

COM661 Full Stack Strategies and Development

Practical D4: Reactive Forms and HTTP POST

Aims

- 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
- To implement an `http.post()` request

Table of Contents

D4.1 RETRIEVING SUB-DOCUMENTS.....	2
D4.1.1 UPDATE THE WEB SERVICE	2
D4.1.2 UPDATE THE COMPONENT	2
D4.1.3 UPDATE THE TEMPLATE	3
D4.2 FORMS	5
D4.2.1 CREATE THE FORM	5
D4.2.2 USING ANGULAR FORMBUILDER.....	7
D4.2.3 SUBMITTING FORM VALUES.....	9
D4.3 VALIDATION WITH REACTIVE FORMS	10
D4.4 RETRIEVING AND POSTING FORM DATA.....	16
D4.4.1 RETRIEVING THE DATA.....	16
D4.4.2 SENDING THE POST REQUEST	19
D4.5 FURTHER INFORMATION.....	22

D4.1 Retrieving Sub-documents

So far, the **Biz Directory** sample application provides functionality to page through a collection of business elements 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.

D4.1.1 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.

First, 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 simply return as the result of the function.

File: D4/src/app/web.service.ts

```
...

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

D4.1.2 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: D4/src/app/business.component.ts

```
...

    reviews: any = [];

...

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

...
```

D4.1.3 Update the template

Now, we can update the **BusinessComponent** template to subscribe to the Observable containing the 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.

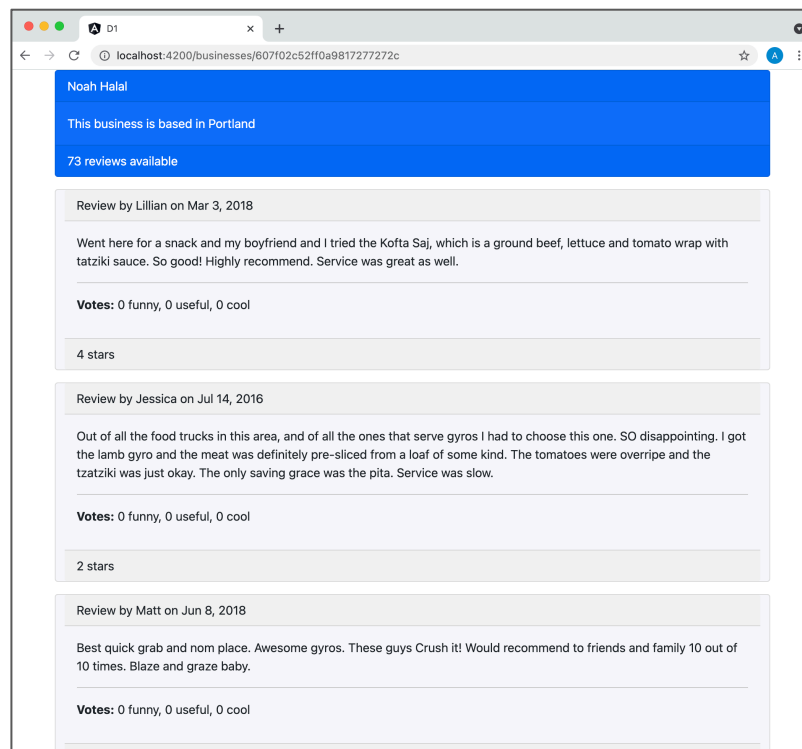


Figure D4.1 Displaying reviews

File: D4/src/app/business.component.html

```
...

<div class="container">
  <div class="row">
    <div class="col-sm-12">
      <div class="card bg-light mb-3"
        style = "width: 30rem; margin:auto"
        *ngFor = "let review of reviews | async">
        <div class="card-header">
          Review by {{ review.name }}
            on {{ review.date.substr(0,10) | date }}
        </div>
        <div class="card-body">
          {{ review.text }}
          <hr>
          <p><strong>Votes:</strong>
            {{ review.funny }} funny,
            {{ review.useful }} useful,
            {{ review.cool }} cool </p>
        </div>
        <div class="card-footer">
          {{ review.stars }} stars
        </div>
      </div>
    </div>
  </div>
</div>
```

Note how we extract the first 10 characters of the date value (which will be in the form “YYYY-MM-DD”) so that we can apply it to the Angular **date** pipe to obtain a meaningful textual representation of the date. Checking in the browser reveals that the collection of reviews for the business are displayed as intended, as illustrated by Figure D4.1, above.

Do it now!	Update the Web Service, Component and Template as shown above so that the collection of reviews is presented below the details of a single business
-------------------	---

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.
--------------	--

D4.2 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 **ReactiveFormsModule** class in the application's main *app.module.ts* file.

File: D4/src/app/app.module.ts

```
...

import { BusinessComponent } from './business.component';
import { ReactiveFormsModule } from '@angular/forms';

...

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

...
```

D4.2.1 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 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: D4/src/app/business.component.html

```
...

<div class="container">
  <div class="row">
    <div class="col-sm-12">

      <h2>Please review this business</h2>
      <form>
        <div class="form-group">
          <label for="name">Name</label>
          <input type="text" id="name" name="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>
  </div>
</div>
```

D4.2.2 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 **formControlName** properties in the template.

File: D4/src/app/business.component.ts

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

...

export class BusinessComponent {

    reviewForm: any;

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

    ngOnInit() {

        this.reviewForm = this.formBuilder.group({
            name: '',
            review: '',
            stars: 5
        });

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

    ...
}
```

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: D4/src/app/business.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 D4.2 below.

The screenshot shows a web browser window with the address bar displaying 'localhost:4200/businesses/607f02c52ff0a981727266d'. The page content includes a review by 'Yanni' on 'Dec 28, 2011' with a text description and a 5-star rating. Below this, there is a section titled 'Please review this business' with a form containing a 'Name' field, a text area for the review, a 5-star rating selector, and a 'Submit' button.

Figure D4.2 A review form

This already demonstrates 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!	Implement the review form as described in the sections above. Ensure that you see the form below the reviews for a business as shown in Figure D4.2.
-------------------	--

Try it now!	Change the default values provided for the formBuilder.group definition in business.component.ts and see how they are then used as the initial values in the <form> when it is displayed.
--------------------	--

D4.2.3 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: D4/src/app/business.component.html

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

File: D4/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 D4.3 below.

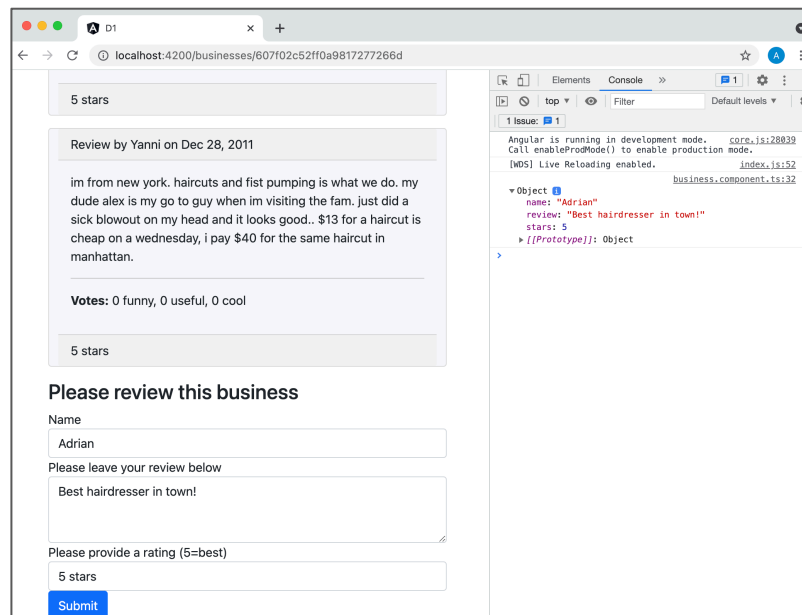


Figure D4.3 Form values submitted

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

D4.3 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: D4/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 **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: D4/src/app/business.component.ts

```
...

export class BusinessComponent {

  ...

  ngOnInit() {
    this.reviewForm = this.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.

Do it now!	Add the Validators element to the Business Component TypeScript file as shown above. Make sure that the message 'false' is seen in the Browser Console when you submit a review with an empty field.
-------------------	---

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: D4/src/app/business.component.css

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

We also need to check the **@Component** specification in the **BusinessComponent** TypeScript file to make sure the CSS file is being imported.

File: D4/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: D4/src/app/business.component.html

```
...

<input type="text" id="name" name="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: D4/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: D4/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: D4/src/app/business.component.ts

```
...  
  
export class BusinessComponent {  
  
    ...  
  
    isValid(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: D4/src/app/business.component.html

```
...

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

...

<textarea class="form-control" id="review" rows="3"
          FormControlName="review"
          name="review"
          [ngClass]="{'error': isValid('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 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: D4/src/app/business.component.ts

```
...

export class BusinessComponent {

  ...

  isUntouched() {
    return this.reviewForm.controls.name.pristine ||
           this.reviewForm.controls.review.pristine;
  }

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

}
```

Now, we apply this to a new **** 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: D4/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.

The screenshot shows a web browser window with the URL `localhost:4200/businesses/607f02c52f0a9817277266d`. The page displays a review for a business. At the top, it shows 'Votes: 0 funny, 0 useful, 0 cool' and a '5 stars' rating bar. Below this is a review by 'Yanni' on 'Dec 28, 2011' with the text: 'im from new york. haircuts and fist pumping is what we do. my dude alex is my go to guy when im visiting the fam. just did a sick blowout on my head and it looks good.. \$13 for a haircut is cheap on a wednesday, i pay \$40 for the same haircut in manhattan.' This review also has 'Votes: 0 funny, 0 useful, 0 cool' and a '5 stars' rating bar. Below the review is a section titled 'Please review this business'. It contains a 'Name' input field, a 'Please leave your review below' text area, and a 'Please provide a rating (5=best)' dropdown menu currently set to '5 stars'. At the bottom, a message states 'You must complete all fields'.

Figure D4.4 Invalid data and information message

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

D4.4 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.

D4.4.1 Retrieving the data

We have already seen in Section D4.2.3 how the name, review and stars values entered by the user on the form are available in the `reviewForm.value` object of the Business Controller. For clarity, we can repeat that check by modifying the `onSubmit()` method of `business.component.ts` as shown below.

```
File: D4/src/app/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 Web Service method to make the POST request to the API and amend **onSubmit()** to call the new 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: D4/src/app/business.component.ts

```
...

onSubmit() {
    this.webService.postReview(this.reviewForm.value);
    this.reviewForm.reset();
}

...
```

File: D4/src/app/web.service.ts

```
...

postReview(review: any) {}

...
```

Do it now!	Prepare to submit the values to the back end by updating business.component.ts and web.service.ts as shown in the code boxes above.
-------------------	---

We have seen from the Browser console that the **review** object passed to **postReview()** will contain the name, review and stars fields, so we can receive these from the review object and add them as parameters to a new **FormData** object which is used to pass data with the HTTP request. Each parameter is added by the **append()** method of the **FormData** object which takes 2 parameters – the name of the parameter to the HTTP request and the value to be provided. As our API expects the parameters of the POST request to be “*name*”, “*text*” and “*stars*”, we specify these against the fields of the review object.

File: D4/src/app/web.service.ts

```
...

postReview(review: any) {
  let postData = new FormData();
  postData.append("name", review.name);
  postData.append("text", review.review);
  postData.append("stars", review.stars);
}
}
```

Another compulsory field of the review is the date on which it was left, but rather than have this entered by the user, we can obtain it in our desired form of YYYY-MM-YY by using the JavaScript/TypeScript **Date()** methods and append the value to the parameter list as highlighted in the following code box.

File: D4/src/app/web.service.ts

```
...

postReview(review) {
  let postData = new FormData();
  postData.append("name", review.name);
  postData.append("text", review.review);
  postData.append("stars", review.stars);

  let today = new Date();
  let todayDate = today.getFullYear() + "-" +
                  today.getMonth() + "-" +
                  today.getDate();
  postData.append("date", todayDate);
}
}
```

Next, we need to obtain the **_id** value of the business for which we are leaving a review by assigning it to a local variable when the **getBusiness()** function is called.

File: D4/src/app/web.service.ts

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

Do it now!	In <i>web.service.ts</i> , add code to postReview() and getBusiness() as shown in the code boxes above.
-------------------	---

D4.4.2 Sending the POST request

Now that the **businessID** is available, we can retrieve the value and use it in the URL to which we send the POST request.

Note that the **http.post()** method takes two parameters, the URL to which we are sending the request and the **FormData** object (**postData**) containing the parameters of the request.

Note:	Remember that the API endpoint to add a new review expects a JSON Web Token to be provide in an 'x-access-token' header. In order to submit this review at this stage, you should edit your back-end <i>app.py</i> file and remove the @jwt_required decorator from the POST request route to add a new review.
--------------	--

File: D4/src/app/web.service.ts

```
...

postReview(review: any) {
  console.log(review);
  console.log(this.businessID);
  let postData = new FormData();
  postData.append("name", review.name);
  postData.append("text", review.review);
  postData.append("stars", review.stars);

  let today = new Date();
  let todayDate = today.getFullYear() + "-" +
    today.getMonth() + "-" +
    today.getDate();
  postData.append("date", todayDate);

  return this.http.post(
    http://localhost:5000/api/v1.0/businesses/' +
    this.businessID + '/reviews', postData);
}
```

Finally, we want to refresh the list of reviews so that the newly submitted one is shown. We do this by subscribing to the Observable that is returned by **http.post()** and passed back as the result of the **postReview()** function. When the review submission is complete, we can then retrieve the updated list of reviews by a call to the Web Service **getReviews()** function. Since the **async** pipe in the Business Component template subscribes to the **reviews** list that is modified by this call, the display is automatically updated.

File: D4/src/app/business.component.ts

```
...

onSubmit() {
  this.webService.postReview(this.reviewForm.value)
    .subscribe((response: any) => {
      this.reviewForm.reset();
      this.reviews = this.webService.getReviews(
        this.route.snapshot.params['id']);
    });
}

...
```

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

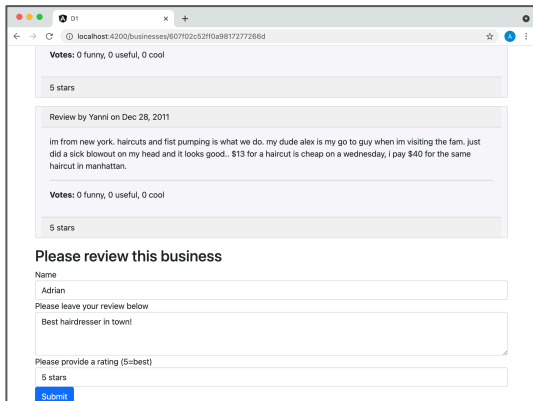


Figure D4.5 Submitting a review

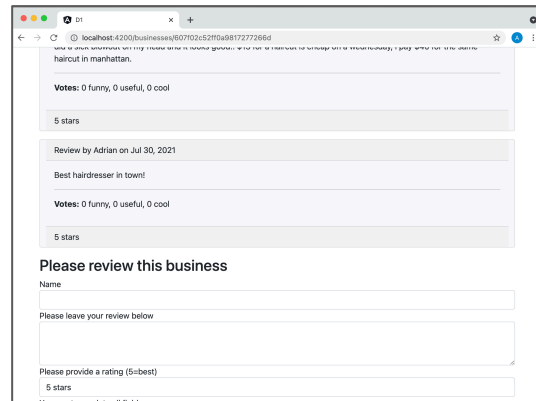


Figure D4.6 Review accepted

Note: This example assumes that each review contains all of the review elements being retrieved and displayed. You will need to make sure that your back-end code to add a new review specifies all elements – including the **date** and **funny/useful/cool** votes that we did not originally implement. We also provide a valid **user_id** as a hard-coded value taken from the **users** collection. Hence, in your back-end program **app.py**, the **new_review** object within **add_new_review()** should be specified as shown below.

```
new_review = {
    "_id" : ObjectId(),
    "user_id" : "6091219c0cb8c687647f180a",
    "name" : request.form["name"],
    "text" : request.form["text"],
    "stars" : request.form["stars"],
    "date" : request.form["date"],
    "funny" : 0,
    "useful" : 0,
    "cool" : 0
}
```

Do it now! Complete **postReview()** by adding the call to **http.post()** as shown above. Try submitting a review and make sure that the review is added to the collection and that the review form is cleared as shown in Figures D4.5 and D4.6 above.

D4.5 Further Information

- <https://coreui.io/docs/components/forms/>
Bootstrap Forms
- https://www.w3schools.com/Bootstrap/bootstrap_forms.asp
Bootstrap Forms – W3Schools
- <https://angular.io/api/common/DatePipe>
Angular DatePipe
- <https://angular.io/guide/reactive-forms>
Angular ReactiveForms – the online manual
- <https://malcoded.com/posts/angular-fundamentals-reactive-forms/>
Reactive Forms with Angular – using easy examples!
- <https://angular.io/api/forms/FormBuilder>
Angular FormBuilder class
- <https://coryryan.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.io/guide/form-validation>
Angular Form Validation
- <https://www.positronx.io/how-to-use-angular-8-httpclient-to-post-formdata/>
How to use the Angular 12 HttpClient API to Post FormData
- <https://www.digitalocean.com/community/tutorials/understanding-date-and-time-in-javascript>
Understanding Time and Date in JavaScript