Components

BUILDING WEB APPLICATIONS USING ANGULAR

Contents

Introduction

- Components are the fundamental building blocks of the interface of Angular applications
- Angular applications are a tree of Components
- A component is made up of:
  - A class which contains the application logic
  - A template which contains the instructions on how to construct the UI
- A component must belong to an NgModule
- The @Component() decorator declares a class to be a Component and tells Angular how to construct, process and used at runtime.

Creating a new component

- Creating a new component is as straight forward as declaring a class with an @Component decorator
- The most commonly used configuration options used within the decorator:
  - selector – the CSS selector that identifies this component in a template. Best practice: this should be an element
  - styles/styleUrls – inline CSS or links to external stylesheets applied to the view
  - template/templateUrl – the template for the view

Data Binding

- Components act as the Controller / ViewModel, where as templates act as the View
- Data binding provides a mechanism for coordinating what the user sees, with data in the application
- Angular has multiple binding types depending on use:
  - One-way (data source to view target)
  - One-way (view target to data source)
  - Two-way

One-way data binding (data source to view target)

- {{expressions}} inserts the result of the template expression at this point in the template
- Property bindings set the DOM element's property to the result of the expression, but only if it's not the name of a known property directive (as with ngClass)

```
{{welcomeMessage}}
<img [src]="imageUrl">
<qa-course [course]="currentCourse"></qa-course>
<div [ngClass]="{remote: isRemote}"></div>
```

- Gotcha! If you omit the [square brackets] then Angular will not evaluate the expression and instead treats it as a string

```
<img src="imageUrl"> <!-- almost certainly a 404 -->
```

One-way data binding (data source to view target)

- What about when an attribute doesn't have a property counterpart, as with aria properties?
- Attribute bindings!

```
<ol role={{role}}> <!-- ERROR: There is no role property to set -->
<ol [attr.role]="role">
```

- Class bindings can be used to manage classes, but it's all or nothing

```
<button [class]="myClasses">…</button>
```

- Individual classes can be managed in an on/off state through a Boolean evaluation

```
<button [class.myClass]="true">…</button>
```

```
<button [class.myClass]="!true">…</button>
```

- This is fine for individual classes but multiple classes should be managed with NgClass

One-way data binding (data source to view target)

- Similarly to class bindings, inline styles can be bound

```
<p [style.color]="selected ? 'red' : 'black'">…</p>
```

- Those with units can be set also

```
<p [style.fontSize.em]="selected ? 2 : 1">…</p>
```

- Again, this is fine for individual styles – but when managing multiple styles we should use NgStyle

One-way data binding (view target to data source)

- Not all users sit passively staring at our websites. Sometimes they like to interact. In these cases data needs to flow from the element to the component

```
<button (click)="onSubmit()" type="submit">Submit</button>
<!-- or canonical form-->
<button on-click="onSubmit()" type="submit">Submit</button>
```

- Angular will check for a matching event property on a known directive first

```
<button (qaClick)="clickEvent=$event" type="submit">Submit</button>
```

- When an event is raised, information about that event is stored in $event, whose shape is dependent on the target event
  - If the target is a DOM element then $event is a native DOM element event

Two-way binding (view target to data source)

- For when you want to display data from the component and update when the user changes
- Combines the syntax of Property [] and Event () binding – giving us: Banana in a box [()] syntax

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

Built-in Directives

- Angular comes packed with built-in directives to help us build our own components
- AngularJS shipped with over 70 built-in directives!
- Angular has attribute directives
  - NgClass
  - NgStyle
- And structural directives
  - NgIf
  - NgFor
  - NgSwitch

NgClass

- NgClass provides a mechanism for us to toggle multiple classes based on our data source
- We need to point NgClass at an object which holds a series of class:boolean entries, these define what class names should be applied, if their Boolean value is true
- We can change these values and see the classes added/removed

```
currentClasses = {
  selected: false,
  valid: false
};
```

```
<section [ngClass]="currentClasses">
<button (click)="currentClasses.valid=true">Validate</button>
```

NgStyle

- NgStyle works similarly to NgClass, with a style:value control object
- With NgStyle the value of each style should resolve to an appropriate value for that style (not true or false as with NgClass)
- We then map the control object to the NgClass directive in the template

```
currentStyles = {}

setCurrentStyles() {
   this.currentStyles = {
      backgroundColor: this.selected ? 'lightgreen' : 'red',
      border: this.valid ? '2px solid green' : '2px solid lightpink'
   }
};
```

```
<section [ngStyles]="currentStyles">
<button (click)="valid=!valid; setCurrentStyles()">Validate</button>
```

NgIf

- Removes elements from the DOM if the expression returns a truthy value

```
<article *ngIf="viewArticle">…</article>
<button (click)="viewArticle=!viewArticle">Toggle Article</button>
```

- If you just want to show/hide something then styles may be more suitable – this will keep the element in memory and Angular may continue to check for changes.

NgFor

- NgFor allows us to build repetitive areas of the template based on the contents of collections

```
<ul>
   <li *ngFor="let user of users">{{user.username}}</li>
</ul>
```

- If you just want to show/hide something then styles may be more suitable – this will keep the element in memory and Angular may continue to check for changes.
- The syntax "let user of users" is called microsyntax and is not a template expression
- You can also include:
  - Index
  - trackby

NgSwitch

- NgSwitch is actually a collection of directives that act together to create something analogous to the JavaScript switch statement
- We use it to display an element based on a switch condition

```
<div [ngSwitch]="instructor">
   <p *ngSwitchCase="'chris'">Chris enjoys cycling!</p>
   <p *ngSwitchCase="'ed'">Ed just looooves React!</p>
   <p *ngSwitchCase="'edsel'">Edsel loves a good group selfie</p>
   <p *ngSwitchCase="'dave'">Dave puts Star Wars references where ever he can</p>
   <p *ngSwitchDefault>Please select an instructor</p>
</div>
```

Input Properties

- Often we want to be able to bind a component property to a value, either hard coded or referencing a parent component's property value. This is the job of Input Properties.
- Input properties are enabled through the use of the @Input decorator which tells Angular that this property will be the target of a binding. If you don't use this decorator you will get a template parse error should you try to bind to it.

```
export class UserComponent {
        @Input()
        user: string;
}
```

- We create the binding in the same way we did earlier in this chapter, this time we are binding to our own defined property

```
<app-user [user]="instructor"></app-user>
```

Input Property Alias

- Sometimes we may wish to expose a different name to the one we use internally in the class. This is done by aliasing the input property by passing a string into the decorator

```
export class UserComponent {
        @Input('user')
        person: string;
}
```

```
<app-user [user]="instructor"></app-user>
```

- We can now use 'person' in the class, whilst in the template we continue to use 'user'

Output Properties

- Whilst Input properties allow us to bind data to a child component's properties, Output properties allow us to inform a parent when a change within the child component has taken place.
- Initially we set up an output property similarly to an input property, using an Output decorator. However we need this property to be an instance of EventEmitter, imported from @angular/core

```
export class UserComponent {
        @Output()
        vote = new EventEmitter<number>();
}
```

- We then bind to this output property in the parent component

```
<app-user (vote)="handleVote($event)"></app-user>
```

Output Properties

- Our parent component then needs the handleVote method which receives the $event

```
handleVote(event) {
    alert(`A vote has been received: ${event}`); //A vote has been received: 1
}
```

- Our child component then simply emits events when it wants to notify the parent of a change

```
upvote(person) {
    this.vote.emit(1);
}

downvote(person) {
    this.vote.emit(-1);
}
```

Output Property Alias

- Output properties may have aliases just as Input properties do. You can add them in the same way.

```
export class UserComponent {
        @Output('change')
        vote = new EventEmitter<number>();
}
```

- We then bind to 'change' but use vote internally as before

```
upvote(person) {
    this.vote.emit(1);
}

downvote(person) {
    this.vote.emit(-1);
}
```

```
<app-user (change)="handleVote($event)"></app-user>
```

Lifecycle hooks

- Components are created and rendered, have their child components created and rendered, and then destroyed by Angular as the application is used
- Lifecycle hooks are functions with special names that we can declare within our components in order to add behaviour at these moments in time

ngOnChanges

- Called whenever an input data-bound property is (re)set. It is passed a SimpleChanges object consisting of property names and their SimpleChange object, which holds the following properties:
  - previousValue: any
  - currentValue: any
  - firstChange: boolean
  - isFirstChange() : Boolean – checks whether the new value is the first value assigned
- This is the first lifecycle hook to be called and the first time the data-bound properties are available to us. Unlike ngOnInit though this method will be called every time those properties change.

ngOnInit

- Called after Angular first display the data-bound properties and sets the component's input properties
- This method is only called once for the whole life of the component, and happens immediately after ngOnChanges
- We should place any significant initialisation logic for the component in this method, leaving the constructor as simple as possible (ideally just property assignments)

ngDoCheck

- This method is called during **every** change detection run, immediately following ngOnChanges (and ngOnInit if this is the first run)
- We should place logic here to detect any changes that Angular can't or won't detect on its own

ngAfterContentInit

- Another hook that is only ever called once, following the first ngDoCheck
- We should place code here to respond to Angular projecting external content into the component's view

ngAfterContentChecked

- This is called after every ngDoCheck (after ngAfterContentInit if it is the first run)
- We should use this method to respond once Angular has checked the content projected into the component

ngAfterViewInit

- Our 3rd once-only lifecycle hook, this gives us opportunity to respond once Angular has initialised the component's view and child views.
- This hook is called once after ngAfterContentChecked

ngAfterViewChecked

- Called once Angular has checked the component's views and child views, this hook runs after every ngAfterContentChecked (after ngAfterViewInit if the first run)

ngOnDestroy

- Our final once-only lifecycle hook this method is called just before Angular destroys the component. This enables us to perform any cleanup required to avoid memory leaks – common tasks will be unsubscribing from Observables and detaching event handlers.

Exercise

BUILDING A COURSES INFORMATION PAGE