COM661 Full Stack Strategies and Development

FE15. Sub-documents and Reactive Forms

Aims

- To generate sample reviews and add to the dataset
- To add functionality to retrieve and display sub-documents
- To introduce Bootstrap form classes
- To use the Angular FormBuilder to connect the form template to the component logic
- To introduce validation with Angular ReactiveForms
- To provide visual feedback associated with form validation
- To introduce two-way data binding

Table of Contents

15.1 Retrieving Sub-documents

So far, we have provided our Angular application with the capability to browse a paginated collection of businesses and select a single business to display in more detail. Each business object also has the facility to store a collection of user reviews, and the next stage of development is to retrieve and display the reviews for a selected business and to provide a means by which a user can add their own review.

15.1.1 Generating Sample Reviews

Just as we provided a collection of dummy data in the file ***businesses.json***, so we can modify the data file to include a random number of randomly-generated reviews for each business. The reviews will be generated by a new function that we will add to the Data Service and invoke when the application runs by adding a call to the new function in the **`ngOnInit()`** method of the App Component. The new Data Service function will be called **`populateReviews()`**, so the first step is to modify the App Component to import the **`DataService`**, make it available by injecting it into the class constructor, and provide an **`ngOnInit()`** method to call the new function.

File: bizFE/src/app.component.ts

```
...
import { DataService } from  './data.service';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, BusinessesComponent, NavComponent],
  providers: [DataService],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {
  title = 'bizFE';

  constructor(private dataService: DataService) { }

  ngOnInit() {
    this.dataService.populateReviews();
  }
}
```

Do it now!	Make the modifications shown above to the App Component. The call to the **`populateReviews()`** function will generate an error at this point as it has not yet been defined, but this will be rectified in the following section.

Next, we add the new **`populateReviews()`** function to the Data Service. In this, we will make use of techniques for generating random data so that the appearance of the review on the web page is as realistic as possible. There is a lot going on in this function, so we will consider it a section at a time.

First, we declare a variable to hold the *Lorem Ipsum* string from which we will randomly select the review text, as well as the dummy review that we will add to the dataset

```
        let loremIpsum = <String>"";
        let dummyReview = <any>{};
```

Next, we make a call to our existing Data Service function **`getLoremIpsum()`** to request a single paragraph of dummy text from the **api-ninjas** endpoint that we have already used. As the call to **`getLoremIpsum()`** returns an **`Observable`**, we need to **`subscribe`** to it so that we can use the result in this function. Once the API produces the text, we copy it to our local **`loremIpsum`** variable.

```
        ...

        this.getLoremIpsum(1).subscribe((response: any) => {
            loremIpsum = response.text;
        }
```

Now that the Lorem Ipsum has been retrieved, we can visit each business in the collection and generate a random number of reviews (in the range 0-9) that we will add to that business.

```
...

this.getLoremIpsum(1).subscribe((response: any) => {
    loremIpsum = response.text;
    jsonData.forEach( function(business) {
       let numReviews = Math.floor(Math.random() * 10);
       for (var i = 0; i < numReviews; i++) {

       }
    })
}
```

Now, to generate the text for each review, we generate the number of characters in the text as a random value in the range 10-300 and then choose a random starting point in the *Lorem Ipsum* making sure that the starting point leaves sufficient text in the string to satisfy the review length that has been generated.

```
...

       for (var i = 0; i < numReviews; i++) {
           let textSize = Math.floor(Math.random() * 290 + 10);
           let textStart = Math.floor(Math.random() *
                          (loremIpsum.length - textSize));

       }
...
```

With all the required values in place, we can now generate the dummy review. The **`username`** field will be "User " plus a random integer in the range 1-9999; the **`comment`** field is a slice of the *Lorem Ipsum* beginning at the calculated **`textStart`** position and containing **`textSize`** characters, and the **`stars`** field is a random integer value in the range 1-5.

Once generated, we add the review by pushing it onto the **`reviews`** list for the business.

```
...

       for (var i = 0; i < numReviews; i++) {
           let textSize = Math.floor(Math.random() * 290 + 10);
           let textStart = Math.floor(Math.random() *
                          (loremIpsum.length - textSize));
           dummyReview = {
                'username' : 'User ' +
                      Math.floor(Math.random() * 9999 + 1),
                'comment' : loremIpsum.slice(textStart,
                                 textStart + textSize),
                'stars' : Math.floor(Math.random() * 5) + 1
            };
            business['reviews'].push(dummyReview);

       }
...
```

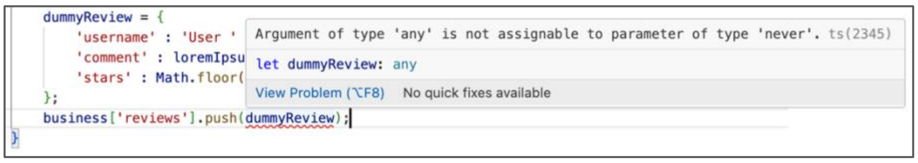
The full **`populateReviews()`** function is shown in the following code box.

Do it now!	Implement the **`populateReviews()`** function in the Data Service as shown in the code box below.

File: bizFE/src/data.service.ts

```
      ...

      populateReviews() {

        let loremIpsum = <String>"";
        let dummyReview = <any>{};

        this.getLoremIpsum(1).subscribe((response: any) => {
          loremIpsum = response.text;

             jsonData.forEach( function(business) {
                let numReviews = Math.floor(Math.random() * 10);
                for (var i = 0; i < numReviews; i++) {
                   let textSize = Math.floor(Math.random() * 290 + 10);
                   let textStart = Math.floor(Math.random() *
                                  (loremIpsum.length - textSize));
                   dummyReview = {
                        'username' : 'User ' +
                              Math.floor(Math.random() * 9999 + 1),
                        'comment' : loremIpsum.slice(textStart,
                                             textStart + textSize),
                        'stars' : Math.floor(Math.random() * 5) + 1
                    };
                    business['reviews'].push(dummyReview);
                }
             })
        }

      ...
```

When you code **`populateReviews()`** you may see that the final operation to push the new review into the reviews list of the business generates a code error as shown in Figure 15.1, below.

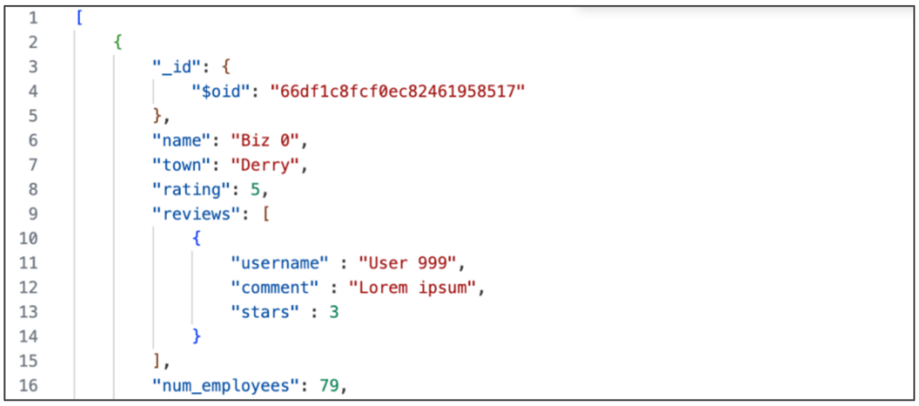


```
    dummyReview = {
        'username' : 'User '
        'comment' : loremIpsu
        'stars' : Math.floor(
    };
    business['reviews'].push(dummyReview);
}
```

```
Argument of type 'any' is not assignable to parameter of type 'never'. ts(2345)
let dummyReview: any
View Problem (⌥F8)   No quick fixes available
```

Figure 15.1 Reviews Object Assigned Type **`never`**.

This error occurs because of the strong typing enforced by TypeScript. When the file ***businesses.json*** contains an empty reviews list for every business, TypeScript assumes that this will always be the case and automatically assigns reviews a type of **`never`** – denoting that the field will never contain a value. Obviously, this is not our intention and, since this is dummy data, the easiest fix is to provide one of the entries in the ***businesses.json*** data file with a dummy review, as shown in Figure 15.2, below.



```
[
    {
        "_id": {
            "$oid": "66df1c8fcf0ec82461958517"
        },
        "name": "Biz 0",
        "town": "Derry",
        "rating": 5,
        "reviews": [
            {
                "username" : "User 999",
                "comment" : "Lorem ipsum",
                "stars" : 3
            }
        ],
        "num_employees": 79,
```

*Figure 15.2 Adding a Dummy Review to the **businesses.json** Dataset*

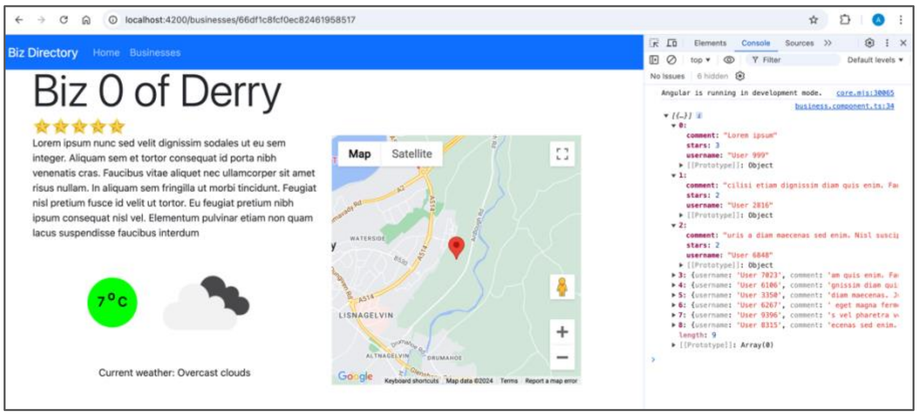
Now that the error has been resolved, we can show that the reviews have been created by logging them to the Browser Console once the business data has been returned from the Data Service by a call from the Business Component **`ngOnInit()`** function.

File: bizFE/src/business.component.ts

```
      ...

          ngOnInit() {
              this.business_list =  this.dataService.getBusiness(
                                  this.route.snapshot.paramMap.get('id');)
              console.log(this.business_list[0]['reviews'])

      ...
```

We can now run the application with the Browser Console open, select one of the businesses, and see how the randomly generated reviews have been generated and retrieved, as shown in Figure 15.3, below.

Do it now!	Modify the dataset by adding a review to one of the businesses and run the application with the Browser Console open. Verify that you see the reviews that have been generated by expanding the element in the Console.



Biz 0 of Derry

Lorem ipsum nunc sed velit dignissim sodales ut eu sem integer. Aliquam sem et tortor consequat id porta nibh venenatis cras. Faucibus vitae aliquet nec ullamcorper sit amet risus nullam. In aliquam sem fringilla ut morbi tincidunt. Feugiat nisl pretium fusce id velit ut tortor. Eu feugiat pretium nibh ipsum consequat nisl vel. Elementum pulvinar etiam non quam lacus suspendisse faucibus interdum

7°C

Current weather: Overcast clouds

```
Angular is running in development mode.        core.mjs:30065
                                          business.component.ts:34
▼ [{…}] 
  ▼ 0:
      comment: "Lorem ipsum"
      stars: 3
      username: "User 999"
    ▶ [[Prototype]]: Object
  ▼ 1:
      comment: "cilisi etiam dignissim diam quis enim. Fa
      stars: 2
      username: "User 2816"
    ▶ [[Prototype]]: Object
  ▼ 2:
      comment: "uris a diam maecenas sed enim. Nisl suscip
      stars: 2
      username: "User 6848"
    ▶ [[Prototype]]: Object
  ▶ 3: {username: 'User 7023', comment: 'am quis enim. Fa
  ▶ 4: {username: 'User 6106', comment: 'gnissim diam qui
  ▶ 5: {username: 'User 3350', comment: 'diam maecenas. J
  ▶ 6: {username: 'User 6267', comment: ' eget magna ferm
  ▶ 7: {username: 'User 9396', comment: 's vel pharetra v
  ▶ 8: {username: 'User 8315', comment: 'ecenas sed enim.
    length: 9
  ▶ [[Prototype]]: Array(0)
```

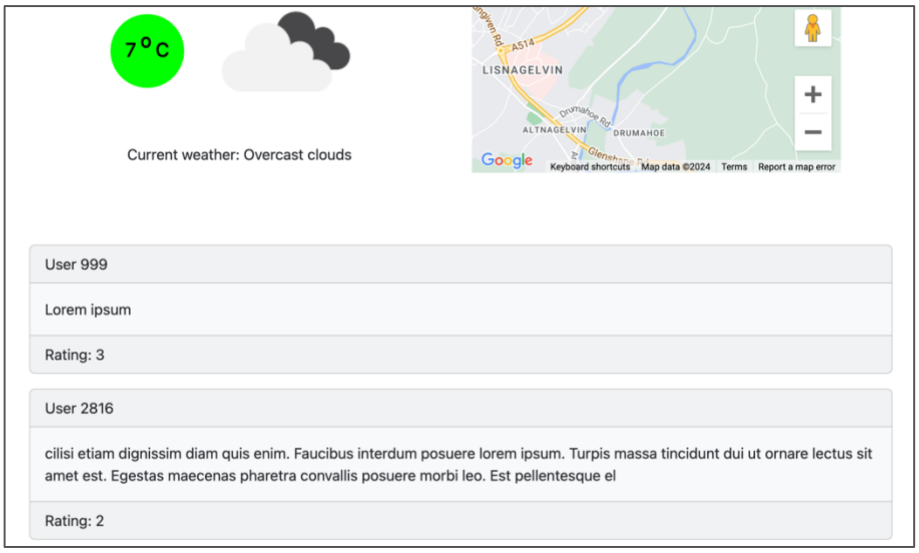
Figure 15.3 Randomly Generated Reviews in the Browser Console

15.1.2 Displaying the Reviews

With the reviews available in the Business Component for display in the browser, we can now add code to the Component HTML template to render them to the display.

We can add this to our existing layout by providing a new Bootstrap row containing a single column that spans the entire container width. Inside this, we use a `@for` block to iterate across each element of the `business.reviews` list. As for the Businesses Component, we will display each review as a Bootstrap `card`, except this time we will use the `bg-light` style class to distinguish a review `card` from a business `card`.

Inside the card, we display the `username` property of the review in the card header, the randomly generated `comment` text in the card body, and an indication of the random `rating` in the card footer. The effect is to display the collection of reviews below the business details as illustrated in Figure 15.4 below.



7°C

Current weather: Overcast clouds

User 999

Lorem ipsum

Rating: 3

User 2816

cilisi etiam dignissim diam quis enim. Faucibus interdum posuere lorem ipsum. Turpis massa tincidunt dui ut ornare lectus sit amet est. Egestas maecenas pharetra convallis posuere morbi leo. Est pellentesque el

Rating: 2

Figure 15.4 Displaying the Reviews

File: bizFE/src/business.component.html

```
      <div class="container"  style="margin-top: 70px">
         @for ( business of business_list; track business.name ) {
         ...

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

                 @for ( review of business.reviews; track review ) {

                    <div class="card bg-light mb-3">

                        <div class="card-header">
                            {{ review.username }}
                        </div>
                        <div class="card-body">
                            {{ review.comment }}
                        </div>
                        <div class="card-footer">
                            Rating: {{ review.stars }}
                        </div>

                    </div>
                 }

              </div>
           </div>


         }
      </div>
```

Do it now!	Add code to the Business Component HTML template as shown in the code box above and verify that you can see the randomly generated collection of reviews as demonstrated by Figure 15.4.

15.2 Forms

In order to invite the user to contribute a new review for the business, we will add an HTML form below the list of reviews. Angular provides a powerful form specification and manipulation facility that implements the form structure as a model and enables two-way binding between the model and the form fields. We will see in this section how this two-way binding provides a powerful validation tool as well as making it easy for us to retrieve data from the form after submission.

As an initial step, we need to import the **`ReactiveFormsModule`** class into the Business Component and add it to the Component **`imports`** list.

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

import { ReactiveFormsModule } from '@angular/forms';

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

...
```

15.2.1 Create the Form

Next, we add a form to the Business Component template with 3 fields

- A user name
- A free text review
- A star rating in the range 1-5

The username is implemented as an HTML **`input`** box, the review is implemented as a **`textarea`** component, and the star rating is provided as a drop-down list (using the **`<select>`** and **`<option>`** tags). All style classes in the code at this stage are basic Bootstrap 5 properties – with the exception of **`formControlName`** which is an Angular property that is used to bind the form field to an element in the model that describes the data structure representing the form.

File: bizFE/src/business.component.html

```
...

<div class="container" style="margin-top: 50px;">
    <div class="row">
        <div class="col-sm-12">
            <h2>Please review this business</h2>
            <form>
                <div class="form-group">
                    <label for="username">Username</label>
                    <input type="text" name="username"
                           id="username″ class="form-control"
                           formControlName="username">
                </div>
                <div class="form-group">
                    <label for="comment">
                        Please leave your review below</label>
                    <textarea name="comment" id="comment"
                              class="form-control"
                              formControlName="comment">
                    </textarea>
                </div>
                <div class="form-group">
                    <label for="stars">
                          Please provide a rating (5 = best)
                    </label>
                    <select name="stars" id="stars"
                            class="form-control"
                            formControlName="stars">
                        <option value="1">1 star</option>
                        <option value="2">2 stars</option>
                        <option value="3">3 stars</option>
                        <option value="4">4 stars</option>
                        <option value="5">5 stars</option>
                    </select>
                </div>
                <button class="btn btn-primary"
                        type="submit"
                        style="margin-top: 20px">
                    Submit
                </button>
            </form>
        </div>
    </div>
</div>
```

15.2.2 Using Angular FormBuilder

Now that the form has been implemented in the template, we update the Component's TypeScript file to import the **`FormBuilder`** class, inject it into the Component **`class`** by providing it as a parameter to the class constructor, and specify the model that describes the data structure represented by the form.

Note that we use '***username'***, '***comment*** and '***stars***' as the field names in the model – matching the values used for **`formControlName`** properties in the template.

File: bizFE/src/business.component.ts

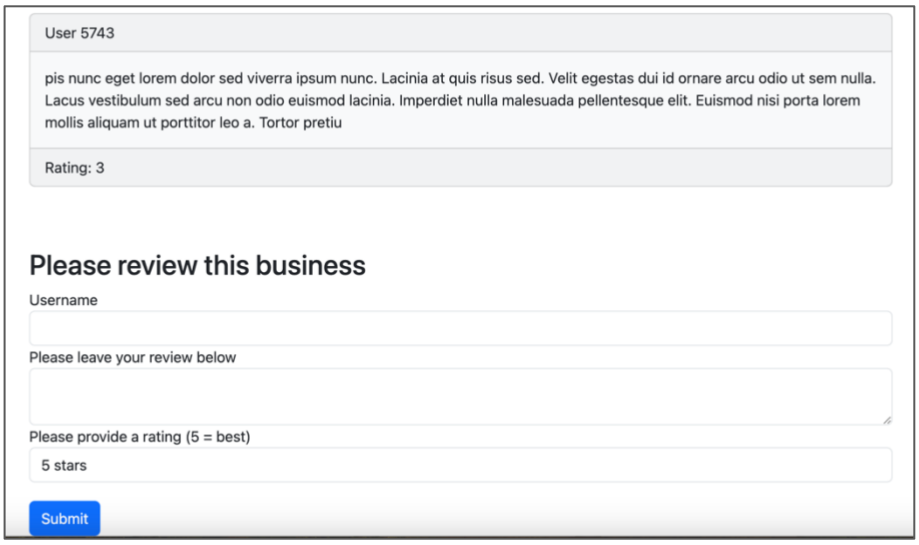
```
...

import { FormBuilder } from '@angular/forms';

...

export class BusinessComponent {

    ...

    reviewForm: any;

    constructor( public dataService: DataService,
                 private route: ActivatedRoute,
                 private formBuilder: FormBuilder) {}

    ngOnInit() {

        this.reviewForm = this.formBuilder.group( {
          username: '',
          comment: '',
          stars: 5
        })

    ...
```

We then complete the connection between the **`formBuilder.group()`** and the form by specifying the **`reviewForm`** model as the value of the **`formGroup`** property in the **`<form>`** tag.

File: bizFE/src/business.component.html

```
      ...

      <div class="col-sm-12">
                  <h2>Please review this business</h2>
                  <form [formGroup] = "reviewForm">
                      <div class="form-group">

      ...
```

Now, when we run the application and click on an individual business entry, we should now see the form to provide a new review displayed at the bottom of the page, as seen in Figure 15.5 below.



User 5743

pis nunc eget lorem dolor sed viverra ipsum nunc. Lacinia at quis risus sed. Velit egestas dui id ornare arcu odio ut sem nulla. Lacus vestibulum sed arcu non odio euismod lacinia. Imperdiet nulla malesuada pellentesque elit. Euismod nisi porta lorem mollis aliquam ut porttitor leo a. Tortor pretiu

Rating: 3

Please review this business

Username

Please leave your review below

Please provide a rating (5 = best)

5 stars

Figure 15.5 A Review Form

This is another demonstration of the binding provided by Angular. Note that we did not specify any of the 5 options for the **`<select>`** element buttons as selected in the form – yet Angular has chosen to use the '5 stars' option as the default. This is actually specified in the **`formBuilder.group()`** definition that we added to the **`BusinessComponent`** TypeScript file.

Do it now!	Add the code to the Business Component HTML template and TypeScript files as shown above to create the review form and display it below the list of reviews for the business display it. Confirm that you see output such as that demonstrated in Figure 15.5, above.

15.2.3 Submitting Form Values

Angular forms are submitted by binding a function to the form's **`ngSubmit`** property. As a first step, we will bind the **`onSubmit()`** function, which we then implement in the **`BusinessComponent`** as a simple **`console.log()`** of the model.

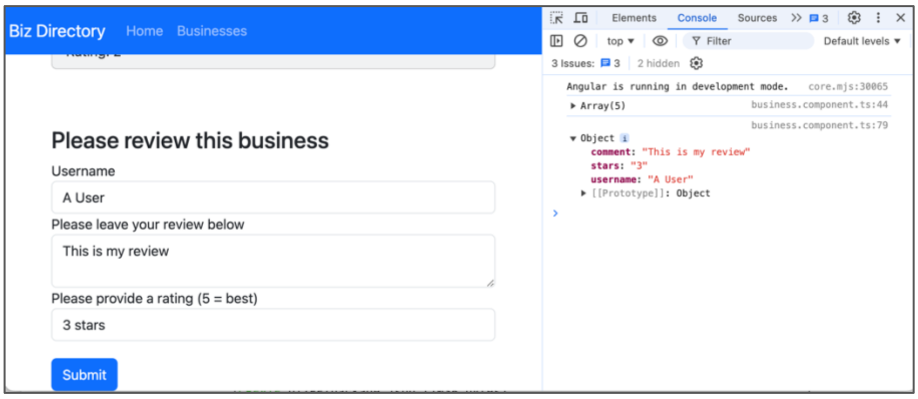
File: bizFE/src/business.component.html

```
...

<div class="col-sm-12">
            <h2>Please review this business</h2>
            <form [formGroup] = "reviewForm"
                  (ngSubmit) = "onSubmit()">
                <div class="form-group">

...
```

File: bizFE/src/business.component.ts

```
...

export class BusinessComponent{

    ...

    onSubmit() {
      console.log(this.reviewForm.value);
    }

...
```

Entering values into the form fields and clicking the submit button generates a browser console message as shown in Figure 15.6 below.



Please review this business

Username

A User

Please leave your review below

This is my review

Please provide a rating (5 = best)

3 stars

```
Angular is running in development mode.        core.mjs:30065
▸ Array(5)                               business.component.ts:44
                                         business.component.ts:79
▾ Object
    comment: "This is my review"
    stars: "3"
    username: "A User"
  ▸ [[Prototype]]: Object
>
```

Figure 15.6 Form Values Submitted

Do it now!	Add the **`onSubmit()`** functionality as described above. Provide values in the review form and make sure that you can see the submitted values in the Browser Console.

15.3 Validation with Reactive Forms

Angular provides a powerful set of validation elements that can be used to change the appearance or operation of the form in response to user input (or lack of input). In order to apply validation, we first need to import the **`Validators`** class into the **`BusinessComponent`**.

Next, we specify that we want to ensure that data is provided for the **`name`** and **`review`** fields of the form by adding the **`Validators.required`** property to the **`name`** and **`review`** fields. We do this by converting the value of each **`formBuilder.group()`** property as a list, where the first element is the default value to be used and the second is the validation requirement. As the **`stars`** rating does not require validation (it is impossible to avoid choosing one of the values), we have no change to make for that element.

Finally, we observe the effect of the validation by modifying the **`console.log`** to output the form's **`valid`** property.

File: bizFE/src/business.component.ts

```
...

import { FormBuilder, Validators } from '@angular/forms';

...

    ngOnInit() {

        this.reviewForm = this.formBuilder.group( {
          username : ['', Validators.required],
          comment: ['', Validators.required],
          stars: 5
        })

...

    onSubmit() {
      console.log(this.reviewForm.valid);
    }

...
```

When we check this in the browser, we see that the value **`false`** is logged to the browser console if the form is submitted with either of the **`username`** or **`comment`** fields left blank, while true is only reported if values are provided for both text fields.

Do it now!	Add validation to the Business Component TypeScript file as shown in the code box above. Make sure that the message **`false`** is shown in the Browser Console when you submit a review with one of the text fields left empty.

We would like to use this validation to provide visual feedback to the user informing them that a value for the field is required. First, we will add a style rule to the ***business.component.css*** file that will be used to apply a light red background to unfilled text entry fields.

File: bizFE/src/business.component.css

```
.error { background-color: #fff0f0 }
```

Now, we bind the new error class to the **`<input>`** box for the user name when the **`invalid`** property of the **`name`** control is true. The **`ngClass`** binding takes a JSON object (in the form **`{ name : value }`**) where the **`name`** element is the style rule and the **`value`** element is the condition that must be satisfied for the rule to be applied. This code can be read as “apply the error class to the name element when the contents of the form field are invalid”. Whether or not the value is invalid is determined by the validation rule described in the **`formBuilder.group()`** definition – i.e. that the value is **`required`**.

File: bizFE/src/business.component.html

```
      ...

      <input type="text" name="username" id="username"
             class="form-control" formControlName="username"
             [ngClass]="{ 'error' :
                          reviewForm.controls.username.invalid }">

      ...
```

Checking in the browser shows that the text box is highlighted even before the user has had an opportunity to enter any data. This is not exactly what we want, so we add another rule to the **`[ngClass]`** definition to apply the background colour only if the field has been **`touched`** (i.e. modified).

File: bizFE/src/business.component.html

```
      ...

      [ngClass]="{ 'error' : reviewForm.controls.username.invalid &&
                             reviewForm.controls.username.touched }">

      ...
```

This is a much more pleasing effect, so we apply the same rule to the **`comment`** textbox.

File: bizFE/src/business.component.html

```
...

<textarea name="comment" id="comment" class="form-control"
     formControlName="comment"
     [ngClass]="
         { 'error' : reviewForm.controls.comment.invalid &&
                     reviewForm.controls.comment.touched }">
</textarea>

...
```

This validation now works as required, but it can be easily seen that if we had a larger number of text inputs on the form, this approach would lead to a large volume of duplicated code. We can avoid this by re-factoring the code to implement the check for invalid input in a function which accepts the name of the form control as a parameter

File: bizFE/src/business.component.ts

```
...

isInvalid(control: any) {
   return this.reviewForm.controls[control].invalid &&
          this.reviewForm.controls[control].touched;
}

...
```

We can then modify the `ngClass` binding rule in the form to call the new function, passing the name of the control as a parameter.

File: bizFE/src/business.component.html

```
      ...

       <input type="text" name="username" id="username"
              class="form-control" formControlName="username"
              [ngClass]="{ 'error' : isInvalid('username') }">

      ...

       <textarea name="comment" id="comment" class="form-control"
                 formControlName="comment"
                 [ngClass]="{ 'error' : isInvalid('comment') }">
       </textarea>

      ...
```

Our final validation stage will be to provide a feedback message if the user attempts to leave a required field blank, while at the same time removing the submit button to prevent invalid submission. First we create new functions **`isUntouched()`** and **`isIncomplete()`** that return **`true`** if either field has not been touched (is still **`pristine`**) or either text input is invalid. The **`isUntouched()`** function covers the initial state where no user input has been provided to either field, while **`isIncomplete()`** combines this with the situation where data has been provided and then removed.

File: bizFE/src/business.component.ts

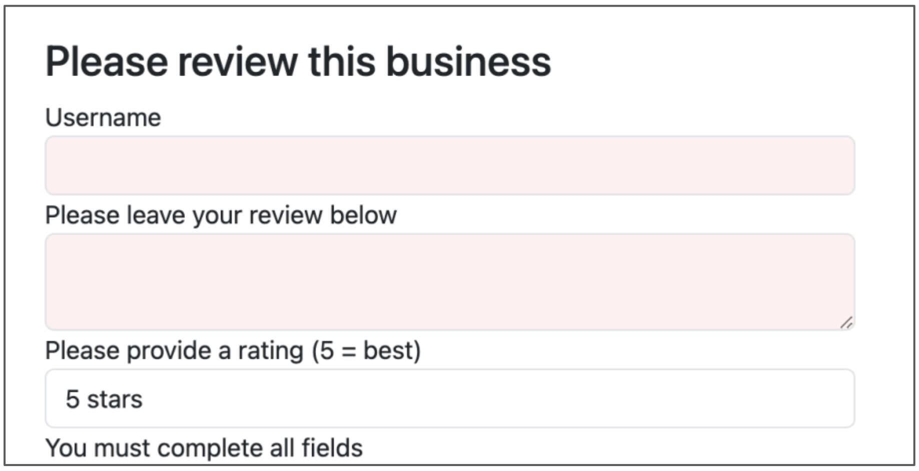
```
      ...

          isUntouched() {
            return this.reviewForm.controls.username.pristine ||
                   this.reviewForm.controls.comment.pristine;
          }

          isIncomplete() {
            return this.isInvalid('username') ||
                   this.isInvalid('comment') ||
                   this.isUntouched();
          }

      ...
```

Now, we apply this to a new **`<span>`** object containing a message – and use the Angular **`*ngIf`** directive to display either this message or the "submit" button. **`*ngIf`** is a very useful feature that can be applied to any HTML element to dynamically modify the structure and content of the page in response to dynamic activity.

File: bizFE/src/business.component.html

```
      ...

         <span *ngIf="isIncomplete()">
             You must complete all fields
         </span>
         <button *ngIf="!isIncomplete()" class="btn btn-primary"
                type="submit"
                style="margin-top: 20px">Submit</button>

      ...
```

Viewing the application in the browser verifies that we now have the desired validation.



Please review this business

Username

Please leave your review below

Please provide a rating (5 = best)

5 stars

You must complete all fields

Figure 15.7 Invalid Data and User Feedback

Do it now!	Implement the validation code as described in the above sections. Test it in the browser and ensure that the correct visual feedback is provided when required fields are not completed.

15.4 Retrieving and POSTing Form Data

Now that the review form is specified and validation is in place, we will add the functionality that allows the values provided by the user to be POSTed to the Data Service.

15.4.1 Retrieving the Data

We have already how the username, comment and stars values entered by the user on the form are available in the **`reviewForm.value`** object of the Business Component. For clarity, we can repeat that check by modifying the **`onSubmit()`** method of ***business.component.ts*** as shown below.

File: bizFE/src/business.component.ts

```
      ...

          onSubmit() {
            console.log(this.reviewForm.value);
          }

      ...
```

If we run the application and check the message in the Browser Console, we can again see that the username, review text and star rating are available. We can now create the new Data Service method **`postReview()`** to add the review to the collection. **`postReview()`** takes as parameters the **`_id`** value of the business to which the review should be attached and the contents of the review form. First, it constructs the **`newReview`** object by extracting the fields from the form. Then, it iterates across the businesses in the collection, checking for that with a matching **`_id`** value. When a match is found, the new review is pushed onto the list of reviews for that business, as shown in the code box below.

File: bizFE/src/data.service.ts

```
      ...

          postReview(id: any, review: any) {
              let newReview = {
                  'username' : review.username,
                  'comment' : review.comment,
                  'stars' : review.stars
              };
              jsonData.forEach( function(business) {
                  if ( business['_id']['$oid'] == id ) {
                      business['reviews'].push( newReview );
                  }
              });
          }

      ...
```

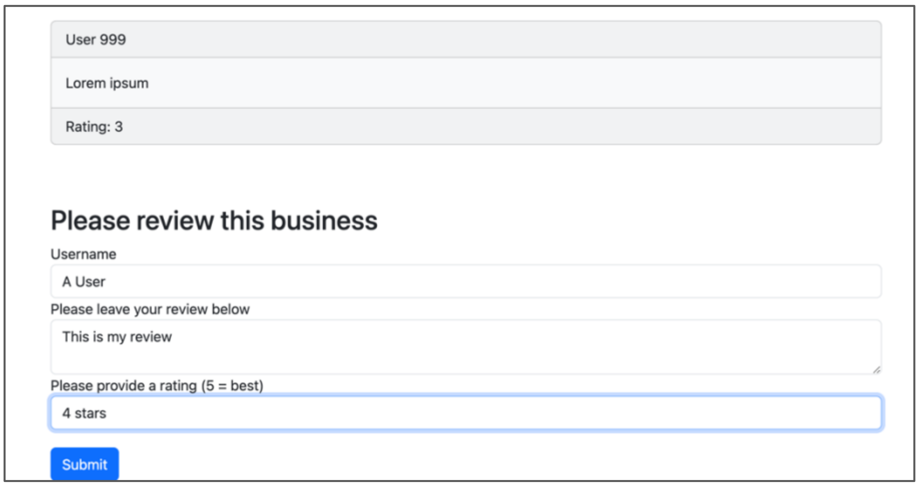
Next, we amend the Business Component **`onSubmit()`** to call the Data Service **postReview()** method. We also include a call to the form's **`reset()`** method to restore the form to its original state once the new review has been accepted.

File: bizFE/src/business.component.ts

```
    ...

        onSubmit() {
            this.dataService.postReview(
                              this.route.snapshot.paramMap.get('id'),
                              this.reviewForm.value);
            this.reviewForm.reset();
        }

    ...
```

Running the application in the browser and submitting a review should confirm that the new functionality is now complete.

Do it now!	Make the amendments to the Business Component and Data Service shown above and run the application. Select a business and observe its reviews displayed on the page. Now submit a review for the business and confirm that it has been added to the collection as shown in Figure 15.8 and 15.9, below.



User 999

Lorem ipsum

Rating: 3

Please review this business

Username

A User

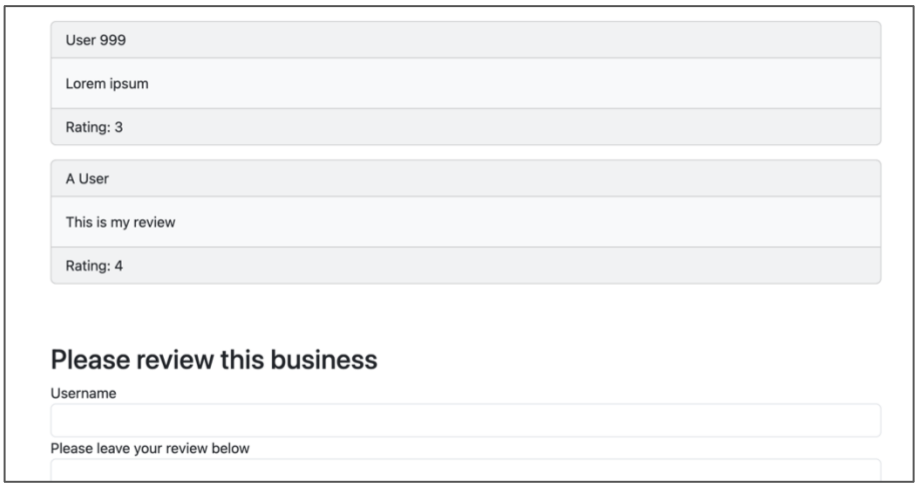
Please leave your review below

This is my review

Please provide a rating (5 = best)

4 stars

Figure 15.8 Submitting a Review



User 999

Lorem ipsum

Rating: 3

A User

This is my review

Rating: 4

Please review this business

Username

Please leave your review below

Figure 15.9 Review Accepted and Added to the Collection

15.5 Further Information

- https://coreui.io/docs/components/forms/
  Bootstrap Forms
- https://www.w3schools.com/Bootstrap/bootstrap_forms.asp
  Bootstrap Forms – W3Schools
- https://angular.dev/guide/forms/reactive-forms
  Angular ReactiveForms – the online manual
- https://malcoded.com/posts/angular-fundamentals-reactive-forms/
  Reactive Forms with Angular – using easy examples!
- https://angular.dev/api/forms/FormBuilder
  Angular FormBuilder class
- https://coryrylan.com/blog/angular-form-builder-and-validation-management
  Angular Form Builder and Validation Management
- https://malcoded.com/posts/angular-reactive-form-validation/
  Validating Reactive forms in Angular
- https://angular.dev/guide/forms/form-validation
  Angular Form Validation
- https://angular.dev/guide/templates/two-way-binding
  Two-way binding in Angular