COM644 Full-Stack Web and App Development

Practical C2: Web Services and Multiple URLs

Aims

- To introduce Angular Web Services as a means of retrieving data from the back end of an application
- To create a first Web Service
- To Introduce JavaScript Promises
- To deminstrate injecting a WebService into a Component
- To use the data returned from a Web Service in the front end of an Angular application
- To appreciate Cross Origin Resource Sharing (CORS)
- To introduce the Angular RouterModule
- To support multiple URLs in a single app via the RouterModule
- To implement internal linking via the routerLink element

Contents

C2.1 ANGULAR WEB SERVICES 2
C2.1.1 CREATING A WEB SERVICE 2
C2.1.2 USING THE WEB SERVICE 3
C2.1.3 CREATING A WEB SERVICE 7
C2.1.4 USING THE DATA RETURNED 9
C2.2 PROVIDING A ROUTING SERVICE FOR MULTIPLE URLS 12
C2.2.1 CREATING A BASIC HOME PAGE 12
C2.2.2 ROUTING IN AN ANGULAR APPLICATION 13
C2.3 AN ADDITIONAL ROUTE AND WEB SERVICE ENDPOINT 16
C2.3.1 PROVIDING THE BUSINESS CONTROLLER 16
C2.3.2 UPDATING THE WEBSERVICE 18
C2.3.3 MAKING THE CONNECTION 18

Our current version of the **We MEAN Business** frontend application displays business information as Bootstrap-styled card elements – but the business data is all hard-coded in our Angular Component.

In this practical, we will see how to retrieve the data from the backend of the application and build a simple database-driven app supporting 3 URLs – one to display a collection of businesses, one to display a single business and one to act as a home page.

C2.1 Angular Web Services

A Web Service is the component of an Angular application that interacts with the backend to retrieve (and store) data. We will deal with saving data to the database in a later practical, but as a starting point, let's create a basic Web Service to retrieve information on a collection of business objects.

C2.1.1 Creating a Web Service

In the interests of code maintainability, it is a good idea to keep the Web Service separate from the rest of the application, so we will create it in a new code file *web.service.ts* within the */src/app* folder.

First, we create and export a class to hold our new Web Service and define the function **`getBusinesses()`** that will return the data from the API. In order to fetch the data, the function will need to make an HTTP call to the backend, so we **`import`** the Angular **`Http`** module. To render the **`Http`** module available to the class, we need to inject it into the class constructor, so we specify a **`constructor()`** function that takes an instance of the **`Http`** object as a private parameter. Note the TypeScript syntax for specifying data types. Here, we are saying that the constructor has a parameter called **`http`** which is of type (is an instance of the) **`Http`** object.

File*: C2/src/app/web.service.ts*

```
import { Http } from '@angular/http';

export class WebService {

    constructor(private http: Http) {}

    getBusinesses() {
    }
}
```

C2.1.2 Using the Web Service

Now that an instance of the **`Http`** object is available within the class, we can use it inside the **`getBusinesses()`** function by invoking its **`get()`** method and passing it the URL of our API endpoint to fetch all businesses.

Note that by default, the **`get()`** method returns an **Observable** – a JavaScript structure we will examine later, but for now, we will adopt a more simple approach and convert the Observable to a **Promise** by passing the result to the **`toPromise()`** method.

Note: A JavaScript **Promise** is an object that represents the eventual result of an asynchronous operation. It allows code execution to continue, without waiting for the result of some time-consuming computation or I/O operation.

In the context presented here, the Promise is used to allow the **`http.get()`** instruction to behave in a non-blocking manner. As JavaScript is single-threaded, this allows execution to proceed more efficiently, with the code that depends on the result of the Promise deferred until the data is available.

Promises are made available through a library called *RxJs*, so we also import this into our Angular Service definition.

File*: C2/src/app/web.service.ts*

```
import { Http } from '@angular/http';
import 'rxjs/add/operator/toPromise';

@Injectable()
export class WebService {

    constructor(private http: Http) {}

    getBusinesses() {
        return this.http.get(
                    'http://localhost:3000/api/businesses')
               .toPromise();
    }
}
```

Finally, we need to specify our new element as a Service. Just as we specified a new component with an **@Component** decorator, so we need to also specify the service with an **@Injectable** decorator, which we also need to **import** from **@angular/core**

File*: C2/src/app/web.service.ts*

```
import { Http } from '@angular/http';
import 'rxjs/add/operator/toPromise';
import { Injectable } from '@angular/core';

@Injectable()
export class WebService {

    constructor(private http: Http) {}

    getBusinesses() {
        return this.http.get(
                  'http://localhost:3000/api/businesses')
               .toPromise();
    }
}
```

Before we can use the Service, we need to register it with the main module. In order to do this, we need to

i) import the **WebService** and **Http** module into *app.module.ts*;

ii) specify that the **WebService** is a Service by including it in the list of **providers**; and

iii) include the **HttpModule** in the **imports** list.

The code box that follows shows these additions made to the main *app.module.ts* file.

File*: C2/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';
import { WebService } from './web.service';
import { HttpModule } from '@angular/http';

@NgModule({
  declarations: [
    AppComponent, BusinessesComponent
  ],
  imports: [
    BrowserModule, HttpModule
  ],
  providers: [WebService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Now, we **`import`** the new **`WebService`** into our **`BusinessesComponent`** and prepare to receive the data from the API by removing the hard-coded JSON collection of businesses.

File*: C2/src/app/businesses.component.ts*

```
import { Component } from '@angular/core';
import { WebService } from './web.service';

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

export class BusinessesComponent {

    constructor(private webService: WebService) {}

    business_list = [
    ];
}
```

Once the **WebService** has been imported into the **BusinessesComponent**, we need to inject it into the constructor just as we did previously in the definition of the Service.

File*: C2/src/app/businesses.component.ts*

```
import { Component } from '@angular/core';
import { WebService } from './web.service';

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

export class BusinessesComponent {

    constructor(private webService: WebService) {}

    business_list = [
    ];
}
```

Next, we call the **WebService** by using the special Angular **ngOnInit()** function that is fired automatically whenever an object has been created

File*: C2/src/app/businesses.component.ts*

```
...

export class BusinessesComponent {

    constructor(private webService: WebService) {}

    ngOnInit() {
      this.webService.getBusinesses();
    }

    business_list = [
    ];
}
```

C2.1.3 Creating a Web Service

If we save this and run it in our browser (remember to start the **database server** and the **B6 backend application** first), we should see an error message in the browser informing us that there has been an `Access-Control-Allow-Origin` error. This means that we need to enable **Cross-Origin Resource Sharing (CORS)** on our backend application. This will allow one application (the frontend) to receive and accept data from another (the backend).

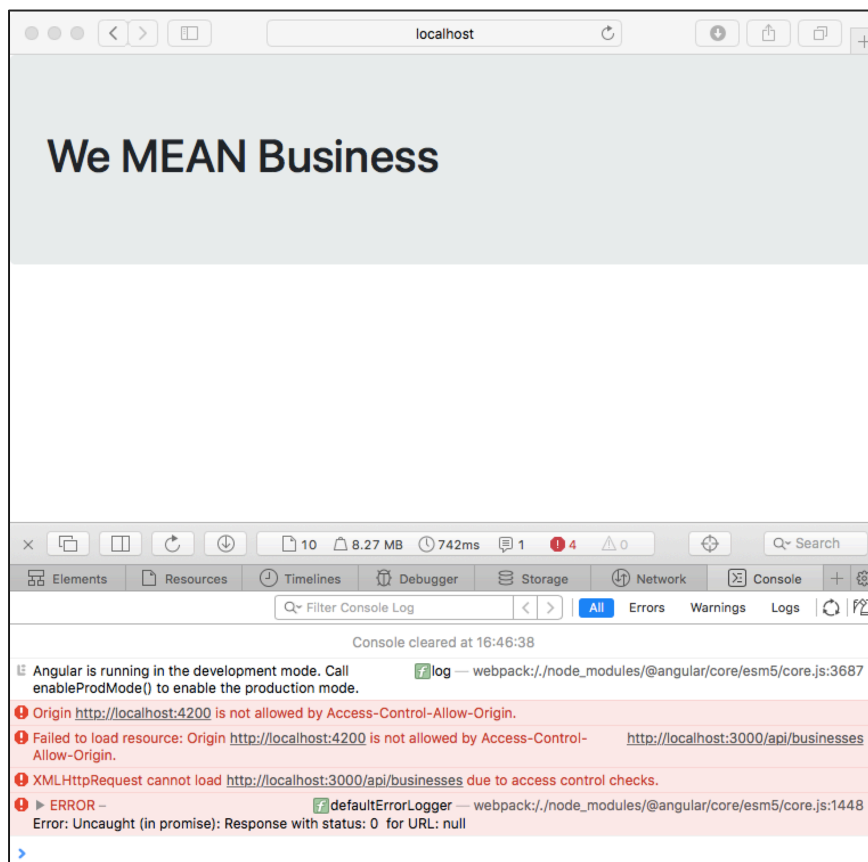

Figure C2.1 Access-Control-Allow-Origin error

This requires us to go back to the B6 application and add the lines identified in the code box below to *app.js*.

File: *B6/api/app.js*

```
...

app.use (function (req, res, next) {
    res.header('Access-Control-Allow-Origin', '*');
    res.header('Access-Control-Allow-Headers',
               'Origin, X-Requested-With, Content-Type,
                Accept');
    next();
});

app.use(express.static(path.join( __dirname, 'public' )));

...
```

Now, if we re-launch the backend application and refresh the browser, we should see that the error message has vanished – but we still have no data returned.

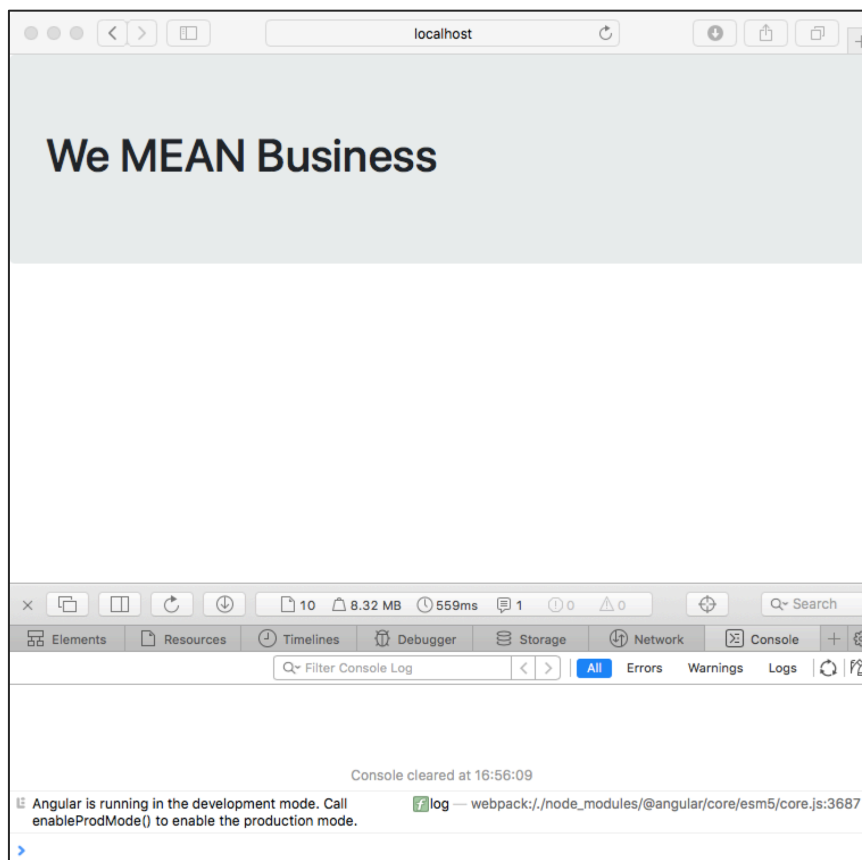

Figure C2.2 Access-Control-Allow-Origin error resolved

C2.1.4 Using the data returned

In order to make use of the data returned from the API, we need to do two things.

First, we need to assign the data returned to a variable (in this case, a variable called **`response`**). Then, we need to be mindful that the **`getBusinesses()`** function in the **`WebService`** returns a promise rather than data, so to use the promise we need to prepend the keyword **`await`** to the call to the function and then designate the function containing the call as being asynchronous by the keyword **`async`**.

File*: C2/src/app/businesses.component.ts*

```
...

export class BusinessesComponent {

    constructor(private webService: WebService) {}

    async ngOnInit() {
      var response = await this.webService.getBusinesses();
    }

    business_list = [
    ];
}
```

There is one final configuration step. Since **`await`** and **`async`** are features of JavaScript **ES6** (the latest version of JavaScript), we need to make sure that Angular is set up to deliver ES6 code. (Remember that all TypeScript is compiled to JavaScript when we issue the **`ng serve`** command).

To verify this, check in the file *tsconfig.json* and make sure that the **`target`** value in the **`compilerOptons`** object is set to **`es6`**.

File*: C2/tsconfig.json*

```
{
  "compileOnSave": false,
  "compilerOptions": {
    "outDir": "./dist/out-tsc",
    "sourceMap": true,
    "declaration": false,
    "moduleResolution": "node",
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "target": "es6",
    "typeRoots": [
      "node_modules/@types"
    ],
    "lib": [
      "es2017",
      "dom"
    ]
  }
}
```

Finally, we can test that the data is being retrieved from the API by using the **`json()`** method to access the data returned as a JSON object and logging it to the console.

File*: C2/src/app/businesses.component.ts*

```
...

export class BusinessesComponent {

    constructor(private webService: WebService) {}

    async ngOnInit() {
      var response = await this.webService.getBusinesses();
      console.log(response.json());
    }

    business_list = [
    ];
}
```

The result can be seen in Figure C2.3 below.

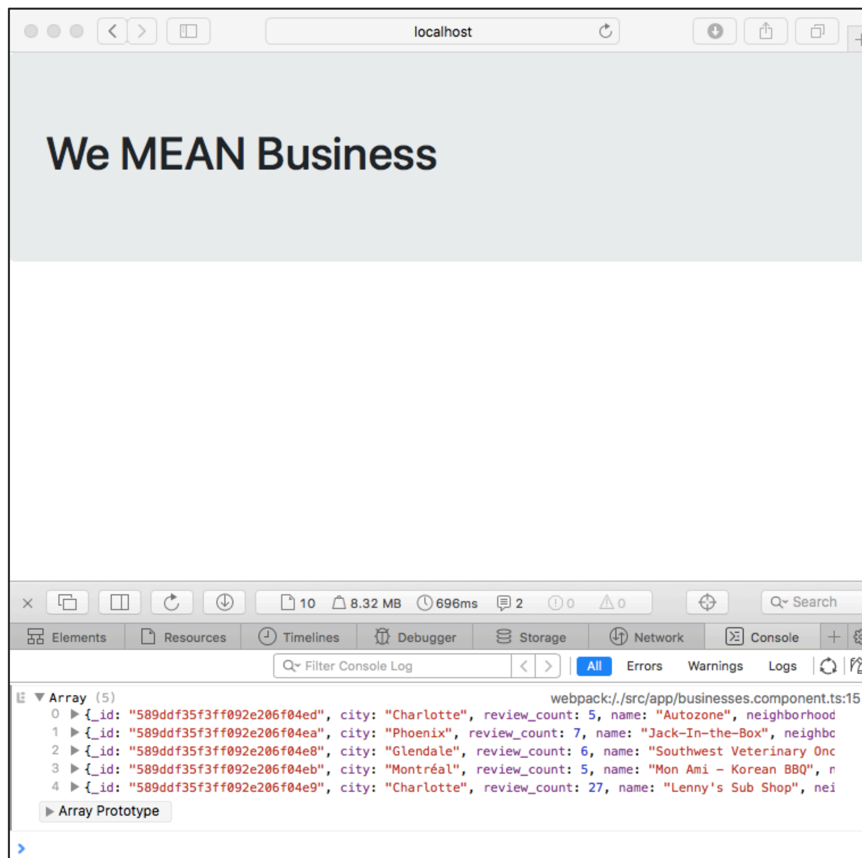

Figure C2.3 Data returned by the API

Now that we are confident that the data is being returned correctly, we can assign our previous **`business_list`** variable to the JSON data fetched and verify that it is being presented in the browser.

File: ***C2/src/app/businesses.component.ts***

```
...
export class BusinessesComponent {

    constructor(private webService: WebService) {}

    async ngOnInit() {
      var response = await this.webService.getBusinesses();
      this.business_list = response.json();
    }

    business_list = [
    ];
}
```

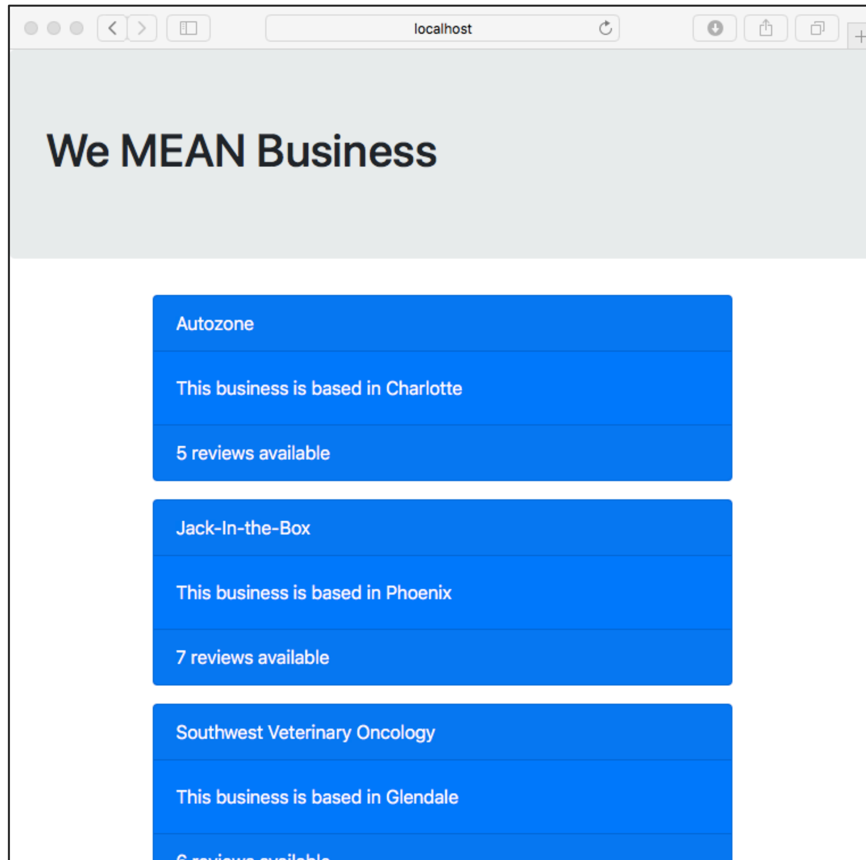

Figure C2.4 Data returned from API and displayed in the browser

C2.2 Providing a Routing Service for Multiple URLs

Although one of the main principles of Angular is that apps are single-page in nature, this essentially means that the browser is never completely re-loaded – even though the application supports multiple URLs to provide different views of the information being delivered.

In this section, we will create a plain page to use as the homepage for our application and then see how to specify different URLs to navigate between it and the previously implemented page that displays details of businesses.

C2.2.1 Creating a Basic Home Page

A static HTML page is easily generated by creating TypeScript, HTML and (optional) CSS files for a new Component, but by providing only the minimal TypeScript definition and having all content served by the HTML template.

Create new files in the **src/app** folder for *home.component.ts*, *home.component.html* and *home.component.css* and provide minimal content as shown below.

Note that the *home.component.ts* file is most easily created by copying the existing *businesses.component.ts* file and deleting/modifying the appropriate elements. We will leave *home.component.css* empty (for now).

File*: C2/src/app/home.component.ts*

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

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

File*: C2/src/app/home.component.html*

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

C2.2.2 Routing in an Angular Application

Angular provides a very useful **Router** module that manages multiple URLs within an application. To make use of this, we need to import it into our *app.module* TypeScript file and then include it in the module's **`imports`** list. Note that the entry in the **`imports`** list requires that we pass a **`routes`** element as a parameter to the **`forRoot()`** method, so we create an (initially) empty **`routes`** list and provide it to the **`RouterModule import`** specification.

File: ***C2/src/app/app.module.ts***

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

var routes = [];

@NgModule({
...

  imports: [
      BrowserModule, HttpModule, RouterModule.forRoot(routes)
  ],
...
```

Next, we need to remove references to the **BusinessesComponent** from the main *app.component* file, since the router will now control what is displayed.

First we remove the **import** statement that refers to the **BusinessesComponent** from *app.componment.ts*

File: ***C2/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';
}
```

and then remove the **<businesses>** element from *app.component.html*.

File: ***C2/src/app/app.component.html***

```
<!-- empty file -->
```

We need to replace this content with somewhere to display the currently selected route, so we add a special Angular element **`<router-outlet>`** to the *app-component.html* file.

File*: C2/src/app/app.component.html*

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

We can add the routes for the two components, setting the **`HomeComponent`** as the default route (/) while the **`BusinessesComponent`** will be accesses by the URL **/businesses**. To achieve this, we return to the **`routes`** list previously defined as empty in *app.module.ts* and add a pair of objects – each specified as a route (path) and a component.

Finally, we import the **`HomeComponent`** into *app.module.ts* and include it in the module's **`declarations`** list.

File*: C2/src/app/app.module.ts*

```
...

import { HomeComponent } from './home.component';

var routes = [
  {
    path: '',
    component: HomeComponent
  },
  {
    path: 'businesses',
    component: BusinessesComponent
  }
];

@NgModule({
  declarations: [
    AppComponent, BusinessesComponent, HomeComponent
  ],

...
```

Now, if we run the application and visit the URL http://localhost:4200 we should see the default home page, while http://localhost:4200/businesses displays the list of business details seen earlier.

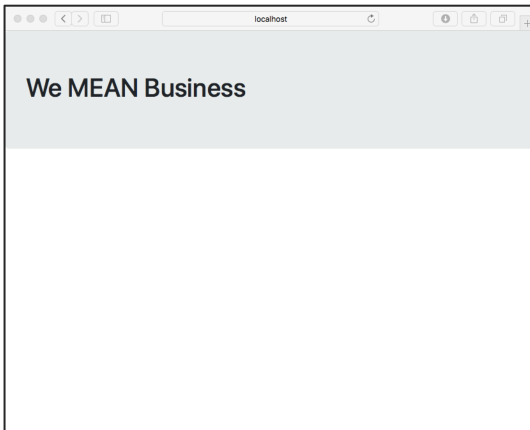

Figure C2.5 Home page
http://localhost:4200/

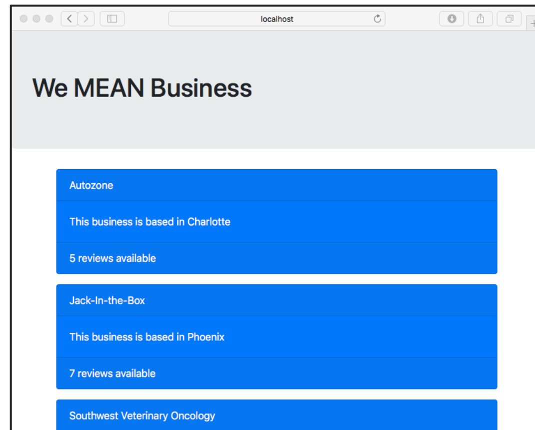

Figure C2.6 Businesses directory
http://localhost:4200/businesses

C2.3 An Additional Route and Web Service Endpoint

As a quick example of providing a new route and extending the **`WebService`**, we will consider the case where we provide a way of displaying only one of the collection of businesses.

C2.3.1 Providing the Business Controller

The Controller to handle the display of a single business is very similar to that displaying multiple businesses. The fastest way to specify these is to create new files for *business.component.ts*, *business.component.html* and *business.component.css* and to copy and modify the contents of the *businesses* equivalents as shown below.

In *business.component.ts*, we update the **`selector`**, **`templateUrl`** and **`styleUrls`** values in the Decorator, before making only minor changes to the class definition.

First, we specify a call to a new **`WebService`** function **`getBusiness()`** that takes an business ID as a parameter. Then, we assign the data returned from that function to a JavaScript object **`business`** (instead of the previous **`business_list`** array)

File*: C2/src/app/business.component.ts*

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

export class BusinessComponent {
    constructor(private webService: WebService) {}

    async ngOnInit() {
      var response = await this.webService.getBusiness(id);
      this.business = response.json();
    }

    business = { }
}
```

The file *business.component.html* will be exactly the same as for the **`BusinessesComponent`** except that we remove the **`<div>`** element containing the **`*ngFor`** directive, so that only a single Bootstrap card is produced with the data held in the class variable **`{{ business }}`**.

File*: C2/src/app/business.component.html*

```
...

<div class="container">
    <div class="row">
        <div class="col-sm-12">
            <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> <!-- col -->
    </div>     <!-- row -->
</div>         <!-- container -->
```

C2.3.2 Updating the WebService

Our code in the **`BusinessComponent`** Typescript file makes reference to a new **`WebService`** endpoint called **`getBusiness()`**, which takes a business ID as a parameter. We can now update the *web.service.ts* code to include this new endpoint as a function that calls the API endpoint http://localhost:3000/businesses/12345, where 12345 is the **`_id`** value of one of the business objects.

File*: C2/src/app/web.service.ts*

```
...

export class WebService {

    constructor(private http: Http) {}

    getBusinesses() {
        return this.http.get(
                    'http://localhost:3000/api/businesses')
                .toPromise();
    }

    getBusiness(id) {
        return this.http.get(
                    'http://localhost:3000/api/businesses/'+id)
                .toPromise();
    }
}
```

C2.3.3 Making the connection

Now that we have the **`BusinessComponent`** and the updated **`WebService`** in place, we can add the route and create a clickable link from the list of businesses.

First, we add the route by importing the new **`BusinessComponent`** into *app.module* and adding it to the **`declarations`** list. We also provide a new entry into the **`routes`** array in *app.module.ts* which specifies the **`id`** parameter using the same **`:id`** syntax that we have previously seen in other instances of application routing.

File*: C2/src/app/app.module.ts*

```
...

import { BusinessComponent } from './business.component';

var routes = [
  {
    path: '',
    component: HomeComponent
  },
  {
    path: 'businesses',
    component: BusinessesComponent
  },
  {
    path: 'businesses/:id',
    component: BusinessComponent
  }
];

@NgModule({
  declarations: [
    AppComponent, BusinessesComponent, HomeComponent,
    BusinessComponent
  ],

...
```

Finally, we need to make the individual card entries in the **BusinessesController** HTML template clickable by adding a **routerLink** entry to each **card** element.

The **routerLink** entry specifes that clicking on the element to which it is applied will result in that destination being displayed in the **<router-outlet>** element provided earlier in *app.component.html*.

There are three important elements to the highlighted code here.

i) We provide the two components to the route ('businesses' and the id value) as an array of values. Note the **business._id** property returning the **_id** value for the selected business.

ii) As we require the actual **business._id** value to be inserted into this array (i.e. we do not want the literal string "business._id"), we enclose the **routerLink** attribute in [ ] brackets. This is a common feature of Angular that we will explain further in a later practical.

iii) We use the inline **`style`** rule to have the cursor change to a pointer when it is positioned over the card object. This is a visual cue to the user that the card is a clickable element on the page.

File*: C2/src/app/businesses.component.html*

```
...

<div class="card text-white bg-primary mb-3"
           [routerLink]="['/businesses', business._id]"
           style="cursor: pointer">
...
```

The final task is to retrieve the **`id`** parameter from the route and insert it into the call to the **`WebService`** function.

First, we include the **`ActivatedRoute`** module into *business.component.ts*, which is the TypeScript file in which the parameter value is required. We also inject this into the constructor by specifying a parameter **`route`** as an instance of **`ActivatedRoute`**.

Finally, we update *business.components.ts* to retrieve the **`id`** value passed in the URL by the somewhat clunky syntax **`route.snapshot.params.id`**.

File*: C2/src/app/business.component.ts*

```
...

import { ActivatedRoute } from '@angular/router';

...

export class BusinessComponent {

    constructor(private webService: WebService,
                private route: ActivatedRoute) {}

    async ngOnInit() {
      var response =
          await this.webService.getBusiness(
                              this.route.snapshot.params.id);
      this.business = response.json();
    }

    business = { }
}
```

Testing the application in the browser now presents the list of businesses as clickable card elements, while clicking on one of them demonstrates our new route and controller that displays information about a single business.

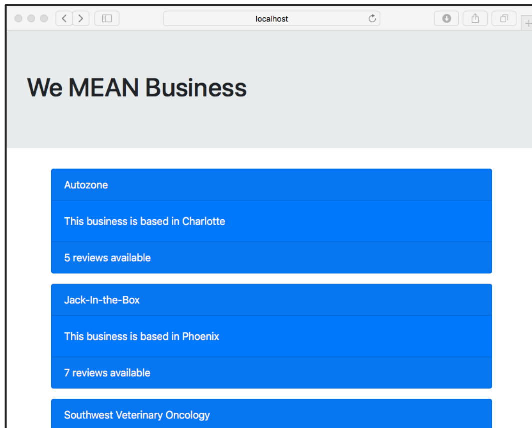

Figure C2.7 A clickable list of businesses http://localhost:4200/businesses

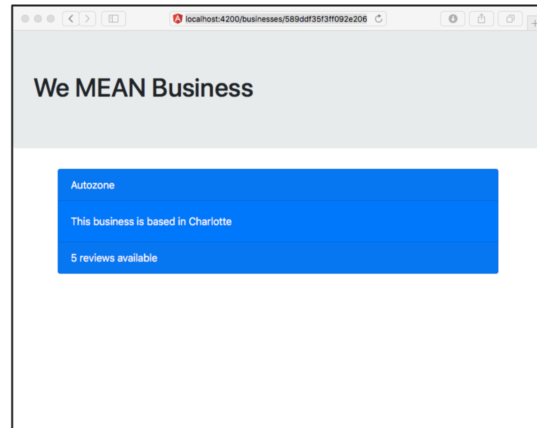

Figure C2.8 Clicking on a single business http://localhost:4200/businesses/12345