COM661 Full Stack Strategies and Development

FE17. Connecting to the Back-end

Aims

- To create a Web Service to handle communication with an external API
- To describe the Single-origin Policy for Web Security and implement Cross-Origin-Resource-Sharing
- To subscribe to Observables and demonstrate retrieval of the data
- To demonstrate using multiple Web Service calls to facilitate flexible update of the content
- To identify opportunities for additonal functionality
- To describe an end-to-end authentication system

Table of Contents

17.1 BUILDING A WEB SERVICE 2

17.1.1 CREATE THE WEB SERVICE 2
17.1.2 CROSS-ORIGIN RESOURCE SHARING 4
17.1.3 DISPLAYING THE DATA RETRIEVED 6

17.2 FETCHING A SINGLE BUSINESS 10

17.2.1 WEB SERVICE 10
17.2.2 BUSINESS COMPONENT 10
17.2.3 FRONT-END 12

17.3 ADDING A REVIEW 13

17.3.1 WEB SERVICE 13
17.3.2 BUSINESS COMPONENT 14
17.3.2 SEPARATING BUSINESSES AND REVIEWS 17

17.4 OTHER FUNCTIONALITY 20

17.5 FURTHER INFORMATION 22

17.1 Building A Web Service

So far, all of the functionality of the front-end Angular application has been driven by hard-coded JSON data provided by the Data Service. In this practical, we explore how to connect the front-end app with the back-end Flask API previously built.

17.1.1 Create the Web Service

Just as our Data Service provides facilities that are accessed by a number of components, we will construct the bridge between the front- and back-ends of the application within a Web Service. The Web Service will not entirely replace the Data Service as the resources accessed from external sites such as the Lorem Ipsum and Google Maps APIs will still be provided there, but the Web Service will be the sole provider of services on businesses and reviews, so that all interaction is with our back-end database rather than the hard-coded data.

As a first step, we build an initial version of the Web Service with a class property to hold the default page size and a method **`getBusinesses()`** to make a HTTP GET request to the API endpoint **GET /api/v1.0/businesses**, passing the page requested and the default page size as query string parameters.

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

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

@Injectable()
export  class WebService{

    pageSize: number = 3;

    constructor(private http: HttpClient) { }

    getBusinesses(page: number) {
        return this.http.get<any>(
            'http://localhost:5000/api/v1.0/businesses?pn=' +
             page + '&ps=' + this.pageSize);
    }
}
```

Do it now!	Add the new Web Service to your front-end application as provided in the code box above. You should be able to verify that no errors have been introduced and that the application still runs as before.

In order to use the **`getBusinesses()`** method of the new Web Service, we need to import it into the Businesses Component and add it to the component's **`providers`** list.

Note:	Remember that when importing elements into Angular components, other **components are added to the imports list**, while **services are added to the providers list**.

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

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

@Component({
  selector: 'businesses',
  standalone: true,
  imports: [RouterModule],
  providers: [DataService, WebService],
  templateUrl: './businesses.component.html',
  styleUrl: './businesses.component.css'
})

...
```

With the new Web Service introduced to the Businesses Component, we can now attempt to make use of its **`getBusinesses()`** method. First, we make the Service available to the Component class by injecting it as a parameter to the Component constructor.

Then, we make a call to the Web Service **`getBusinesses()`** method, recognising that as the **`http.get()`** request invoked by the Web Service returns an Observable that we need to subscribe to. In the subscription function, we accept the response object returned by the Observable and initially log it to the browser console.

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

```
...

constructor(public dataService: DataService,
            private webService: WebService) { }

ngOnInit() {
   if (sessionStorage['page']) {
       this.page = Number(sessionStorage['page']);
   }

   // this.business_list =
   //     this.dataService.getBusinesses(this.page);


   this.webService.getBusinesses(this.page)
       .subscribe((response) => {
            console.log(response);
    })

}

...
```

Do it now!	Update the Businesses Component as shown above and run the application. Make sure that your MongoDB Database service and back-end API are already running.

17.1.2 Cross-Origin Resource Sharing

When we run the modified Angular application (remember to start the MongoDB database server and the back-end application first), we encounter an error message in the browser console when we attempt to fetch a page of businesses. The error message indicates that there has been an Access-Control-Allow-Origin error, as illustrated in Figure 17.1, below.

This error is a result of the **same-origin policy** which is is a web browser security mechanism that aims to prevent websites from attacking each other by restricting scripts on one origin from accessing data from another origin. For API development, however, this behaviour is exactly what we want to provide, so we need to enable **Cross-Origin-Resource-Sharing** (CORS) on our API.

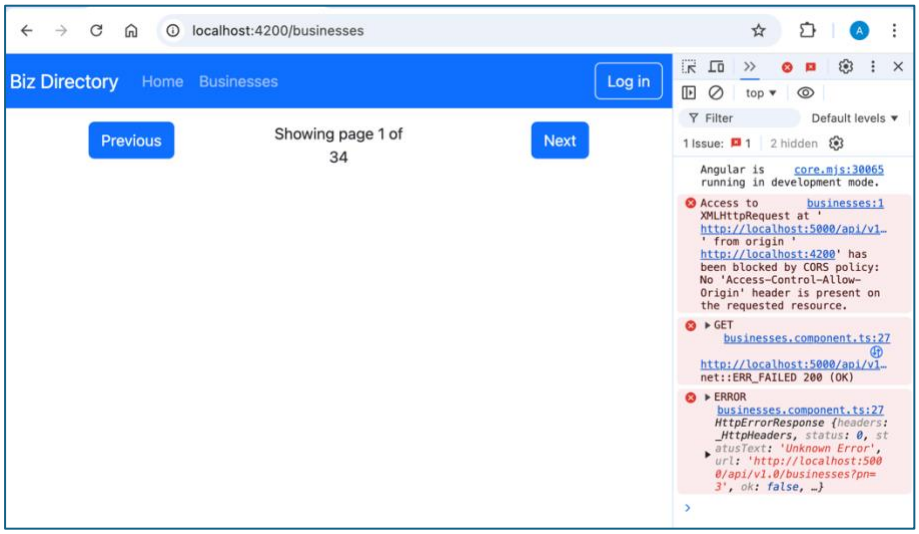

Figure 17.1 Cross Origin Resource Sharing Error

This requires us to stop our API application and install the package **flask_cors** by the command

```
(venv) C:\Desktop\Biz> pip install flask_cors
```

With the package installed, we can enable CORS by adding the code highlighted below to our API

File: biz\app.py

```
from flask import Flask
from flask_cors import CORS

...

app = Flask(__name__)
CORS(app)

...
```

Now, with CORS enabled, we can re-run the application and demonstrate that a page of businesses is being returned from the back-end and displayed in the browser console, as illustrated in Figure 17.2, below.

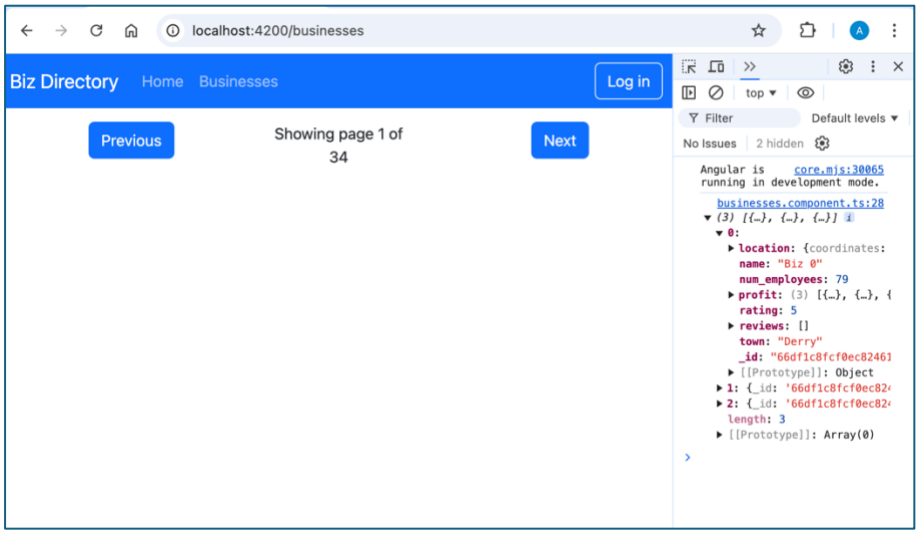

Figure 17.2 CORS Error Resolved

Do it now!	Update the Flask API application by enabling CORS as shown above. Re-run both the back-end and front-end and ensure that the data is retrieved and displayed as shown in Figure 17.2, above.

Note:	When building an API, you should enable CORS by design from the outset. In this example, we wanted to demonstrate the effect of not enabling CORS, so have introduced it deliberately at this stage.

17.1.3 Displaying the Data Retrieved

Logging the data to the console has proved that the communication with the back-end is operating as expected, so we can now display that data on the front-end instead of the data previously returned by the Data Service.

To achieve this, all we need to do is replace the **`console.log()`** of the response from the **`http.get()`** Observable with an assignment to the **`business_list`** property of the Businesses Component.

Making this change and running the application results in the first page of businesses being fetched from the MongoDB database, via the API endpoint and Web Service, and being displayed on the web page, as shown in Figure 17.3 below.

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

```
...

ngOnInit() {
   if (sessionStorage['page']) {
       this.page = Number(sessionStorage['page']);
   }

   /*
   this.business_list =
        this.dataService.getBusinesses(this.page);
   */


   this.webService.getBusinesses(this.page)
       .subscribe((response) => {
            this.business_list = response;
    })

}

...
```

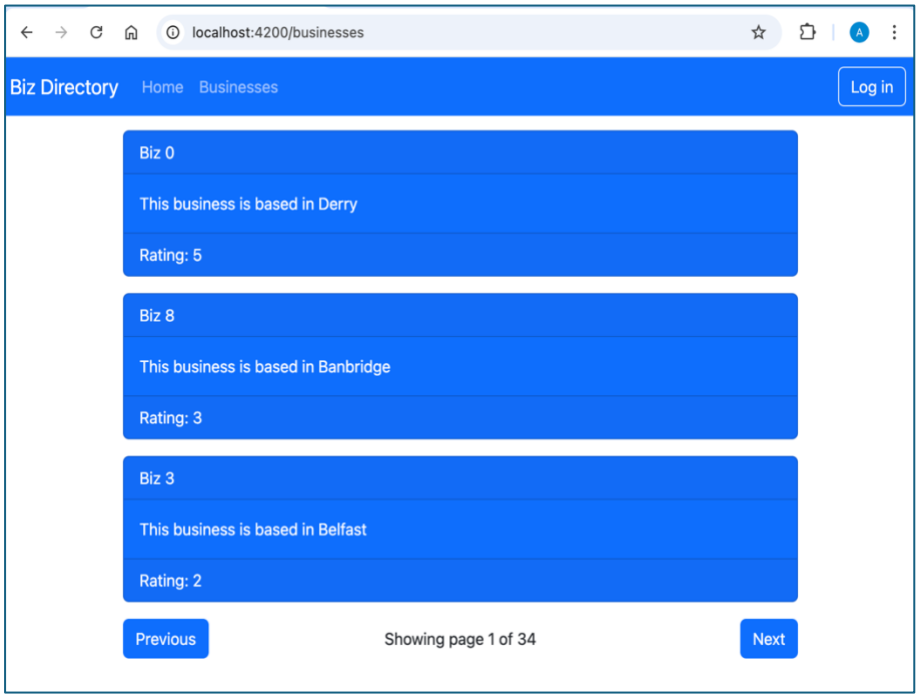

Figure 17.3 Businesses Displayed in Browser

Do it now!	Make the update to the Businesses Component **`ngOnInit()`** method as shown above and confirm that the first page of businesses are displayed in the browser.

Although we can be confident that we have fetched the data from the new Web Service rather than the previous Data Service, the display in the browser is exactly as we have seen previously. However, we can confirm that the data really is being retrieved from the back-end by making a small change to the Web Service that modifies the number of businesses requested per page.

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

```
...

export class WebService {

    pageSize: number = 4;

...
```

Saving this change and re-running the application results in a page of four businesses being displayed – leaving no doubt that the Web Service and the call to the back-end is the source of the data.

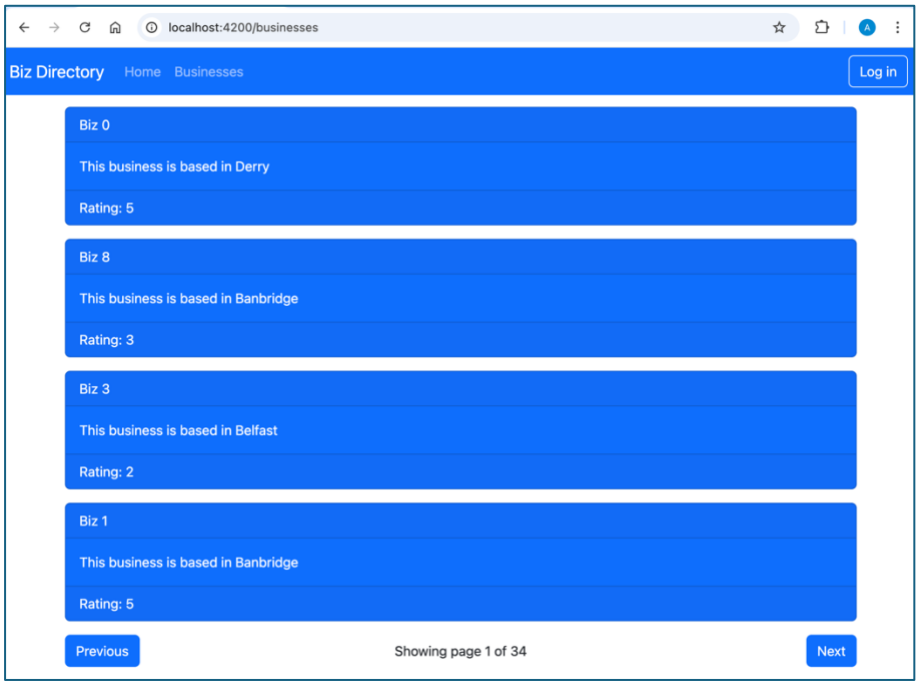

Figure 17.3 Page Size Changed

Do it now!	Change the default page size specified in the Web Service and confirm that the change has the appropriate effect on the number of business returned and displayed.

Note:	You can leave the default page size at four or change it back to three (or any other value) according to your preference.

Now that the collection of businesses is being fetched from the back-end, we need to ensure that all instances of data being fetched are implemented in the same way. Apart from the initial fetch operation in the Businesses Component **`ngOnInit()`** method, we also retrieve a fresh page of businesses following a "next page" or "previous page" operation. The final modification for the Businesses Component is to update these as shown below.

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

```
...

    previousPage() {
        if (this.page > 1) {
            this.page = this.page - 1;
            sessionStorage['page'] = this.page;
            this.webService.getBusinesses(this.page)
                .subscribe((response: any) => {
                   this.business_list = response;
                })
        }
    }

    nextPage() {
        if (this.page < this.dataService.getLastPageNumber()) {
            this.page = this.page + 1;
            sessionStorage['page'] = this.page;
            this.webService.getBusinesses(this.page)
            .subscribe((response: any) => {
               this.business_list = response;
            })
        }
    }

...
```

Do it now!	Change the **`previousPage()`** and **`nextPage()`** functions as shown above and re-run the application. Confirm that you can navigate forward and backwards through the collection of businesses.

17.2 Fetching a Single Business

With the Web Service established and demonstrated to correctly retrieve a page of businesses from the API, we can now add the functionality to retrieve a selected business by ID. This is tackled in three phases – adding new functionality to the Web Service, modifying the Business Component to use the Web Service rather than the Data Service, and updating the front-end HTML template.

17.2.1 Web Service

Providing the Web Service with the means of retrieving a single business by ID is straightforward, and simply involves the specification of a new method that accepts the business id as a parameter and uses it in an **`http.get()`** request to the API endpoint that returns a specified business.

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

```
    ...

        getBusiness(id: any) {
            return this.http.get<any>(
                'http://localhost:5000/api/v1.0/businesses/' + id);
        }
    }
```

Do it now!	Add the **`getBusiness()`** method to the Web Service as shown above.

17.2.2 Business Component

The first update to the Business Component is to import the Web Service and make it available to the Component class. This is carried out in the now familiar steps as highlighted in the code box below.

File: bizFE/src/app/business.component.ts

```
...

import { WebService } from './web.service';

@Component({
  selector: 'business',
  standalone: true,
  imports: [CommonModule, GoogleMapsModule,
            ReactiveFormsModule],
  providers: [DataService, WebService],
  templateUrl: './business.component.html',
  styleUrl: './business.component.css'
})

...

constructor( public dataService: DataService,
             private route: ActivatedRoute,
             private formBuilder: FormBuilder,
             public authService: AuthService,
             private webService: WebService) {}

...
```

Now, we can replace the existing call to the Data Service **`getBusiness()`** method with one to the Web Service equivalent. However, we need to be careful to recognize the fundamental difference between the methods. As the Data Service **`getBusiness()`** method returned the JSON object representing the business, we could assign the value returned from **`getBusiness()`** directly to the **`business_list`** component property. However, since the **`http.get()`** request returns an Observable, we need to subscribe to it in order to retrieve the value.

This has a significant consequence for the remainder of the code in **`ngOnInit()`**. As the requests to the **OpenWeatherMap** and **Google Maps** APIs depend on coordinate values that are retrieved from the business object returned, these API requests cannot be allowed to proceed until after the call to the Biz Directory API has returned. To ensure that this is the case, the code that uses external API requests to generate additional presentation information is moved inside the subscription function of the **`getBusiness()`** API call.

This is demonstrated in the following code box.

File: bizFE/src/app/business.component.ts

```
...

this.webService.getBusiness(
                         this.route.snapshot.paramMap.get('id'))
     .subscribe( (response: any)  => {
          this.business_list = [response];

          this.business_lat = ...
          this.business_lng = ...

          ...
          // remaining ngOnInit() code inside the subscription
          // function

     }
}
```

Do it now!	Update the ***business.component.ts*** file by importing the Web Service and modifying **`ngOnIOnit()`** as shown in the code boxes above.

17.2.3 Front-end

The only change required to the front end is in the Businesses Component rather than the Business Component. As the data was previously fetched from the Data Service, the **id** field that identified each business and which was specified in the **`routerlink`** property of the Bootstrap card was the **`business._id.$oid`** field (as this is how it is presented when the dataset is exported using **mongoexport**).

Now that the data is being retrieved directly from the database, the field is simply **`_id`**, so we can make the corresponding change in the Business Component HTML template.

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

```
...

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

...
```

Now, when we run the application and select a business, its data is fetched from MongoDB via the API and Web Service and is displayed in the browser as illustrated by Figure 17.5, below.

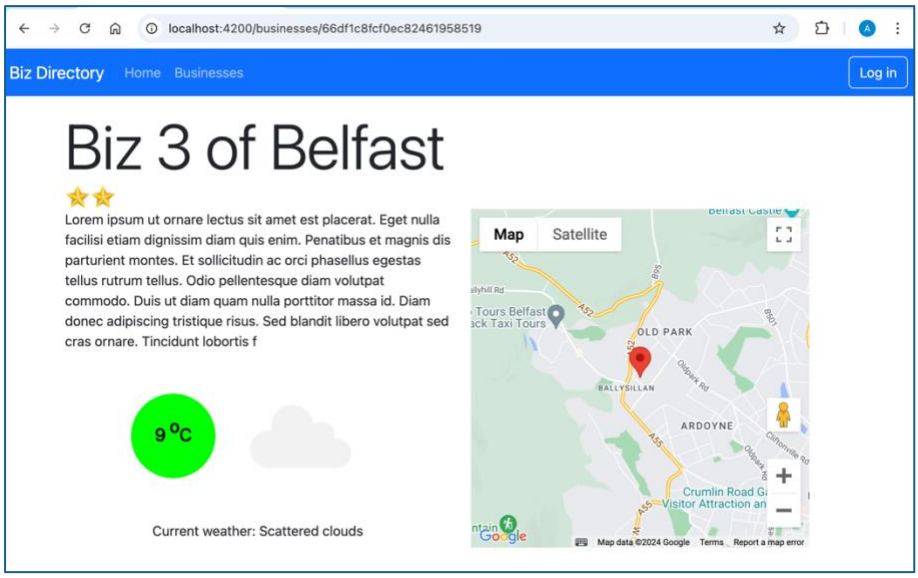

Figure 17.5 Business Retrieved

Do it now!	Update the ***businesses.component.html*** file as shown in the code box above and verify that you can navigate the collection of businesses and choose a business for display in detailed form as shown in Figure 17.5, above

17.3 Adding a Review

The next element of Data Service functionality that needs to be replaced with a Web Service equivalent is the ability to add a user review to the collection for the business being displayed. At the minute, the review is added to the collection maintained by the Data Service, but as this data is refreshed from the file in the assets folder every time the application is run, any review added are lost. By switching to the Web Service, the reviews will be added to the MongoDB database and will persist over the life of the application.

17.3.1 Web Service

The first step is to add an new method to the Web Service that accepts a business ID and a review data structure as parameters, and makes an **`http.post()`** request to the API endpoint that adds a review to the collection.

The TypeScript **`FormData`** object is a representation of a form that is structurally equivalent to an HTML form sent using the **`multipart/form-data`** encoding. This means that we can send it as the second parameter of a **`http.post()`** request and it will be received by the API just as if it had originated from an HTML **`<form>`** (or from a Postman request).

The Web Service **`postReview()`** method adds the user provided data to the **`FormData`** object by taking each field in turn and appending it with a field name. Hence, in the code box below, the value of the field "*username*" from the review form is added to the **`FormData`** object, again using "*username*" as the field name.

The full specification of **`postReview()`** is provided in the code box below.

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

```
...

postReview(id: any, review: any) {
   let postData = new FormData();
   postData.append("username", review.username);
   postData.append("comment", review.comment);
   postData.append("stars", review.stars);
   return this.http.post<any>(
          'http://localhost:5000/api/v1.0/businesses/' +
           id + "/reviews", postData);
}
```

Do it now!	Add the **`postReview()`** method to the Web Service as shown in the code box above.

17.3.2 Business Component

With the Web Service functionality in place, we can now move to the Business Component and invoke the new method, passing it the review form object and the _id of the business for which the review is intended. Again, as the **`http.post()`** request returns an Observable, we need to subscribe to it, moving the statement to reset the form inside the subscription function.

File: bizFE/src/app/business.component.ts

```
...

onSubmit() {
      this.webService.postReview(
                this.route.snapshot.paramMap.get('id'),
                this.reviewForm.value)
          .subscribe( (response) => {
             this.reviewForm.reset();
          });
}

...
```

When we save the file and run the application, sending a review generates a **401 Unauthorised Request** error which can be seen in the Browser Console as shown in Figure 17.6, below.

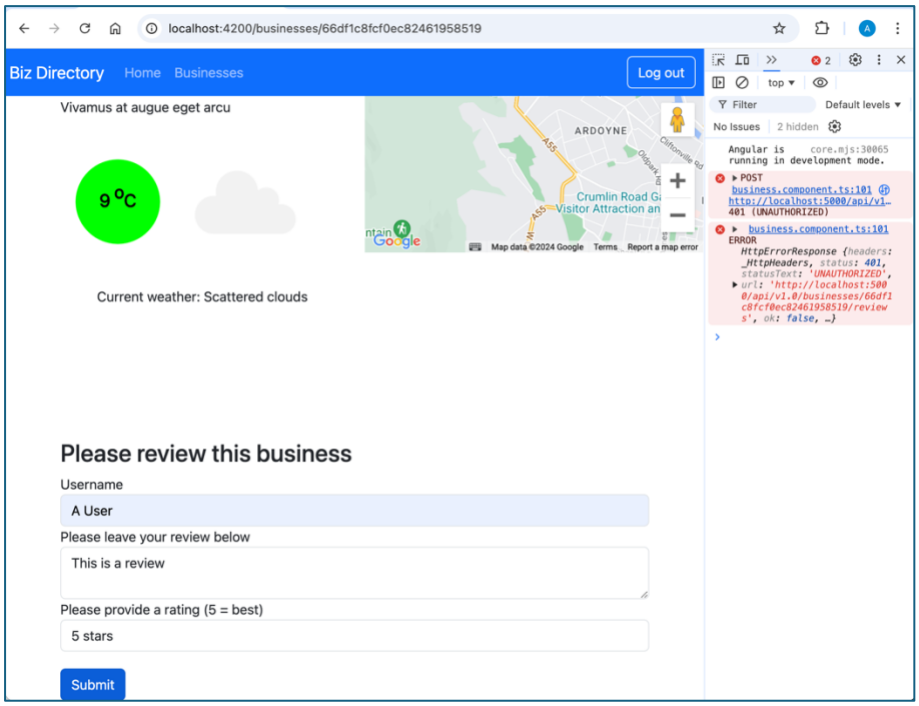

Figure 17.6 Unauthorised Access Error

Do it now!	Modify the `onSubmit()` method in the Business Component as shown in the code box above. Run the application with the Browser Console open and verify that you receive the 401 error code as shown in Figure 17.6, above.

The error is occurring because we specified in the API that the POST review endpoint should only be available to requests that enclosed a valid JWT token. We will discuss how to achieve this later, but for now we will progress by removing the requirement for a token from the API endpoint.

File: biz/bueprints/reviews/reviews.py

```
...

@reviews_bp.route("/api/v1.0/businesses/<string:id>/reviews",
                  methods=['POST'])
# @jwt_required - COMMENTED OUT (for now)
def add_new_review(id):
    new_review = {

...
```

Now when we re-run the application and submit a review, we can navigate back to the business and confirm that the review has been added, as seen in Figure 17.7, below.

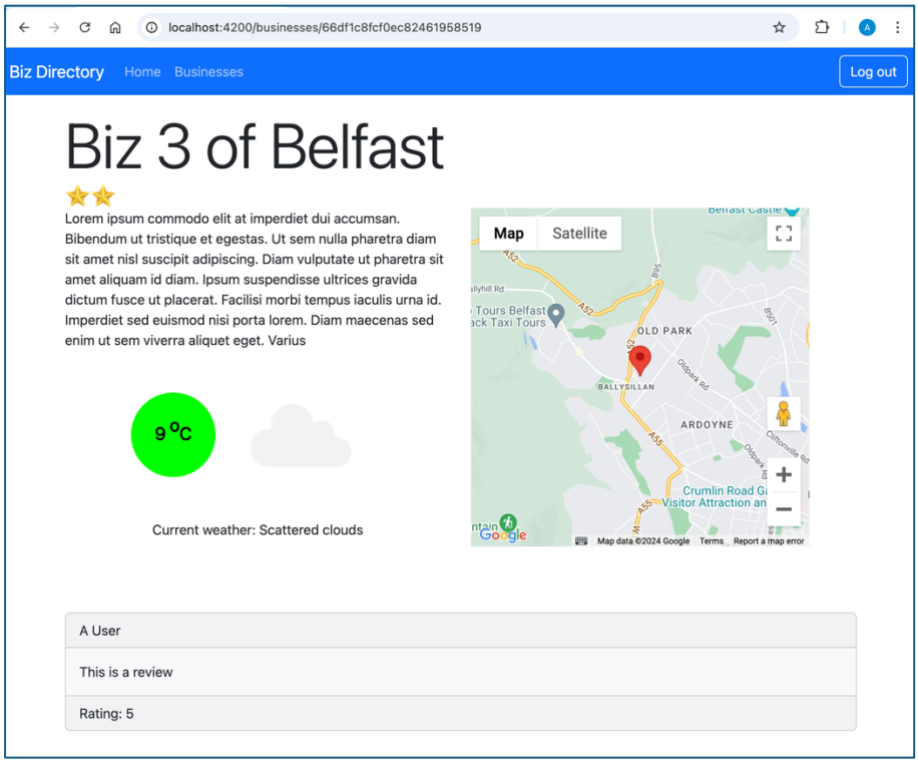

Figure 17.7 Review Added

Do it now!	Modify the API as shown above and verify that the review is added by refreshing the page after it has been submitted.

17.3.2 Separating Businesses and Reviews

A limitation of the **`postReview()`** implementation is that we need to navigate away from the business and return, or refresh the page in order to see the updated list of reviews. We could simply fetch the business again and update the browser that way, but fetching a business also involves fetching the description, weather information and map from 3rd party APIs so would be a very inefficient solution.

A better approach is to separate fetching a business from fetching its reviews, so that we can request that the reviews only are refreshed when a new review is added.

The first step is to add a Web Service method to retrieve the reviews of a specified business from the API.

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

```
...

    getReviews(id: any) {
        return this.http.get<any>('
                http://localhost:5000/api/v1.0/businesses/' +
                id + '/reviews');
    }}

...
```

Do it now!	Add the new **`getReviews()`** method to the Web Service as shown above.

Then, we update the Business Component to add a new property to hold the reviews list and to make a call to the new Web Service method that fetches the reviews immediately after the call to the method that fetches the business. Once again, since the **`http.get()`** request returns an Observable, we subscribe to it and assign the response returned from the Observable to the **`reviews_list`** property of the business.

Then, in the **`onSubmit()`** function, in the subscription function for the call to **`postReview()`**, we add a call to **`getReviews()`** to fetch the updated reviews list for the business. In the subscription function for **`getReviews()`** we retrieve the response from the Observable and update the **`review_list`** with this new value. This has the effect of updating the **`review_list`** after each new review is added, without requiring that the rest of the page is refreshed. Since the reviews displayed in the Business Component HTML

template are bound to the **`review_list`** variable, the list of reviews displayed is automatically updated every time a new review is posted.

File: bizFE/src/app/business.component.ts

```
...

export class BusinessComponent {

    business_list: any;
    ...
    review_list: any;

...

   this.webService.getBusiness(
                   this.route.snapshot.paramMap.get('id'))
       .subscribe( (response: any)  => {
            this.business_list = [response];

            ...
   }

   this.webService.getReviews(
                   this.route.snapshot.paramMap.get('id'))
       .subscribe( (response) => {
            this.review_list = response;
   });
}

onSubmit() {
     this.webService.postReview(
          this.route.snapshot.paramMap.get('id'),
          this.reviewForm.value)
        .subscribe( (response) => {
            this.reviewForm.reset();

            this.webService.getReviews(
            this.route.snapshot.paramMap.get('id'))
                 .subscribe( (response) => {
                      this.review_list = response;
            });
     });
}
```

Finally, we bind the reviews displayed in the browser to the content of the new **`review_list`** property (rather than the **`reviews`** element of the current business) as shown in the code box below.

File: bizFE/src/app/business.component.html

```
...

 <div class="row" style="margin-top: 70px">
      <div class="col-sm-12">

           @for ( review of review_list; track review ) {

...
```

Now when we run the application and visit a business, we can leave a review and see that the page is automatically updated without any other page elements requiring a refresh

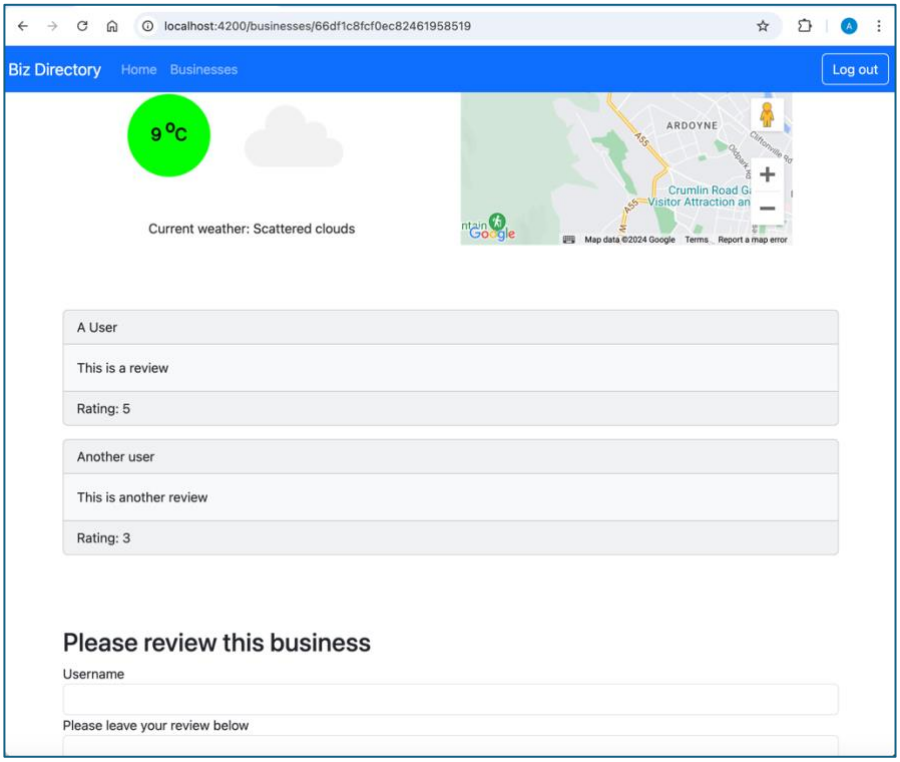

Figure 17.7 Review Added and Displayed Instantly

Do it now!	Make the modifications to the Business Component TypeScript and HTML files as shown above and verify that submitted reviews are automatically added to the displayed collection.

17.4 Other Functionality

As we build the Web Service and migrate the businesses and reviews functionality from the Data Service, we identify additional opportunities for development that require that either the back-end adapts to the requirements of the front-end, or the front-end adapts to the provision of the back-end.

We will identify one example of each here, but the actual implementation is left for you as an exercise and an opportunity to contribute original functionality to your own applications.

17.4.1 Providing Navigation Information

In our page that provides navigation of the collection of businesses, we have a message "Showing page x of y" which gives the user awareness of the scale of the scale of the collection and their current position within it.

Figure 17.7 Navigation Information Message

This message is facilitated by the Data Service method **`getLastPageNumber()`** which calculates and returns the number of the last page of information according to the size of the collection and the number of businesses requested per page. The same method also provides the value used in the Businesses Component method **`nextPage()`** which prevents the user from navigating past the last page of content.

This functionality is not possible with the current version of the Biz Directory API as there are no endpoints that return information about the collection. This could be provided in a variety of ways as follow:

i) Provide a new endpoint such as **GET /api/v1.0/businessesInfo** that returns a packet of information about the businesses collection. This packet could contain the size of the collection as well as other data that might be useful such as the different types of businesses represented, the different locations, or other meta-data that could be useful in building the front-end of the application.

ii) Restructure the data returned from the **GET /api/v1.0/businesses** endpoint so that summary information is returned alongside the businesses data. For example, the data returned might be in the form

```
{
  'thisPage' : 5,
  'totalPages' : 70,
  'businesses' : [ { business1 }, { business2 }, etc. ]
}
```

Note:	In general, we want to avoid making changes to the back-end to support front-end functionality as we might compromise other applications that use the API in its existing form. We should ensure that any changes we make are non-breaking for other users, otherwise they should be incorporated into a new version of the API that exists alongside the old one. For example, the endpoint **GET /api/v1.0/businesses** might return data in the current form, while the endpoint **GET /api/v2.0/businesses** could return the expanded data packet suggested above.

17.4.2 Passing a JWT Token

When we implemented our **`postReview()`** Web Service method, we had to remove the @jwt_required decorator from the API application so that the review could be accepted. If we want to make use of back-end JWT tokens at the front end, we need to make Web Service calls to our back-end authentication methods. The following technique could be implemented alongside the Auth0 authentication or (more likely) it could replace Auth0 authentication completely so that we provide our own "end-to-end" authentication service.

i) Provide a front-end interface (for example a button on the NavBar) that makes a call (via the Web Service) to your back-end **login** endpoint.

ii) Subscribe to the Observable returned by the login endpoint and store the JWT token received in browser storage.

iii) When making a request to an endpoint that requires a JWT token, retrieve the token from browser storage in the corresponding Web Service method and add it to the headers of the http request. (You can see the technique for adding request headers by reviewing the request for Lorem Ipsum content in Practical FE14).

17.5 Further Information

- https://angular.dev/guide/http
  Angular HttpClient

- https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Access-Control-Allow-Origin
  Access-Control-Allow-Origin HTTP Response Header

- https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy
  Same-origin Policy

- https://web.dev/articles/same-origin-policy
  Same-origin Policy – WebDev Article

- https://flask-cors.readthedocs.io/en/stable/api.html
  The Flask-CORS Package

- https://www.techiediaries.com/angular-formdata/
  Working with FormData in Angular 18

- https://www.telerik.com/blogs/angular-basics-introduction-observables-rxjs-part-2
  Angular Basics – Introduction to Observables

- https://aidankmcbride.medium.com/observables-and-subscriptions-in-angular-c0c703bb910d
  Observables and Subscriptions in Angular

- https://tekloon.medium.com/how-to-add-http-header-to-angular-httpclient-d38d4ccd0eb8
  How to Add Http Header to Angular HttpClient