Reactive Forms

BUILDING WEB APPLICATIONS USING ANGULAR

Contents

- What are Reactive Forms
- Async v sync
- Essential Form Classes
- Reactive Form Components
- Form Groups
- Form Builder
- Inspecting Form Data
- Populating Forms
- Using FormArray
- Validating a Reactive Form
- Saving Form Data

Reactive Forms (1)

- Technique for creating and building forms in a reactive style
  - Favours explicit management of data flowing between non-UI data-model and UI-oriented form model
    - Data typically retrieved from the server
    - Form model retains state and value of each HTML form control in the UI
  - Allows reactive patterns, testing and validation
- Create tree of Angular form control objects in component and bind to native form control elements in component template
- Form control objects can be created and manipulated directly in component class
  - Component can immediately access data model and form control structure
    - Allows pushing of data model value to form controls
    - Allows pulling of user-edited values back out
    - Component can observe changes in from control and react to the changes

Reactive Forms (2)

- Advantages:
  - Value and validity updates are always synchronous and under control
    - No timing issues sometimes found when using template-driven forms
  - Easier to unit test
- Reactive Paradigm
  - Component preserves immutability of data model
    - Treats data model as pure source of original values
  - Component extracts user changes and forwards changes to external component or service for handling
    - Does not update the model directly
- Not necessary to follow all reactive principles – useful if they are to be adopted

Async vs Sync

- Reactive forms are SYNCHRONOUS
  - Template-driven forms are asynchronous
- Entire form control tree coded
  - Can immediately update a value or transverse descendants of parent form because all controls are omnipresent

- Neither of the form types are better
  - Just different architecture with relative advantages and disadvantages
  - Often decided by the most suitable approach for the application
  - Can have both approaches in the same application

4 Essential Reactive Form Classes

- AbstractControl:
  - Abstract base class for 3 concrete form control classes (described below)
  - Provides common behaviours and properties (some of which are observable)
- FormControl:
  - Tracks the value and validity status of an individual form control
  - Corresponds to an HTML form control
- FormGroup:
  - Tracks values and validity status of a groups of AbstractControl instances
  - Group's properties include its child control – top level form is often FormGroup
- FormArray
  - Tracks value and validity status of a numerically index array of AbstractControl instances

Reactive Form Components

- Requires FormControl to be imported from the @angular/forms module

```
import { FormControl } from '@angular/forms';
```

- Form Controls can then be defined in the Component

```
export class SomeComponent {
   formcontrolname = new FormControl();
}
```

- The template can then be created to use the form control

```
<label>FormControlName
   <input [formControl]="formcontrolname">
</label>
```

- The ReactiveFormsModule also needs to be imported into the appropriate module

Form Groups

- Useful to register multiple FormControls on a form
- Need to import and add FormGroup to the component and label it in the template
- FormControls are wrapped in a FormGroup in the Component class

```
export class SomeComponent {
   formname = new FormGroup({
      formcontrolname = new FormControl()
   });
}
```

- formGroup reactive directive associates existing FormGroup with an HTML element

```
<form [formGroup]="formname">
   <label>FormControlName
      <input formControlName="formcontrolname">
   </label>
</form>
```

Form Builder

- Helps to reduce repetition and clutter by handling details of control creation
- Need to import FormBuilder into component, inject into constructor and then use a method that uses FormBuilder to create the form, calling it in the constructor

```
…
import { FormBuilder, FormGroup } from '@angular/forms';
…
export class SomeComponent {
   myForm: FormGroup;
   constructor(private fb: FormBulder) {
      this.createForm();
   }
   createForm() {
      this.myForm = this.fb.group({
         myFormControlName: ''
      });
}
```

Form Control Properties

- When using a FormControl the following properties can be accessed:
  - .value – returns the value of the FormControl named before the .
  - .status – returns the validity of the FormControl named before the .
    - Can be: VALID, INVALID, PENDING or DISABLED
  - .pristine – returns true if the user HAS NOT changed the value of the FormControl preceding the . in the UI
  - .dirty – returns true if the user HAS changed the value of the FormControl preceding the . In the UI
  - .untouched – returns true if the user HAS NOT entered the HTML control and triggered the blur event
  - .touched – returns true if the user HAS entered the HTML control and triggered the blur event
- These are commonly used for Validation (see later)

Inspecting Form Data

- When using a FormControl without a FormGroup (or a just a FormGroup) the following syntax can be used within the template (and the Component) to access the control properties:

```
myControlProperty.controlProperty

// Eg.
name.value
formName.status
name.pristine
formName.touched
```

- When using a FormGroup the .get() method is used to access an individual FormControl and then the control properties can be accessed (again can be used on the template or in the Component):

```
myFrom.get('myControlProperty').controlProperty

// Eg.
formName.get('name').status
formName.get('topProperty.nestedProperty').touched
```

Populating Forms (1)

- Copying values from a data model to a form model has 2 important implications:
  1. Understanding how the data model's properties map to the form model's properties
  2. Changes in the UI flow from the DOM elements to the form model – the form controls never update the data model
- Data and form model structures need not match exactly
  - Often want to present a subset of data from the data model in the UI
  - Easier if the from model almost matches the data model
- Can initialise form data when a control is created
  - Can also use setVaue and patchValue

Populating Forms (2) – setValue

- The following data model and form model are pretty close in structure
  - The form model does not have the id property

```
// Data Model

export class MyDataModel {
   id = 0;
   property1 = '';
   property2: boolean;
}
```

```
// Component Class (excerpt)

this.myForm = this.fb.group({
   property1: '',
   property2: false
});
```

- With setValue, every form control value is assigned at once by passing data object with exactly the same properties in the form model

```
// Component Class (excerpt)
this.myForm.setValue({
   property1: this.myDataModelInstance.property1,
   property2: this.myDataModelInstance.property2
});
```

Populating Forms (3) – patchValue

- With patchValue, form control value can be set for specific controls
  - Done by supplying a key/value pair for controls that need to be populated
- Fails silently if structure or values are missing
- More flexibility for working with differing data and form models
- Only the property1 is set in the code shown here:

```
// Component Class (excerpt)
this.myForm.patchValue({
   property1: this.myDataModelInstance.property1
});
```

Populating Forms (4) – when to set values

- Depends on when the the component has access to the data model values
- Could be when the user selects a new data item to view
  - Selected data item is passed to the Component by binding an specific instance of the data model to it
  - In this case, use setValue in the ngOnChanges lifecycle hook
    - OnChanges and Input need to be imported into the Component

```
import { Component, Input, OnChanges } from '@angular/core';
```

    - Add the data model instance as an input

```
@Input myDataModelInstance: MyDataModel;
```

    - Define the ngOnChanges method to call setValue

```
ngOnChanges() {
   this.myForm.setValue({
      … // values set here
   });
}
```

Populating Forms (5) – how to reset values

- Reset a form to ensure that previous values are removed and that the status flags are returned to their default state
  - Call reset at the top of the ngOnChanges method

```
this.myForm.reset()
```

  - Optional state value so that status and control values can be set at the same time
    - Actually calls setValue with the argument when provided

```
ngOnChanges() {
   this.myForm.reset({
      property1: this.myDataModelInstance.property1,
      property2: this.myDataModelInstance.property2
   });
}
```

Using FormArray (1) – Nomenclature

- Sometimes necessary to present an unknown number of controls or groups
- An Angular FormArray can display an array of FormGroups or FormControls
- To use FormArray
  1. Define the FormControls or FormControls in the array
  2. Initialise the array with items created from data in the data model
  3. Add and remove items as the user requires

Using FormArray (2) – Data Models and FormArray

- Data models sometimes include other objects (and indeed arrays of those objects) as

```
// Data Model

export class MyDataModel {
   id = 0;
   property1 = '';
   property2: boolean;
   Property3: SomeObject[];
}

export class SomeObject {
   someProperty1 = '';
   someProperty2 = '';
   someProperty3 = '';
}
```

- These can be included in a form model as a nested form group:

```
// Component Class excerpt

this.myForm = this.fb.group({
   prop1: '',
   prop2: false,
   prop3: this.fb.array([])
});
```

- prop3 is an empty FormArray
- Note that the form model does not have to match the data model for this to work
  - Just needs to be some form of sensible relationship within the application domain

Using FormArray (3) – Replacing FormArrays

- Default form will display with nothing in the prop1 and prop3 form fields
- Need a method to populate (or repopulate) the prop3 with actual values when the Component sets a MyComponent.myDataModelInstance input property to a new MyDataModel
- The setProp3 method replaces the prop3 FormArray with a new FormArray initialised by an array of SomeObject FormGroups

```
// Component excerpt

setSomeObjects(someObjects: SomeObject[]) {
   const someObjectFGs = someObjects.map(someObject => this.fb.group(someObject));
   const someObjectFormArray = this.fb.array(someObjectFGs);
   This.myForm.setControl('prop3', someObjectFormArray);
}
```

- Note the use of setControl here rather than setValue
  - This is because a control is being set and not the value of a control

Using FormArray (4) – Getting

- Use FormArray's get() method to receive a reference to the FormArray
  - Convenience properties are used for clarity and re-use

```
// Component excerpt

get prop3(): FormArray {
   return this.myForm.get(prop3) as FormArray;
}
```

Using FormArray (5) – Displaying

- The *ngFor structural directive is useful here
  - Write in the following way:
    1. Wrap the element with the *ngFor in another element (e.g. a)
    2. Set the wrapping element's formArrayName directive name to the name of the FormArray from the component
       - Establishes the FormArray as the the context for form controls in the inner HTML element
       - Source of repeated items is FormArray.controls not FormArray itself and each item is a FormGroup
    3. Each repeated FormGroup needs a unique formGroupName which must be the index if the

```
<!-- Template excerpt -->
<div formArrayName="prop3">
   <div *ngFor="let someObject of prop3.controls; let i = index" [formGroupName]="i" >
      <!- Repeated template for someObject -->
   </div>
<div>
```

Using FormArray (6) – Adding to the FormArray

- Create a method in the component that gets the FormArray and appends a new object FormGroup to it

```
// Component excerpt

addProp3() {
   this.prop3.push(this.fb.group(new SomeObject());
}
```

- Add a button on the template so the user can add a new prop3 object, wiring it to the addProp3

```
<!-- Template excerpt -->

<button (click)="addProp3()" type="button">Add a prop3</button>
```

Validating a Reactive Form (1) – Using Validators

- Source of truth is component class
  - Validator functions added directly to from control model
  - Angular calls functions whenever the value of the control changes
- 2 types of Validator functions:
  - **Sync validators** – take a control instance and immediately return set of validation errors (or null)
    - Can pass these as second argument when you instantiate a FormControl
  - **Async validators** – take a control instance and return a Promise or Observable that then emits a set of validation errors (or null)
    - Can pass these as third argument when you instantiate a FormControl

Validating a Reactive Form (2) – Built-in Validators

- Same built-in validators as available for template-drive forms can be used in functions from Validators class
  - Eg. required, minlength, maxlength, email, min, max, etc **
  - When creating a FormControl, a single validator or an array of built-in validators can be passed to each

```
// Component excerpt
this.myForm = new FormGroup({
   prop1: new FormControl(this.myDataModel.property1, [
      Validators.required,
      Validators.minLength(10)
   ]),
   prop2: new FormControl(this.myDataModel.property2, Validators.requiredTrue)
});
```

  - In the template, *ngIf directives can be used to access the FormControl errors property to conditionally

```
<input id="prop1" formControlName="prop1" required>
<div *ngIf="prop1.error.required">This field is required</div>
```

Validating a Reactive Form (3) – Custom Validators

- Built-in validators don't always do what you need them to
- Custom validators can be created – usually in their own .directive.ts file and imported for use

```
// forbidden-string.directive.ts

export function forbiddenStringValidator(stringRe: RegExp): ValidatorFn {
   return (control: AbstractControl): {[key: string]: any} => {
      const forbidden = stringRe.test(control.value);
      return forbidden ? {'forbiddenString': {value: control.value}} : null;
   };
}
```

- FormControl string cannot match the given regular expression
- Actually factory function to detect a specific forbidden string and returns a validator function

```
// Component excerpt
prop1: new FormControl(this.myDataModel.property1, [
   Validators.required,
   forbiddenStringValidator(/someForbiddenString/i)
])
```

Save Form Data (1) - Saving

- Saving and reverting unsaved changes are common operations on forms
- Saving form data is done through passing an instance of the data model to a save method on the injected service triggered by a 'Save' button on the UI

```
// Component excerpt
onSubmit() {
   this.myDataModel = this.prepareSaveDataModel();
   this.myDataModelService.updateData.subscribe(/* error handling */);
   this.ngOnChanges();
}
prepareSaveDataModel(): MyDataModel{
   const formModel = this.myForm.value;
   const prop3DeepCopy: SomeObject[] = formModel.prop3.map(
      (someObject: SomeObject) => Object.assign({}, someObject)
   )
   const saveMyData: MyDataModel = { id : this.myDataModelInstance.id, property1:
   formModel.prop1 as string, property2: formModel.prop2, property3: prop3DeepCopy}
   return saveMyData;
}
```

Save Form Data (2) - Reverting

- Reverting form data is done simply by supplying the user with a 'Revert' button and calling ngOnChanges from the handling method using the original data model

```
// Component excerpt
revert() { this.ngOnChanges(); }
```

- The triggering of the onSubmit and revert methods would be done as in Template-driven forms
  - Button types of 'submit' tigger the ngSubmit event – calling the attached handler (onSubmit in our case)

Exercise

CREATE A REACTIVE FORM

Appendix – Observe Control Changes

- Angular does not call ngOnCHanges when the user modifies a FormControl
  - Changes can be monitored by subscribing to one of the form control properties that raise an event
    - Such as valueChanges – returns an RxJS Observable
    - Add methods to log the changes:

```
// Component excerpt

prop1ChangeLog: string[] = [];
logProp1Change() {
   const prop1Control = this.myForm.get(prop1);
   prop1Control.valueChanges.forEach(
      (value: string) => this.prop1ChangeLog.push(value)
   )
}
```

    - Call method in the constructor – outputting the prop1ChangeLog in the template will show all keystrokes

Appendix – Dynamic Forms (1)

- Reactive Forms created based on metadata that describes the business object model
- Can use FromGroup to dynamically render forms with different control types and validation
- Allows creation of forms on-the-fly without changing the application code

RECIPE…

1. Define an object model that can describe all scenarios needed by the form functionality
2. Derive classes based on the types of inputs required on the form
   - Idea is that form will be bound to specific input types and render the appropriate controls dynamically
3. Create a service to convert the input types into a FormGroup
   - Form group consumes the meta data from the object model and allow specification of default values/validation
4. Create components that represent the dynamic form and a parent component for the form
   - Using *ngSwitch directive can determine which of the components to display

Appendix – Dynamic Forms (2)

RECIPE (ctd)…

5. The wrapping component expects the input types in the form of an array bound to an @Input
    - These can be retrieved by the service defined
    - Key point is that inputs are controlled entirely through objects returned from the service
        - Input maintenance is simply a matter of adding, updating and removing objects from the inputs array
6. Use the service in the component that will contain the dynamic form, using it to set the array of inputs to be used on the form

- A worked example of this can be found at: https://angular.io/guide/dynamic-form