COM644 Full-Stack Web and App Development

Practical C3: Using Observables

Aims

- To introduce Observables as a tool for asynchronous development
- To convert a component from using Promises to Observables
- To appreciate potential security issues with public variables
- To introduce RxJs Subjects for transmission of information between components
- To use the `async` pipe to subscribe directly to an asynchronous data source
- To build a basic pagination interface
- To implement browser storage using the `localStorage` and `sessionStorage` objects

Contents

C3.1 Observables

In the previous practical, we saw how the call to the back-end API operates in a non-blocking manner, with programme code continuing to execute after the call is made. In order to assign the data returned from the API to a variable, we had to append the **`toPromise()`** method to the call, which in turn required us to use the JavaScript **`await`** and **`async`** keywords to identify the function and call as being asynchronous.

In this practical we will examine a better technique for handling non-blocking activity that allows is to subscribe to asynchronous events and automatically react to them each time they occur.

C3.1.1 Preparation

The current version of our Web Service contains a pair of functions to retrieve information on a collection of businesses and to retrieve information on a single business. We will eventually convert both of these to use Observables rather than Promises, but initially we will deal only with the retrieval of a collection of businesses.

As a first step, then, we will comment out the definition of the **`getBusiness()`** function in the **`WebService`** as well as the code in the **`BusinessComponent`** that calls the function. This will enable us to work on the **`getBusinesses()`** operation in isolation.

File*: C3/src/app/web.service.ts*

```
...

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

...
```

File*: C3/src/app/business.component.ts*

```
...

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

...
```

C3.1.2 From Promises to Observables

Now that only a single **`WebService`** function remains, we will convert it to use an Observable rather than a Promise.

Actually, the **`http.get()`** method returns an Observable by default, so all we need to do is remove the **`toPromise()`** call that was appended to it.

File*: C3/src/app/web.service.ts*

```
...

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

...
```

Now, in the **`BusinessesComponent`** Typescript file, we remove the **`await`** and **`async`** keywords that were required to use the previous Promise.

Now, since we are no longer waiting from the result of a promise, we no longer assign the result to a response variable – so we remove that code as well to leave the **`ngOnInit()`** function as shown in the code box below.

File*: C3/src/app/businesses.component.ts*

```
...

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

One of the features of this development stage is to simplify the Component and move the complexity into the **`WebService`** (as potentially many Components can make use of the same Service), so we modify the **`http.get()`** call in the **`WebService`** to subscribe to the Observable.

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 JSON data returned from the API and copies it to the new local variable **`business_list`**.

File*: C3/src/app/web.service.ts*

```
...

    business_list = [];

    constructor(private http: Http) {}

    getBusinesses() {
        return this.http.get(
                    'http://localhost:3000/api/businesses')
            .subscribe(response => {
                this.business_list = response.json();
            })
    }

...
```

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 parameter 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.

Now that the **`WebService`** is modified, 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*: C3/src/app/businesses.component.html*

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

Running the front-end application and visiting http://localhost:4200/businesses in the web browser (don't forget to run the **`mongod`** database server and the B6 back-end application) should confirm that the conversion from Promise to Observable is complete and the application works as before.

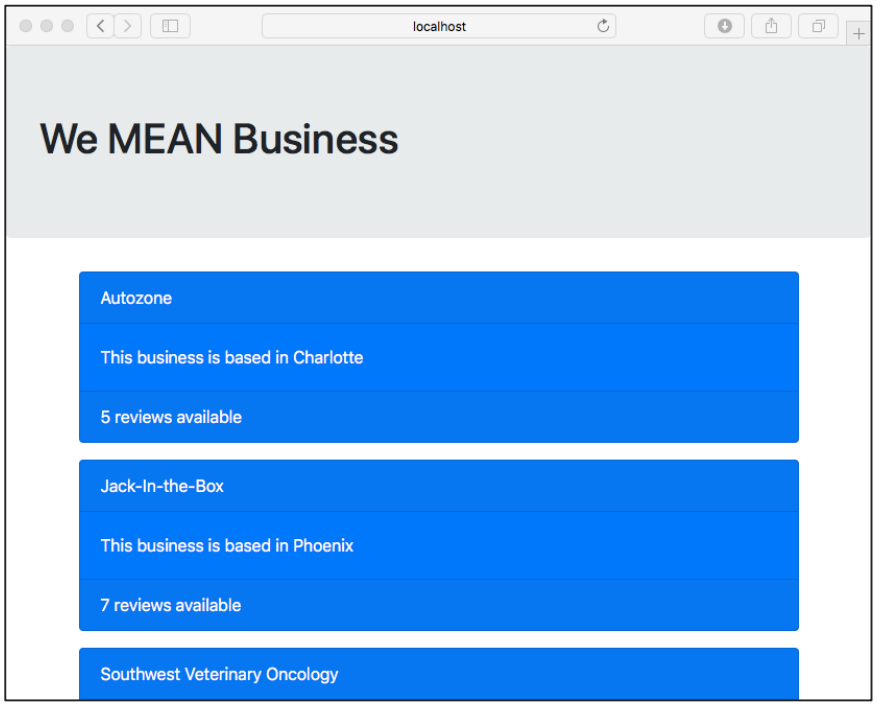

Figure C3.1 Subscribed to the Observable

C3.2 Broadcast with Subject

Although we have successfully converted the **`WebService`** to use an Observable, we have introduced a potential security flaw in that the **`business_list`** element in the **`WebService`** is publically available – so any other module could potentially access it – and modify its components

C3.2.1 Observables and Subjects

We can remedy this by introducing an **`RxJx Subject`**. This is a special type of Observable that allows values to be multicast to many receivers. While plain Observables are unicast (each subscriber owns an independent execution of the Observable), Subjects can be subscribed to by as many observers as are required.

First, we import the **`Subject`** class from **`rxjx/Rx`** and create a new Subject in the **`WebService`** class. Next, we add a line of code to the **`subscribe()`** function that calls the **`next()`** method on the Subject to alert any module that is listening to the Subject that new data is available. Finally, we can make the **`business_list`** variable private to this class to prevent outside interference.

File*: C3/src/app/web.service.ts*

```
...

import { Subject } from 'rxjs/Rx';

@Injectable()
export class WebService {

  private business_list = [];
  businessesSubject = new Subject();

  constructor(private http: Http) {}

  getBusinesses() {
     return this.http.get(
               'http://localhost:3000/api/businesses')
          .subscribe(response => {
              this.business_list = response.json();
              this.businessesSubject.next(this.business_list);
          })
  }

...
```

Next, we update the **`BusinessesComponent`** to subscribe to the new Subject and provide a fat arrow function that updates the local **`business_list`** variable with a copy of the data retrieved from the API.

File*: C3/src/app/businesses.component.ts*

```
...

export class BusinessesComponent {

    constructor(private webService: WebService) {}

    ngOnInit() {
      this.webService.getBusinesses();
      this.webService.businessesSubject
          .subscribe(businesses => {
               this.business_list = businesses
          })
    }

    business_list;
}
```

C3.2.2 Addressing the Subject

Finally, we can update the template so that the data is retrieved once more from the local variable rather than from the **`WebService`**.

File*: C3/src/app/businesses.component.html*

```
...

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

...
```

C3.3 Using the async Pipe

Although we have fixed the security flaw that made the **business_list** variable publically available, we now have a situation where any module could potentially invoke the **next()** method on the Subject.

The solution is to make the Subject private to the **WebService** class but to create an Observable on the subject that the modules that consume the data can subscribe to.

First, we make **businessesSubject** a private variable and create a public **business_list** variable to hold the data. At this point, it is easier to change the name of our private **business_list** variable – so we rename it to **business_private_list**

Now we can create a new public **business_list** object as an Observable on the **businessesSubject**.

File*: C3/src/app/web.service.ts*

```
...

export class WebService {

    private business_private_list = [];
    private businessesSubject = new Subject();
    business_list = this.businessesSubject.asObservable();

    constructor(private http: Http) {}

    getBusinesses() {
        return this.http.get('
                    http://localhost:3000/api/businesses')
            .subscribe(response => {
                this.business_private_list = response.json();
                this.businessesSubject.next(
                                this.business_private_list);
            })
    }

...
```

Now we return to the **BusinessesComponent** and instead of subscribing to the **businessesSubject** (which is no longer publically available), we subscribe instead to the new **business_list** Observable.

File*: C3/src/app/businesses.component.ts*

```
...

    ngOnInit() {
      this.webService.getBbusinesses();
      this.webService.business_list
          .subscribe(businesses => {
               this.business_list = businesses
          })
    }

...
```

Running the application and checking in the browser should confirm that everything still works as expected.

The final optimization is to access the data directly in the template without having to subscribe. First, remove the **`subscribe()`** operation from the **`BusinessComponent`** TypeScript file

File*: C3/src/app/businesses.component.ts*

```
...

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

...
```

Then, in the HTML template, we access the **`WebService`** **`business_list`** object directly and use the **`async`** pipe to indicate that the data source will change asynchronously and that the view should update automatically when a change is detected.

File*: C3/src/app/businesses.component.html*

```
...

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

...
```

This is a very efficient solution with minimal code in the Component function and secure, moderated access to the data provided by the **`WebService`**. You should be able to confirm its operation by checking in the browser that the details of the businesses still load as expected.

Try it now!

Make the same changes to the component that displays details of a single business. You may find it easier to modify the **`getBusiness()`** **`WebService`** function so that it returns an array containing a single business. This will allow you to use the same HTML template structure as for a collection of businesses.

C3.4 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.

C3.4.1 Specifying the slice to be taken

Our back-end application returns details on the first 5 businesses in the collection by default. We also provided optional querystring parameters **`number`** and **`start`** that allow us to change the number of businesses returned and the position from which we begin to read data. For example, a call to http://localhost:3000/api/businesses?count=10 would return the first 10 businesses, while http://localhost:3000/api/businesses?count=10&start=20 would return 10 businesses after skipping the first 20.

In this example, we will keep the default of 5 businesses per page, but provide “Next” and “Previous” buttons to add or subtract 5 from the value of **`start`** each time.

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

File*: C3/src/app/businesses.component.ts*

```
...

export class BusinessesComponent {

    constructor(private webService: WebService) {}

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

    start = 5;
}
```

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

File*: C3/src/app/web.service.ts*

```
...

 getBusinesses(start) {
   return this.http.get('
       http://localhost:3000/api/businesses?start=' + start)

...
```

Running the application and checking the web browser should confirm that the parameter is accepted and that the 2nd set of 5 businesses is returned and displayed.

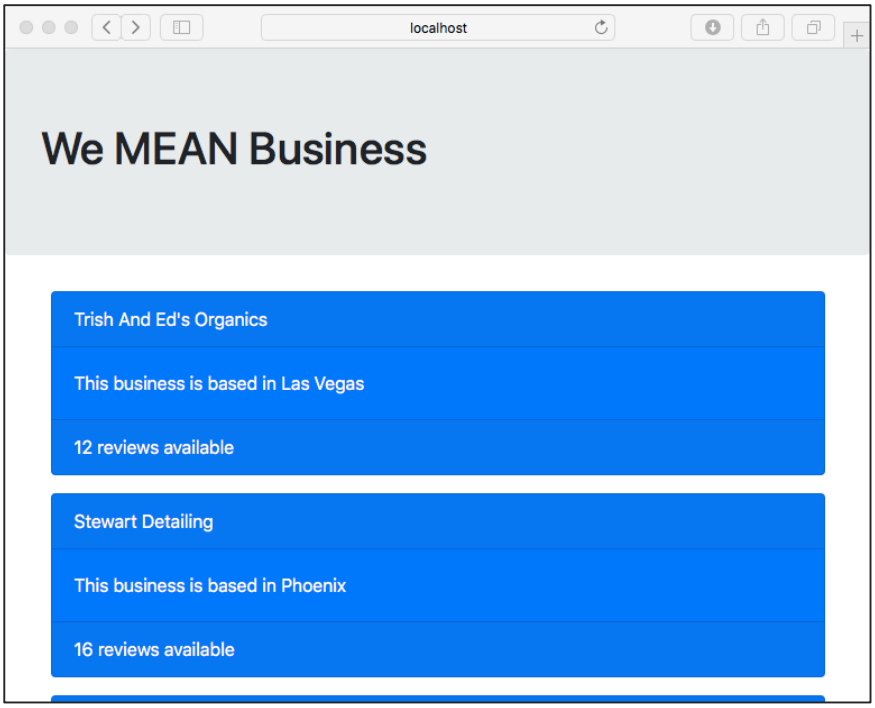

Figure C3.2 The second set of businesses

C3.4.2 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`** keywork.

File*: C3/src/app/businesses.component.html*

```
...

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

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

The buttons should be aligned to the left and right hand sides of the display as shown in Figure C3.3 below.

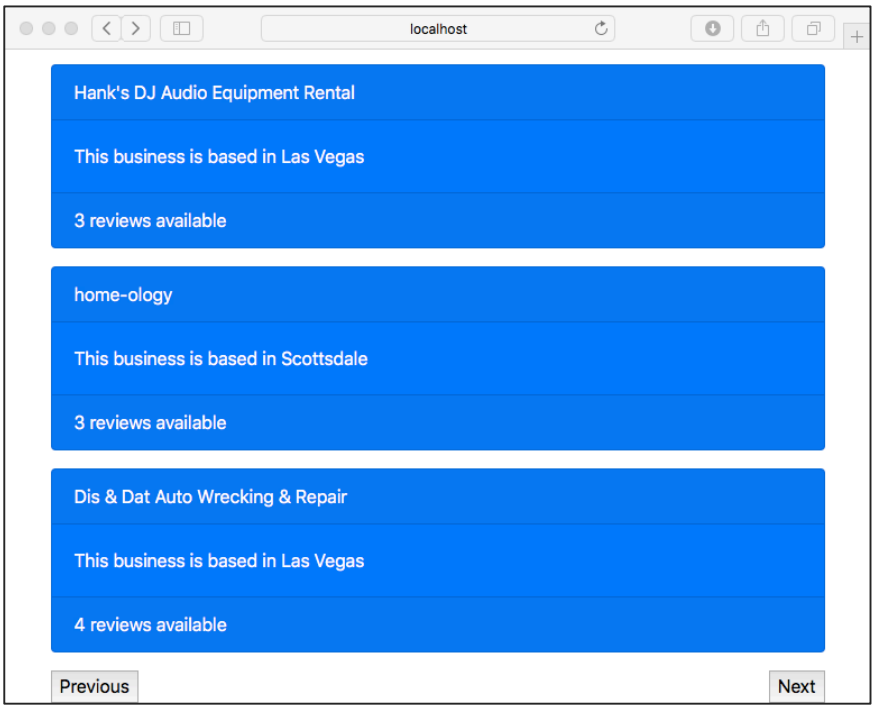

Figure C3.3 Adding Pagination Buttons

C3.4.3 Adding Component Functionality

Now, we add the **`nextPage()`** and **`previousPage()`** functions to move forward or backward through the collection of business data.

In each case, all that is required is to change the value of start accordingly 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 businesses in the collection, we could use this value in the test – but this is left for you as an exercise.

File*: C3/src/app/businesses.component.ts*

```
...

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

nextPage() {
   this.start = Number(this.start) + 5;
   this.webService.getBusinesses(this.start);
}

previousPage() {
   if (this.start > 0) {
      this.start = Number(this.start) - 5;
      this.webService.getBusinesses(this.start);
   }
}

start = 0;

...
```

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 click on an entry and load the page displaying details of that business, we find that the browser 'Back' button returns us to the first page of information – not the page that we most recently visited. This is because the **`BusinessesComponent`** is re-initialised when the page is loaded, resetting the value of **`start`** to zero.

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.

File*: C3/src/app/businesses.component.ts*

```
...

  ngOnInit() {
    if (sessionStorage.start) {
      this.start = sessionStorage.start;
    }
    this.webService.getBusinesses(this.start);
  }

  nextPage() {
      this.start = Number(this.start) + 5;
      sessionStorage.start = Number(this.start);
      this.webService.getBusinesses(this.start);
  }

  previousPage() {
      if (this.start > 0) {
        this.start = Number(this.start) - 5;
        sessionStorage.start = Number(this.start);
        this.webService.getBusinesses(this.start);
      }
  }

...
```

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