COM644 Full-Stack Web and App Development

Practical C4: Using HTTP Post

Aims

- To add functionality to retrieve and display sub-documents
- To introduce Bootstrap form classes
- To use the Angular FormBuilder to connecg 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
- To implement an http.post() request

Contents

C4.1 Retrieving Sub-documents

So far, the **WeMEANBusiness** sample application provides functionality to page through a collection of business element and display details on a single business. The data in our back-end database also provides for a collection of reviews for each business, so in this session we will first add functionality to display the collection of reviews with each business description and then enable the user to add a new review for the business.

C4.1.1 Update the Web Service

First, we need to add functionality to display the reviews for a business below the business information. The retrieval operation is quite straightforward and follows the same pattern as retrieving information on the collection of businesses and on a single business. (Implementation of this was left for you as an exercise in a previous practical.)

File: ***C4/src/app/web.service.ts***

```
...
    private businesses_private_list = [];
    private businessesSubject = new Subject();
    businesses = this.businessesSubject.asObservable();

    private business_private_list = [];
    private businessSubject = new Subject();
    business = this.businessSubject.asObservable();

    private reviews_private_list = [];
    private reviewsSubject = new Subject();
    reviews = this.reviewsSubject.asObservable();
...
```

Next, we add the **`WebService`** function that calls the API endpoint to retrieve the reviews of a business.

Here, we construct the API by appending the **`_id`** value of the business in question to the URL and invoking the **`http.get()`** method. As before, the **`http.get()`** method returns an Observable that we subscribe to with an arrow function that accepts the JSON data returned from the API and broadcasts it in the **`reviewsSubject`** by invoking the Subject's **`next()`** method.

File: *C4/src/app/web.service.ts*

```
 ...

 export class WebService {

 ...

    getReviews(id) {
         this.http.get(
              'http://localhost:3000/api/businesses/' + id +
              '/reviews')
         .subscribe(
            response => {
                this.reviews_private_list = response.json();
                this.reviewsSubject.next(
                                 this.reviews_private_list);
            }
        )
    }
}
```

C4.1.2 Update the component

As the reviews will be displayed as part of the business description, we then update the **`BusinessComponent`** by adding the call to the new **`WebService`** function to the existing call to retrieve details of the business.

File: *C4/src/app/business.component.ts*

```
 ...

   ngOnInit() {
     this.webService.getBusiness(
                     this.route.snapshot.params.id);
     this.webService.getReviews(
                     this.route.snapshot.params.id);
   }

 ...
```

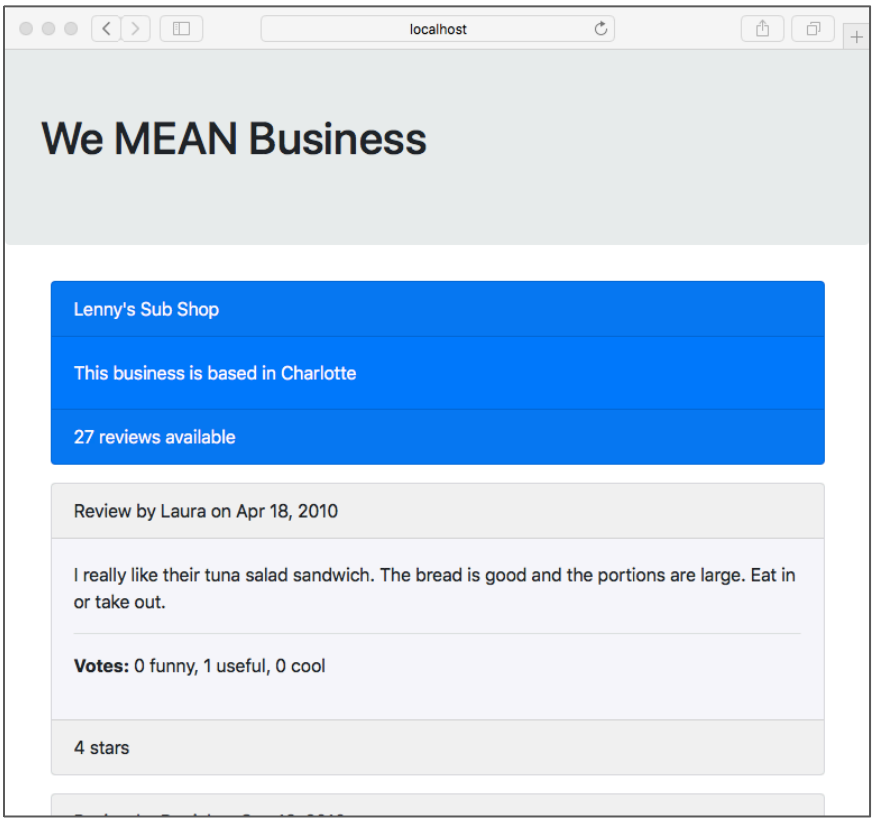
C4.1.3 Update the template

Now, we can update the **`BusinessComponent`** template to accept the list of reviews from the Observable and display each review in a separate Bootstrap card below the card that shows details of the business. We distinguish the review cards from the business information card by specifying that reviews are presented in cards with the **`bg-light`** class applied.

File: ***C4/src/app/businesses.component.html***

```
...

<div class="container">
  <div class="row">
     <div class="col-sm-12">
        <div>
           <div class="card bg-light mb-3"
                *ngFor =
                  "let review of webService.reviews | async">
              <div class="card-header">
                Review by {{ review.username }}
                       on {{ review.date | date }}
              </div>
              <div class="card-body">
                {{ review.text }}
                <hr>
                <p><strong>Votes:</strong>
                  {{ review.votes.funny }} funny,
                  {{ review.votes.useful }} useful,
                  {{ review.votes.cool }} cool </p>
              </div>
              <div class="card-footer">
                 {{ review.stars }}
                 stars
              </div>
           </div>
        </div>
     </div> <!-- col -->
  </div>    <!-- row -->
</div>
```

Checking in the browser reveals that the collection of reviews for the business are displayed as intended.


localhost

We MEAN Business

Lenny's Sub Shop

This business is based in Charlotte

27 reviews available

Review by Laura on Apr 18, 2010

I really like their tuna salad sandwich. The bread is good and the portions are large. Eat in or take out.

Votes: 0 funny, 1 useful, 0 cool

4 stars

Figure C4.1 Displaying reviews

Note: Our backend API only allows for the entire collection of reviews to be displayed at once and does not allow us to specify the order in which they are retrieved. Two potential enhancements would be (i) to allow the user to page through the reviews and (ii) to update the API so that the reviews are sorted in order of date with the most recent reviews shown first. These are left for you as an exercise.

C4.2 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 include the **`FormsModule`** and **`ReactiveFormsModule`** classes in the application's main *app.module.ts* file.

File*: C4/src/app/app.module.ts*

```
...

import { WebService } from './web.service';
import { HttpModule } from '@angular/http';
import { FormsModule, ReactiveFormsModule }
       from '@angular/forms';

...


  imports: [
    BrowserModule, HttpModule, RouterModule.forRoot(routes),
    FormsModule, ReactiveFormsModule
  ],

...
```

C4.2.1 Create the form

Next, we add a form to the **`BusinessComponent`** template with 3 fields

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

The user name 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 4 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: ***C4/src/app/businesses.component.html***

```
...

    </div> <!-- col -->
 </div>    <!-- row -->

 <h2>Please review this business</h2>
 <form>
   <div class="form-group">
     <label for="name">Name</label>
     <input type="text"id="name"
            class="form-control"
            formControlName="name">
   </div>
   <div class="form-group" >
     <label for="review">Please leave your review below
     </label>
     <textarea id="review" rows="3" name="review"
               class="form-control"
               formControlName="review"></textarea>
   </div>
   <div class="form-group">
     <label for="stars">Please provide a rating
                       (5 = best)</label>
     <select id="stars" name="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 type="submit"
           class="btn btn-primary">Submit</button>
 </form>

<div>
```

C4.2.2 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`** and specify the model that describes the data structure represented by the form.

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

File*: C4/src/app/business.component.html*

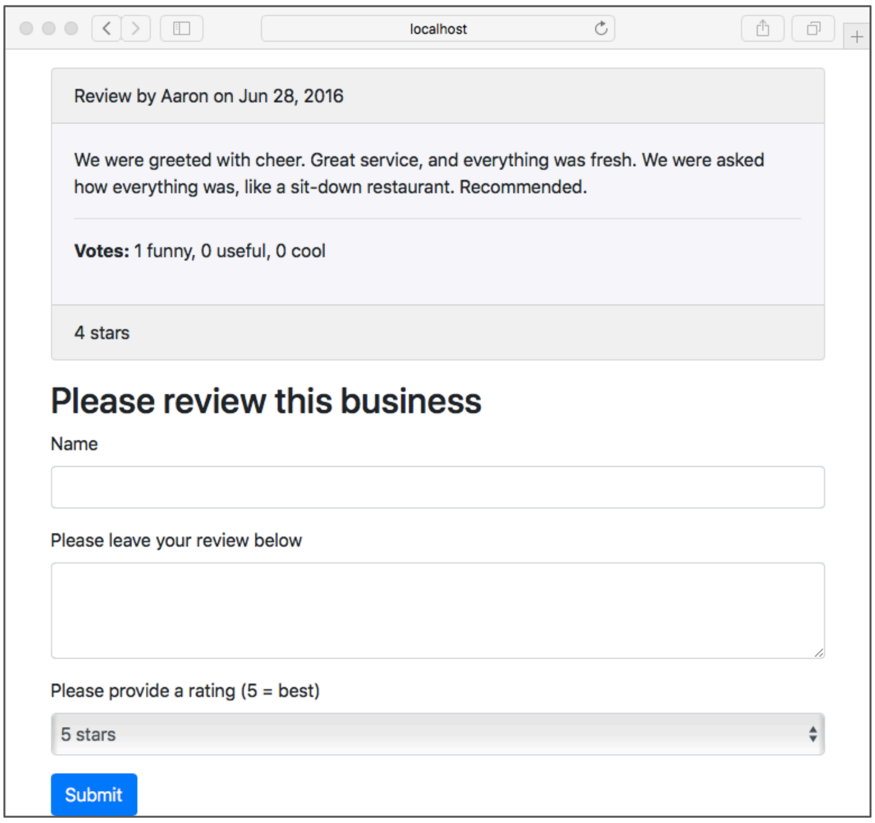
```
import { Component } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
import { WebService } from './web.service';
import { FormBuilder } from '@angular/forms';

...

export class BusinessComponent {

  reviewForm;

  constructor(private webService: WebService,
              private route: ActivatedRoute,
              private formBuilder: FormBuilder) {

    this.reviewForm = formBuilder.group( {
       name: '',
       review: '',
       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*: C4/src/app/businesses.component.html*

```
...

<h2>Please review this business</h2>
<form [formGroup]=”reviewForm”>

...
```

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 C4.2 below.


localhost

Review by Aaron on Jun 28, 2016

We were greeted with cheer. Great service, and everything was fresh. We were asked how everything was, like a sit-down restaurant. Recommended.

Votes: 1 funny, 0 useful, 0 cool

4 stars

Please review this business

Name

Please leave your review below

Please provide a rating (5 = best)

5 stars

Submit

Figure C4.2 A review form

This already demonstrates the binding provided by Angular. Note that we did not specify any of the 5 radio 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.

Try it now!

Change the default values provided for the model in *business.component.ts* and see how they are then used as the initial values in the **`<form>`** when it is displayed.

C4.2.3 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.

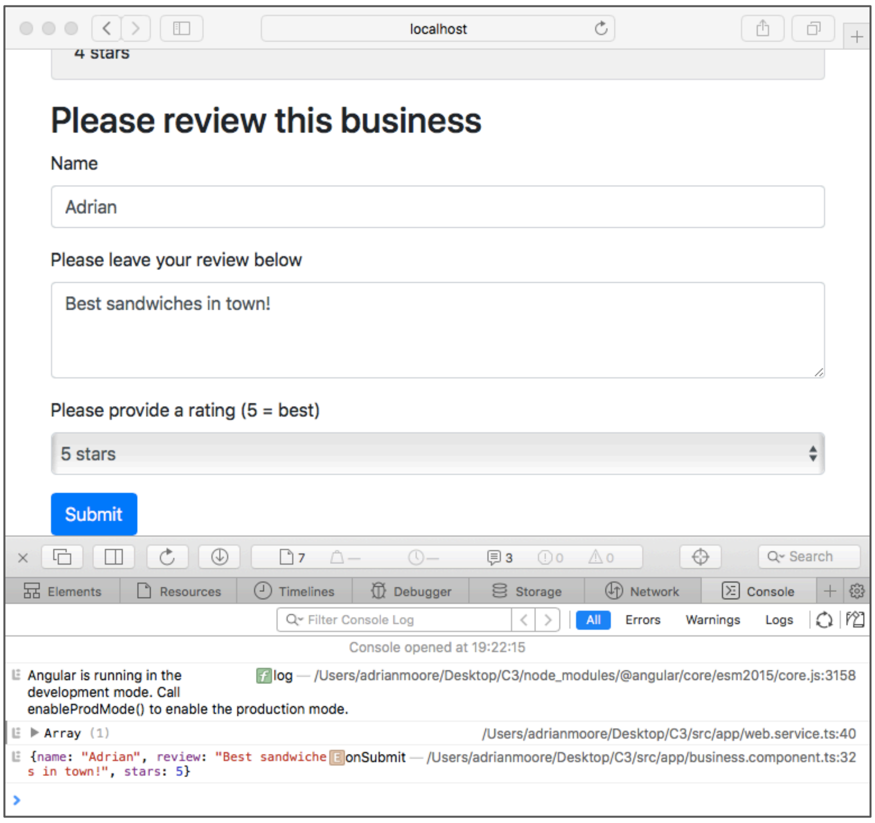
File*: C4/src/app/business.component.html*

```
...

  <h2>Please review this business</h2>
  <form [formGroup]="reviewForm" (ngSubmit)="onSubmit()">

...
```

File*: C4/src/app/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 C4.3 below.



Please review this business

Name

Adrian

Please leave your review below

Best sandwiches in town!

Please provide a rating (5 = best)

5 stars

Submit

```
Console opened at 19:22:15
Angular is running in the          log — /Users/adrianmoore/Desktop/C3/node_modules/@angular/core/esm2015/core.js:3158
development mode. Call
enableProdMode() to enable the production mode.
▶ Array (1)                                    /Users/adrianmoore/Desktop/C3/src/app/web.service.ts:40
{name: "Adrian", review: "Best sandwiche onSubmit — /Users/adrianmoore/Desktop/C3/src/app/business.component.ts:32
s in town!", stars: 5}
>
```

Figure C4.3 Form values submitted

C4.3 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`**.

File: ***C4/src/app/business.component.ts***

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

Next, we specify that we want to ensure that data is provided for the **`name`** and **`review`** fields of the form by adding the **`Validator.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*: C4/src/app/business.component.ts*

```
...

export class BusinessComponent {

  ...
    this.reviewForm = formBuilder.group( {
        name: ['', Validators.required],
        review: ['', 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 *name* or *review* fields left blank.

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*: C4/src/app/business.component.css*

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

We also need to update the **`@Component`** specification in the **`BusinessComponent`** TypeScript file to import the new CSS file.

File*: C4/src/app/business.component.ts*

```
...

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

...
```

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*: C4/src/app/business.component.html*

```
...

  <input type="text"  id="name"
         class="form-control"
         formControlName="name"
         [ngClass]="{ 'error':
                      reviewForm.controls.name.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*: C4/src/app/business.component.html*

```
...

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

...
```

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

File*: C4/src/app/business.component.html*

```
...

  <textarea id="review" rows="3" name="review"
         class="form-control"
         formControlName="review"
         [ngClass]="{ 'error':
                      reviewForm.controls.name.invalid  &&
                      reviewForm.controls.name.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*: C4/src/app/business.component.ts*

```
...

export class BusinessComponent {

  ...

  isInvalid(control) {
    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*: C4/src/app/business.component.html*

```
...

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

...

  <textarea class="form-control" id="review" rows="3"
         formControlName="review"
         name="review"
         [ngClass]="{'error': isInvalid('review')}" >
  </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 a new function **`isIncomplete()`** that returns **`true`** if either text input is invalid.

File: ***C4/src/app/business.component.ts***

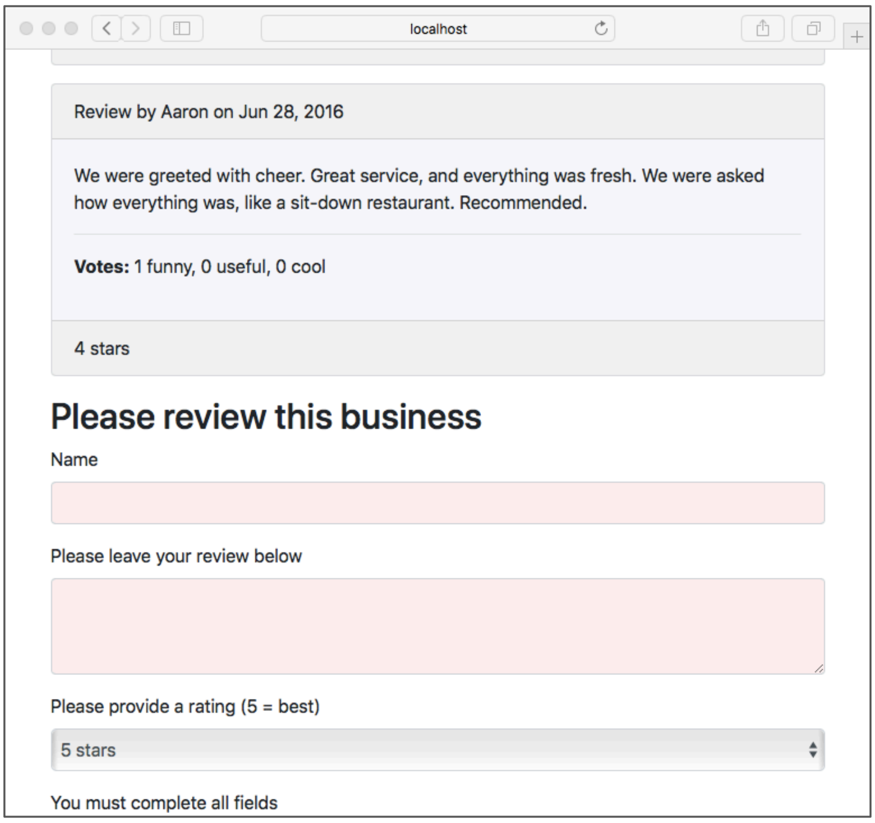
```
...

export class BusinessComponent {

  ...

  isIncomplete() {
    return this.isInvalid('name') ||
           this.isInvalid('review');
  }

...

}
```

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: ***C4/src/app/business.component.html***

```
...

    <span *ngIf="isIncomplete()">
          You must complete all fields</span>

    <button *ngIf="!isIncomplete()" type="submit"
          class="btn btn-primary">Submit</button>

  </form>

...
```

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



Review by Aaron on Jun 28, 2016

We were greeted with cheer. Great service, and everything was fresh. We were asked how everything was, like a sit-down restaurant. Recommended.

Votes: 1 funny, 0 useful, 0 cool

4 stars

Please review this business

Name

Please leave your review below

Please provide a rating (5 = best)

5 stars

You must complete all fields

Figure C4.4 Invalid data and information message

C4.4 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 API.

C4.4.1 Two-way data binding

We have already seen binding between the **`BusinessComponent`**'s **`formBuilder.group()`** specification and the HTML form, but Angular provides a much more extensive two-way binding that allows us to bind the form fields to a data structure that can be used to provide direct access to form field values.

We implement this by first defining a **`review`** object that will be passed to the **`WebService`** to specify the data needed to POST the review to the API.

File: ***C4/src/app/business.component.ts***

```
...

export class BusinessComponent {

  reviewForm;

  review = {
    businessID: '',
    name: '',
    review: '',
    stars: 5
  }

...

}
```

Now we specify that this object is to be tied to the form fields by adding **`ngModel`** binding between each form field and the relevant property in the new object.

File: ***C4/src/app/business.component.html***

```
...

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

  ...

  <textarea class="form-control" id="review" rows="3"
         formControlName="review"
         name="review"
         [ngClass]="{'error': isInvalid('review')}"
         [(ngModel)]="review.review"></textarea>

  ...

  <select class="form-control" id="stars"
         formControlName="stars"
         name="stars"
         [(ngModel)]="review.stars">
```

Note the **[(   )]** syntax in the binding. This specifies that the binding is active in both directions – i.e. if the value entered into the form field changes, then the object property is also automatically changed; **BUT ALSO** if the value of the model is changed programmatically, then the value of the form field is immediately updated to reflect this.

We can check that the binding is in place by amending the **`console.log()`** in the **`onSubmit()`** function to output the value of the new **`review`** object.

File*: C4/src/app/business.component.ts*

```
...

onSubmit() {
    console.log(this.review);
}

...
```

If we run the application and check the message in the browser console, we can see that the username, review text and star rating are available – but the **`_id`** value that identifies the business is not present (as it is not part of the form specification).

There are a number of ways in which we could rectify this (e.g. providing a hidden field for the **`_id`** value and pre-populating it when the form is constructed), but the easiest approach is simply to retrieve the business **`_id`** value from the Web Service.

First, we update the Web Service to include a public **`businessID`** variable that is populated when the **`getBusiness()`** function is called.

File*: C4/src/app/web.service.ts*

```
...

    businessID;

...

    getBusiness(id) {
         this.http.get(
             'http://localhost:3000/api/businesses/' + id)
         .subscribe(
            response => {
              this.business_private_list = [];
              this.business_private_list.push(
                                              response.json());
              this.businessSubject.next(
                                   this.business_private_list);
              this.businessID = id;
            }
        )
    }

...
```

Now that the **`businessID`** is available, we can retrieve it from the **`WebService`** and add it to the **`review`** object before performing the **`console.log()`**.

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*: C4/src/app/business.component.ts*

```
...

onSubmit() {
    this.review.businessID = this.webService.businessID;
    console.log(this.review);
    this.reviewForm.reset();
}

...
```

Testing the application in the browser should verify that all required data is now available in the **`review`** object and that the form is reset once it has been submitted.

C4.4.2 Posting the data to the API

Now that all required information is available, we can create the **`postReview()`** function in the **`WebService`** that will accept a **`review`** object as a parameter and generate the **`http.post()`** request that accesses the API endpoint.

The **`http.post()`** method accepts a URL and a **`URLSearchParams`** object containing the data to be POSTed. As for **`http.get()`**, the **`post()`** method returns an Observable to which we subscribe with a function specifying the action to be taken once the **`post()`** request is complete. As we would like the new review to be automatically added to those displayed, we specify a call to the **`getReviews()`** function that causes the updated collection of reviews to be fetched and displayed.

File: ***C4/src/app/web.service.ts***

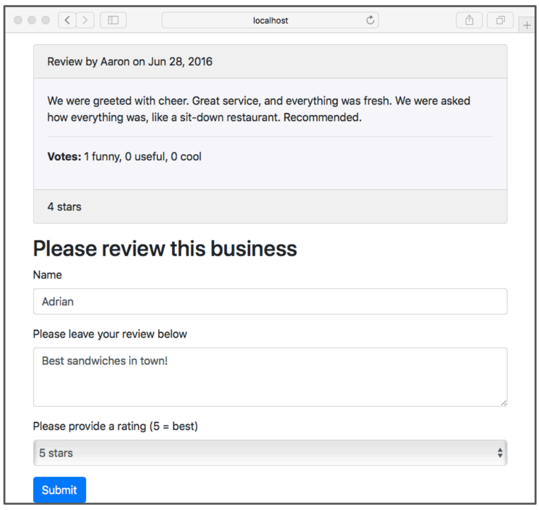
```
import { Http, URLSearchParams } from '@angular/http';

...

   postReview(review) {
      let urlSearchParams = new URLSearchParams();
      urlSearchParams.append('username', review.name);
      urlSearchParams.append('text', review.review);
      urlSearchParams.append('stars', review.stars);

      this.http.post(
            "http://localhost:3000/api/businesses/" +
            review.businessID + "/reviews",
            urlSearchParams)
         .subscribe(
             response => {
                  this.getReviews(review.businessID);
         }
      )
    }

...
```

Finally, we can call the new **`postReview()`** function from the **`onSubmit()`** function within the **`BusinessComponent`**.

File: ***C4/src/app/business.component.ts***

```
...

  onSubmit() {
    this.review.businessID = this.webService.businessID;
    this.webService.postReview(this.review);
    this.reviewForm.reset();
  }

...
```

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



Review by Aaron on Jun 28, 2016

We were greeted with cheer. Great service, and everything was fresh. We were asked how everything was, like a sit-down restaurant. Recommended.

Votes: 1 funny, 0 useful, 0 cool

4 stars

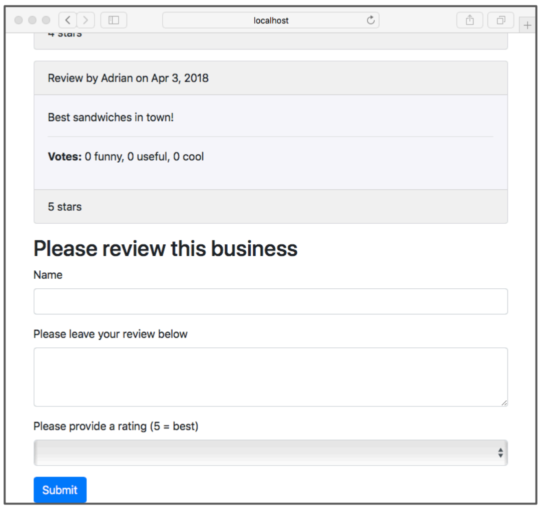
Please review this business

Name: Adrian

Please leave your review below: Best sandwiches in town!

Please provide a rating (5 = best): 5 stars

Figure C4.5 Submitting a review



Review by Adrian on Apr 3, 2018

Best sandwiches in town!

Votes: 0 funny, 0 useful, 0 cool

5 stars

Please review this business

Name

Please leave your review below

Please provide a rating (5 = best)

Figure C4.6 Review accepted