COM661 Full Stack Strategies and Development

FE12. Building a Data Service

Aims

- To demonstrate external JSON data files in an Angular application.
- To introduce Angular Services as a means of providing general purpose functionality to an application
- To create a first Data Service
- To demonstrate injecting a Service into a Component
- To use the data returned from a Data Service in the front end of an Angular application
- To build a basic pagination interface
- To implement browser storage using the Local Storage and Session Storage objects

Table of Contents

12.1 Importing JSON Data

JSON (JavaScript Object Notation) is a text-based format for representing structured data. It is very often used for communicating data between web servers and clients and is particularly useful in our work because of the close relationship to the format of the MongoDB database architecture that we will integrate to the front-end soon. In this practical, we will show how JSON data files can be used as the secondary storage for our applications and build a Data Service that provides information storage and retrieval functionality.

12.1.1 Application Assets

The assets of a software application are those elements other than the program code that decorate or otherwise support the functionality. They might include videos, images, stylesheets or other contextual information, but in this case, we will explore how our **bizFE** application can be populated with an external collection of business information specified by a JSON data file. As a first step, lets create a new Angular application to investigate the import of JSON data.

Do it now!	Create a new Angular app called **FE12** by running the command **`ng new FE12`** from a terminal prompt. Remember that due to the volume of data generated, it is better to navigate to a non-networked drive (such as the desktop on the lab PCs) and create the new application there.

12.1.2 Importing JSON

In order to maintain the theme of working with business data, we will create a JSON data structure that reflects the database previously generated. Using the **mongoexport** tool, we can generate a JSON data file that reflects the current contents of the **biz** collection in our **bizDB** database and use it as the basis for our experiments. We will then create an ***assets*** sub-folder within the ***src*** folder of our new application and copy the file of businesses data to there.

Do it now!	Obtain a JSON representation of your MongoDB **biz** collection by using **mongoexport** to write the content of the collection to the file ***businesses.json*** (or use the version supplies in the FE12 Practical Files). Create a folder ***assets*** as a sub-folder of ***FE12/src*** and place the file ***businesses.json*** into the ***assets*** folder.

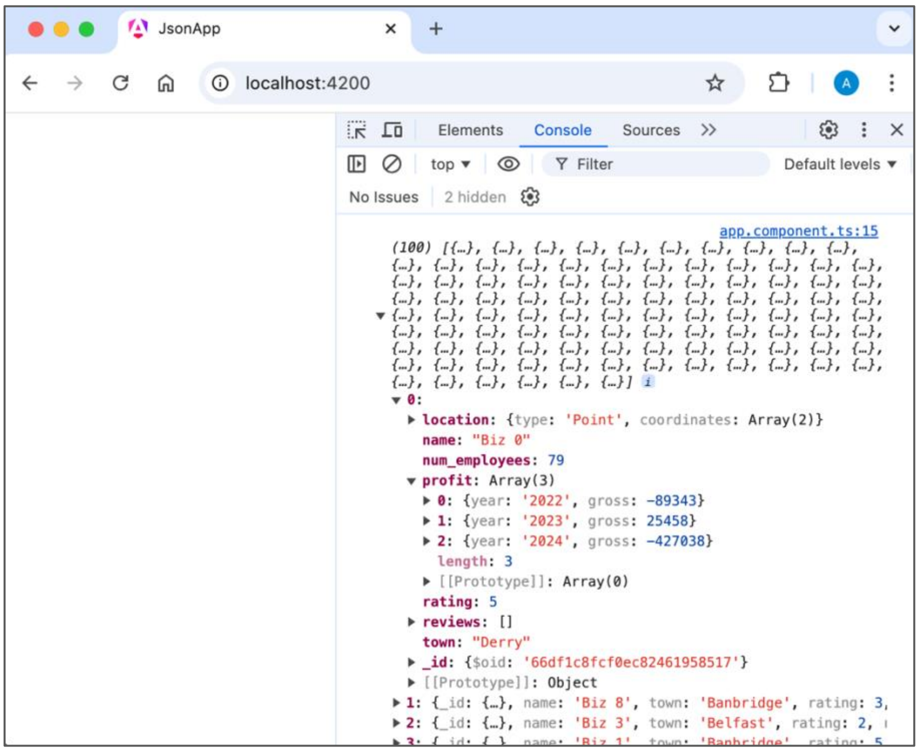
Note: Where you have different types of assets in your application, it may be useful to organize these in sub-folders of ***assets*** such as ***assets/data***, ***assets/images***, ***assets/logos***, ***assets/videos*** etc. In such cases, the path in the **`import`** statement would change accordingly, for example
`import jsonData from ‘../assets/data/businesses.json’`

Now that our database collection is represented in an external JSON file, we can demonstrate how to read the contents of that file into a Python variable. In many languages we would be required to read the file line by line and re-build the structure, but as the file contains a single JSON structure (a JSON array), we are able to import it in its entirety in a single statement. This is illustrated in the code box below, where the contents of the file ***src/assets/businesses.json*** is read into the Python variable **`jsonData`** which can then by output to the browser console by a **`console.log()`** statement. Note the definition of the **`ngOnInit()`** method within the **`AppComponent`** class. This is a method that is called automatically by Angular once a Component has been initialized and is often used to initialize variables or to set up connections (for example to back-end APIs).

File: FE12/src/app/app.ts

```
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import jsonData from ‘../assets/businesses.json’;

@Component({
  selector: 'app-root',
  imports: [RouterOutlet],
  templateUrl: './app.html',
  styleUrl: './app.css'
})
export class App {

    ngOnInit() {
       console.log(jsonData);
    }
}
```

Do it now!	Modify ***app.ts*** as shown above to import the JSON data file from the ***assets*** folder and log the contents to the browser console. Verify that the data has been imported by running the application with **`ng serve`**, visiting http://localhost:4200 in the web browser, and opening the Browser Console. Check that you see data such as that in Figure 12.1, below.



```
                                                    app.component.ts:15
  (100) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
  {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
  {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
  {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
▼ {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
  {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
  {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
  {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
  {…}, {…}, {…}, {…}, {…}, {…}] i
  ▼ 0:
    ▶ location: {type: 'Point', coordinates: Array(2)}
      name: "Biz 0"
      num_employees: 79
    ▼ profit: Array(3)
      ▶ 0: {year: '2022', gross: -89343}
      ▶ 1: {year: '2023', gross: 25458}
      ▶ 2: {year: '2024', gross: -427038}
        length: 3
      ▶ [[Prototype]]: Array(0)
      rating: 5
    ▶ reviews: []
      town: "Derry"
    ▶ _id: {$oid: '66df1c8fcf0ec82461958517'}
    ▶ [[Prototype]]: Object
  ▶ 1: {_id: {…}, name: 'Biz 8', town: 'Banbridge', rating: 3,
  ▶ 2: {_id: {…}, name: 'Biz 3', town: 'Belfast', rating: 2,
  ▶ 3: {_id: {…}, name: 'Biz 1', town: 'Banbridge', rating: 5
```

Figure 12.1 JSON Data in the Browser Console

Note:	It is a good idea to keep the Browser Console open when developing Angular applications as it is often the source of helpful error messages. In addition, use of the **`console.log()`** statement in your code can help you trace the flow of execution and trace the source of bugs.

12.2 Using a Data Service

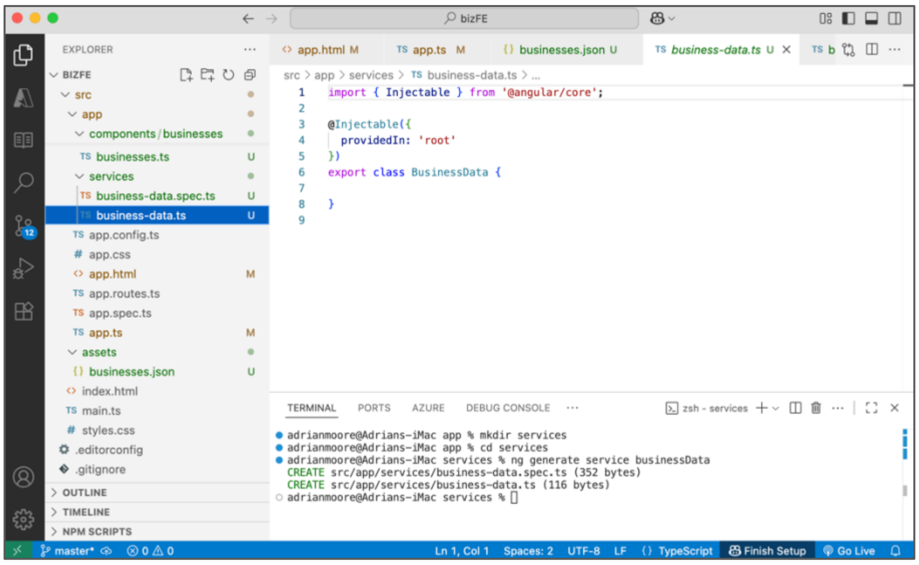
Components in Angular relate to elements that will have a physical presence in the application and that are incorporated into the application by using the tag defined as the component selector (for example **`<app-businesses></app-businesses>`** in our **bizFE** application). They comprise of a chuck of HTML, CSS and TypeScript that encapsulates some part of the application that we want to display.

Angular **Services** on the other hand, are used for common functionality that can be used across multiple parts of the application. They can be injected into any Component that requires that functionality so that the operation only has to be defined once. In this section, we introduce Angular Services by creating a Data Service that provides access to the collection of businesses.

Do it now!	Close the **FE12** application and stop the Angular application running with CTRL-C. Now, return to your **bizFE** application, create the new folder ***src/assets*** and copy your file ***businesses.json*** to the new folder.

12.2.1 Creating the Data Service

Angular Services can be generated by the command **`ng generate service`** which automatically provides the default file structure and content. As with our custom components it is best to manage these in a separate folder that we will call ***app/services***, so we open an Integrated Terminal on the ***app*** folder, create the ***services*** folder by the command **`mkdir services`** and then navigate into it by the command **`cd services`**. Now we can generate our new Data Service called **`BusinessData`** with **`ng generate service businessData`**, as shown in Figure 12.2 below.



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

@Injectable({
  providedIn: 'root'
})
export class BusinessData {

}
```

```
adrianmoore@Adrians-iMac app % mkdir services
adrianmoore@Adrians-iMac app % cd services
adrianmoore@Adrians-iMac services % ng generate service businessData
CREATE src/app/services/business-data.spec.ts (352 bytes)
CREATE src/app/services/business-data.ts (116 bytes)
adrianmoore@Adrians-iMac services % 
```

Figure 12.2 Creating the Data Service

Do it now!	Create the new Service as shown above and view the file ***business-data.ts*** that is generated.

Opening the newly generated file ***services/business-data.ts*** shows the default TypeScript file that is created. The method by which Services can be used in our Components is known as **injection**, and it can be seen that the **`@Injectable`** decorator is applied to the Service class as shown in the code box below.

File: services/business-data.ts

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

@Injectable({
  providedIn: 'root'
})
export class BusinessData {

}
```

Now we modify the **`BusinessData`** Service file ***business-data.ts*** by **`import`**ing the JSON file from the assets folder and creating a new method **`getBusinesses()`** to return the JSON collection as shown below.

File: services/business-data.ts

```
import { Injectable } from '@angular/core';
import jsonData from '../../assets/businesses.json';

@Injectable({
  providedIn: 'root'
})
export class BusinessData {

  getBusinesses() {
    return jsonData;
  }
}
```

Do it now! Make the modifications shown above to the file ***business-data.ts***.

12.2.2 Connecting to the Data Service

In order to use the new Service, we need to **`import`** it into the relevant Component and inject it into the Component class. This is carried out in three stages as follows.

First, the Service is **`import`**ed into the Component.

File: components/businesses/businesses.ts
```
import { Component } from '@angular/core';
import { BusinessData } from '../../services/business-data;
...
```

Next, we include the Service in the **`providers`** list within the Component decorator.

File: components/businesses/businesses.ts
```
...

@Component({
  selector: 'app-businesses',
  providers: [BusinessData],
  templateUrl: './businesses.component.html',
  styleUrl: './businesses.component.css'
}))

...
```

Finally, we inject the Service into the Component (i.e. make it available for use) by providing it as a parameter to the **`constructor()`** method of the Component. Here, we are making available an instance of the **`BusinessData`** Service under the name **`businessData`**. The **`private`** specifier demotes that the **`businessData`** object will only be available within the current class definition.

File: components/businesses/businesses.ts
```
...

export class Businesses {
    ...

    constructor(private businessData: BusinessData) { }

    ...
}
```

Now that the Service is available, we can use it by re-defining the local Component variable **`business_list`** as an empty list (rather than the previous hard-coded list of businesses) and populating it from the values returned from the **`BusinessData`** Service by making a call to the Service's **`getBusinesses()`** method from within the Component's **`ngOnInit()`** method.

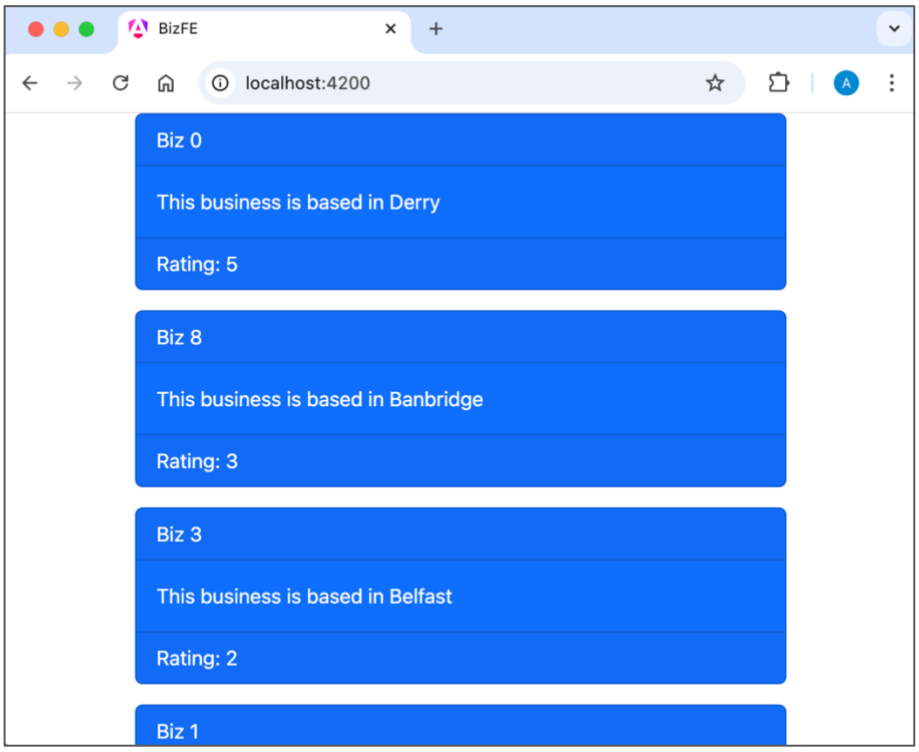
The full modified listing of ***businesses.ts*** is provided for convenience below.

File: components/businesses/businesses.ts

```
import { Component } from '@angular/core';
import { BusinessData } from '../../services/business-data';

@Component({
  selector: 'app-businesses',
  providers: [BusinessData],
  templateUrl: './businesses.html',
  styleUrl: './businesses.css'
})
export class Businesses {
  business_list: any = [ ];

  constructor(private businessData: BusinessData) { }

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

Now, when we run the application, the browser should display all of the business data returned from the **`BusinessData`** Service.

Do it now!	Connect the **`Businesses`** component to the **`BusinessData`** Service as described in the steps above. Run the application and make sure that the browser displays the complete collection of business data as a stack of Bootstrap cards, as illustrated in Figure 12.3, below.



Biz 0

This business is based in Derry

Rating: 5

Biz 8

This business is based in Banbridge

Rating: 3

Biz 3

This business is based in Belfast

Rating: 2

Biz 1

Figure 12.3 Data Service Provides Business Data

12.3 Front-end Pagination

The initial version of our **`BusinessData`** Service returns the entire collection as a single list. Depending on our application and data, this might be too much information to display within our interface in a single view, so it may be beneficial to paginate the data so that only a slice at a time is retrieved.

Pagination can be provided either at the front-end, the back-end, or both. We have already implemented back-end pagination in our API, but it can be beneficial to have all data available to the front-end and paginate there to make it easier to implement searching and filtering with minimal network traffic.

12.3.1 Specifying the Slice to be Taken

We will modify our **`BusinessData`** Service so that only a specified portion of the dataset is returned, controlled by two values. The **`BusinessData`** local variable **`pageSize`** will specify how many businesses are to be returned, while the **`getBusinesses()`** method will accept a parameter called **`page`** which determines the page of businesses required.

Then, in **`getBusinesses()`** we calculate the values **`pageStart`** and **`pageEnd`**, and return the slice of the JSON array that is bounded by these values.

File: services/business-data.ts

```
import jsonData from '../assets/businesses.json'

export class DataService {
    pageSize: number = 3;

    getBusinesses(page: number) {
        let pageStart = (page - 1) * this.pageSize;
        let pageEnd = pageStart + this.pageSize;
        return jsonData.slice(pageStart, pageEnd);
    }
}
```

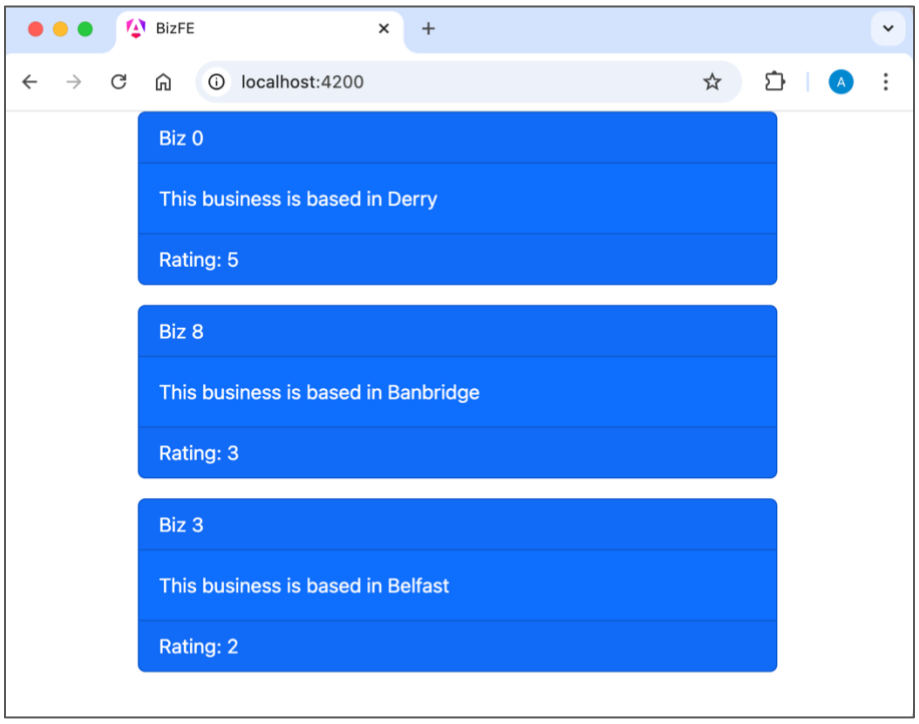
Now, we can update the **`Businesses`** component to include a local variable **`page`** that will store the page number currently being displayed and modify the call to the **`BusinessData`** **`getBusineses()`** method so that the desired page is selected. As the **`Businesses`** component HTML template will automatically display all data that is returned, no updates to the template are required at this stage.

File: components/businesses/businesses.ts

```
...

export class Businesses {
    business_list: any = [ ];
    page: number = 1;

    constructor(private businessData: BusinessData) { }

    ngOnInit() {
       this.business_list =
            this.businessData.getBusinessea(this.page);
    }
}
```

Running the application now should confirm that only the first page of three businesses is retrieved and displayed.

Do it now!	Modify the **`Businesses`** component and **`BusinessData`** service as shown to have the application display the first page of 3 businesses. Verify that you receive output such as that shown in Figure 12.4, below.



BizFE

localhost:4200

Biz 0

This business is based in Derry

Rating: 5

Biz 8

This business is based in Banbridge

Rating: 3

Biz 3

This business is based in Belfast

Rating: 2

Figure 12.4 A Page of Three Businesses Returned

Try it now!	Verify the effect of the **`page`** variable in the **`Businesses`** Component and the **`pageSize`** property in the **`BusinessData`** Service by assigning them different values and observing the effect in the browser.

12.3.2 Providing Front-end Navigation

The next step is to add a pair of buttons to the **`Businesses`** Component 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: components/businesses/businesses.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 -->
```

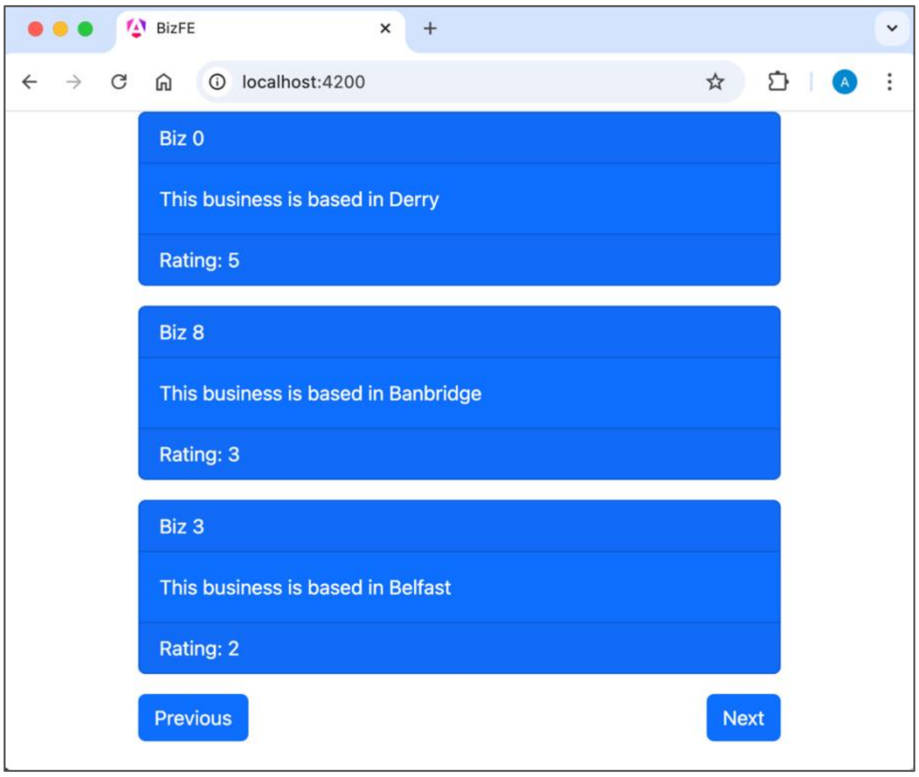
Since we have specified calls to functions **`previousPage()`** and **`nextPage()`**, we need to provide(initially) empty definitions for these in the component TypeScript file.

File: components/businesses/businesses.ts

```
...

export class Businesses {
  ...

  previousPage() {
  }

  nextPage() {
  }

}
```

The application should now run, and we can verify that the buttons are added and are aligned to the left and right of the stack of business cards as shown in Figure 12.5 below.

Do it now!	Follow the steps above to add “Previous” and “Next” buttons to the end of the **`Businesses`** Component HTML template



Biz 0
This business is based in Derry
Rating: 5

Biz 8
This business is based in Banbridge
Rating: 3

Biz 3
This business is based in Belfast
Rating: 2

Figure 12.5 Adding Pagination Buttons

12.3.3 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 **`BusinessData`** method that retrieves the data. As each of **`nextPage()`** and **`previousPage()`** request a new slice of data from **`getBusinesses()`** at the end of each call, the display will automatically change each time a new page of data is fetched.

Do it now!	Add the pagination functionality by providing code for **`previousPage()`** and **`nextPage()`** as shown below. Verify that the **Previous** and **Next** buttons now allow the user to navigate forwards and backwards through the dataset.

Try it now!	Attempt to navigate past the boundaries of the dataset by requesting the previous page when the first is being displayed or the next page when the last is being displayed.

File: components/businesses/businesses.ts

```
...

export class Businesses {
  ...

  previousPage() {
    this.page = this.page - 1
    this.business_list =
                  this.businessData.getBusinesses(this.page);
  }

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

}
```

When scrolling forward and backwards through the pages of data, we notice a problem when we attempt to go back past the first page or forward past the last page. In these cases, there is nothing to prevent the **`BusinessData`** Service from attempting to take a slice of data from outside the boundaries of the array. We need to fix these edge cases by testing the page requested against the range of pages available. In the case of the "Previous" button, it is easy – we only allow the previous page to be selected if a page other than the first is currently being displayed.

File: components/businesses/businesses.ts

```
...

export class Businesses {
  ...

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

For the “Next” button, the concept is equally simple – do not allow navigation past the last page – but how can we tell which page is the last? If we know the number of elements in the dataset then it is straightforward, but in an application where the number of elements may change over time, we need additional support.

One technique is to add functionality to the **`BusinessData`** Service to return the number of the last page of data currently stored in the collection. The calculation is to divide the length of the dataset by the size of each page and to return that result – adding 1 if there is a remainder in the division. Hence for a page size of 10, a dataset size of 20 will require two pages, which a dataset of between 21-29 will require an extra page to hold those after the first two.

TypeScript provides a useful function **`Math.ceil()`** that performs this calculation by rounding up the result of a division to the next whole number and we can use it in a new **`BusinessData`** Service method **`getLastPageNumber()`** that returns the number of pages available in the dataset.

File: services/business-data.ts

```
import jsonData from '../assets/businesses.json'

export class DataService {
    pageSize: number = 3;

    getBusinesses(page: number) {
        let pageStart = (page - 1) * this.pageSize;
        let pageEnd = pageStart + this.pageSize;
        return jsonData.slice(pageStart, pageEnd);
    }

    getLastPageNumber() {
        return Math.ceil( jsonData.length / this.pageSize );
    }
}
```

With the new **`BusinessData`** capability in place, we can return to the **`Businesses`** Component and add the check for the end of the dataset to the **`nextPage()`** method.

File: components/businesses/businesses.ts

```
...

export class BusinessesComponent {
  ...

  nextPage() {
    if (this.page < this.businessData.getLastPageNumber()) {
       this.page = this.page + 1
       this.business_list =
                  this.businessData.getBusinesses(this.page);
    }
  }

}
```

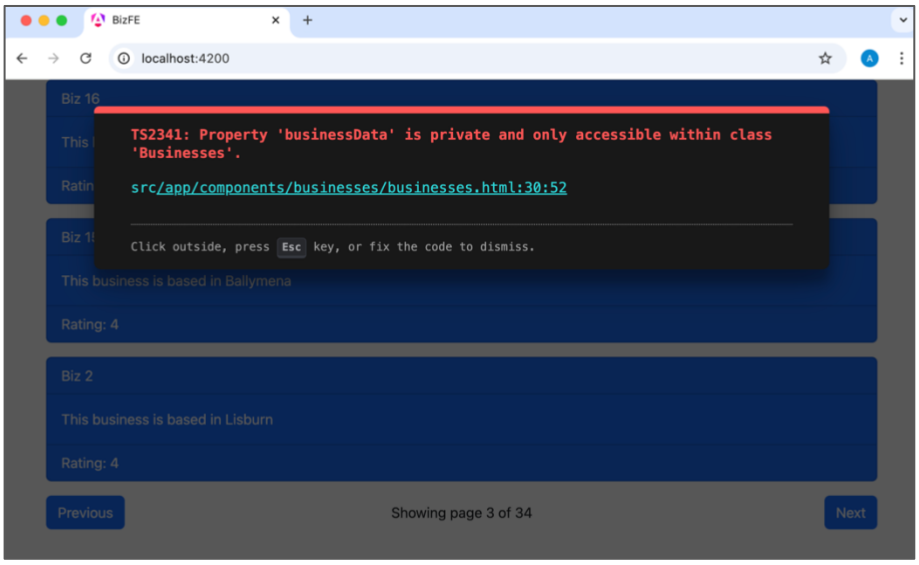
Do it now! Add the pagination control logic by making the amendments to the **`BusinessData`** Service and the **`Businesses`** Component as shown above. Verify that the **Previous** button no longer has any effect when the first page is being displayed and that the **Next** button has no effect when the last page is being displayed.

An additional useful modification would be to add code to the **`Businesses`** Component HTML template to display the current page number as well as the number of pages available. We can make use of the space between the “Previous” and “Next” buttons by converting the Bootstrap row to one of three columns and displaying a message incorporating the current value of the Component **`page`** variable as well as the result of a call to the **`BusinessData`** Service method **`getLastPageNumber()`**.

File: components/businesses/businesses.html

```
      ...

        <div class="row">
            <div class="col-sm-4">
                <button class="btn btn-primary"
                            (click)="previousPage()">Previous</button>
            </div>
            <div class="col-sm-4 text-center align-self-center">
                Showing page {{ this.page }}
                of {{ this.businessData.getLastPageNumber() }}
            </div>
            <div class="col-sm-4 text-end">
                <button class="btn btn-primary"
                            (click)="nextPage()">Next</button>
            </div>
        </div>

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

Running the application now reveals an error as the **`businessData`** object was declared to be **`private`** when it was injected into the **`Businesses`** Component, as seen in Figure 12.6, below.



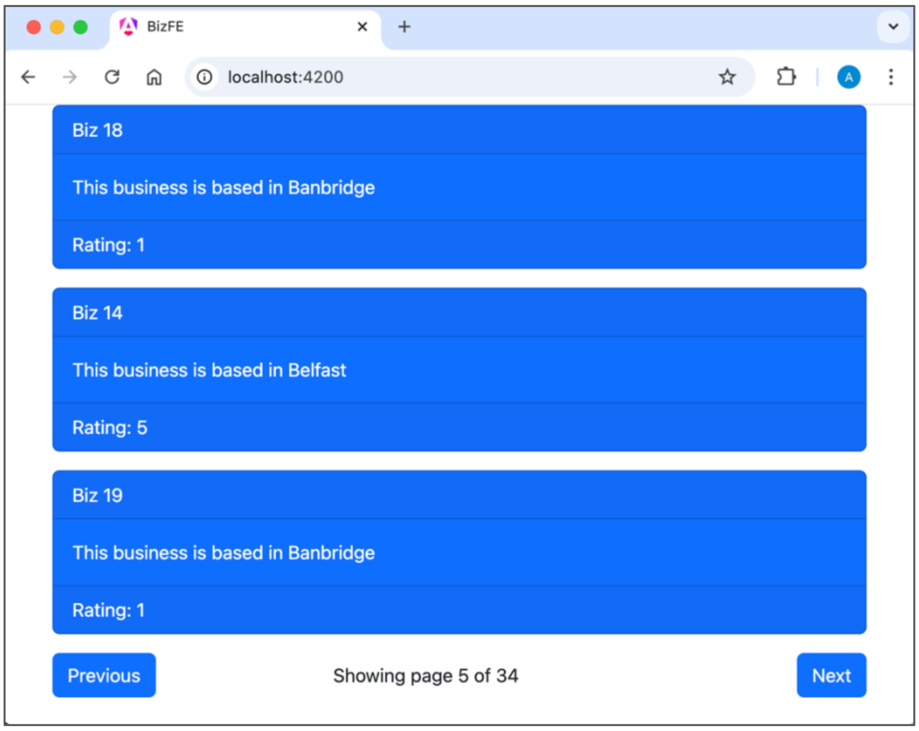
```
TS2341: Property 'businessData' is private and only accessible within class
'Businesses'.

src/app/components/businesses/businesses.html:30:52

Click outside, press Esc key, or fix the code to dismiss.
```

Figure 12.6 **`BusinessData`** *is Private*

However, this is easily addressed by returning to the definition of the **`Businesses`** Component **`constructor()`** and changing the **`businessData`** definition to be **`protected`** instead, thus allowing it to be used within the Component HTML template.

File: components/businesses/businesses.ts

```
...

export class Businesses {
    business_list: any = [];
    page: number = 1;

    constructor(protected businessData: BusinessData) { }

    ...
}
```



BizFE

localhost:4200

Biz 18

This business is based in Banbridge

Rating: 1

Biz 14

This business is based in Belfast

Rating: 5

Biz 19

This business is based in Banbridge

Rating: 1

Previous | Showing page 5 of 34 | Next

Figure 12.7 Display Pagination Status Information

Do it now! Add the pagination status information by making the changes shown above and verify that it displays the page currently being viewed and the number of pages available, as seen in Figure 12.6, above. Change the **`pageSize`** property in the **`BusinessData`** Service and observe the effect on navigation.

12.4 Using Browser Storage

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, we are returned to the first page of information – not the page that we most recently visited. This is because the **`Businesses`** Component is re-initialised every time it is loaded, resetting the value of page to 1.

12.4.1 HTML5 Data Storage

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.

Do it now!	Illustrate the bug in our current pagination scheme by navigating to a page other than the first and refreshing the browser. Observe how the re-initialisation of the **`Businesses`** Component reverts to displaying the first page of information.

12.4.2 Using Session Storage to Improve Pagination

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.

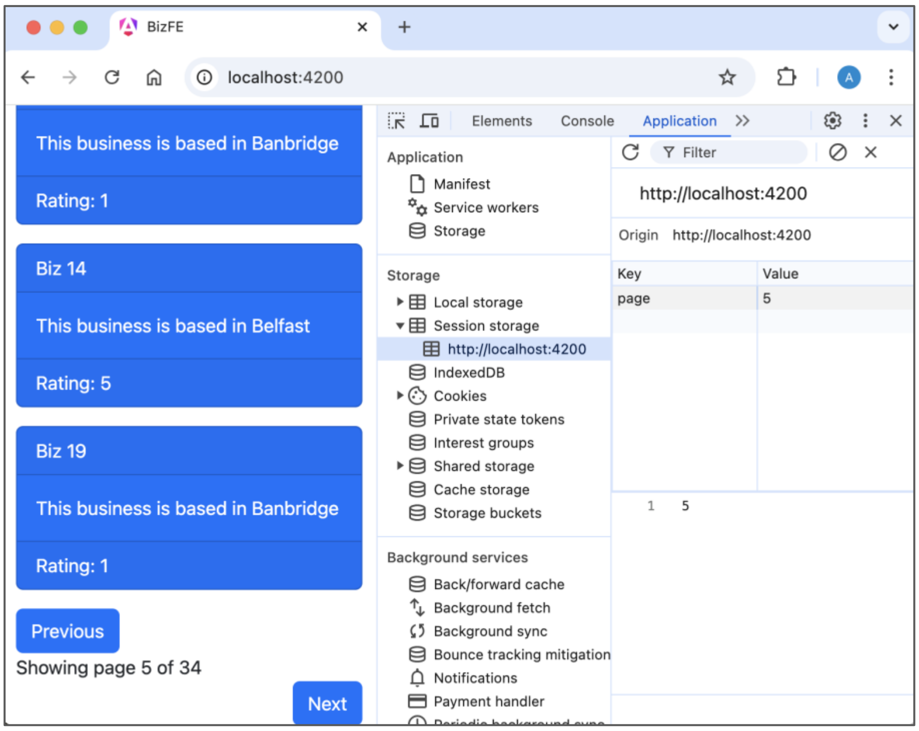
The modifications to the **`Businesses`** Component TypeScript file are presented in the following code box.

File: components/businesses/businesses.ts

```
...

export class Businessest {
  ...

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

  previousPage() {
    if (this.page > 0) {
       this.page = this.page - 1
       sessionStorage['page'] = this.page;
       this.business_list =
                   this.businessData.getBusinesses(this.page);
    }
  }

  nextPage() {
    if (this.page < this.businessData.getLastPageNumber()) {
       this.page = this.page + 1;
       sessionStorage['page' = this.page;
       this.business_list =
                  this. businessData.getBusinesses(this.page);
    }
  }
}
```

Now, running the application with the browser console open and displaying the application storage allows us to track the value of page that is maintained in **`sessionStorage`** as the application runs.

Do it now!	Add the code shown above to ***businesses.ts*** to maintain a copy of the current page number in the browser Session Storage. Now run the application with the browser console open and observe how the value for **`page`** in Session Storage updates as you navigate through the pages as illustrated in Figure 12.8, below. Observe also how refreshing the browser causes the last stored version of **`page`** to be retrieved so that the user is presented with the most recently viewed page.


This business is based in Banbridge

Rating: 1

Biz 14

This business is based in Belfast

Rating: 5

Biz 19

This business is based in Banbridge

Rating: 1

Previous

Showing page 5 of 34

Next

http://localhost:4200

Origin http://localhost:4200

Key	Value
page	5

Figure 12.8 Showing the Page Number in Session Storage

Note:	There may be occasions when reverting to the first page of data is the desired response. For example, if the user navigates away from the directory of businesses and then returns to it, you may prefer them to be returned to the first page. This is easily achieved by either deleting the value in Session Storage or re-initialising it to 1 whenever you want the pagination to be reset.

12.5 Further Information

- https://www.thirdrocktechkno.com/blog/how-to-read-local-json-files-in-angular/
  How to read local JSON files in Angular

- https://developer.chrome.com/docs/devtools/console
  Using the Browser Console

- https://angular.dev/tutorials/first-app/09-services
  Angular Services

- https://angular.dev/guide/di
  Dependency Injection in Angular

- https://angular.dev/guide/components
  Angular – Introduction to Components

- https://angulardive.com/blog/angular-services-vs-components-understanding-when-to-use-each/
  Angular Services vs Components: Understanding When to Use Each

- https://angular.dev/api/core/OnInit
  Using **`ngOnInit()`**

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

- https://www.tutorialrepublic.com/html-tutorial/html5-web-storage.php
  HTML5 Web Storage

- https://www.geeksforgeeks.org/what-are-the-different-types-of-storage-in-html5/
  Different Types of Storage in HTML5

- https://medium.com/@hassaanali.dev/efficient-data-pagination-in-angular-implementing-smooth-and-performant-pagination-for-large-data-637fe994bbbf
  Efficient Data Pagination in Angular