Template Driven Forms

BUILDING WEB APPLICATIONS USING ANGULAR

Contents

- Form templates
- Events
- Template Reference Variables
- NgModel
- Feeding back to the user
- Form submission

Template-Driven Forms

- As it suggests, forms built using Angular's template syntax with form specific directives and techniques
- Suitable for building almost any form that may be required
  - Allows laying out of controls creatively, binding them to data, specifying validation rules, displaying validation errors, conditionally enabling or disabling specific controls, triggering built-in visual feedback, etc
- Creating a form usually consists of the following steps:
  - Creating a form component and template
  - Using ngModel to create 2-way data binding for reading and writing input-control values
  - Tracking the state and validity of form controls
  - Providing visual feedback using special CSS classes
  - Displaying validation errors and enabling/disabling form controls
  - Sharing information across HTML elements using template reference variables

Introduction

- The first step to creating a form is to import the Angular Forms module

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

@NgModule({
   imports: [FormsModule]
}) export class SomeModule
```

- This gives us access to the directives we need to build an interactive form

Form Template

- A form template is little more than HTML combined with Angular directives we are already partly familiar with

```
<form>
   <label for="firstname">First name</label>
   <input type="text" id="firstname" name="firstname" required>

   <label for="surname">Surname</label>
   <input type="text" id="surname" name="surname" required>

   <label for="age">Age</label>
   <input type="number" id="age" name="age">

   <button type="submit">Submit</button>
</form>
```

- There's nothing particularly 'Angular' about this form. Until we furnish it with some Angular directives.

Events

- Events are a critical part of any form. Focus, keypress and submit events (amongst others) are all integral to producing a good user experience on any form
- As seen previously we can react add event listeners in our templates, and pass the $event object back

```
<input type="number" id="age" name="age" (blur)="onBlur($event)">
```

- This can often be a bad idea though as it can result in tightly coupling our model to our view

```
onBlur(evt: FocusEvent) {
   if (parseInt((<HTMLInputElement>evt.target).value) < 18) {
      console.log('Under age');
   }
}
```

- We need to know about the HTML implementation in order to extract the right information

Template Reference Variables

- Template reference variables provide direct access to an element within the templates using a simple # notation

```
<input type="number" id="age" name="age" #age keypress="0">
```

- The template reference variable “age” now refers to the input element itself

```
{{age.value}}
```

NgModel

- ngModel allows us to create two way data binding between our form control components and the model

```
<input type="number" id="age" name="age" [(ngModel)]="user.age">
```

- Which we can use in the model or in the view (usually for diagnostic purposes)

```
{{user.age}}
```

- ngModel will implicitly attach an NgForm directive to the <form> tag if the input is used as part of a form

Form Validation

- HTML form validation can be added to forms using the usual HTML attributes
  - Angular uses directives to match these with validator functions in the framework
- When a value changes in a form control, validation is run by Angular
  - Generates a list of validation errors resulting in a INVALID status or null (VALID)
  - Can then use template variables by exporting ngModel to it and inspecting the control state

Form control state – Form Validation

- Angular tracks the state of our form and its controls through the NgModel
- Depending on the state of the control element (and the form as a whole) Angular will furnish the elements with various classes

State	True	False
Control has been focussed	ng-touched	ng-untouched
Control's value has been changed	ng-dirty	ng-pristine
Control's value is valid	ng-valid	ng-invalid

Form control state – Form Validation

- By adding CSS on these classes we can easily provide feedback to our users

```
input.ng-dirty.ng-invalid {
   border: 2px solid red;
}
```

Form control state – Form Validation

- We can also test the NgModel directive in our template using template reference variables

```
<input type="text" id="firstname" name="firstname" #firstname="ngModel" required >
<div *ngIf="firstname.dirty && firstname.invalid">Please provide a firstname</div>
```

- The NgModel directive has an “exportAs” property that is equal to “ngModel” and hence why we use that in the template reference variable.
- In this way we can hide/show error messages to our users very simply

Custom Validators

- Need to add a directive to the template (a further validation function is also needed – see next slide)
  - Example shows a Validator directive to recognise forbidden text within a control

```
// forbidden-string.directive.ts
@Directive({
   selector: ['appForbiddenString'],
   providers: [{provide: NG_VALIDATORS, useExisting:
        ForbiddentValidatorDirective, multi: true}]
   })
export class ForbiddenValidatorDirective implements Validator {
   @Input() forbiddenString: string;

   validate(control: AbstractControl): {[key: string]: any} {
      Return this.forbiddenString ? forbiddenStringValidator(new
      RegExp(this.forbiddenString, 'i'))(control) : null;
   }
}
```

  - Add the forbiddenString selector with the forbidden value to any input element to activate it

Form submission

- Submission can be handled through the ngSubmit directive

```
<form #addUserForm="ngForm" (ngSubmit)="onSubmit()>
```

- You'll likely want to prevent submission if the form isn't valid

```
<button type="submit" [disabled]="addUserForm.form.invalid">Submit</button>
```

- And remove the form once it's been submitted

```
<form #addUserForm="ngForm" (ngSubmit)="onSubmit()" *ngIf="!submitted">
```

Exercise

CREATING A FORM