class="gretta-mobile-vanish"> </p>
          <figure class="figure">
            <img src="../../assets/images/payments-01.svg" class="figure-img img-fluid rounded"
              alt="Payments page for annual subscription." height="451" width="451">
          </figure>
        </div>
      </div>
```

```
      <div class="col-lg-7 col-lg-pull-5 gretta-banner-col2">
        <div class="gretta-banner-content-wrap pb-5">
          <h5 *ngIf="verificationFailure" class="mb-5">We couldn't verify your email address. Please
contact us.</h5>
          <h2 class="gretta-content-header">Subscriptions</h2>
          <div class="gretta-content">
            <p class="text-left">
              <b>Subscription Payment Form</b><br>
              <span *ngIf="verificationSuccess">Thanks for verifying your email address!</span> Please
click on the following button to pay your €100 yearly subscription for Castletown AC. Payment is
made securely by credit card or debit card.
            </p>
          </div>
          <div class="gretta-description-about">
            <form id="payment-form" #paymentForm>
              <div id="card-element"><!--Stripe.js injects the Card Element--></div>
              <button id="submit" #submit class="gretta-dglass-blue">
                <div class="spinner hidden" id="spinner" #spinner></div>
                <span id="button-text" #buttonText>Pay €100</span>
              </button>
              <p id="card-error" role="alert" #cardError></p>
              <p class="result-message hidden" #resultMessage>Payment succeeded, thanks!</p>
            </form>
          </div>
        </div>
      </div>
    </div>
  </div>
</div>
<app-footer></app-footer>
```

The only thing of significance to us here is Angular's **NgIf** structural directive. It displays the element it sits on only if the condition you bind to it is true. If the parameters in the link we have sent the member by email are invalid, an error message will be displayed - otherwise, a success message.

Once again, update the routing module with this payments component, so that it will be inserted into the view when we require it. And that's the last adjustment you'll have to make to the routing module for this project.

We now need to go to Wordpress. Create the first endpoint/route above in the component TypeScript file - 'verification' - by registering it in **api-custom-routes.php** as before. Then create the PHP **verification** function as follows, not forgetting to **require** it in **api-custom-routes.php**:

themes/castletown/process/api-includes/verification.php

```
<?php
function verification($request) {

#---------------------------------------------------------------
# Get request from Angular
#---------------------------------------------------------------
```

```
$params = $request->get_params();
$user_login = $params['userLogin'];
$user_key = $params['userKey'];

#--------------------------------------------------------------
# Verify user
#--------------------------------------------------------------
$result = check_password_reset_key( $user_key, $user_login );
if(is_wp_error($result)) {
        return rest_ensure_response( 'verification error' );
        exit;
}

#--------------------------------------------------------------
# Send response to Angular
#--------------------------------------------------------------
return rest_ensure_response('verification success');

}
```

Not much explaining needed here. If the email link we send the member has parameters that are invalid, we reject them, and block access to the Stripe payments form i.e disable the submit button. Otherwise, we give the green light. Next we need to create the 'stripe-test' route and end-point. Register it as normal in **api-custom-routes.php** - and create as follows:

themes/castletown/process/api-includes/stripe-test.php

```
<?php
function stripe_test($request) {

$params = $request->get_params();
$user_id = $params['user_id'];
#--------------------------------------------------------------
# Load the Stripe PHP library
#--------------------------------------------------------------
require(get_template_directory().'/dependencies/vendor/autoload.php');

#--------------------------------------------------------------
# Get the user
#--------------------------------------------------------------
$user = get_user_by('id', $user_id);

#--------------------------------------------------------------
# Create the paymentintent and send the client secret key back to Angular
#--------------------------------------------------------------
try {
\Stripe\Stripe::setApiKey('sk_test_xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx');

$customer = \Stripe\Customer::create(array(
   'email' => $user->user_email,
        'name' => $user->display_name
));
```

```
$paymentIntent = \Stripe\PaymentIntent::create(array(
    'amount' => 10000,
    'currency' => 'eur',
    'customer' => $customer->id
));

$client_secret = $paymentIntent->client_secret;

#---------------------------------------------------------------
# Send HTTP response back to Angular
#---------------------------------------------------------------
return rest_ensure_response(array('clientSecret' => $client_secret));

#---------------------------------------------------------------
# Check for errors
#---------------------------------------------------------------
} catch (Error $e) {
  http_response_code(500);
  $error = $e->getMessage();
  return rest_ensure_response($error);
}

}
```

First of all, this is our test file, hence the name. When this is working correctly and you are receiving payments successfully in test mode, you can create a duplicate and use that for the real thing - and swap out the API keys.

So let's quickly explain the function above. We retrieve the WP user id from the request, and then find that user's email and name. We create the Stripe customer based on that info, and then set up the PaymentIntent object, extracting the client secret key from that and sending it back to our Angular app. Now, for any of this work, we will of course need to install the Stripe PHP library on our server. Do it as follows, with Composer:

```
composer require stripe/stripe-php
```

If you refer to the file tree diagram of our project in Chapter 2, I install this in a folder **dependencies**, and I recommend you do the same. If you are new to Composer, see the appendix of this book for instatllation instructions. Ok, once that's done, you should be good to go. But you will want to test your payments page, of course. Stripe provides test card numbers for this purpose. Visit www.stripe.com/docs/testing for several choices. You can then see the results in the events section of your Stripe dashboard - in test mode of course.

CHAPTER 10

CONCLUSION

Before I send you on your merry way, there is one final feature I want to add to this website i.e. add the two most recent news items to the homepage. If you go to www.castletownac.com you will see these posts, but with images that have different dimensions from previously. Wordpress automatically crops the images for us to this size, because we specified this in the functions file. This addition gives the website a more polished and complete look.

So go ahead and open up the homepage component file, and change it such that it looks like this:

castletown/src/app/pages/homepage/homepage.component.ts

```
import { Component, OnInit } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Component({
  selector: 'app-homepage',
  templateUrl: './homepage.component.html',
  styles: [
  ]
})
export class HomepageComponent implements OnInit {

  constructor(private http: HttpClient) { }

  firstPostTitle = ''
  secondPostTitle = '';
  firstPostContent = '';
  secondPostContent = '';
  firstPostImg = '';
  secondPostImg = '';

  ngOnInit(): void {
    this.http.get('https://wp.castletownac.com/wp-json/castletown/v1/news')
      .subscribe(postArray => {
        this.firstPostTitle = postArray[0].post_title;
        this.secondPostTitle = postArray[1].post_title;
        this.firstPostContent = postArray[0].post_content;
        this.secondPostContent = postArray[1].post_content;
        this.firstPostImg = postArray[0].img2;
```

```
        this.secondPostImg = postArray[1].img2;
        this.firstPostContent = this.firstPostContent.split(' ').slice(0, 50).join(" ");
        this.secondPostContent = this.secondPostContent.split(' ').slice(0, 50).join(" ");
      });
  }

}
```

This will make more sense if you refresh your memory of the Wordpress endpoint function, so have a look:

themes/castletown/process/api-includes/news.php

```
function news() {

  $new_posts = array();
  $posts = get_posts(array('cat' => 1, 'author' => 1, 'post_status' => 'publish' ));

  foreach ($posts as $post) {
    $user = get_user_by('id',$post->post_author);
    $post->{'display_name'} = $user->display_name;
    $img1 = get_the_post_thumbnail_url($post->ID);
    $post->{'img1'} = $img1;
    $img2 = get_the_post_thumbnail_url($post->ID,'square');
    $post->{'img2'} = $img2;
    $new_posts[] = $post;
  }

  return rest_ensure_response($new_posts);

}
```

We are constructing a new array **$new_posts** from the original query **$posts** because it doesn't include the featured image URLs. We create **$new_posts** with that info, and send it back to Angular. We receive it in Angular as an array, and so in our component file we extract the properties that we want from the WP post objects in that array e.g. **post_title** etc. Note that we can't loop through these post objects like we did in our news component (with **NgFor**) because here we are displaying the second post differently from the first i.e. they are on different sides of the page. So in the component file we store the two different post object properties separately. You will note that we cut the post content down to 50 words, but without splitting any words, with a succint combination of JavaScript string methods. Now change our template as follows:

castletown/src/app/pages/homepage/homepage.component.html

```
<app-header></app-header>
<div class="jumbotron gretta-jumbotron gretta-blue-wrapper">
  <div class="container gretta-banner-container">
    <div class="row gretta-banner-row">
```

```
    <div class="col-lg-5 col-lg-push-7 gretta-banner-col1">
      <div class="gretta-banner-image">
        <img src="../../assets/images/homepage-01.svg" alt="Castletown AC Homepage Image"
itemprop="image"/>
      </div>
    </div>
    <div class="col-lg-7 col-lg-pull-5 gretta-banner-col2">
      <div class="gretta-banner-content-wrap">
        <h1 class="gretta-banner-heading" itemprop="name">Train With Us!</h1>
        <p><span class="gretta-description" itemprop="description">Welcome to Castletown Athlec-
tics Club website, one of Ireland's largest and most successful athletics and running clubs!</span></
p>
        <div class="gretta-mailing-list"><a routerLink="/training" role="button" class="btn btn-lg
btn-primary gretta-dglass-orange mb-5">Training Times</a><a routerLink="/contact" role="button"
class="btn btn-lg btn-primary mb-5 ml-5 gretta-dglass-blue">Contact us</a></div>
      </div>
    </div>
  </div>
 </div>
</div>
<div id="gretta-category-list2" class="gretta-home-category-wrapper mt-5">
 <div class="container gretta-home-category-container mt-0 pt-0 pb-0">
  <div class="row">
   <div class="col text-center gretta-content">
    <h2 class="gretta-content-header">Castletown AC, Dublin City</h2>
    <p><img src="../../assets/images/home.jpg" class="figure-img img-fluid rounded" alt="Castle-
town AC Profile Photo" width="700"></p>
    <p>Situated in Castletown Stadium, just minutes from Dublin city centre, we have a membership
of about 600 athletes, and we welcome all ages, from juvenile, through junior and senior, to masters.
We also cater for all standards, from beginners and recreational runners, to elite athletes, Olympians
and everything in between.</p>
   </div>
  </div>
 </div>
</div>
<!-- news content begins -->
<div id="gretta-category-list" class="gretta-home-category-wrapper mt-5">
 <div class="container gretta-home-category-container">
  <div class="row align-items-center">
   <div class="col-md-6">
    <figure class="figure">
   <img src="{{this.firstPostImg}}" class="figure-img img-fluid rounded" alt="{{this.firstPostTitle}}
Photo" height="451" width="451">
  </figure>
   </div>
   <div class="col-md-6 gretta-content-wrapper">
    <div class="gretta-content">
     <h2 class="gretta-content-header"><a routerLink="/news">{{this.firstPostTitle}}</a></h2>
     <p>{{this.firstPostContent}}...<a href="/news">Read more.</a></p>
    </div>
   </div>
  </div>
 </div>
 <div class="container gretta-home-category-container">
  <div class="row align-items-center">
  <div class="col-md-6 order-first order-md-last">
   <figure class="figure">
    <img src="{{this.secondPostImg}}" class="figure-img img-fluid rounded" alt="{{this.secondPost-
```

```
Title}} Photo" height="451" width="451">
      </figure>
    </div>
    <div class="col-md-6 gretta-content-wrapper">
      <div class="gretta-content">
        <h2 class="gretta-content-header"><a routerLink="/news">{{this.secondPostTitle}}</a></h2>
          <p>{{this.secondPostContent}}...<a href="/news">Read more.</a></p>
        </div>
      </div>
    </div>
  </div>
</div>
<!-- news content ends -->
<div id="gretta-category-list1" class="gretta-home-category-wrapper gretta-blue-container mt-5">
  <div class="container gretta-home-category-container pb-5">
    <div class="row">
      <div class="col text-center gretta-content">
      <h1 class="gretta-story-header gretta-banner-heading text-center">Get running now!</h1>
          <p>A club dedicated to excellence in fitness and health.</p>
          <a routerLink="/contact" role="button" class="btn btn-lg btn-primary gretta-dglass-orange
mb-5 mt-3">Join now</a>
      </div>
    </div>
  </div>
</div>
<app-footer></app-footer>
```

You will note from the comments where to insert the new HTML. Save and check your browser. You should now be elegantly pulling the latest 2 posts from Wordpress. So that completes our project. Well done! You are now the proud creator of a web app utilizing the Angular and Wordpress frameworks.

APPENDIX A

INSTALLING ANGULAR

To install Angular, you first of all need to install Node.js on your machine. Node.js is a server side language. You can download it from www.nodejs.org and during the installation, by default, NPM will automatically be installed for you. You will use NPM (Node Package Manager) to install the Angular CLI (which you use to set up an Angular app) or any number of other JavaScript packages in the NPM registry. During this course, we have used NPM to install Bootstrap and jQuery. It is essentially a command line interface allowing developers to install and publish public JavaScript modules of code.

So once you have NPM installed, open up a terminal window and insert the following command:

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

This will install the Angular CLI globally on your machine. Next, let's create an Angular app:

```
ng new my-app
```

The Angular CLI now installs the necessary Angular packages and other dependencies from the NPM registry inside a new folder **my-app**. This will take a short while to complete. It will create a new Angular app, entitled **my-app**, which is ready to run and has a starter welcome template - as you saw at the beginning of this book. We used 'castletown' instead of 'my-app'.

APPENDIX B

INSTALLING WORDPRESS & COMPOSER

Choosing a good host server for your application is the single most important decision you will make i.e. it is imperative that your app is served quickly to the client browser, that all communication between your app and the server is speedy, and that your host server provides good support and has a good UI for all your server needs. A slow delivery speed can negate all the good work you have put in, will affect the user's experience and will negatively influence your rating on search engines. So don't overlook this! If you need any recommendations, feel free to contact me at info@dnuapublishing.com and I will point you in the right direction. It's not the place of this book to openly support any commercial operation.

Once you have a server chosen, you will upload the Wordpress files available at wordpress.org/download to your server, and it's simply a matter then of following the on-screen instructions once you open up your root domain in a browser. Some server hosts will have a facility for installing Wordpress without the need for manual upload i.e. by means of Cpanel, if your server provides it.

The next step is to create your new theme folder. For this project, we call the folder **castletown** and place this empty folder within the **themes** folder, which you will find within **wp-content** in your Wordpress installation. There will already be a theme there by default i.e. the standard Wordpress theme for that year, and it will be active on your domain. Delete it. Every Wordpress theme requires at least 2 files, **index.php** and **style.css**, so go ahead and create them. The former has no content for this project, so you don't need to do anything with it other than place it within the **castletown** folder. Insert the following into **style.css** so that it looks like this:

themes/castletown/style.css

```
/*
Theme Name: Castletown
Theme URI: https://www.angular-wordpress.com
Author: John Manners
```

```
Author URI: https://www.angular-wordpress.com
Description: Castletown AC
Requires at least: 6.1
Tested up to: 6.1
Requires PHP: 5.6
Version: 1.0
License: GNU General Public License v2 or later
License URI: https://www.gnu.org/licenses/old-licenses/gpl-2.0.html
Text Domain: castletown
Tags: one-column, custom-colors, custom-menu, custom-logo, editor-style, featured-images,
full-site-editing, block-patterns, rtl-language-support, sticky-post, threaded-comments, transla-
tion-ready, wide-blocks, block-styles, accessibility-ready, blog, portfolio, news
*/
```

So, no actual code, but comments that are all important for the Wordpress engine. Change the values as necessary for your own requirements. And that's it! You have received instructions earlier in this book on what extra files you need to create to complete the Castletown project.

Now for Composer. It is not entirely dissimilar to NPM. It differs in a few important ways nonetheless. It is a dependency management tool for PHP, as opposed to a package management tool for JavaScript like NPM. Download it from getcomposer.org/download and use it as instructed at the end of Chapter 9. It is the best way to install all sorts of dependencies for your Wordpress/PHP project, such as the Youtube API etc.

dnuapublishing.com

www.dnuapublishing.com | Teicneolaíocht

ISBN 978-1-3999-4589-9

9 781399 945899

DUBHLINN NUA PUBLISHING