Chapter 1 - Model Driven Forms

Objectives

This chapter includes the following model driven forms topics:

- Setup
- FormGroup initialization
- FormControl object
- Validation
- SubForms

1.1 Model Driven Forms Overview

- Angular has two ways to define forms, "template driven" forms and "model driven" forms
  - Model driven forms are also referred to as "Reactive forms"
- Model driven forms create and manipulate form control objects directly in the component class
  - The component can observe changes in form control state and react to those changes, thus "reactive" forms
- There are still form elements in the template of the component but not as much code to define the form
  - The template just binds to the form control elements defined in the component

1.2 Setup for Model Driven Forms

- Similar to that for template driven forms
- To use the full features of model driven forms, use the 'ReactiveFormsModule' instead of the regular 'FormsModule'
- Add the following in `app.module.ts` boot file:

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

```
...
@NgModule({
  imports: [ BrowserModule, ReactiveFormsModule ],
```

1.3 Form Component Setup

- Typical imports required in the form component:

```
import { FormGroup, FormControl, FormBuilder,
  Validators } from  '@angular/forms';
@Component({
 ...
})
```

1.4 Setup Main FormGroup

- In the component class
  - Have a property of FormGroup type to reference the form
  - Construct the form during component construction or initialization

```
myForm: FormGroup;
ngOnInit(){
  this.myForm = new FormGroup({
    first: new FormControl('Jim', []),
    last: new FormControl('Doe',[Validators.required])
  });
}
```

- In HTML Template
  - The <form> element references the form with the [formGroup] property
  - The individual form elements reference the form elements from the component with the 'formControlName' property

```
    <form [formGroup]="myForm">
      First: <input formControlName="first" ><br>
      Last : <input formControlName="last" ><br>
      <input type=button value=submit (click)=onSubmit()>
    </form>
```

1.5 formControlName

- The `formControlName` directive is used to bind input fields in the HTML template to FormControl objects in the component class:
  - formControlName directive:

    ```
    <input formControlName="first" >
    ```
  - FormControl Object

    ```
    first: new FormControl('Jim', []),
    ```
- These two are bound by the name "`first`"

1.6 FormControl Object

- There are two ways to create the FormControl object bound to the input element in a form:
  - Constructed during FormGroup initialization

```
this.myForm = new FormGroup({
  first: new FormControl('Jim'), ...
```

  - Then bound to form using 'formControlName'

```
<input formControlName="first" >
```

  - Created as a separate property of the component class

```
firstName = new FormControl('Jim');
```

  - Then bound to using '[formControl]'

```
<input [formControl]="firstName" >
```

- You can only use '[formControl]' in the template when the FormControl is available as a property of the component
  - If the FormControl is only a child of the FormGroup, you must use 'formControlName' to reference it

1.7 Getting Form Values

- Defining a form is only useful if you can get the values entered in the form and use them in the rest of the Angular application
- In JavaScript code in the component, you can get the value of the entire form with the '.value' property of the form

```
data = myForm.value;
```

  - Or of a individual component within the form using the 'get' method

```
firstName = myForm.get('firstName').value;
```

- The "value" of the entire form is an object with property names matching the form control names and values containing the form data

```
{ first: "Bob",          // myForm.value
  last: "Builder"   }
```

- One advantage of model driven forms is that data is available synchronously, meaning you don't have to wait for Angular template directives to update the form
  - This also simplifies testing since you don't need to manipulate forms asynchronously in a test

1.8 FormBuilder Form Initialization

- Using the 'FormBuilder' class simplifies form initialization
  - You don't need to manually call the FormControl constructor
- To use the FormBuilder to initialize a form:
  - Import the class with the other '@angular/forms' classes
  - Inject a FormBuilder into the component constructor

```
constructor(private fb: FormBuilder) {
    this.createForm();   // calling custom method
}
```

  - Call the 'group' method of FormBuilder with an object where the properties are the names of form controls and values are the parameters you would pass if calling the FormControl constructor
    - Multiple FormControl constructor parameters are passed as an array

```
createForm() {
  this.heroForm = this.fb.group({
    first: '', // <--- the FormControl called "first"
    last: ['', Validators.required ] // parameter array
  });
}
```

1.9 Validation

- One of the biggest differences between template driven forms and model driven forms is where validation is declared
  - Template driven forms have validation properties defined in the HTML template
  - Model driven forms define validation by customizing the construction of

FormControl objects in the component

- When FormControl objects are created, there is an optional, second parameter to the constructor
  - We can pass a single validator or an array of validators as the second parameter

```
fieldname: new FormControl ( default_value,
                       validator | validator[] )
```

- Example validators array:

```
[ Validators.minLength(5), Validators.required ]
```

1.10 Built-In Validators

- Angular has a few built-in validators:

```
Validators.required
Validators.minLength(minLen: number)
Validators.maxLength(maxLen: number)
Validators.pattern(regex-pattern: string)
```

- Usage examples:

```
name: new FormControl('', Validators.required)),
street: new FormControl('', Validators.minLength(3)),
city: new FormControl('', Validators.maxLength(10)),
zip: new FormControl('', Validators.pattern('[A-Za-z]{5}'))
```

Notes

Corresponding usage in Template Driven Forms:

```
<form novalidate>
  <input type="text" name="name" ngModel required>
  <input type="text" name="street" ngModel minlength="3">
  <input type="text" name="city" ngModel maxlength="10">
  <input type="text" name="zip" ngModel pattern="[A-Za-z]{5}">
</form>
```

1.11 Custom Validator

- Custom Validators:
  - Take a Control as input
  - Return null when the control is valid
  - Return an object with validation info when control is invalid
- Example (`email.validator.ts`):

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

export function validateEmail(control: FormControl)
{
  if( typeof(control.value) === 'string'){
    if(control.value.includes("@")){
      return null;
    }else{
      return {validateEmail: {valid: false}}
    }
  }
}
```

Notes

To create a custom validator:

- create a separate file for the validator
- import FormControl so we can reference it
- create and exporting the validation function
- have the validation function accept a FormControl parameter
- have the validation function return null or object
-

The example validator above simply checks for the presence of the '@' symbol in the string.

1.12 Using a Custom Validator

- Steps:
  - Import the validator function

    ```
    import { validateEmail } from './email.validator';
    ```

  - Reference the function in a FormControl

    ```
    this.myForm = fb.group({
      email: ["james@abc.com", [ validateEmail ]]
    });
    ```

Notes:

Full code of validator function (email.validator.ts):

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

export function validateEmail(control: FormControl) {
  if( typeof(control.value) === 'string'){
    if(control.value.includes("@")){
      return null;
    }else{
      return {validateEmail: {valid: false}}
    }
  }
}
```

Full code of component that uses the custom validator (custom.validator.component.ts):

```
import { Component, OnInit } from '@angular/core';
import { validateEmail } from './email.validator';
import { FormGroup, FormControl, Validators } from '@angular/forms';

@Component({
selector: 'customvalform',
template: `
<h3>Custom Validator Form</h3>
<form [formGroup]="myForm">
 Email:
 <input formControlName="email" ><br>
 <input type=button value=submit (click)=onSubmit() >
```

```
</form>
Form Valid: {{myForm.valid}}<br>
Form Value: {{myForm.value|json}}
`,
})
export class CustomValidatorComponent implements OnInit {

  myForm: FormGroup;

  ngOnInit(){
    this.myForm = new FormGroup({
      email: new FormControl('Jim@abc.com', [ validateEmail ]),
    });
  }

  onSubmit(){
    console.log( "Valid: " + this.myForm.valid );
    console.log("Value: " + JSON.stringify(this.myForm.value, null, 2));
  }
}
```

1.13 Useful FormGroup and FormControl Properties/Functions

- Several properties of FormGroup and FormControl are inherited from the AbstractControl class
  - .value
  - .status – String of the control status
  - .disabled/.enabled
  - .reset(...)
  - .valid/.invalid
  - .pristine/.dirty – If the UI value has changed
  - .touched/.untouched – If the user has selected/entered
- There are also '.setValue' and '.patchValue' methods
  - These behave the same on a FormControl and change the value of the control
  - With a FormGroup they take an object with control names as keys
    - With 'setValue' the object must contain exactly the form structure or

an error is thrown

- 'patchValue' can take a super-set or sub-set of the form data and will match what it can without throwing an error

1.14 Sub FormGroups - Component Class

- Component Class Code:

```
export class FormComponent implements OnInit {

  myForm: FormGroup;
  ngOnInit(){
    this.myForm = new FormGroup({
      first: new FormControl('Jim', []),
      last: new FormControl('Doe', []),
      address: new FormGroup({
        address: new FormControl('Main Street', []),
        city: new FormControl('Newark', []),
        state: new FormControl('NJ', []),
        zip: new FormControl('07102', []),
      })
    });
  }
  onSubmit(){
    console.log( JSON.stringify(
                   this.myForm.value, null, 2));
  }
}
```

1.15 Sub FormGroups - HTML Template

- In HTML Template

```
<form [formGroup]="myForm">
    First: <input formControlName="first" ><br>
    Last : <input formControlName="last" ><br>
  <div formGroupName="address">
    Address: <input formControlName="address" ><br>
    City: <input formControlName="city" size=12 >
    State: <input formControlName="state" size=2 >
    Zip: <input formControlName="zip" size=5 >
  </div>
  <input type=button value=submit (click)=onSubmit()>
</form>
```

- In the above HTML template we have two fields (first, last) at the root level of the main form group "myForm"
- Then we have four fields (address, city, state, zip) in the sub Form Group Address.
  - Note the sub FormGroup is referenced by the 'formGroupName' property as 'formGroup' can only reference something declared as a component property and not the nested FormGroup 'address'

1.16 Why Use Sub FormGroups

- It formats the form data for us:

```
/* this.myForm.value */
{
  "first": "Jim",
  "last": "Doe",
  "address": {
    "address": "Main Street",
    "city": "Newark",
    "state": "NJ",
    "zip": "07102"
  }
}
```

- Validation can be applied separately to different sub groups.

1.17 Summary

- Model driven forms define more of a form's properties in the code of the related component
- Components with model driven forms still have form elements in the template but mainly to match up with the form elements defined in the component
- A FormGroup object is the root of a form with FormControl objects representing individual form elements
- Using the FormBuilder class can simplify the initialization of a form
- Validation of model driven forms is done with setting 'Validators' properties in the form controls
- Sub forms can model a more complex structure with FormGroup objects nested inside other FormGroups