Class 19 - Handling Forms in Angular Apps

Class 19 Course Content

Lesson Outline

Today we will learn:

1. Two ways we can handle forms in Angular Applications.
2. The differences between both approaches.
3. How to implement a Template Driven Approach.
4. How to implement a Reactive Driven Approach.

Lesson Notes

- **Template Driven Approach:** The *template driven approach* to forms is the more intuitive and simpler way to implement form functionality in Angular. This method infers the structure of the "form" object from the DOM.
- **Reactive Driven Approach:** The *reactive driven approach* to forms is more explicit. This way of handling forms is more complicated, but, you have gain more fine-tuned control and flexibility.

Angular Forms Course Project Steps - TEMPLATE DRIVEN

STEP 1: Registering the Controls

Terminal:

- Create a new component called "BookFormTDComponent" inside the "bookshelf" folder.

```
ng g c book-form-td
```

bookshelf/bookshelf-editor/bookshelf-editor.component.ts:

- Place the `<book-form-td>` tag as the only content in this file.

bookshelf/book-form-td/book-form-td.html file:

- *Note*: Make sure you have added the "FormsModule" inside the "imports" array in *app.module.ts* file.
- *Note*: All inputs need an "id" attribute and a "name" attribute.
- On all the inputs you want to be submitted, add `ngModel` as an attribute.

```

```

```
<!-- ~~~ !TEMPLATE DRIVEN APPROACH! ~~~ -->
<form>
  <div id="book-data">
    <!-- Title -->
    <div class="form-group">
      <label for="title">Title</label>
      <input type="text" id="title" name="title" class="form-control"
ngModel />
    </div>

    <!-- Author -->
    <div class="form-group">
      <label for="author">Author</label>
      <input
        type="text"
        id="author"
        name="author"
        class="form-control"
        ngModel
      />
    </div>

    <!-- Genre -->
    <div class="form-group">
      <label for="genre">Genre</label>
      <select id="genre" name="genre" class="form-control" ngModel>
        <option value="mystery">Mystery</option>
        <option value="self-help">Self-Help</option>
        <option value="fantasy">Fantasy</option>
        <option value="science-fiction">Science Fiction</option>
      </select>
    </div>

    <!-- isBestSeller -->
    <div class="form-group">
      <p>Is this book a best seller?</p>

      <div class="flex-col">
        <!-- True -->
        <div>
          <input
            type="radio"
            id="true"
            name="isBestSeller"
            value="true"
            ngModel
          />
          <label for="true" class="ml-3">True</label>
        </div>

        <!-- False -->
        <div>
          <input
```

```
              type="radio"
              id="false"
              name="isBestSeller"
              value="false"
              ngModel
            />
            <label for="false" class="ml-3">False</label>
          </div>
        </div>
      </div>

      <!-- Submit Button -->
      <button class="btn btn-primary" type="submit">Submit</button>
    </div>
  </form>
```

STEP 2: Submitting the User Form

book-form-template/book-form-template.html file:

- Inside the "form" element tag, listen to the `(ngSubmit)` event that calls an `onFormSubmit()` function that we will create next.

```
<form (ngSubmit)="onFormSubmit()">
  <!-- . . . -->
</form>
```

book-form-template/book-form-template.ts file:

- Create a new function called `onFormSubmit()`.
- Create a `console.log()` inside the form that outputs the text "Submitted!" to the console.

```
onFormSubmit() {
    console.log('Submitted!');
}
```

book-form-template/book-form-template.html file:

- Now inside our html "form" tag. . . We want to add a local reference variable and set it equal to "ngForm".
- Pass this local refference through the function call on `onFormSubmit(templateFormRef)`.

```
<form (ngSubmit)="onFormSubmit(templateFormRef)"
#templateFormRef="ngForm">
```

```
  <!-- . . . -->
</form>
```

book-form-template/book-form-template.ts file:

- The `onFormSubmit(formObj: NgForm)` Now needs to take in a "formObj" as an argument. This argument should be of type "NgForm".

```
onFormSubmit(formObj: NgForm) {
    console.log('Submitted!', formObj);
}
```

STEP 3: Validating the Inputs

book-form-td/book-form-td.html file:

- Add "required" as an attribute for all fields you definitely want to have every time. (We will place it on the "title", "author", and "genre" inputs.)
- Add a span tag below each input that displays an appropriate error message. This should only be shown if the input is invalid and has been touched. We do this by adding an element referrence on each element tag and set it equal to "ngModel".
- On the submit button, bind to the `[disabled]` property and set it to be disabled if our form isn't valid.

```
<!-- . . . -->
<input name="title" #titleRef="ngModel" ngModel required />
<span class="help-block text-danger" *ngIf="!titleRef.valid &&
titleRef.touched"
  >Please enter a valid title!</span
>
<!--  . . . -->
<input name="author" #emailRef="ngModel" ngModel required />
<span class="help-block text-danger" *ngIf="!emailRef.valid &&
emailRef.touched"
  >Please enter a valid email!</span
>
<!--  . . . -->
<select name="genre" #genreRef="ngModel" ngModel required></select>
<span class="help-block text-danger" *ngIf="!genreRef.valid &&
genreRef.touched"
  >Please enter a valid genre!</span
>
<!-- . . . -->
<button type="submit" [disabled]="!templateFormRef.valid">Submit</button>
```

book-form-td/book-form-td.css file:

- Create a class selector called `form .ng-invalid.ng-touched` that sets the "border" property to be "1px solid red". (This will put a red border on any element inside a form that is invalid and that the user has touched.)

```
form .ng-invalid.ng-touched {
  border: 1px solid red;
}
```

STEP 4: Default Values & Two-way-databinding

book-form-td/book-form-td.html file:

- For the select element inside our form, bind to the `[ngModel]` property and set it equal ot the string "'mystery'". (This will set the default option of this select to the option with a value of 'mystery'.)
- On the title input, use two-way-binding on the `[(ngModel)]` property and set it equal to a new variable "bookTitle".
- On the submit button, output the bookTitle variable after the word "submit".
-

```
<input
  type="text"
  id="title"
  name="title"
  class="form-control"
  #titleRef="ngModel"
  ngModel="title"
  required
/>
<!-- . . . -->
<select
  id="genre"
  name="genre"
  class="form-control"
  #genreRef="ngModel"
  ngModel="'mystery'"
  required
></select>
<!-- . . . -->
<button
  class="btn btn-primary"
  type="submit"
  [disabled]="!templateFormRef.valid"
>
```

```
  Submit
</button>
```

STEP 5: Displaying our Data && Resetting the Form

book-form-template/book-form-template.html file:

- Create a new row with a column taking up the full-width. Inside the column, add a heading or paragraph tag to display data for all of the fields on the form. (We will create a new object in our typescript that contains our values.)
- Add an `*ngIf` conditional to the row so it only displays when the form has been submitted.

```
<!-- RESULTS -->
<div class="row mt-4" *ngIf="formHasBeenSubmitted">
  <div class="col-xs-12">
    <h3>Title: {{ bookDetails.title }}</h3>
    <h4>Author: {{ bookDetails.author }}</h4>
    <p>Genre: {{ bookDetails.genre }}</p>
  </div>
</div>
```

book-form-template/book-form-template.ts file:

- Create a new object variable that contains all the required values on our form. Set them to be empty values.
- Create a boolean variable "formHasBeenSubmitted" and set it to be false. When we call the "onFormSubmit" function, set this variable to true.
- In the "onFormSubmit" function, set all the values on our form to the values in the new object we just created.
- Reset our form at the end of the "onFormSubmit" function.

```
export class BookFormTemplateComponent {
  formHasBeenSubmitted = false;
  bookDetails = {
    title: "",
    author: "",
    genre: "",
  };

  onFormSubmit(formObj: NgForm) {
    // console.log('Submitted!', formObj);

    // Update boolean variable to true
    this.formHasBeenSubmitted = true;
```

```
    // Set Local "bookDetails" object to the values on the form inputs
    this.bookDetails.title = formObj.value.title;
    this.bookDetails.author = formObj.value.author;
    this.bookDetails.genre = formObj.value.genre;
    this.formHasBeenSubmitted = true;

    // Reset the form
    formObj.reset();
  }
}
```

Angular Forms Project Steps - REACTIVE DRIVEN

STEP 1: Initializing the form

Note: Make sure you have added the "ReactiveFormsModule" inside the "imports" array in *app.module.ts* file.

book-form-reactive/book-form-reactive.ts file:

- Create a new variable "reactiveForm" and set it to type "FormGroup".
- Inside `ngOnInit()`, set your "reactiveForm" variable equal to a `new FormGroup({})`.
- Add all of your input names with default values using the `new FormControl()`.
- *Note*: Make sure you are importing `{ FormGroup, FormControl }` from "@angular/forms".

```
export class BookFormReactiveComponent implements OnInit {
  reactiveForm: FormGroup;

  constructor() {}

  ngOnInit(): void {
    this.reactiveForm = new FormGroup({
      title: new FormControl(null),
      author: new FormControl(null),
      genre: new FormControl("mystery"),
      isBestSeller: new FormControl(null),
    });
  }
}
```

STEP 2: Syncing the Form in Typescript to HTML

book-form-reactive/book-form-reactive.html file:

- Add the `[formGroup]` directive and pass our form variable as a referrence.
- Tell Angular which form inputs should be set to the typescript form values by adding the "formControlName" properties.

```
<!-- ~~~ !REACTIVE DRIVEN APPROACH! ~~~ -->
<form [formGroup]="reactiveForm">
  <div id="book-data">
    <!-- Title -->
    <div class="form-group">
      <label for="title">Title</label>
      <input
        type="text"
        id="title"
        name="title"
        class="form-control"
        formControlName="title"
      />
    </div>

    <!-- Author -->
    <div class="form-group">
      <label for="author">Author</label>
      <input
        type="text"
        id="author"
        name="author"
        class="form-control"
        formControlName="author"
      />
    </div>

    <!-- Genre -->
    <div class="form-group">
      <label for="genre">Genre</label>
      <select
        id="genre"
        name="genre"
        class="form-control"
        formControlName="genre"
      >
        <option value="mystery">Mystery</option>
        <option value="self-help">Self-Help</option>
        <option value="fantasy">Fantasy</option>
        <option value="science-fiction">Science Fiction</option>
      </select>
    </div>

    <!-- isBestSeller -->
    <div class="form-group">
      <p>Is this book a best seller?</p>
```

```
      <!-- True -->
      <input
        type="radio"
        id="true"
        name="isBestSeller"
        value="true"
        formControlName="isBestSeller"
      />
      <label for="true" class="mr-3">True</label>

      <!-- False -->
      <input
        type="radio"
        id="false"
        name="isBestSeller"
        value="false"
        formControlName="isBestSeller"
      />
      <label for="false">False</label>
    </div>

    <!-- Submit Button -->
    <button class="btn btn-warning" type="submit">Submit</button>
  </div>
</form>
```

STEP 3: Submitting the Form

book-form-reactive/book-form-reactive.html file:

- Place the `(ngSubmit)` event binding and set the value equal to a function `onFormSubmit()`

```
<form [formGroup]="reactiveForm" (ngSubmit)="onFormSubmit()">
  <!-- . . . -->
</form>
```

book-form-reactive/book-form-reactive.ts file:

- Create the `onFormSubmit()` function and log to the console our "reactiveForm" variable.

```
onFormSubmit() {
    console.log('Submitted!', this.reactiveForm);
}
```

STEP 4: Adding Validation

book-form-reactive/book-form-reactive.ts file:

- Inside the first three `FormControl()`'s, pass in a second argument... the "Validators.required" argument.

```
ngOnInit(): void {
    this.reactiveForm = new FormGroup({
        title: new FormControl(null, Validators.required),
        author: new FormControl(null, Validators.required),
        genre: new FormControl('mystery', Validators.required),
        isBestSeller: new FormControl(null)
    });
}
```

book-form-reactive/book-form-reactive.html file:

- Add a span tag below each input to show the error message.
- Add the conditional `*ngIf` statements to check if the input is valid and has been touched.

```
<input formControlName="title" />
<span
  class="help-block text-danger"
  *ngIf="
          !reactiveForm.get('title').valid &&
reactiveForm.get('title').touched
        "
  >Please enter a valid title!</span
>
<!-- . . . -->
<input formControlName="author" />
<span
  class="help-block text-danger"
  *ngIf="
          !reactiveForm.get('author').valid &&
          reactiveForm.get('author').touched
        "
  >Please enter a valid author!</span
>
<!-- . . . -->
<select formControlName="genre"></select>
<span
  class="help-block text-danger"
  *ngIf="
          !reactiveForm.get('genre').valid &&
reactiveForm.get('genre').touched
        "
  >Please enter a valid genre!</span
>
```

book-form-reactive/book-form-reactive.css file:

- Add the same CSS selector to display a red border when the input is invalid and touched.

```
form .ng-invalid.ng-touched {
  border: 1px solid red;
}
```

STEP 5: Displaying the Data && Resetting the Form

book-form-reactive/book-form-reactive.html file:

- Create a new row with a column taking up the full-width. Inside the column, add a heading or paragraph tag to display data for all of the fields on the form. (Use the `reactiveForm.value` properties to display the dynamic data.)
- Add an `*ngIf` conditional to the row so it only displays when the form has been submitted.

```
<!-- RESULTS -->
<div class="row mt-4" *ngIf="formHasBeenSubmitted">
  <div class="col-xs-12">
    <h3>Title: {{ reactiveForm.value.title }}</h3>
    <h4>Author: {{ reactiveForm.value.author }}</h4>
    <p>Genre: {{ reactiveForm.value.genre }}</p>
  </div>
</div>
```

book-form-reactive/book-form-reactive.ts file:

- Create a boolean variable "formHasBeenSubmitted" and set it to be false. When we call the "onFormSubmit" function, set this variable to true.
- In the "onFormSubmit" function, set all the values on our form to the values in the new object we just created.
- Reset our form at the end of the "onFormSubmit" function.

```
export class BookFormReactiveComponent {
  reactiveForm: FormGroup;
  formHasBeenSubmitted = false;

  onFormSubmit() {
    // console.log('Submitted!', this.reactiveForm);

    // Change boolean form submitted variable to true
    this.formHasBeenSubmitted = true;
```

```
      // Reset Form after 5 seconds
      setTimeout(() => {
        this.reactiveForm.reset();
        this.formHasBeenSubmitted = false;
      }, 5000);
    }
  }
```

Additional Notes

Class Exercise

1. Download staring code for class 10 Angular Example
2. In the *app.component.html* file, create a container div with a row inside that renders both of our components side-by-side.
3. Inside the "BookFormTemplateComponent": Create a template-driven form that, when submitted, displays all the relevant data below the form.
    - Title, Author, and Genre should all be required inputs that display an error message with a red border when touched and invalid.
    - Set 'mystery' to be the default value on the genre select box.
    - Use two-way-binding to display the name of the book after the word "Submit" on the submit button.
    - Display the title, author, and genre below the form when a user clicks "Submit". (Also, reset the form.)
4. Inside the "BookFormReactiveComponent": Create a reactive-driven form that, when submitted, displays all the relevant data below the form.
    - Title, Author, and Genre should all be required inputs that display an error message with a red border when touched and invalid.
    - Set 'mystery' to be the default value on the genre select box.
    - Display the title, author, and genre below the form when a user clicks "Submit". (Also, reset the form.)
5. Publish to Github. Inside the README.md file, write your experience of the different approaches.
    - Define the "Template-Driven-Approach" and the "Reactive-Driven-Approach".
    - Explain their differences and what each excel at doing.
    - Write about what you find easy and what is challenging about each.

Form Approaches Overview

Template Driven Approach

- *Note*: Make sure you import the `Forms Module` and add it to your NgModule's imports array. When Angular detects a form element, it creates a JavaScript object representation of the form. However, inputs are not automatically detected.

Submitting the Form:

- Add `ngModel` attribute to inputs you would like to add as a form control. Add a "name" attribute to register a control to the JavaScript object representation of our form.
- To access the JavaScript representation of this form, create a local reference of `ngForm` on the form element. Add `(ngSubmit)` directive to the form element as well, and call a custom method (created by you in the component), passing the local reference to the form.

Accessing the Form:

- Add the submit method you called with ngSubmit to your *.component.ts* file, taking in the form as a parameter.
- Alternatively, you can use `@ViewChild()` and assign the form to a variable in your component.
- Explore properties of the form object by logging to the console.
- We can utilize the form state with property binding and conditional rendering to disable the sumbit button, or display a message if the form is invalid.
- You can use built in and custom validators. Here is a list of built in validators.
- Angular dynamically adds CSS classes based on the state of the form.
- Utilize Two-Way-Binding with `ngModel` to react to or output a user's input instantly.
- You can use `ngModelGroup` to group form controls.
- To use radio buttons, create a div with `ngFor` looping through the options, and render an input with a value set to the option. Use one way binding with ngModel if you want a default option selected.

Form Values:

- NgForm's `setValue()` method can be used to prepopulate form inputs. However, it will overwrite data entered by the user.
- To only set parts of the form, use the `patchValue()` method. It is available on the form object that is wrapped by NgForm.
- Extract the form data in your compoenent by accessing the form value, followed by the name you used for that input in the HTML template.
- To reset the form, simply use the reset method on the your form variable.

Reactive Driven Approach

- *Note*: Make sure you import the `Reactive Forms Module` and add it to the imports array in `app.module.ts`.

Import "FormGroup" and "FormControl" from "@angular/forms" into the *.component.ts* file where you will create the form.

Creating the Form:

- Create a variable set to a new `FormGroup` with an object with control names(key) labeled with quotes, and set to a new `FormConrtol()`. You can initialize with null or data you want to prepopulate the field, as well as validators.
- Sync the form to the HTML by binding `formGroup` to the variable you created, and add the `formControlName` attribute to the corresponding input elements with the name you create the form with.

Submitting the Form:

- We cannot and don't need to use a local reference to access this form. Call your submit method with `ngSubmit`, and access the FormGroup variable you created.

Adding Validation:

- When you initialize your FormGroup object, you can add validators to your FormControls by importing the Validators object from '@angular/forms'.

Accessing Controls in the HTML:

- Access controls by passing the control name (as a string) to FormGroup's get method.
- You can access properties and state for conditional rendering.

Grouping Controls:

- You can nest FormGroups, but you must structure your HTML page the same way.
- Access grouped controls as a JavaScript object.
- FormArray holds an array of conrols, allowing users to enter mulitple inputs.

Custom Validation:

- We can create our own validators by creating a function to check if our control meets a condition, returning nothing or null if the control is valid.
- If your validator fucntion references `this` You must `bind(this)` when adding to FormControls, to avoid problems with scope.

Error Codes:

- Angular adds error codes (returned by you in custom function) on the individual control. This is beneficial to provide users with detailed error information.

Async Validators:

- Asynchronous validators provide a pending state when your app needs to wait for a server response to determine the validity of user input.
- Asynchronous validators return a "Promise" or an "Observable".
- State of control is pending while awaiting a response.

Status and Value Changes:

- You can `subscribe` to the "statusChanges" and "valueChanges" observables provided by FormGroup.

Setting, Patching and Resetting Values:

- As with the TD approach, we can access the `setValue()`, `patchValue()`, and `reset()` methods. However, the FormGroup object is not wrapped in this instance, and we can access these methods directly.

Resources

- Angular Resources - Built in HTML Validators
- Angular Docs - Introduction to Forms
- Angular Guide - Reactive Forms