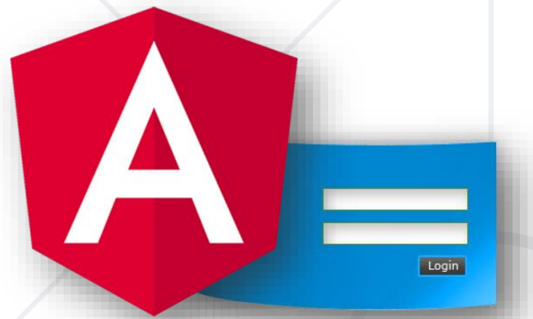
Directives and Forms

Creating Directives. Handling Forms.



SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Have a Question?

sli.do

#angular

Table of Contents

Directives

Manipulating the DOM in Angular

Directives Overview

- There are **three** types of **directives** in Angular
  - **Components** – directives with template
  - **Attribute** directives - change the **appearance** or behavior of an element, component or another directive (**ngStyle** and **ngClass**)
  - **Structural** directives - change the DOM **layout** by **adding** and **removing** DOM elements (*<b>ngIf</b> and *<b>ngFor</b>)

Directives Comparison

- **Attribute** Directives
- Look like HTML attributes
- Only affect/change the **element** they are **added to**
- Example - **ngStyle**, **ngClass**

- **Structural** Directives
- Have a leading *****
- Affect a **whole area** in the DOM
- Examples - ***ngIf**, ***ngFor**

Build a Simple Attribute Directive

- An attribute directive minimally requires building a controller class **annotated** with **@Directive**

```
import { Directive } from '@angular/core'
```

- Surround the **selector** with **square** brackets

```
@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
 constructor() { }
}
```

Import the directive in **declarations** array

Attach Styles to Referenced Elements

- Now **inject** the **referenced** element and **change** its background style

```
export class HighlightDirective implements OnInit {
 constructor(private el : ElementRef) {}

 ngOnInit() {
    this.el.nativeElement.style.backgroundColor = 'yellow';
 }
}
```

Warning - Use Renderer2

- It's not a good practice to **directly access** DOM elements via **ElementRef**
- Angular is **not limited** to run only on the browser (could run with service workers)
- Services Worker – environment where the DOM is **inaccessible**
- Use **Renderer2** to manipulate DOM elements

```
import { Renderer2 } from '@angular/core'
```

Renderer2 Usage

- Inject the renderer and access its **methods** to change the DOM

```
constructor( private renderer: Renderer2) {  }
ngOnInit() {
  this.renderer.setStyle(
    this.el.nativeElement,
    'background-color',
    'red'
  );
}
```

Respond to Events

- A directive can be more **dynamic** and **detect** user events

```
import { HostListener } from '@angular/core'
```

- Attach host **listeners** to handle different **DOM events**

```
@HostListener('mouseenter') onMouseLeave(e) {
  this.highlight('yellow');
}
@HostListener('mouseleave') onMouseLeave(e) {
  this.highlight('blue');
}
```

Using HostBinding

- Bind to DOM properties **without** Renderer

```
import { HostBinding } from '@angular/core'
```

```
export class BasicHighlightDirective {
  @HostBinding('style.backgroundColor')
    backgroundColor: string;

  highlight(color: string) {
    this.backgroundColor = color;
  }
}
```

Handling Forms

Template-Driven Forms

Forms Overview

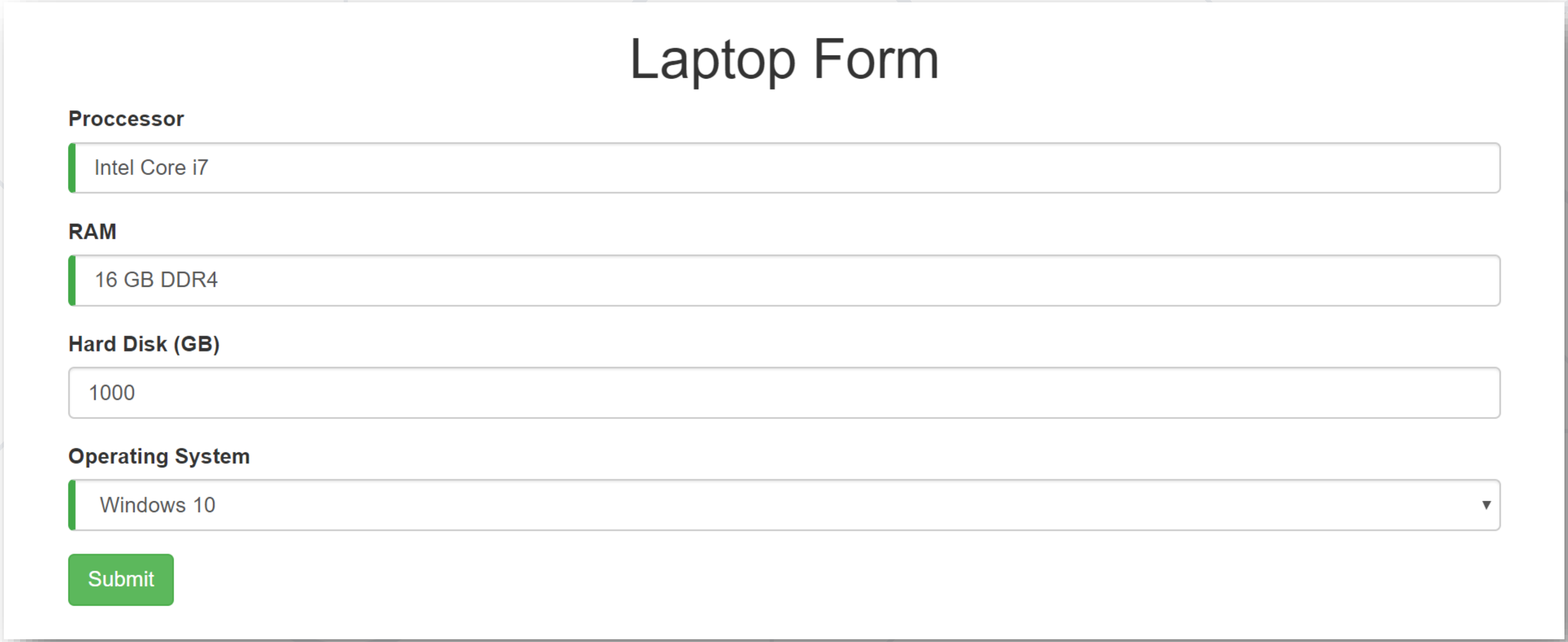
- Forms are the **mainstay** of **business** applications
- We use Forms to
  - Register/Log in
  - Submit a **help** request
  - **Place** an order
  - **Book** a flight and more
- Guide the user **efficiently** and **effectively** when creating forms

Template-Driven Forms

- Build a Form by writing **templates** using the Angular **template syntax**
  - Track **state** changes (**validity** of form controls)
  - Provide **visual** feedback **using** special **CSS** classes
  - Display **validation** errors when **needed**
  - Use **reference variables** to share information

Problem: Create a Template-Driven Form

- Create a **Template-Driven** Form looking like this



Laptop Form

Proccessor

Intel Core i7

RAM

16 GB DDR4

Hard Disk (GB)

1000

Operating System

Windows 10 ▾

Import Bootstrap

- Bootstrap is the most **popular** open-source **front-end** framework for **designing** web **sites** and web **apps**
- Install via **npm** and import it inside **angular.json**

```
"styles": [
  "node_modules/bootstrap/dist/css/bootstrap.min.css",
  "src/styles.css"
]
```

- Create **containers**, **form-groups**, **form-controls**, **style** buttons and errors

Introducing Forms Module

- Angular is **module based** and to handle Forms (**ngModel**, **ngSubmit**, **ngForm**) we need **Forms Module**
- Import the following in **app.module.ts**

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

@NgModule({
  imports: [
   BrowserModule,
   FormsModule
  ]
})
export class AppModule {  }
```

Create Form Component

- An **Angular** form has **two** parts
  - An HTML-based **template**
  - Component **class** to **handle** data

```
@Component({…})
export class LaptopFormComponent {
  operatingSystems: string[] = [
    'Windows 10',
    'Linux',
    'Mac OS'
  ];
}
```

Initial HTML Template

```html
<div class="container">
  <h1>Laptop Form</h1>
  <form>
    <div class="form-group">
      <label for="processor">Proccessor</label>
      <input type="text" class="form-control" id="processor"
      required>
    </div>
    <div class="form-group">
      <label for="ram">RAM</label>
      <input type="text" class="form-control" id="ram"
      required>
    </div>
    <div class="form-group">
      <label for="hardDisk">Hard Disk (GB)</label>
      <input type="number" class="form-control" id="hardDisk">
    </div>
  </form>
</div>
```

Initial HTML Template

```
<div class="form-group">
  <label for="os">Operating System</label>
    <select class="form-control"
      id="os"
      required>
        <option *ngFor="let os of operatingSystems"
               [value]="os">{{os}}</option>
    </select>
</div>

<button type="submit" class="btn btn-success">Submit</button>
```

The NgModel Directive

- We need to **display**, **listen** and **extract** data at the same time
  - This is done by using the **ngModel** directive

```
<input type="text"
  class="form-control" id="processor"
  ngModel />
```

- The following directive will **not** work without a **name attribute**

```
<input name="processor"/>
```

The NgForm Directive

- Declare a **template** variable to the form

```
<form #f="ngForm">
```

- Angular **automatically** attaches an **NgForm directive**
- The NgForm directive adds **additional** features
  - **Monitors** properties
  - **Validates** properties
  - It holds a **valid** property which is **true** only if **all controls** are valid

Access the Local Reference

- Use **@ViewChild** to access the local reference

```
@Component({…})
export class LaptopFormComponent implements AfterViewInit {
  @ViewChild('f') form: NgForm

  ngAfterViewInit() {
    console.dir(this.form);
  }
}
```

Submit a Form

- To submit a form bind **ngSubmit** event property to form component's **onSubmit** method

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

- The **onSubmit** method should send the **control values** directly to an **API** through a **service** of some sort

```
onSubmit() {
 const content = this.form.value;
  // Send model to API
}
```

Tracking Form State

- The **NgForm** Directive
  - Tracks if the user **touched** the control
  - Tracks if the user **changed** the control
  - Tracks if the control is **valid**
- The directive doesn't just track state
  - It can **update** the control with special **Angular CSS** classes
  - Leverage those class names to **change** appearance

Track Control State

State	Class if true	Class if false
The control was visited	ng-touched	ng-untouched
The control's value was changed	ng-dirty	ng-pristine
The control is valid	ng-valid	ng-invalid

Add Custom CSS for Visual Feedback

- You can mark **required** fields and **invalid** data at the **same** time with a **colored** bar on the **left** of the input box

styles.css

```
input.ng-valid {
  border-left: 5px solid #42A948; /* green */
}

input.ng-invalid.ng-touched {
  border-left: 5px solid #A94442; /* red */
}
```

Add Validation

- Add **HTML 5 attributes** to input fields for validation
- Angular tracks most attributes and **changes** the **state** depending on what the user enters

```
<input type="text" class="form-control"
 id="processor"
 required
 minLength="5"
 ngModel
 name="processor">
```

List of Validators/Third-party Validators

- Angular is shipped with the following validators
  - https://angular.io/api/forms/Validators
- For template-driven forms you will need directives
  - https://angular.io/api?type=directive
- There are multiple npm packages for custom validators
  - https://www.npmjs.com/package/ng5-validation

Outputting Error Messages

- The user should know **exactly** what went wrong
- Leverage the control's **state** to reveal a helpful **message**
- Add a template reference **variable** in the **input**

```
<input type="text" class="form-control"
 id="processor"
 required
 ngModel
 name="processor"
 #processor="ngModel">
```

Outputting Error Messages

- Create a div and **display** it **only** when the control state is **invalid**
- Use the reference **variable** to **check** the state
- Add a **helpful** message **inside** the div

```
<div *ngIf="processor.invalid && processor.touched"
alert alert-danger">
   Processor is required!
</div>
```

Form Overall Validity

- We can **bind** the form's overall **validity** using the **reference variable** declared in the **<form>** tag
- **Block** the submit button in case a control has **invalid** state

```
<button type="submit" class="btn btn-success"
[disabled]="f.invalid">
 Submit
</button>
```

Two-way Data Binding

- Instantly react to any changes using **two-way** data binding

```
<input type="text" class="form-control"
 id="processor"
 required
 [(ngModel)]="laptop.processor"
 name="processor"
 #processor="ngModel">
```

```
constructor() {
 this.laptop = new Laptop()
}
```

The NgModelGroup Directive

- Group similar input fields using **ngModelGroup**
- Useful for input fields that have the **same validation**
  - Password/Confirm password

```
<div
 ngModelGroup="passData"
 #passData="ngModelGroup">
```

```
<div *ngIf="passData.invalid && passData.touched">
  Both passwords must be valid!
</div>
```

Setting and Patching Form Value

- Use **setValue()** or **patchValue()** to change the form from **inside** the component or add default values

```
changeInput() {
  this.laptopForm.form.patchValue({
    ram: '16 GB'
    processor: 'Intel Core i7'
  });
}
```

Resetting the Form

- After a form is submitted resetting is necessary to **clear** all input fields and **reset** the **track state**

```
onSubmit() {
  const body = this.form.value;
  // Send body to an API
  this.form.reset();
}
```

Handling Forms

Reactive Forms

Reactive Forms Overview

- There are some scenarios that can't be **resolved** using template-driven forms
  - **Using** Form Arrays
  - Dynamic Form **Creation**

Reactive Forms Module

- In order to **use** reactive forms, we need the **Reactive Forms Module**

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

- Now we have **access** to all the needed **directives**
  - **formGroup**
  - **formControl** and **formControlName**
  - **formGroupName**
  - **formArrayName**

The Component Class

- The component class will create **instances** of **FormGroup** and **FormControl** that will bind later in the template
- The core **idea** is to **transfer** most of the logic from the **template** inside the **component** class

```
import { FormGroup, FormControl } from '@angular/forms'
```

```
this.laptopForm = new FormGroup({
  processor : new FormControl('Intel Core i7'),
  ram : new FormControl('16 GB DDR4')
});
```

The Template

- In the **template** we have to **mark** the main **formGroup** and after that add **formControlName** to each form control

```
<form (ngSubmit)="save()" [formGroup]="laptopForm">
```

```
<input type="text" class="form-control" id="processor"
 required
 formControlName="processor">
```

The **name** of the **key** instance

Accessing Form Model Properties

- Two ways to **access** the **properties** of the form model

```
laptopForm.controls.processor.valid
```

```
laptopForm.get('processor').valid
```

- The idea is to **shorten** the **template** and **transfer** such logic in the **component** when using **reactive forms**

Using Form Builder

- Use **FormBuilder** service to avoid create **instances** of **FormGroup** and **FormControl** name

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

- Inject it **into** the constructor

```
constructor(private fb : FormBuilder) { }
```

```
this.laptopForm = this.fb.group({
  processor : 'Intel Core i7',
  ram : '16 GB DDR4'
});
```

Validation

- In reactive forms we can add validation more **dynamically** based on user **action**
- We can **adjust** rules at **runtime**
- We can create **custom** validators
  - Custom validators excepting **parameters**
- **Cross-field** validations and more

Setting Up Build-in Validation

- Defining our **FormGroup** with a **FormBuilder** allows us to add an **array** of validations using the **Validators** class

```
this.laptopForm = this.fb.group({
  processor : [
   'Intel core i7', [
     Validators.required,
     Validators.minLength(10)
    ]
  ]
});
```

Adjust the Template

- The **formGroup** directive has an **errors** property which can be used to **show** errors only when **needed**

```
<div *ngIf="(laptopForm.get('processor').touched
|| laptopForm.get('processor').dirty)
&& laptopForm.get('processor').errors" class="alert alert-danger">
<span *ngIf="laptopForm.get('processor').errors.required">
  Processor is required!
</span>
<span *ngIf="laptopForm.get('processor').errors.minlength">
  Processor should be at least 10 symbols long!
</span>
</div>
```

Watching and Reacting to Changes

- Using **Reactive Forms**, we have the ability to **watch** and **react** to changes on form **groups** and form **controls**
- Whenever a **value** of an input **changes** we can **subscribe** to that event and handle the **observable**

```
this.laptopForm.get('os')
.valueChanges
.subscribe(console.log);
```

Reactive Transformations Example

- Import **throttleTIme** from the following library

```
import { throttleTime } 'rxjs/operators';
```

- Attach the **throttleTIme** function to a form control's **valueChanges** event

```
processorControl.valueChanges
.pipe(throttleTime(1500))
.subscribe(value => {
   console.log(value);
});
```

Summary

- There are three types of Directives
  - Components, Structural, Attribute
- There are two ways to handle forms in Angular
  - Template-driven Forms (two-way binding)
  - Reactive Forms (more dynamic approach)
- Directives are integrated into Form Modules

Questions?

Software University

SoftUni

Software University

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

Finance Academy

SoftUni Diamond Partners

Software University

SmartIT

Coca-Cola HBC Bulgaria

SOFTWARE GROUP

DXC TECHNOLOGY

createX

Postbank

Решения за твоето утре

SUPER HOSTING .BG

DRAFT KINGS

VIVACOM

INDEAVR

Serving the high achievers

AMBITIONED

PHAR VISION

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, softuni.org
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity

License