COM644 Full-Stack Web and App Development

Practical C1: Introducing Angular

Aims

- To appreciate the purpose of Angular in the MEAN stack
- To create a default Angular application
- To understand the code structure of an Angular applicaton
- To introduce the structure and role of an Angular controller
- To demonstrate injection of data values into the web presentation
- To introduce Angular directives
- To install Bootstrap through the Node Package Manager
- To style the initial application using Bootstrap cards

Contents

C1.1 INSTALLATION AND FIRST USE 2
C1.1.1 AngularJS and Angular 2
C1.1.2 Getting Started 2
C1.1.3 Basic Structure of an Angular App 5
C1.2 SPECIFYING CUSTOM COMPONENTS 7
C1.2.1 Creating a Component 7
C1.2.2 Connecting a Component to the Application 9
C1.3 USING ANGULAR DIRECTIVES 11
C1.3.1 Manipulating JSON Data 11
C1.3.2 Using the *ngFor Directive 12
C1.4 USING BOOTSTRAP 13
C1.4.1 Installing Bootstrap 13
C1.4.2 Styling with Bootstrap 14

C1.1 Installation and First Use

Angular is a front-end framework for the development of modern, responsive and scalable single-page Web applications. In this section of the module, we will use Angular to build a front-end to the API for the WeMeanBusiness application that we created in sections A and B.

C1.1.1 AngularJS and Angular

AngularJS was first developed at Google in 2009 as an MVC (Model, View, Controller) JavaScript-based framework for the rapid development of front-end web applications. It was publically launched as Version 1.0 in 2012 and quickly grew in popularity as an alternative to jQuery for the development of complex interactions.

The release of Version 2 in 2014 was the result of a complete re-write of the infrastructure and significant syntax changes, resulting in incompatibility between versions 1 and 2. As Version 1 had already become the most popular JavaScript framework by 2011 (Ref: https://www.infoworld.com/article/2612250/application-development/application-development-the-10-hottest-javascript-framework-projects.html) the change was met with some resistance, but simplifications in code structure and performance improvements saw the updated framework (now known simply as Angular) maintain its popularity.

The current version of Angular is v5, but it is important to recognize that versions v2, v4 and v5 (there was no v3!) are compatible with each other. Most changes were concerned with improvements under the bonnet – rather than in syntax and code structure. To emphasise this, v1 (which maintains a large user base) is commonly known as AngularJS, while versions 2+ are known simply as Angular.

One of the major changes between AngularJS and Angular is the introduction of TypeScript as a replacement for JavaScript. TypeScript was developed by Microsoft as a superset of JavaScript, adding optional data typing to the language. All JavaScript code is valid TypeScript and, although we will use TypeScript in this section of the module, we will not formally cover it – but will point out significant features as we meet them.

C1.1.2 Getting Started

Note: Angular has been provided for you on the lab machines but if you need to install it on your own computer, you can do so (as long as **`npm`** has previously been installed) by issuing the command

```
U:\> npm install –g @angular/cli
```

We will build up our front-end application in 6 stages, so create a directory called C1 for the first stage, navigate into it and build a default Angular application by the command

```
U:\> ng new C1
```

(Note that this will take a little while – even up to a couple of minutes on a high-powered computer – but we only need do it once!)

Figure C1.1 Creating a new Angular application

Although we have not yet provided any code, we can test that the application has installed properly by navigating into the C1 directory and launching it with the command

```
U:\C1> ng serve
```

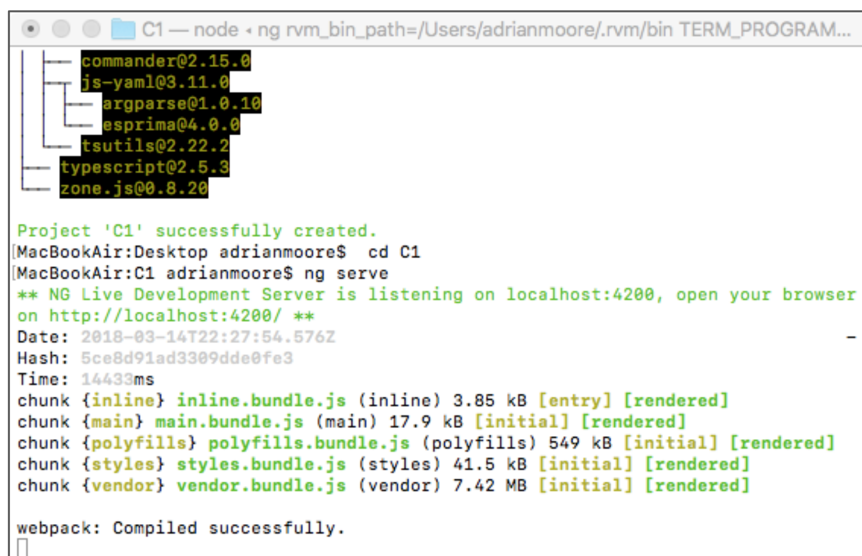

Figure C1.2 Running the frontend Server

Note: On the lab machines, you may see an error message such as

`Error – cannot find module '@angular-devkit/core'`

If so, you need to take the following steps…

1. Run the command **`npm update -g @angular/cli`**
2. Edit **package.json**, changing the entry for **`@angular/cli`** from "**`1.6.4`**" to "**`^1.6.4`**"
3. Run the command **`npm update`**

You should now be able to run **`ng serve`** and see the result as in Figure C1.2.

Now that the server is running, we can open a web browser and load the URL http://localhost:4200 to see the default Angular homepage as shown below.

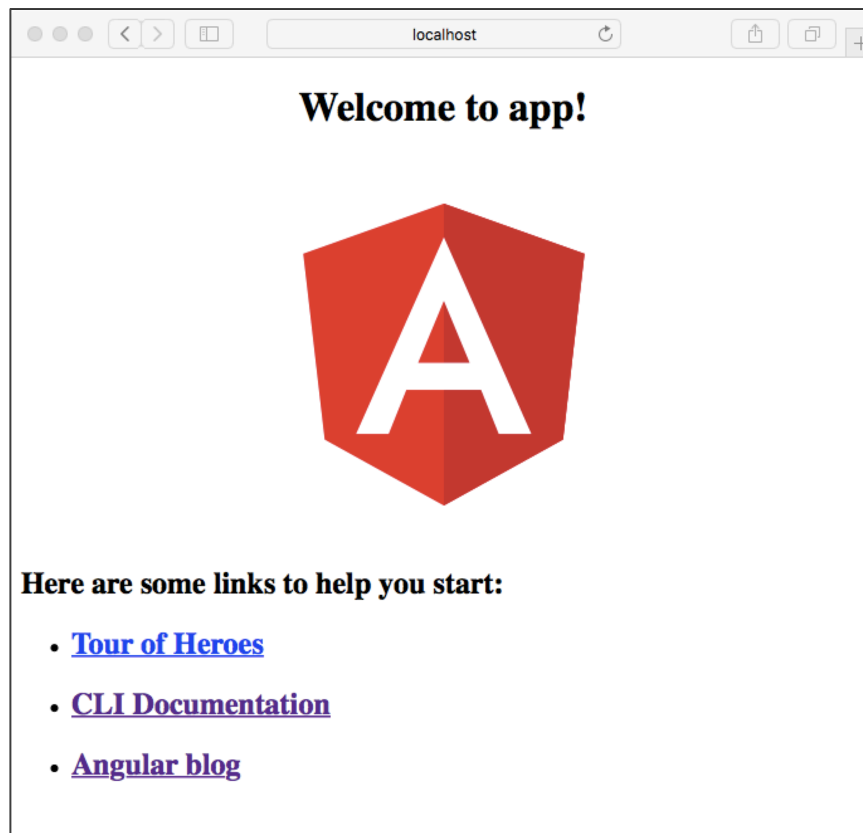

Figure C1.3 Default Server Output

C1.1.3 Basic Structure of an Angular App

The Angular CLI (Command Line Interface – invoked by the **`ng new`** command) creates all of the folders and files that will support our application development. All of the files that we will edit (and add) will be in the **src** directory – the other directories and files are Angular system files to support the application's development and execution.

The base of the frontend application is the file index.html, located in the **src** directory. If you examine the code in this file (shown in the code box below), you will see a standard HTML skeleton for a web application – enhanced by a curious **`<app-root></app-root>`** tag in the **`<body>`** section. This is an example of an ***Angular Controller*** – the basic building block of an Angular application. Controllers enable us to inject content onto the page at a position of our choosing and it is worth exploring the code to illustrate how this default controller generates the display shown in the browser.

File*: C1/src/index.html*

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Frontend</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>
```

A Controller is specified by a **TypeScript** file and an optional **HTML** template and **CSS** stylesheet. The files for controllers are defined in the ***app*** sub-directory of ***src***, so first look at the default controller TypeScript file, *app.component.ts*.

File*: C1/src/app/app.component.ts*

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  title = 'app';
}
```

The controller TypeScript file consists of 3 sections

i) The **`import`** statement that enables the file to make use of the facilities of the Angular **`Component`** library

ii) The **Decorator** that specifies the selector (HTML tag) to be used to refer to the component (here, **`app-root`** – corresponding to the **`<app-root>`** tag used in *index.html*) and the optional URLs for an HTML template and stylesheets to be used when rendering the component. Note that there is a single HTML template while the stylesheets are specified as a list.

iii) Finally, the **class definition** that contains the logic for the component. Here, we set up a single variable called **`title`** with the value '**app**'

Now, examine the code for the HTML template that renders the component in response to the **`<app-root>`** tag on index.html.

File*: C1/src/app/app.component.html*

```
<div style="text-align:center">
  <h1>
    Welcome to {{ title }}!
  </h1>
...
```

Note here how the component variable **`title`** defined in the class definition can be used within the template. As we see in the browser, the value of the variable is inserted into the **`<h1>`** tag by enclosing it in **`{{ }}`**.

Try it now!

Test the interaction of these files and develop your understanding of Angular code structure by attempting the following:

i) Change the value of the `title` variable in the component TypeScript file.

ii) Change the HTML template by removing some existing content or adding some of your own (but leave the `<h1>` element in place).

iii) Add a style rule to the CSS template file to have the `<h1>` element displayed in red text on a yellow background.

iv) Make the required changes so that the `<app-root>` component is called by the modified tag `<app-title>`. (Note: return it to `<app-root>` when you have shown that you can change it – we will refer to this element again later)

Note how the changes take effect in the browser immediately when you save the source file – the application is automatically re-loaded after each change.

C1.2 Specifying Custom Components

In this section, we will add a custom component to our front-end application. Eventually, this component will house our collection of businesses, but initially we will just display a simple heading.

C1.2.1 Creating a Component

The first stage is to create the files that will be required to house our component. Our component will be called '*BusinessesComponent*', so the first step is to create 3 empty files called

- *businesses.component.ts*,
- *businesses.component.html* and
- *businesses.component.css*

in the ***src/app*** folder of the ***C1*** application.

Now, we specify the content to be rendered by the component by adding a simple `<h1>` element to the HTML file

File*: C1/src/app/businesses.component.html*

```
<h1>We MEAN Business</h1>
```

and create the component TypeScript file by copying and pasting the content from *app.component.ts* into our new *businesses.component.ts* and changing the selector, template, style and class names to identify the new component as shown below.

File*: C1/src/app/businesses.component.ts*

```
import { Component } from '@angular/core';

@Component({
  selector: 'businesses',
  templateUrl: './businesses.component.html',
  styleUrls: ['./businesses.component.css']
})
export class BusinessesComponent { }
```

Next, we use the new selector by adding a **<businesses></businesses>** tag to the *app.component.html* file to render our new BusinessesComponent component.

File*: C1/src/app/app.component.html*

```
<div style="text-align:center">
  <h1>
    Welcome to {{ title }}!
  </h1>
</div>

<businesses></businesses>
```

If we now examine the browser and open the browser console, we can see that the application fails to display any content but instead produces an error message reporting that the **businesses** element is unknown.

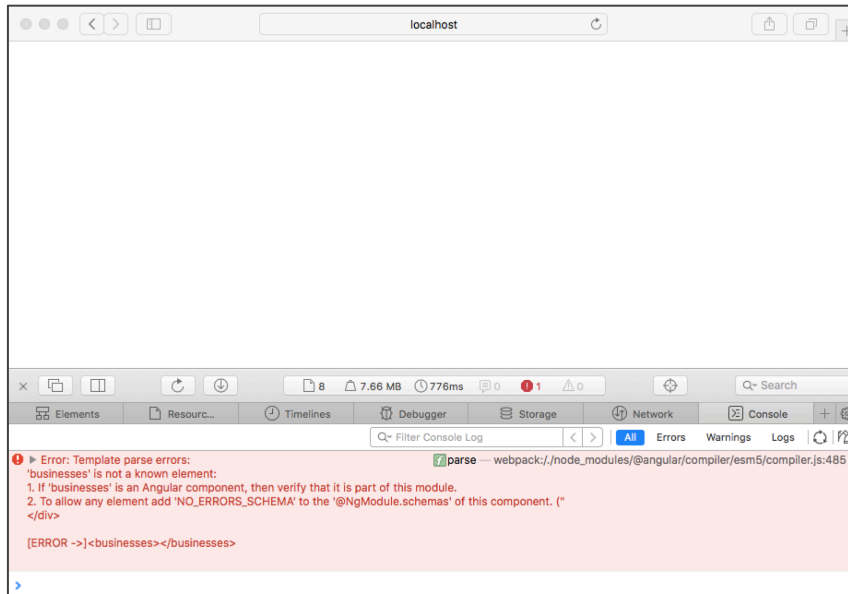

Figure C1.4 Unregistered component

C1.2.2 Connecting a Component to the Application

There are two further things we need to do to register the new component with the application. First, we need to import the new **BusinessesComponent** component into the *`app.component.ts`* file as it is rendered within its template.

File*: C1/src/app/app.component.ts*

```
import { Component } from '@angular/core';
import { BusinessesComponent } from './businesses.component';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'my app';
}
```

Then, we need to register the new component in the main *app.module.ts* file by importing it and adding it to the **`app.module`** declarations list.

File: ***C1/src/app/app.module.ts***

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';
import { BusinessesComponent } from './businesses.component';

@NgModule({
     declarations: [
       AppComponent, BusinessesComponent
     ],
     imports: [
       BrowserModule
     ],
     providers: [],
     bootstrap: [AppComponent]
 })

export class AppModule { }
```

Now, when we view the content in the browser, we can see that our new component has been properly rendered on the page.

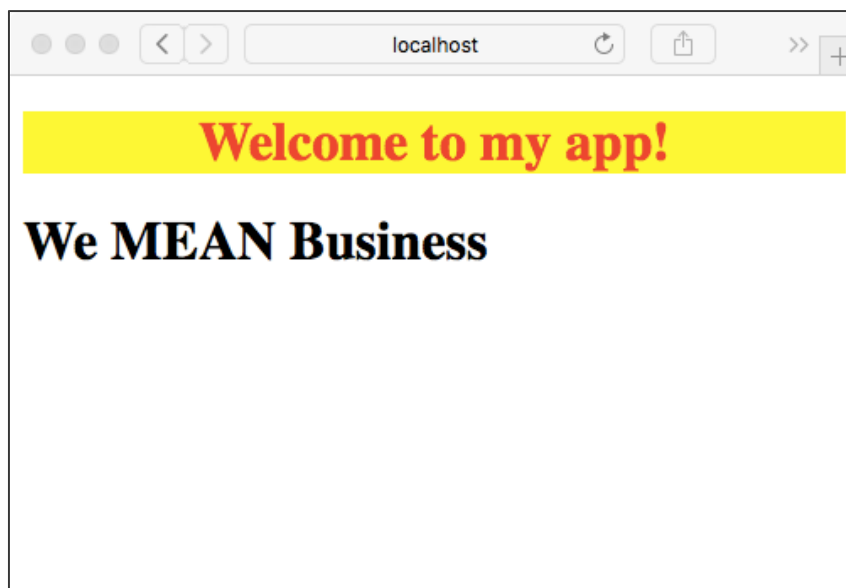

Figure C1.5 New component correctly rendered

C1.3 Using Angular Directives

So far, although our new component is connected to the application, its impact is limited to the display of a simple, static HTML element. We will now enhance the functionality of the component by having it process JSON data that we specify in the controller's TypeScript file.

C1.3.1 Manipulating JSON Data

In the **`BusinessesComponent`** class of the *businesses.component.ts* file, we define a variable **`business_list`** as a JSON object holding information about 3 business objects. For each business, we specify value for fields "*name*", "*city*" and "*review_count*" (reflecting 3 of the fields in our data set from the WeMEANBusiness application).

File*: C1/src/app/businesses.component.ts*

```
...

export class BusinessesComponent {

    business_list = [
        { "name": "Pizza Place",
          "city": "Coleriane",
          "review_count": 10 },
        { "name": "Wine Lake",
          "city": "Ballymoney",
          "review_count": 7 },
        { "name": "Beer Tavern",
          "city": "Ballymena",
          "review_count": 12 }
    ];

 }
```

Now, we add code to the *businesses.component.html* file to display the size of the JSON structure (i.e. the number of elements it contains) and the **`name`** property of the first element.

File*: C1/src/app/businesses.component.html*

```
<h1>We MEAN Business</h1>

<h2>There are {{ business_list.length }}
    businesses in the directory</h2>

<h2>The first business is
    {{ business_list[0].name }}</h2>
```

Examining the browser contents reveals that the information from the JSON structure is now embedded into the code that specifies the web page.

C1.3.2 Using the *ngFor Directive

Angular provides a number of very useful **directives** that enable us to introduce common programming constructs into our HTML specification. One of the most useful of these is the `*ngFor` directive, that provides a looping mechanism.

`*ngFor` takes a value in the form "`let` ***`variable`*** `of` ***`collection`***", by which the `variable` iterates across the `collection`, taking on each value in turn. To illustrate this, consider the code box below where the `*ngFor` element is applied to a `<div>` element that contains a paragraph that presents the *name*, *city* and *review_count* properties of each element of `business_list` in turn.

File*: C1/src/app/businesses.component.html*

```
...

<div *ngFor="let business of business_list">
    <p>Name: {{ business.name }} <br>
       City: {{ business.city }} <br>
       Reviews: {{ business.review_count }}
    </p>
</div>
```

The effect of this is to generate separate `<div>` elements for each business as shown in Figure C.6 below.

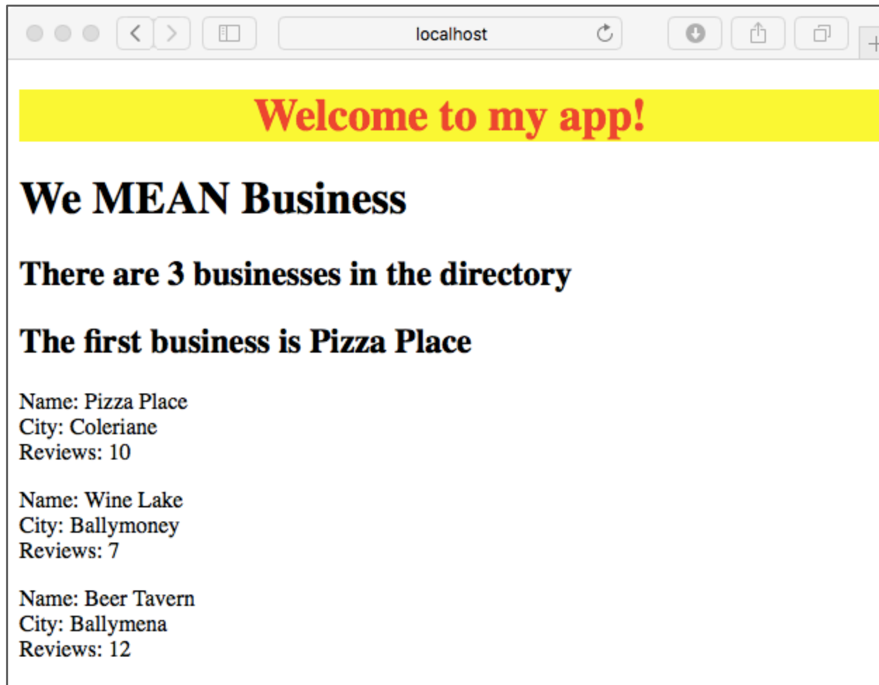

*Figure C1.6 Using *ngFor to Traverse a Data Set*

C1.4 Using Bootstrap

As we build the front-end of the *WeMEANBusiness* application, we will style the interface using Bootstrap. This will provide an easy way of creating an attractive interface, as well as automatically providing dynamic adjustment for different devices and browser characteristics.

C1.4.1 Installing Bootstrap

We could use Bootstrap by linking to a local copy or a CDN, but npm also provides it as a package that we can install into our application. To install Bootstrap and its dependent packages jQuery and Popper, kill any currently running Angular server, navigate back to the **C1** directory and issue the command

```
U:\C1> npm install bootstrap@4 jquery popper --save
```

Now, we need to tell the Angular application that the Bootstrap CSS library is available by adding it to the list of global stylesheets. Open the file **/.angular-cli.json** in the code editor and locate the `styles` entry. Now add the location of the file *bootstrap.min.css* to the styles entry as shown below.

File: ***C1/.angular-cli.json***

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

We can now check that Bootstrap has been installed by running the application by running **`ng serve`**, opening the browser console on the *Elements* tab and examining the **`head|styles`** section. Expanding the **`styles`** area verifies that Bootstrap has been included, as shown in Figure C.7.

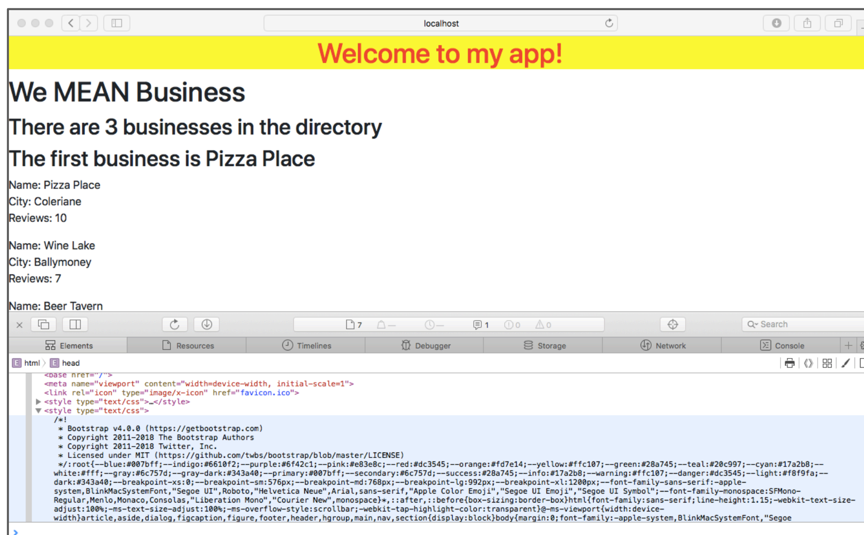

Figure C1.7 Verifying that Bootstrap is installed

C1.4.2 Styling with Bootstrap

As a quick example of Bootstrap styling, we will present our JSON data as Bootstrap **`card`** elements with a Bootstrap **`jumbotron`** header. A Bootstrap **`card`** is a flexible and extensible content container. It includes options for headers and footers, a wide variety of content, contextual background colors, and powerful display options.

The card is specified by applying style classes to **`<div>`** elements that define the card and its options header, body and footer components as shown below.

```
<div class = "card">
    <div class = "card-header">
        <!-- the card header text -->
    </div>

    <div class = "card-body">
        <!-- the card body text -->
    </div>

    <div class = "card-footer">
        <!-- the card footer text -->
    </div>
</div>
```

The remaining CSS classes applied (e.g. **text-white**, **bg-primary**, etc.) are Bootstrap classes to set text and background colours. A full specification of Bootstrap cards can be seen at https://getbootstrap.com/docs/4.0/components/card/. The code box below shows the revised specification for *businesses.component.html*.

File*: C1/src/app/businesses.component.html*

```
<div class="jumbotron">
    <h1>We MEAN Business</h1>
</div>

<div class="container">
  <div class="row">
     <div class="col-sm-12">
        <div *ngFor = "let business of business_list">
           <div class="card text-white bg-primary mb-3">
              <div class="card-header">
                   {{ business.name }}
              </div>
              <div class="card-body">
                This business is based in
                {{ business.city }}
              </div>
              <div class="card-footer">
                 {{ business.review_count }}
                 reviews available
              </div>
           </div>
        </div>
     </div> <!-- col -->
  </div>    <!-- row -->
</div>      <!-- container -->
```

Finally, we remove all remaining default code from *app.component.html* so that the entire presentation consists of our own work in the ***businesses*** component.

File: ***C1/src/app/app.component.html***

```
<businesses></businesses>
```

The effect of these changes can be seen in Figure C1.8, below.

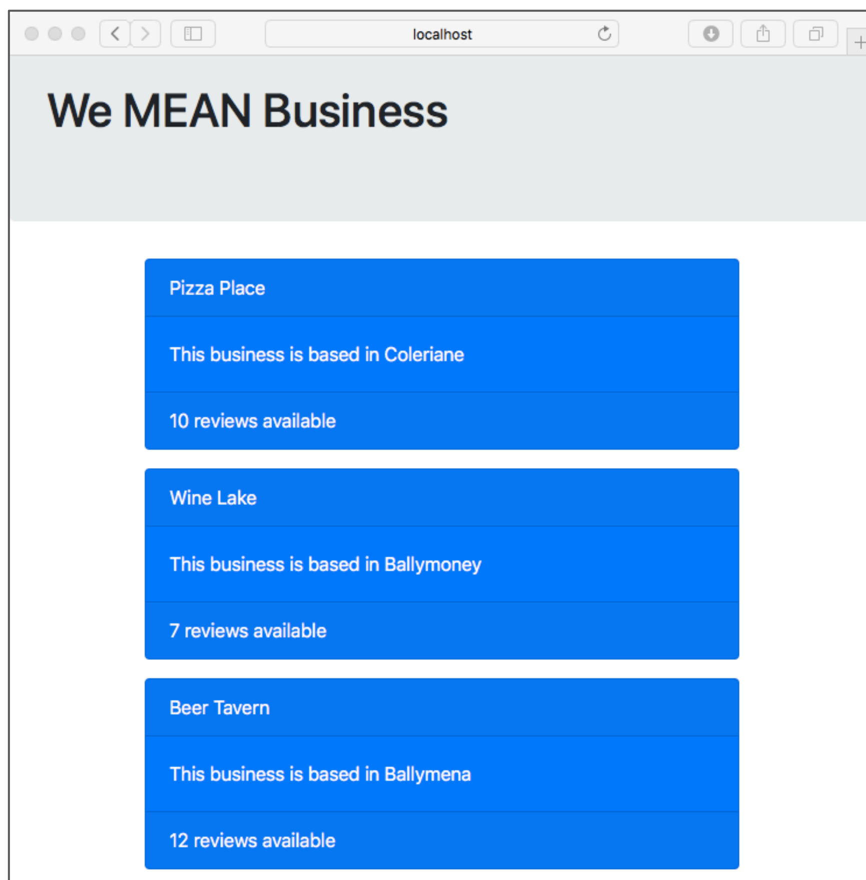

Figure C1.8 Using Bootstrap

Try it now!

Add more businesses to the data set and experiment with Bootstrap and cards to generate the most attractive layout with cards displayed 3 or 4 to a row.