COM661 Full Stack Strategies and Development

Practical D2: 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 Observables
- To demonstrate 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

Contents

D2.1 ANGULAR WEB SERVICES 2

D2.1.1 CREATING A WEB SERVICE 2
D2.1.2 USING THE WEB SERVICE 3
D2.1.3 RUNNING A WEB SERVICE 8
D2.1.4 USING THE DATA RETURNED 9

D2.2 PROVIDING A ROUTING SERVICE FOR MULTIPLE URLS 11

D2.2.1 CREATING A BASIC HOME PAGE 11
D2.2.2 ROUTING IN AN ANGULAR APPLICATION 12

D2.4 FURTHER INFORMATION 14

D2.1 Angular Web Services

Our current version of the **Biz Directory** front-end 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 back-end application developed in Sections B and C of the module and build a simple database-driven app supporting 2 URLs – one to display a collection of businesses, and one to act as a home page.

Note:	This practical (and future practicals) will use the API developed as the demonstrator for the assessment – i.e. the version of the Biz API that uses the **`businesses`** collection in the **`yelpDB`** database. You should make sure that your back-end application is up-to-date before proceeding.

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, we will create a basic Web Service to retrieve information on a collection of business objects.

D2.1.1 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 **`HttpClient`** module. To render the **`HttpClient`** 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 **`HttpClient`** 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) **`HttpClient`** object.

File: D2/src/app/web.service.ts

```
import { HttpClient } from '@angular/common/http';

export class WebService {
    constructor(private http: HttpClient) {}

    getBusinesses() {}
}
```

Do it now!	Create the file ***web.service.ts*** and populate it with the code as shown above.

D2.1.2 Using the Web Service

Now that an instance of the **`HttpClient`** 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 that acts as a producer of multiple variables, "pushing" them to subscribers. We will see how to subscribe to the stream of data returned soon, but for now, we will create the remainder of the code infrastructure.

File: D2/src/app/web.service.ts

```
import { HttpClient } from '@angular/common/http';

export class WebService {

    constructor(private http: HttpClient) {}

    getBusinesses() {
        return this.http.get(
                  'http://localhost:5000/api/v1.0/businesses'
                );
    }
}
```

Next, 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: D2/src/app/web.service.ts

```
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';

@Injectable()
export class WebService {
    constructor(private http: HttpClient) {}

    getBusinesses() {
        return this.http.get(
                  'http://localhost:5000/api/v1.0/businesses'
                );
    }
}
```

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 **`HttpClient`** modules into *app.module.ts*;

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

iii) include the **`HttpClientModule`** in the **`imports`** list.

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

File: D2/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 { HttpClientModule } from '@angular/common/http';

@NgModule({
  declarations: [
    AppComponent, BusinessesComponent
  ],
  imports: [
    BrowserModule, HttpClientModule
  ],
  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: D2/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 {
  business_list: any;
}
```

Note:	Note also how we provide a type of **any** for the local **business_list** variable. This is required when we don't assign a default value, as TypeScript's strict type checking needs to know what type of data to expect. Setting the type to **any** allows us to use the variable as we see fit later.

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: D2/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 {
    business_list: any;

    constructor(private webService: WebService) {}
}
```

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

File: D2/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 {
    business_list: any;

    constructor(private webService: WebService) {}

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

Finally, for this stage, we need to add to our Web Service the capability to **subscribe** to the stream of data that will be returned from the API.

The function provided as the parameter to the **`subscribe()`** method will be automatically called each time the Observable is activated (i.e. each time the **`http.get()`** method returns from the back-end with data). When the function is invoked, it accepts the data returned from the API and copies it to the new local variable **`business_list`**. We also echo the response from the back-end to the browser console for visual confirmation that it is being returned.

File: D2/src/app/web.service.ts

```
...

    business_list: any;

    constructor(private http: HttpClient) {}

    getBusinesses() {
        return this.http.get(
                    'http://localhost:5000/api/v1.0/businesses')
            .subscribe((response: any) => {
                this.business_list = response;
                console.log(response)
            })
    }

...
```

Note how the function provided as a parameter to **`subscribe()`** is written in the "*fat arrow*" style. Fat arrow functions (sometimes just called "arrow functions") are a concise notation for short (often single line) functions that avoid use of the **`function`** and **`return`** keywords. The parameter to the function is on the left-hand side of the **=>** symbol, while the code to be executed is on the right-hand side. Fat arrow functions also have one important property that makes them particularly useful in this case – unlike "normal" functions, they do not create a new scope layer – i.e. the value of the keyword **`this`** is the same inside the function as outside. If the **`subscribe`** function were written in the normal style, we would either need to bind the value of **`this`** to the calling function or pass a reference to the function as an additional parameter.

Do it now!	Follow the steps described above to add the HTTP GET request to the Web Service and connect the service to the Angular application.

D2.1.3 Running a Web Service

If we save this and run it in our browser (remember to start the **database server** and the **back-end application** first), we should see an error message in the browser console informing us that there has been an **`Access-Control-Allow-Origin`** error as shown in Figure D2.1, below. This means that we need to enable **Cross-Origin Resource Sharing (CORS)** on our backend application. This will allow one application (the front-end) to receive and accept data from another (the back-end).

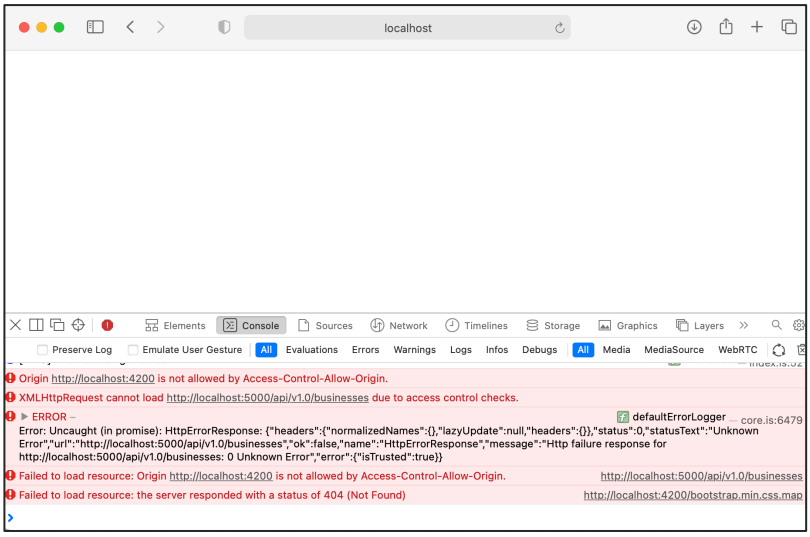

Figure D2.1 Access-Control-Allow-Origin error

This requires us to go stop the API and install the package **flask_cors** by the following command.

```
(venv) P:\Biz\B5> pip install flask_cors
```

Now we can add the lines highlighted in the code box below to *app.py*.

File: B5/app.py

```
...
from flask_cors import CORS

app = Flask(__name__)
CORS(app)
...
```

Now, if we re-launch the backend application and refresh the browser, we should see that the error message has vanished – and the data logged to the console verifies that it is being retrieved from the server.

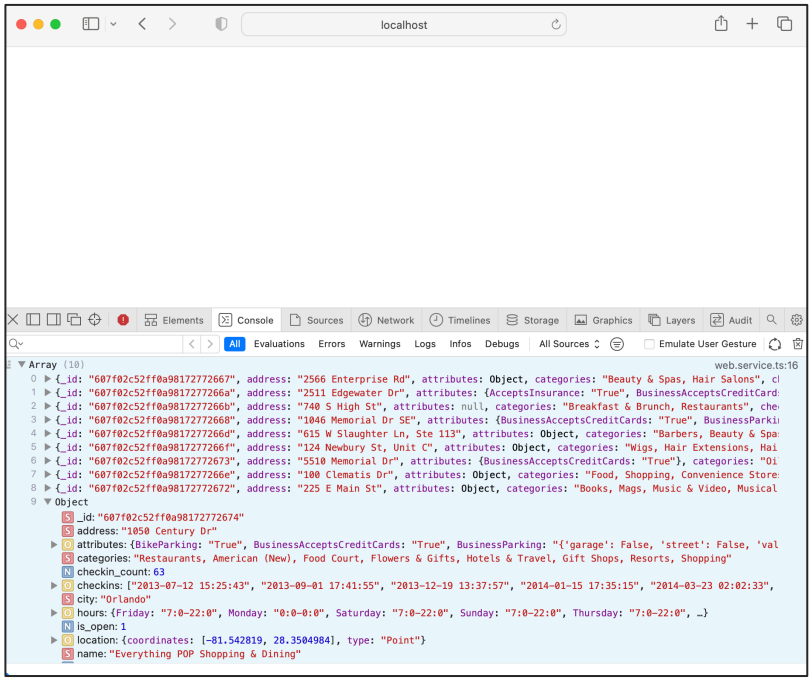

Figure D2.2 Access-Control-Allow-Origin error resolved

Do it now!	Follow the steps above to remove the Access-Control-Allow-Origin error and ensure that both the back- and front-end apps are running without error.

Note:	Depending on your installation of Bootstrap, you may also see an error *"failed to load sourcemap bootstrap.min.css.map"*. This is non-critical and can be ignored, but is easily removed by deleting the line **`/*# sourceMappingURL=bootstrap.min.css.map */`** from the file node_modules/bootstrap/dist/css/bootstrap.min.css and the line **`/*# sourceMappingURL=bootstrap.css.map */`** from the file node_modules/bootstrap/dist/css/bootstrap.css.

D2.1.4 Using the data returned

Now that the **`WebService`** is in place, we need to make a change to the **`BusinessesComponent`** HTML template to read the data from the new **`business_list`** variable created in the Service.

File: D2/src/app/businesses.component.html

```
...

<div *ngFor = "let business of webService.business_list">

...
```

As we now want to have the **`webService`** object available in the HTML template, we need to change its modifier in the constructor from **`private`** to **`public`**.

File: D2/src/app/businesses.component.ts

```
...

    constructor(public http: HttpClient) {}

...
```

Running the front-end application and visiting http://localhost:4200/ in the web browser (don't forget to run the **`mongod`** database server and the back-end application) should confirm that the connection to the back-end via the Web Service is complete.

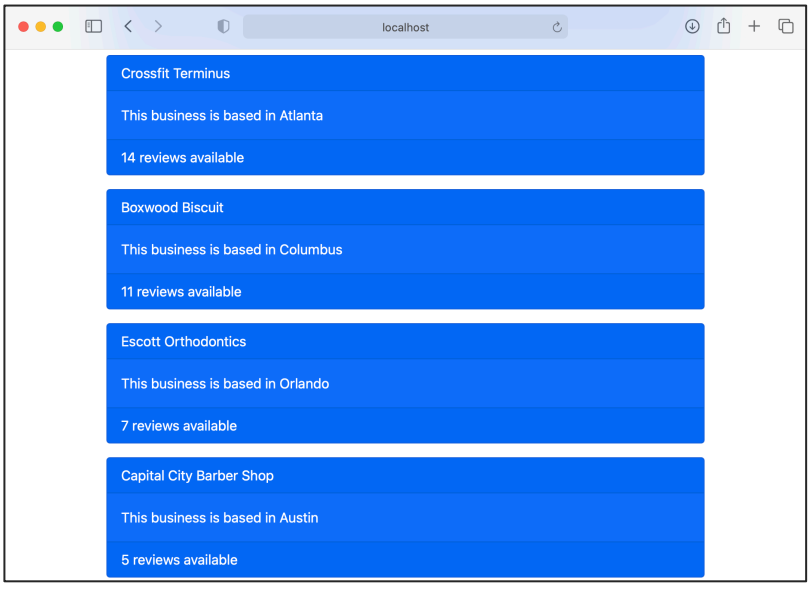

Figure D2.3 Data Retrieved and Displayed

Do it now!	Follow the steps described above to have the Web Service receive the data returned from the API and pass it to the Businesses Component for display on the web page. Ensure that you receive output such as that illustrated by Figure D2.4, above.

D2.2 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.

D2.2.1 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: D2/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: D2/src/app/home.component.html

```
<h1>Biz Directory</h1>
```

Do it now!	Create the files for the new Home Component as shown above.

D2.2.2 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: D2/src/app/app.module.html

```
...
import { WebService } from './web.service';
import { HttpClientModule } from '@angular/common/http';
import { RouterModule } from '@angular/router';

var routes: any = [];

@NgModule({
...

  imports: [
      BrowserModule, HttpClientModule,
      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.component.ts*

File: D2/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 {}
```

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

File: D2/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: D2/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 accessed 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: D2/src/app/app.module.ts

```
...

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

var routes: any = [
  {
    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 information seen earlier.

Figure D2.4 Home page
http://localhost:4200/

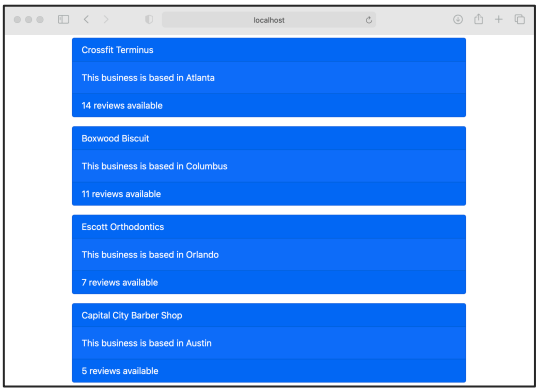

Figure D2.5 Businesses directory
http://localhost:4200/businesses

Do it now!	Follow the steps described above to add the router to the application and verify that the application now supports the two routes illustrated in Figures D2.4 and D2.5 above.

D2.4 Further Information

- https://medium.com/ramsatt/angular-7-crud-part-3-creating-web-services-using-httpclient-128f1947913c
  Creating Angular Web Services using HttpClient

- https://www.metaltoad.com/blog/angular-5-making-api-calls-httpclient-service
  Making API calls with the HttpClient service

- https://angular.io/api/core/OnInit
  `ngOnInit()` – The Angular Documentation

- https://indepth.dev/the-essential-difference-between-constructor-and-ngoninit-in-angular
  Difference between **`ngOnInit()`** and **`constructor`**

- https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS
  Cross-Origin Resources Sharing (CORS)

- https://pypi.org/project/Flask-Cors/
  Flask-Cors home page

- https://angular.io/guide/router
  Routing and Navigation in Angular

- https://www.w3schools.com/cssref/pr_class_cursor.asp
  CSS **`cursor`** properties