COM661 Full Stack Strategies and Development

Practical D3: Asynchronous Pipes and Information Management

Aims

- To use the `async` pipe to subscribe directly to an asynchronous data source
- To introduce another controller and route
- To extend the existing Web Service
- To build a basic pagination interface
- To implement browser storage using the `localStorage` and `sessionStorage` objects

Table of Contents

D3.1 Using the `async` Pipe

Our current implementation of the front-end Angular application uses a "fat arrow" function to subscribe to the Observable returned by the **HTTP GET** request to the **Get Businesses** endpoint. The data obtained from the subscription is copied to a Web Service variable **`business_list`**, from where it can be accessed by the **Businesses Component** HTML template and displayed in the browser window.

We can simplify the implementation further by making use of the Angular **`async`** pipe component that subscribes to an Observable and returns the latest value that has been generated. This enables us to omit the explicit subscription code that was added to the Web Service and return it to the simpler form shown below.

File: D3/src/app/web.service.ts

```
...

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

...
```

Next, we update the Business Component TypeScript file so that the output from the Web Service function is assigned to the local variable **`business_list`**. Note that the **`business_list`** variable in the Businesses Component may already be available as we used it in the original version that operated on hard-coded data.

File: D3/src/app/ businesses.component.ts

```
...

  business_list: any = [];

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

...
```

Finally, we subscribe to the Observable returned from the Web Service by connecting the **async** pipe to the **business_list** variable in the ***ngFor** directive. In this way, every time the Web Service returns new data to the **business_list** variable, the component template is automatically updated – without any additional code to manage the subscription.

File: D3/src/app/ businesses.component.html

```
...

  <div *ngFor = "let business of business_list | async">

...
```

Do it now!	Follow the steps illustrated above to introduce the **async** pipe to the Businesses Component template. Verify that the list of businesses is displayed on the page as before.

D3.2 An Additional Route and Web Service Endpoint

Now that the functionality to return a collection of business information is complete, we will now consider the case where we provide a way of displaying only one of the collection of businesses.

D3.2.1 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: D3/src/app/business.component.ts

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

export class BusinessComponent {

    business_list: any;

    constructor(private webService: WebService) {}

    ngOnInit() {
       this.business_list = this.webService.getBusiness(id);
    }
}
```

The file ***business.component.html*** will be exactly as for the Businesses Component equivalent. Note that we maintain the **`*ngFor`** directive, even though we expect details of only a single business to be produced. This is required because the subscription to the Observable returns an iterable – i.e. an object that we can loop over – so the **`*ngFor`** directive provides this iterative structure.

The ***business.component.css*** file can remain blank at this stage, though we may choose to apply CSS to the component later.

File: D3/src/app/business.component.html

```
...
<div class="container">
    <div class="row">
        <div class="col-sm-12">
            <div *ngFor =
                 "let business of business_list | async">
                <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 -->
```

Do it now!	Follow the steps described above to create the new Business Component and populate it with the code shown.

D3.2.2 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:5000/api/v1.0/businesses/12345, where 12345 is the **`_id`** value of one of the business objects.

File: D3/src/app/web.service.ts

```
...

export class WebService {

    constructor(private http: Http) {}

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

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

Do it now!	Update the Web Service with the new method to fetch details of a single business.

D3.2.3 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.ts*** and adding it to the **`declarations`** list. We also provide a new entry into the **`routes`** array in ***app.module.ts*** which specifies the variable **`:id`** parameter that will act as a marker for a business **`_id`** value.

File: D3/src/app/app.module.ts

```
...

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

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

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

...
```

Do it now!	Update ***app.module.ts*** with the new path and component as shown in the code above.

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 specifies 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.

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: D3/src/app/businesses.component.html

```
...

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

...
```

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: D3/src/app/business.component.ts

```
...

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

...

export class BusinessComponent {

    business_list: any = [];

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

    ngOnInit() {
       this.business_list = this.webService.getBusiness(
                            this.route.snapshot.params['id']);
    }
}
```

Do it now!	Update the Business Component HTML and TypeScript files with the new code as shown above.

Finally, we need to make a minor change to the back-end application so that the value returned to the Observable is an iterable. This is easily accomplished to have the endpoint return a list containing a single business rather than the individual object.

File: B7/app.py

```
...

def show_one_business(id):

   ...

   return make_response( jsonify( [ business ] ), 200 )
```

Note how the simple addition of the list brackets **[ ]** to the output converts the data to an iterable and enables us to use the existing **`*ngFor ... async`** structure in the Business Component HTML template..

Now, 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.

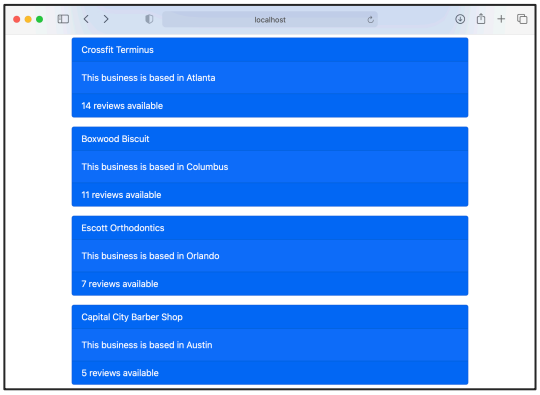

Figure D3.1 A clickable list of businesses
http://localhost:4200/businesses

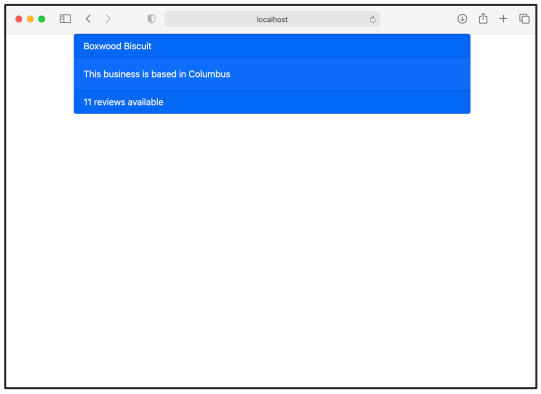

Figure D3.2 Clicking on a single business
http://localhost:4200/businesses/12345

Do it now!	Update the back-end application as shown above and re-run it. Launch the front-end application and ensure that you obtain output such as that shown in Figures D3.1 and D3.2, above.

D3.3 Basic Pagination

Pagination is a common requirement of catalogue-type applications where the collection of data is too large to be displayed in a single view. There are many Angular pagination components that you can download and use (this is left for you as an exercise), but it is useful to work through the development of simple "Next" and "Previous" buttons as a number of important Angular concepts are raised.

D3.3.1 Specifying the slice to be taken

Our back-end application returns details on the first 10 businesses in the collection by default. We also provided optional querystring parameters **`pn`** (page number) and **`ps`** (page size) that allow us to change the number of businesses returned on a page and the page number from which we begin to read data. For example, a call to http://localhost:5000/api/v1.0/businesses?pn=1&ps=5 would return the first 5 businesses, while http://localhost:5000/api/v1.0/businesses?pn=3&ps=10 would return 10 businesses after skipping the first 20.

In this example, we will keep the default of 10 businesses per page but provide "Next" and "Previous" buttons to add or subtract 1 from the value of **pn** (the page number) each time.

First, we introduce a local variable **page** to the **BusinessesComponent** class and add it as a parameter to the call to the **WebService** function **getBusinesses()**.

File: D3/src/app/businesses.component.ts

```
...
export class BusinessesComponent {
  constructor(public webService: WebService) {}

  ngOnInit() {
    this.business_list =
                      this.webService.getBusinesses(this.page);
  }

  business_list: any = [];
  page: number = 1;

}
```

Next, we modify the **WebService** function to accept the **page** parameter and append it to the URL in the call to the API.

File: D3/src/app/web.service.ts

```
...

getBbusinesses(page: number) {
   return this.http.get('
       http://localhost:3000/api/businesses?pn=' + page)

...
```

Running the application and checking the web browser should confirm that the parameter is accepted and that the 1st set of 10 businesses is returned and displayed. If you then change the initialization of **page** in ***businesses.component.ts*** (e.g. to 2) and save the code you should be able to confirm the effect of the variable by receiving details of a different set of pages.

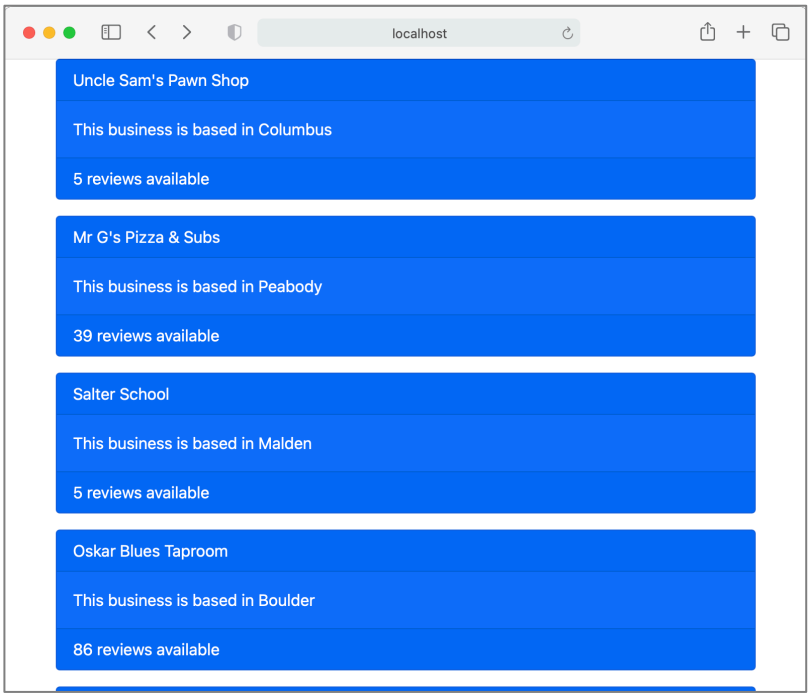

Figure D3.3 The second set of businesses

D3.3.2 Modify the front end

The next step is to add a pair of buttons to the **`BusinessesComponent`** HTML template to trigger the previous and next pages of data to be displayed. Note the **`(click)`** notation that assigns the event handler to the button. This is the Angular equivalent to the familiar JavaScript **`onClick`** keyword.

File: D3/src/app/businesses.component.html

```
...

  <div class="row">
     <div class="col-sm-6">
        <button class="btn btn-primary"
                (click)="previousPage()">Previous</button>
     </div>
     <div class="col-sm-6 text-end">
        <button class="btn btn-primary"
                (click)="nextPage()">Next</button>
     </div>
  </div>

</div>      <!-- container -->
```

Since we have specified calls to functions **`previousPage()`** and **`nextPage()`**, we need to provide(initially) empty definitions for these in the component TypeScript file.

File: D3/src/app/businesses.component.ts

```
...

export class BusinessesComponent {

  ...

  previousPage() {
  }

  nextPage() {
  }

  ...

}
```

The application should now run and we can verify that the buttons are added and are aligned to the left- and right-hand sides of the display as shown in Figure D3.4 below.

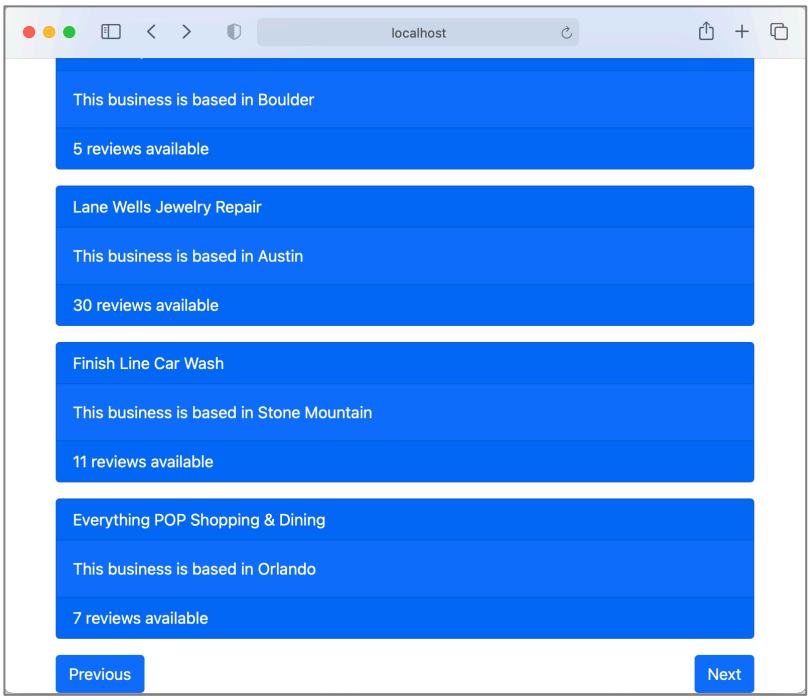

Figure D3.4 Adding Pagination Buttons

Do it now!	Follow the steps above to add “Previous” and “Next” buttons to the end of the Businesses Component HTML template and to pass the **page** value from the Component to the Web Service.

D3.3.3 Adding Component Functionality

Now, we add the **nextPage()** and **previousPage()** methods to move forward or backward through the collection of business data. In each case, all that is required is to change the value of **page** accordingly (by either adding or subtracting 1) and then to call the **WebService** function that retrieves the data. As the HTML template subscribes to an Observable that will update each time the API returns new data, the display will automatically change each time a new page of data is fetched.

Note:	We are able to prevent the user from scrolling back beyond the first page by checking that the value of start is never allowed to be less than zero. However, we do not have a similar check for the last page. If our API had provided an endpoint that returned the number of pages in the collection (or had returned the value as part of the response to a **GET /businesses** call), we could use this value in the test – but this is left for you as an exercise.

File: D3/src/app/businesses.component.ts

```
...
ngOnInit() {
 this.business_list = this.webService.getBusinesses(this.page);
}

previousPage() {
  if (this.page > 1) {
    this.page = this.page - 1;
    this.business_list =
                      this.webService.getBusinesses(this.page);
  }
}

nextPage() {
  this.page = this.page + 1;
  this.business_list =
                      this.webService.getBusinesses(this.page);
  }

business_list: any = [];
page: number = 1

}
```

When we try the pagination in the browser it appears to work as expected. We are able to move forward and backward through the data one page at a time.

However, if we refresh the browser or click on an entry and load the page displaying details of that business and then click the browser 'Back' button, we are returned to the first page of information – not the page that we most recently visited. This is because the **`BusinessesComponent`** is re-initialised every time it is loaded, resetting the value of **`page`** to 1. The solution to this is to store the value of start in the browser's **`sessionStorage`** object each time it is updated and to retrieve the most recent value when the Component is initialized.

HTML5 provides **`localStorage`** and **`sessionStorage`** as repositories for data on the client. The information is never passed to the server and can be used when we want to programmatically maintain the state of an application. Data stored in **`localStorage`** has no expiry date – it is not deleted when the browser is closed and will be available at any point in the future, while data in **`sessionStorage`** is removed when the browser tab is closed.

We can improve the usability of our application by using **`sessionStorage`** to store the page currently being displayed. Every time that the **`nextPage()`** or **`previousPage()`** methods are invoked, the new version of page is copied to **`sessionStorage`**. Then, when the component is loaded and **`ngOnInit()`** runs, we can check for the presence of a **`page`** value in **`sessionStorage`** and restore the value of **`page`** (first making sure that it is interpreted as a numeric value and not a string) if a previous value exists.

File: D3/src/app/businesses.component.ts

```
...

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

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

nextPage() {
  this.page = this.page + 1;
  sessionStorage['page'] = this.page;
  this.business_list =
                    this.webService.getBusinesses(this.page);
}

...
```

The values stored in **`sessionStorage`** can be examined by opening the Browser Console, selecting the **Storage** tab and clicking on the **Session Storage** option, as seen in Figure D3.5.

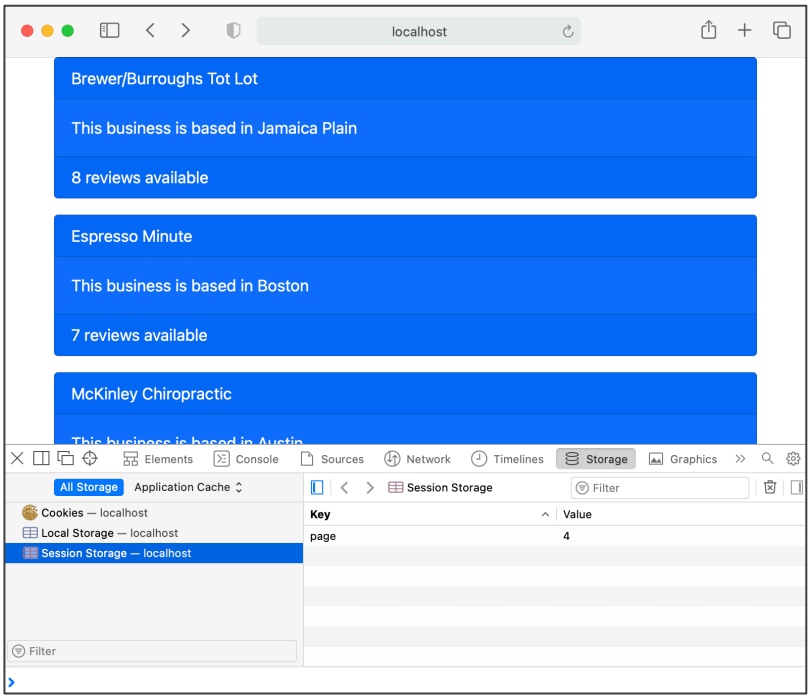

Figure D3.5 Storing the page number in Session Storage

Do it now!	Add the **`nextPage()`** and **`previousPage()`** methods to the Businesses Component and ensure that you can page through the collection of businesses data. Use the Browser Console to verify that the value of **`page`** is being maintained.

Note:	There are many Angular Components available online that implement full pagination, filtering and other display aids. It is left as an exercise for you to research these with a view to including more extensive navigation in your own submissions.

D3.4 Further Information

- https://medium.com/@luukgruijs/understanding-creating-and-subscribing-to-observables-in-angular-426dbf0b04a3
  Understanding, Creating and Subscribing to Observables in Angular

- https://malcoded.com/posts/angular-async-pipe/
  Async Pipe: How to use it properly in Angular

- https://angular.io/guide/dependency-injection
  Dependency Injection in Angular – using the Constructor

- https://www.w3schools.com/bootstrap/bootstrap_buttons.asp
  Bootstrap Buttons

- https://www.tutorialrepublic.com/html-tutorial/html5-web-storage.php
  HTML5 Web Storage – Using LocalStorage and SessionStorage