Exercise 9 - Creating Angular Reactive Forms

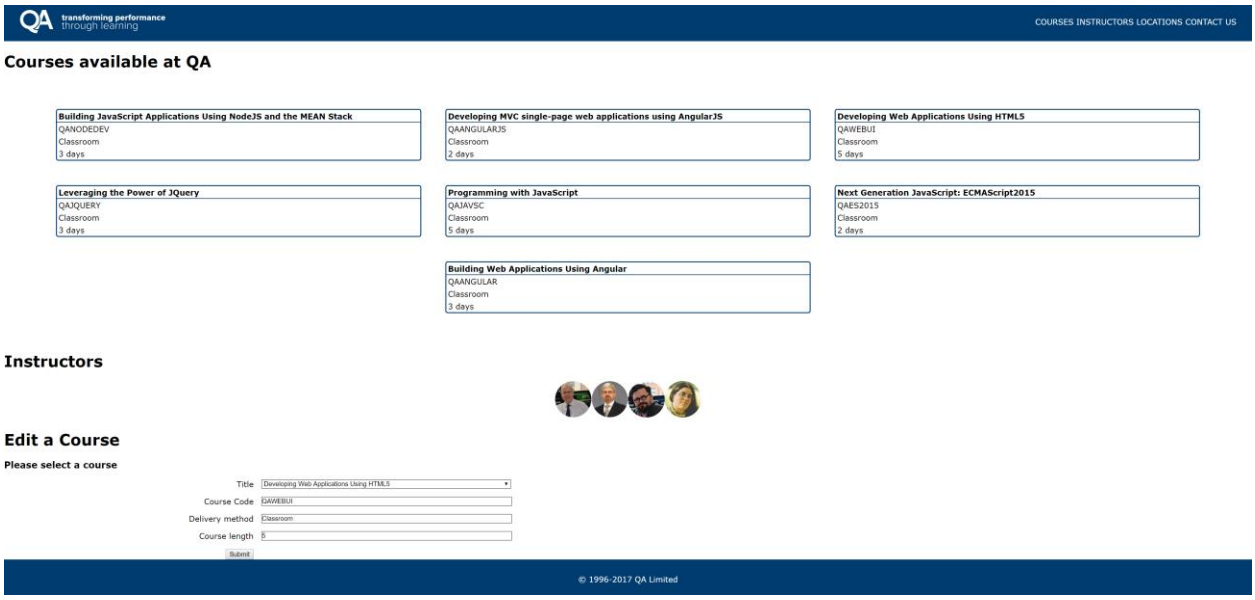
Objective

To create a new component within our Angular application that is able to receive input from the user through the use of a reactive form.

Overview

In this exercise, you are going to modify a form which will contain very little "Angular". We will then furnish it with various Angular directives to provide feedback to the user as well as restrict their data entry to fit with our requirements. The end result should look something like this. If you're having déjà vu here it's because this exercise produces the same result as the Template Forms exercise, just as a Reactive Form instead!

Running the application (after an **npm install**, if required) should produce an application that runs without errors, the form allows you to select different courses from the list but no other functionality for it is yet available.



Instructions

Part 1 – Editing the course-editor component file

To avoid duplication of creating components, you will find that the component and the form are already created. They do not, however, contain any of the code needed to create a reactive form. In this part, we will add the imports needed for creating the reactive form.

1. In **src/app/course/course-editor**, open **course-editor.component.ts**

2. Add **imports** for `FormBuilder` and `FormGroup` from `@angular/forms`
3. In the class itself, add the following declaration:

    - `courseForm` of type `FormGroup`

Modifying the constructor

4. In the `constructor`, **inject** a `private fb` as a `FormBuilder`
5. Call, an as yet uncreated function called `createForm()`

The createForm() method

6. After the constructor declare a class method called `createForm()`
7. Assign `courseForm` to a `new FormBuilder` Group with the following attributes in the supplied object argument:

    - `title` set to an empty string
    - `code` set to an empty string
    - `delivery` set to an empty string
    - `days` set to `null`

Modifying the template

8. In **course-editor.component.html** add the **attribute** `[formGroup]` to the `<form>` tag and set it to `courseForm`
9. For each **input** (including the select group) add a `formControlName` attribute set to *whatever value it represents* in the component

Adding ReactiveFormsModule to the Courses Module

10. For the Reactive directives to work, the application needs to know about the `ReactiveFormsModule` from `@angular/forms`, so add it as an **import** to the **course.module.ts** file

(Note: The FormsModule will be used later in the exercise, so it is imported already)

Save changes so far and run the application – it should still function in the same was as before.

Part 2 - Selecting a course for editing

Now that we have a form to display the courses we now need to bind it to the model such that when we select a course using the select element, its details are displayed in the appropriate fields of the form. To do this, we need to set an Observable on the title field and then update the rest of the from whenever this changes.

11. Open **course-editor.component.ts** (or switch to it if it's already open in your editor!)
12. Add a call to a method called `onChanges()` – this will be defined next!
13. Define a method called `onChanges()` that has a `void` return type
14. Subscribe to value changes in the title field of the form using the following code:

```
this.courseForm.get('title').valueChanges.subscribe();
```

15. Make the argument to subscribe an anonymous function that takes a variable called value and has a body that:

    - Loops through each course in the course array checking to see if the current course title is the same as value and setting the class member course to the course being iterated if it is
    - Conditionally calls `patchValue()` on the form, setting `code`, `delivery` and `days` to the current value held in the class' course IF it has been the class' course has been set

Hints:

- A for…of loop is most efficient (coding-wise) here.
- patchValue is called as shown:

```
this.courseForm.patchValue({attribute1: value1, attribute2, value2…});
```

Save all files and then check to see if the whole form now updates when a change is made in the Title field.

Part 3 – Validating the user's input

The form now functions to display the different course details when the user makes a selection. To add the user-friendly validation, a little more work is needed. In this section, we will make use of Validators from Reactive Forms to recognise when an input is not valid. We will also hide the bottom 3 fields of the form until a user has made a selection. Let's do that now…

16. Under the `<div>` that encloses the **Title** section of the course, wrap the other constituent parts of the form in a `<div>` that uses a *structural directive* to only display **IF** a *value for course exists*

Preparing the template for validation

17. For each `<input>` tag add an **attribute** of `required`
18. For the `days` `<input>` add an **attribute** of `min` and set it to `1`
19. Under each `<input>` tag, add a `<div>` that uses a *structural directive* to display if the control for the field is **invalid AND (dirty OR touched)**, so for `code`:

```
<div *ngIf="code.invalid && (code.dirty || code.touched)"></div>
```

20. Inside each `<div>` created, add another `<div>` that displays if the field is left empty using the reactive validator syntax, so for `code`:

```
<div *ngIf="code.errors.required">Course code is required</div>
```

21. For the `days` `<input>` add a second `<div>` which checks for an **error on min** and displays a suitable method:

```
<div *ngIf="days.errors.min">Course Length should be at least 1</div>
```

Once you have done this, you might be tempted to see if it is working in the browser, but don't be surprised to find undefined errors relating to invalid. The next section will fix this!

Preparing the Component for Validation

22. Add Validators to the list of imports from @angular/forms
23. In the `createForm()` method, change `code` and `delivery` so that they are an *array* whose first element is the *original value* and the second is `Validators.required`, so for code:

```
code: ['', Validators.required],
```

24. Change `days` so that the second element in the array is *another array* with element values of `Validators.required` and `Validators.min(1)`

```
days: [null, [Validators.required, Validators.min(1)]]
```

25. Add 3 `get` methods to the class, *one for each of the validated controls* that simply returnd the field, so for `code`:

```
get code() { return this.courseForm.get('code') }
```

Save all files and return to the browser. You should see that the 3 validated fields are hidden until you make a selection and that the validation works if you remove any of the values. Check in the Course Duration field that an entry of 0 produces the desired validation method.

Part 4 – Submitting the changes

Unlike Template Driven forms, the values entered on the form are not automatically passed back to the component. To do this, we must add some code to retrieve the form values and then set a Course object to hold these for submission. First, we must prepare the form for the actions to take on submission.

Prepare the form

26. In **course-editor.component.html**, add an `ngSubmit` event that calls `onSubmit()` to the `<form>` tag
27. At the bottom of the form, in the `<button>` tag, add an **attribute** that *evaluates* **disabled** against whether the form is **invalid OR** (**code AND delivery AND days**) are **pristine**

```
[disabled]="courseForm.invalid || (code.pristine && delivery.pristine
&& days.pristine)"
```

This ensures the button is disabled unless there are changes to submit.

Create the onSubmit method in the Component

28. In **course-editor.component.ts**, add a method called `onSubmit()` to the class
29. The body of this method should:

    - Set the *current course* to a call to, an as yet unwritten method, `prepareCourse()`
    - Log out the *current course values*
    - Call `onChanges()`

Create the prepareCourse method

30. Create a method called `prepareCourse()` that **returns** a `Course`
31. The method body should:

    - Declare a `const` called `formModel` that is *assigned* to the *current form value*
    - Declare a `const` called `saveCourse` that is of type `Course` and *assigns* the following:
        - `title` to be the *current course title*
        - `code` to be the *value of code on the form as a string*
        - `delivery` to be the *value of delivery on the form as a string*
        - `days` to be the *value of days on the form as a number*

So, for `code`:

```
Code: formModel.code as string,
```

    - Return `saveCourse`

Save all files and check that the button works as expected.

Obviously, the submit function would make a call to a 'Course Model' service to update the actual values held for the application in real life but not explained here.

If you have time…Custom Validators

The custom validator has already been included, the instructions are here for completeness, BUT…you will need to complete from the section called:

- **'Modifying the code to use the custom validator'**

to see it working, so skip to that if you did this as part of the exercise on Template Forms or read on until you reach the section!

Create a custom validator that would ensure that the Course Code supplied by the user starts with the letters QA (or qa) and has at least 3 more letters after this.

The Regular Expression needed for this is:

```
[Qq]{1}[Aa]{1}[A-Z]{3,}
```

Within the code, we will ensure that this is case insensitive.

Start by creating a Custom Validator Directive:

32. Navigate to the **courses** folder and create a new **directive** called `allowed-course-codes` using the Angular-CLI
33. In the **allowed-course-codes.directives.ts** file, add an import for `Input` from `@angular/core`
34. Imports the following from `@angular/forms`:

- `AbstractControl`
- `NG_VALIDATORS`
- `Validator`
- `ValidatorFn`
- `Validators`

The Validator Function

35. Under the import statements, export and declare a function called `allowedCourseCodeValidator` that:
    - Takes a **RegExp** called `allowedCourseCodeRe` as an argument
    - Has a return type of `ValidatorFn`
    - Returns an anonymous function that:
        - Takes an **AbstractControl** called `control` as an argument
        - Has a return type `{[key: string]: any}`

- Declares a **const** called `allowed` in the function body that is assigned to the result of **testing** the *supplied RegExp* against `control.value`
- Returns the ternary evaluation of **'NOT'** `allowed` – this should return `{'allowedCourseCode': {value: control.value}}` or `null`

The Validator Decorator

36. Change the selector in the @Directive decorator so that it is `'[allowedCourseCode][ngModel]'`
37. Add `providers` that set:

```
[{provide: NG_VALIDATORS, useExisting: AllowedCourseCodeDirective, multi: true}]
```

The Validator Class

38. Make the `AllowedCourseCodeDirective` class **implement** `Validator`
39. In the class body, remove the `constructor() {}` and add an `@Input()` decorator with `allowedCourseCode` as a **string**
40. Still in the class body, add a **method** called `validate` that:

    - Takes an **AbstractControl** called `control` as an argument
    - Has a return type `{[key: string]: any}`
    - Has a body that **returns** a ternary statement evaluating `this.allowedCourseCode` with `allowedCourseCodeValidator(new RegExp(this.allowedCourseCode, 'i'))(control)` or `null` as the return values

Modifying the Course Editor code to use the Custom Validator

41. Add an **import** for `allowedCourseCodeValidator` from **allowed-course-codes.directive**
42. Change the **value** given to code in the `createForm()` method so that the second element is an array
43. In the *newly created Validators array*, add another element that is a call to the `allowedCourseCodeValidator` function with the regular expression `/[Q]{1}[A]{1}[A-Z]{3,}/i` as it's argument – the line of code should be:

```
code: ['', [Validators.required, allowedCourseCodeValidator(/[Q]{1}[A]{1}[A-Z]{3,}/i)]],
```

Modifying the template to display a validation error

44. Under the `<div>` for a **required** error in `code`, add another `<div>` that:

- Uses a *structural directive* that checks for `code.errors.allowedCouseCode` *but not* **required** *errors*
- Displays a message to explain why it is invalid and the correct course code format

```
<div *ngIf="code.errors.allowedCourseCode && !code.errors.required">
    Course code must be QA followed by at least 3 letters
</div>
```

Save all of the code and check that all of your validation works.