Reactive Data Driven Forms

CS569 – Web Application Development II

Maharishi International University

Department of Computer Science

Associate Professor Asaad Saad

Maharishi International University - Fairfield, Iowa

Forms in Angular

A lot of applications are very form-intensive, especially for enterprise development.

Forms can end up being really complex:

- Form inputs are meant to **modify data**, both on the page and the server
- Users cannot be trusted in what they enter, so you need to **validate** values
- The UI needs to clearly **state expectations** and errors
- **Dependent fields** can have complex logic
- We want to be able to **test** our forms, without relying on DOM selectors

Form states such as being: **`valid`**, **`invalid`**, **`pristine`**, **`dirty`**, **`untouched`**, **`touched`**, **`disabled`**, **`enabled`**, **`pending..etc`**

Two Modules

Angular has the **`@angular/forms`** package with two modules:

`FormsModule` (**Template Driven Forms**)

We create a form by placing directives in the template. We then use data bindings to pass the data from and to that form.

`ReactiveFormsModule` (**Data Driven Forms**)

We define a form in the component class and bind it to elements in the template. Because we use reactive programming to pass data in and out of the form, we call it “reactive”.

Importing Form Module

```
import { ReactiveFormsModule } from '@angular/forms';
@NgModule({
        declarations: [..],
        imports: [ BrowserModule,
                    ReactiveFormsModule ]
})
```

By importing `ReactiveFormsModule` into our `NgModule` means we can use all directives exported from these modules in our view template.

High-Level Form Programming

Form Model

The form model is a UI-independent representation of a form. There are three building blocks: **`FormControl`**, **`FormGroup`**, and **`FormArray`**.

Form Directives

These directives connect the form model to the DOM.

DOM

These are DOM inputs, checkboxes, and radio buttons.

Angular Form Models

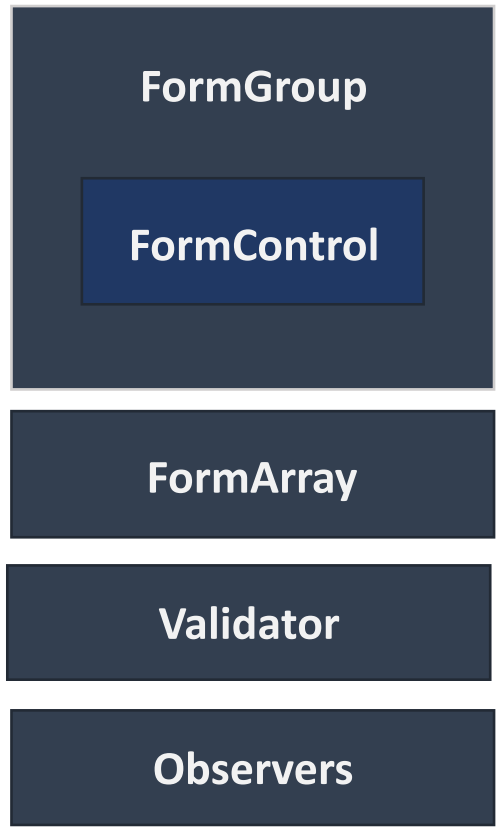
`FormControl` an indivisible part of the form, an atom. It usually corresponds to a simple UI element, such as an input. It has a **`value`**, **`status`**, and a map of **`errors`**

`FormGroup` A fixed-size collection of controls. The value of a group is just an aggregation of the values of its children.

`FormArray` A collection of the same control type as `FormGroup` of a variable length.

`Validator` give us the ability to validate inputs.

`Observers` let us watch our form for changes and respond accordingly.



Why Form Model?

The form model is a UI-independent way to represent user input consist of simple controls (`FormControl`) and their combinations (`FormGroup` and `FormArray`), where each control has a value, status, validators, errors, it emits events and it can be disabled.

Having this model has the following advantages:

- Form handling is a complex problem. Splitting it into UI-independent and UI-dependent parts makes them easier to manage. And we can test form handling without rendering UI.
- Having the form model makes reactive forms possible.

Data Driven Forms – Form Model

```
const formgroup = new FormGroup({
        login: new FormControl(''),
        passwords: new FormGroup({
                password: new FormControl('', Validators.required),
                passwordConfirmation: new FormControl('')
        })
});
```

Using `FormBuilder` Service

`FormBuilder` is a service that helps us build forms and give us a lot of customization options.

Forms are made up of **`FormControl`** and **`FormGroup`** and the **`FormBuilder`** helps us create them (you can think of it as a factory object).

Data Driven Forms Directives

We want to link our `<form>` UI to the `myForm` object that we created in our Component.

When we want to bind an existing **FormGroup** to a **form** we use the directive
formGroup
When we want to bind an existing **FormControl** to an **input** we use the directive
formControl

Reactive Forms with FormBuilder

To start building a Data-Driven Form we inject **`FormBuilder`** service in the constructor of our component class.

During injection an instance of **`FormBuilder`** will be created.

```
@Component({...})
export class DataDrivenComponent {
        myForm: FormGroup;
        constructor(private fb: FormBuilder) {
                this.myForm = fb.group({
                        'email': ['asaad@miu.edu'] });
        }
        onSubmit(): void {
                console.log('you submitted value: ', this.myForm.value);
        }
}
```

There are two main methods on FormBuilder :
`control()` - creates a new **`FormControl`**
`group()` - creates a new **`FormGroup`**

Using myForm in the view

```
<form [formGroup]="myForm" (ngSubmit)="onSubmit()">
        <input type="text"
                name="email"
                [formControl]="myForm.get('email')">
        <div *ngIf="!myForm.get('email').valid">Invalid email</div>
        <div *ngIf="myForm.get('email').hasError('invalid')">Error</div>
        <button type="submit" [disabled]="!myForm.valid">Submit</button>
</form>
```

```
formControlName="email"
```

Inspect the code and see how angular adds all kind of state change classes to the form elements.

Main Points

Data Driven Forms

To bind to an existing **`FormGroup`** and **`FormControl`** from your component to your template we use the following directives:

- **`formGroup`**
- **`formControl`**

Form Validation

Validators change the status of **`FormControl/FormGroup.`**
To assign a validator to a **`FormControl`** object we simply pass it as the **second argument** to our **`FormControl`** constructor:

```
@Component({...})
export class DataDrivenComponent {
   myForm: FormGroup;
   constructor(fb: FormBuilder) {
        this.myForm = fb.group({
           'email': ['asaad@miu.edu', Validators.required] });
   }
}
```

To use more than one validator:

The compose function will execute all the validators and merge the errors.

```
'email': ['asaad@miu.edu', Validators.compose([Validators.required, Validators.email]) ]
```

Custom Validators

A validator is a function that takes a **`FormControl`** as an input and returns a **`StringMap<string, boolean>`** where the key is error code and the value is `true` if it fails

```
exampleValidator(control: FormControl): {[s: string]: boolean} {
        if (control.value === 'Example') {
                return {'invalid': true};
        }
        return null;
}
```

Returning null means validation is valid.
Returning anything else means validation is invalid.

Custom Asynchronous Validators

Asynchronous validator is a function that takes a **`FormControl`** as its input and returns a **`Promise<any>`** or an **`Observable<any>`**

```
asyncValidator(control: FormControl): Promise<any> | Observable<any> {
   const promise = new Promise<any>((resolve, reject) => {
       setTimeout(() => { if (control.value === 'Example') {
                              resolve({'invalid': true});
                          } else {
                              resolve(null);
                          }
                  }, 1500);
   });
   return promise;
}
```

While the promise or the observable being resolved, the status of the form will be PENDING

Watching For Changes

Both **`FormGroup`** and **`FormControl`** have an **`EventEmitter`** that we can use to observe changes.

Any time a control updates, it will emit the value.

To watch for changes on form/control we get access to the **`EventEmitter/Observable`** by calling **`valueChanges`** or **`statusChanges`** (valid, invalid, pending).

Since **`valueChanges`** and **`statusChanges`** are RxJS observables, we can use the rich set of RxJS operators to implement powerful user interactions in a just a few lines of code.

Example

```
import { Component } from '@angular/core';
import { FormGroup, FormBuilder } from "@angular/forms";

@Component({...})
export class DataDrivenComponent {
        myForm: FormGroup;
        constructor(private formBuilder: FormBuilder) {
                this.myForm = formBuilder.group({});
                this.myForm.statusChanges.subscribe(
                              (data: any) => console.log(data) );
        }
}
```

Main Point

Angular forms can be created via two ways: by using the **template-driven** approach or the **data-driven** approach.

In the **data-driven** approach, you create the form as Model in the component body), and then link it to the UI/DOM by using **`formGroup`** for the form and **`formControl`** for the individual controls.