FORMS

Ori Calvo, 2017

oric@trainologic.com

https://trainologic.com

Introduction

- Handling forms inside SPA is considered challenging
- In general it breaks the unidirectional data flow principle
- Angular provides a dedicated package named @angular/forms
- Two approaches
  - Template Driven
  - Reactive Form

Template Driven vs. Reactive Form

- Template driven approach infers the form structure directly from the DOM
  - Therefore less testable
- Reactive form approach allows you to programmatically define the form and synchronize it with the DOM
  - More testable
  - More code

Template Driven

- Start by installing @angular/form and add FormModule to imports list
- Continue with attaching ngModel directive to inputs

```
<div>
  <input [(ngModel)]="name">

  <button (click)="add()">Add</button>
</div>
```

The [(ngModel)] syntax

- When specifying the following

```
<input [(ngModel)]="name">
```

- Angular converts is to

```
<input [ngModel]="name" (ngModelChange)="name = $event">
```

- It means that two way data binding can be used only when the directive supports the expected convention

CSS classes

- ngModel tracks the status of the input and changes its CSS classes

```
<input class="ng-pristine ng-valid ng-touched">
```

- Once the end user type something the CSS classes change

```
<input class="ng-valid ng-touched ng-dirty">
```

- You should use those CSS classes and give the correct UI feedback

Validation

- Is achieved using standard HTML5 attributes

```
<input [(ngModel)]="name" required>
```

- ngModel sets the ng-invalid/ng-valid classes

```
<input class="ng-pristine ng-invalid ng-touched">
```

- Once typing something

```
<input class="ng-valid ng-touched ng-dirty">
```

Display Validation Messages

- Do not use CSS classes for conditional display
- ngModel offers a simple API for querying validation flags

```
<input [(ngModel)]="name" #nameNgModel="ngModel" required>
```

- Inside code

```
export class AppComponent {
 name: string;
 @ViewChild("nameNgModel") nameNgModel: NgModel;

 save() {
  if(!this.nameNgModel.valid) {
   return;
  }
 }
}
```

Display Validation Messages

- Inside template

```
<span class="validation-message" *ngIf="nameNgModel.invalid && nameNgModel.touched">
  Name is required
</span>
```

Multiple Validation Messages

- Use the errors bag

```
<div *ngIf="nameNgModel.errors && (nameNgModel.dirty || nameNgModel.touched)">

 <span class="validation-message" *ngIf="nameNgModel.errors.required">
  Name is required
 </span>

 <span class="validation-message" *ngIf="nameNgModel.errors.maxlength">
  Too long
 </span>

</div>
```

errors is null when there are no errors

Display error messages only after the user typed something

Grouping

- A “common” form contains multiple inputs
- Each input has its own validation logic
- The submit button need to validate all inputs
- Therefore, Angular allows you to group multiple inputs under a single form tag
- The form can queried
  - It holds aggregation of all inputs

Form


```
<form #form="ngForm">
 <input [(ngModel)]="name" name="name" #nameNgModel="ngModel" required maxlength="5">

 <button type="submit" (click)="add()">Add</button>

 <div *ngIf="nameNgModel.errors && (nameNgModel.dirty || nameNgModel.touched)">
 <span class="validation-message" *ngIf="nameNgModel.errors.required">
  Name is required
 </span>

  <span class="validation-message" *ngIf="nameNgModel.errors.maxlength">
  Too long
 </span>
 </div>
</form>
```

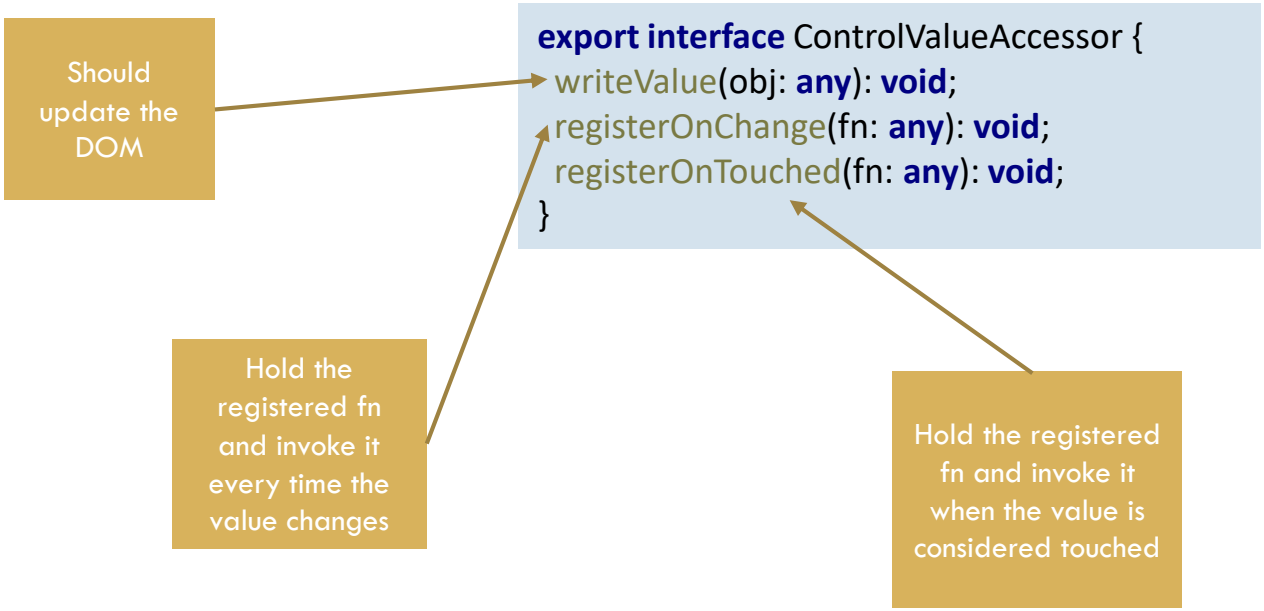
Import the ngForm directive and use its API

```
export class AppComponent {
 @ViewChild("form") form: NgForm;

 add() {
  if(!this.form.invalid) {
    return;
  }
 }
}
```

Custom component & ngModel

- By default ngModel cannot be attached to custom component
- A component should implement ControlValueAccessor



Implement Value Accessor


```
export const VALUE_ACCESSOR = {
  provide: NG_VALUE_ACCESSOR,
  useExisting: forwardRef(() => EmailComponent),
  multi: true
};

@Component({
  providers: [VALUE_ACCESSOR]
})
export class EmailComponent implements ControlValueAccessor {
  writeValue(obj: any): void {…
  }

  registerOnChange(fn: any): void {…
  }

  registerOnTouched(fn: any): void {…
  }
}
```

Must register the value accessor and point it to self

Grouping Inputs

- Use ngModelGroup to group multiple inputs into single value and single valid flag

```
<form #f="ngForm" (ngSubmit)="onSubmit(f)">
 <p *ngIf="nameCtrl.invalid">Name is invalid.</p>

 <div ngModelGroup="name" #nameCtrl="ngModelGroup">
  <input name="first" [ngModel]="name.first" minlength="2">
  <input name="last" [ngModel]="name.last" required>
 </div>

 <input name="email" ngModel>

 <button>Submit</button>
</form>
```

```
export class NgModelGroupComp {
 name = {first: 'Nancy', last: 'Drew'};

 onSubmit(f: NgForm) {
  console.log(f.value);  // {name: {first: 'Nancy', last: 'Drew'}, email: ''}
  console.log(f.valid);  // true
 }

 setValue() { this.name = {first: 'Bess', last: 'Marvin'}; }
}
```

Reactive Forms

- You create the form control model in code
- Angular binds the model to template elements
- Offers the following
  - No two way data binding
  - Can change validation functions on the fly
  - Manipulate the control model
  - Test easily
  - HTML is cleaner

Reactive Forms – Getting Started

- Import ReactiveFormModule
- form element is bound to [formGroup]
  - No ngForm
- Inputs are bound to formControlName
  - No ngModel
- Validation metadata moves into code

Reactive Forms


```
export class AppComponent {
 formGroup: FormGroup;

 constructor(builder: FormBuilder) {
  this.formGroup = builder.group({
    name: ["", [
       Validators.required
     ]
    ]
  });
 }

 add() {
  if(!this.formGroup.valid) {
    console.log("Not valid");

    return;
  }

  console.log("Saving ...");
 }
}
```

```
<form [formGroup]="formGroup">
 <input formControlName="name">

 <button (click)="add()">Add</button>
</form>
```

No Two way data binding

- Latest form value is stored inside the FormGroup
- It is updated automatically when DOM changes
- Use setValue in order to change value from code

Does the same thing

```
change() {
  this.formGroup.setValue({
    name: "XXX"
  });

this.formGroup.controls.name.setValue("XXX");
}
```

Change Control Model

- [formGroup] creates a live binding
- We can recreate the formGroup at runtime and all settings are reapplied

```
export class AppComponent {
 formGroup: FormGroup;

 constructor(private builder: FormBuilder) {
  this.formGroup = builder.group({
    name: ["", [
       Validators.required,
       Validators.maxLength(5)]]});
 }

 change() {
  this.formGroup = this.builder.group({
    name: ["", [
       Validators.required]]});
 }
}
```

Validation is easier

- Each FormControl has a list of validators
- Each validator is a simple function
- You can easily define and reuse validators

```
function validate(control:FormControl):{[key:string]:boolean} {
  if(control.value == "11111") {
    return null;
  }

  return {'xxx': true};
}
```

Subscribe

- You can subscribe to any change inside the FormGroup

```
export class AppComponent {
 formGroup: FormGroup;

 constructor(private builder: FormBuilder) {

  this.formGroup.valueChanges.subscribe(value => {
   console.log("valueChanges", value);
  });
 }

 add() {
  if (!this.formGroup.valid) {
   console.log("Not valid", this.formGroup.value);

   return;
  }

  console.log(this.formGroup.value);
 }
}
```

Subscribe to any change

Retrieve latest value

Summary

- Forms & Validation is based around the concept for control model
- Template driven creates the model from HTML
- Reactive driven creates the models from code
  - More effort
  - More flexible