Angular Forms

Siddharth Ajmera **@SiddAjmera**

Content:

- Form Building Strategies in Angular
- Template Driven Forms
- Form Validations
- Model Driven/Reactive Forms
- Data binding
- Binding with different control types
- Custom Form Validations

Angular Form API

Template Drive

- Fully programmed in component's template
- Angular is responsible for generating the JS Object Representation of the form.
- Template defines structure of the form
- Validation rules are also defined in the template

Reactive forms

- Form model created programmatically in the code (typescript code)
- Template can be dynamically generated based on the model
- FormControls and FormGroups can be added dynamically in FormArrays

Template Driven form

- Directives like ngForm, ngModel and ngModelGroup are used to create forms.
- NgForm represents a form, NgModelGroup represents a group of form controls, and NgModel represents a single form control.
- It is simpler and uses classes from the FormsModule in Angular.
- Form data is exported as JSON values when submit method is called.
- Native HTML5 validation attributes are used for validations.
- Custom Directives can be used for custom validations.

Steps - Template Driven

- import { FormsModule } from '@angular/forms' and add it to imports array.
- <form #formName="ngForm" (ngSubmit)="submit(formName)">
- <input required
  minlength="8"
  maxlength="20"
  pattern="John Doe"
  name="name"
  ngModel
  #name="ngModel">
- ngModelGroup="address" to collate the FormControls as a group.
- Use the template variables to check whether there's an error in that FormControl.

Connecting model to input control

- The directive NgModel connects a model property to an input control
- Unlike AngularJS, NgModel is not distributed with the core Angular module
- NgModel is distributed via the @angular/forms package, and must be registered with the application module to be used
- Once imported, the directive can be used in templates throughout the application
- NgModel is applied to a form control such as an input control
- NgModel assigns the value of the model to the control, and it outputs the new value to update the model

Data Binding

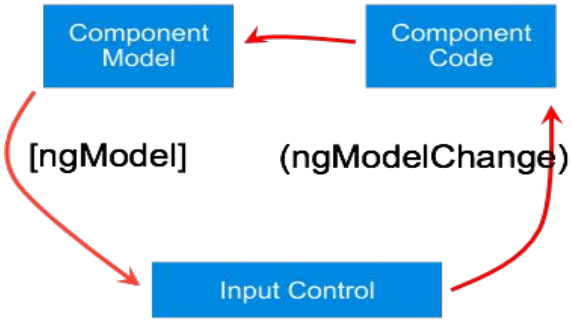
- The value of the message property, 'Hello World!' is display with a template variable, and populates the input control
- When the value in the input control is modified, the template variable is updated immediately

```
import { Component } from "@angular/core";

@Component({
  selector: "main",
  template: `
    Value: {{message}}<br>
    <input type="text" [(ngModel)]="message">
  `,
})
export class AppComponent {
   public message: string = "Hello World!";
 }
```

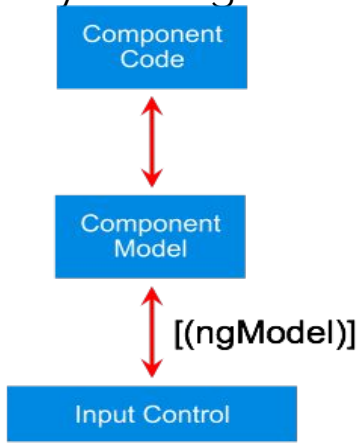
Binding Pattern

- Angular forms have three parts which make up the whole
  - Form and Control Elements
  - Form and Control Objects
  - Component Model
- NgModel connects the Component Model directly to the Elements
- The connection created by NgModel is called two-way binding



One-Way Data Binding

The key difference is the location of the component code and the bi-directional arrows. Both, approaches can update the model, the key is managing the updates through component code versus automatic bindings. Two-way results in more than one source of truth for the component model.



Two-Way Data Binding

Reactive form

Reactive programming is a programming paradigm oriented around data flows and propagation of change

- With Reactive Forms, the component directly manages the data flows between the form controls and data models
- Reactive forms are code driven versus template driven
- Reactive forms break from the most traditional declarative approach Angular has used in the past, its more similar to React
- Reactive forms eliminate the anti-pattern of updating the data model via two-way data binding
- Typically, Reactive form control creation is synchronous, and can be unit tested with synchronous programming techniques

Steps - Reactive

- import { ReactiveFormsModule } from '@angular/forms';
- Add it to the imports array like so: imports: [ReactiveFormsModule]
- In the TypeScript Class, import { FormGroup, FormControl, FormArray, Validators } from '@angular/forms';
- In the ngOnInit lifecycle hook:
- new FormGroup({
  someKey: new FormControl('Initial Value', [SyncValidators], [AsyncValidators])
  })
- new FormControl('Initial Value', [SyncValidators], [AsyncValidators]),
- new FormArray([])
- (<FormArray>this.userForm.get('hobbies')).push(new FormControl(''));
- (<FormArray>this.userForm.get('hobbies')).removeAt(index);
- get username() {
  return this.userForm.get('username');
  }

Steps - Reactive Cntd...

- Bind the form to the template like this: <form [formGroup]="userForm">
- Bind a FormControl to an input using <input formControlName="name">
- Bind a FormGroup to a group of form controls using <fieldset formGroupName="address">
- Loop through a FormArray's FormControls or FormGroups using:

```
<div *ngFor="let hobbyControl of userForm.get('hobbies').controls; let i = index">
        <input [formControlName]="i">
</div>
```

Reactive VS Template

The form control tree is created synchronously with code

The form control tree is created asynchronously as part of the compilation process as directives are processed

The form control tree is available immediately even before the child form elements have been created

The form control tree is available after the child form elements have been created

Good for complex dynamic forms

Old approach for 2 way data binding

Reactive form classes

- When building Reactive Forms, the FormGroup, FormControl & FormArray classes are the fundamental building blocks of a form
- To simplify the creation of forms, Angular provides a FormBuilder service
- A Form Group is a collection of Form Controls
- A Form Control is the programmatic connection between the form control in the template and the TypeScript code for the component
- A Form Array supports a dynamic number of form controls
- The Form Builder service uses an object literal to configure an entire form

Setting up Reactive form

- Import ReactiveFormsModule
- Import form specific functions in component code

```
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
```

```
export class ReactiveFormComponent implements OnInit {

  reactiveForm: FormGroup;
  post: any;
  description:string = '';
  name:string = '';

  constructor(private fb: FormBuilder) {
    this.reactiveForm = fb.group({
      'name' : ['', Validators.required],
      'description' : ['', Validators.compose([Validators.required,
        Validators.minLength(30), Validators.maxLength(500)])]
    });
  }

  ngOnInit() {
  }

}
```

- reactiveForm is defined as FormGroup
- Our form has 2 input fields name and description
- Form control is called ‘name’ and ‘description’. No other name is specified
- Validators can be composed as combination of multiple validaators

Setting up Reactive form


```
<div>
    <form [formGroup]="reactiveForm" novalidate>
        <div>
            <label for="new-name-input">Name:</label>
            <input type="text" id="new-name-input" formControlName="name">
        </div>

        <div>
            <label for="new-description-input">Description:</label>
            <input type="text" id="new-description-input" formControlName="description">
        </div>
    </form>
</div>
```

Dynamically adding and removing form controls

- The FormGroup and FormArray classes support the ability to add form controls dynamically
- Creating dynamic forms involves updating the form control tree and the DOM
- Directives such as ngFor can read the form control tree to create new input elements when new controls are added

```
let newAddress = this.reactiveForm.controls['addressGroups'] as FormArray;
  newAddress.push(new FormGroup({
    streetControl: new FormControl(''),
    cityControl: new FormControl(''),
    stateControl: new FormControl(''),
    zipCodeControl: new FormControl(''),
  }));
```

When the new form group is added, the ngFor adds a new div

```
  <fieldset formArrayName="addressGroups">
  <legend>Address</legend>
  <div
    *ngFor="let addressGroup of reactiveForm.controls['addre
    [formGroup]="addressGroup">
      Street: <input type="text" formControlName="streetCont
      City: <input type="text" formControlName="cityControl"
      State: <input type="text" formControlName="stateContro
      Zip Code: <input type="text" formControlName="zipCodeC
  </div>
</fieldset>
</div>
```

Extracting form data

- Unlike ngModel and Template Forms, Reactive Forms do not update the model directly
- Rather, the Form Control object tree is updated, and the data is explicitly extracted from the tree and then updates the model
- To extract the collected data from the FormGroup, FormControl and FormArray object, the value property is used
- For FormGroups and FormArrays, which are containers for other Reactive Form objects, their value properties return all of the form control data they contain

```
public profileForm: FormGroup;
```


```
console.dir(this.profileForm.value);
```

Form Validation

- Validation occurs at the control level, and the validation status is tracked on each control and is aggregated to the group and form level
  Example: If a single control is invalid, its group and form are invalid
- Three states are tracked for each control: pristine, valid and untouched
  - Pristine/Dirty – has any of the controls been modified
  - Valid/Invalid – is the data in controls valid according to the validation rules
  - Untouched/Touched – has the control fired its blur event
- Through these statuses form validation is performed
- For Template Forms, the Form Control Object tree can be accessed either through template reference variables or by accessing the NgForm with ViewChild
- For Reactive Forms, the Form Control Object tree is created as part of building the form, and if made available as a component property, it can be referenced from there in the template

Showing Error messages

- ng-invalid class will be paired with the ng-touched class to show error messages for a control
- CSS classes work for Template Forms and Reactive Forms
- Each Form, Group and Control object in the Form Control object tree contains the following pairs of boolean properties
  - valid / invalid
  - pristine / dirty
  - touched / untouched
- These properties can be used with template variables and directives such as NgIf to display validation messages