Advanced Component in Angular

Index

Styling components (with encapsulation)

- ViewEncapsulation: Emulated
- ViewEncapsulation: Native
- ViewEncapsulation: None

Creating a Popup: Referencing and Modifying Host Elements

- An host element is an element to which the directive or component is bound
- For example, we can have a Popup directive that will attach behavior to its host element which will display a message when clicked
- Make a difference between Components and Directives:
    - Component: Components are directives and Components always have a view
    - Directives: Directives may or may not have a view

Creating a Popup: Popup Structure

- receive the message attribute from the host
- be notified when the host element is clicked

```
@Directive({
    selector: '[popup]'
})
export class PopupDirective {
    constructor() {
        console.log('Directive bound');
    }
}
```

Creating a Popup: Attach the Directive

```
@Component({
    selector: 'app-popup-demo',
    template: `
    <div class="ui message" popup>
        <div class="header">
            Learning Directives
        </div>

        <p>
            This should use our Popup directive
        </p>
    </div>
    `
})
export class PopupDemoComponent1 {}
```

Creating a Popup: Using ElementRef

Example of using elementRef:

```
@Directive({
    selector: '[popup]'
})
export class PopupDirective {
    constructor(_elementRef: ElementRef) {
        console.log(_elementRef);
    }
}
```

Creating a Popup: Do something when the host is clicked

- In this case is necessary the HostListener decorator: this allows a directive to listen to events on its host element

The new code for the directive:

```
@Directive({
    selector: '[popup]'
})
export class PopupDirective {
    @Input() message: String;

    constructor(_elementRef: ElementRef) {
        console.log(_elementRef);
    }

    @HostListener('click') displayMessage(): void {
        alert(this.message);
    }
}
```

The new code for the component:

```
@Component({
    selector: 'app-popup-demo',
    template: `
        <div class="ui message" popup
            message="Clicked the message">

            <div class="header">
                Learning Directives
            </div>

            <p>
                This should use our Popup directive
            </p>
        </div>

        <i class="alarm icon" popup
            message="Clicked the alarm icon"></i>
    `
})
export class PopupDemoComponent3 {}
```

Creating a Popup: Adding a Button using exportAs

```
@Directive({
    selector: '[popup]',
    exportAs: 'popup', // <-- this is the important step
})
export class PopupDirective {
    @Input() message: String;

    constructor(_elementRef: ElementRef) {
        console.log(_elementRef);
    }

    @HostListener('click') displayMessage(): void {
        alert(this.message);
    }
}
```

```
template: `
    <div class="ui message" popup #popup1="popup"
        message="Clicked the message">
        <div class="header">
            Learning Directives
        </div>

        <p>
            This should use our Popup directive
        </p>
    </div>

    <i class="alarm icon" popup #popup2="popup"
        message="Clicked the alarm icon"></i>
`
```

```
<div style="margin-top: 20px;">

    <button (click)="popup1.displayMessage()" class="ui button">
        Display popup for message element
    </button>

    <button (click)="popup2.displayMessage()" class="ui button">
        Display popup for alarm icon
    </button>

</div>
```

Creating a Message Pane with Content Projection

- Sometimes when we are creating components we want to pass inner markup as an argument to the component.
- This technique is called content projection
- The idea is that it lets us specigy a bit of markup that will be expanded into a bigger template

Our goal is: make this message

```
<div message header="My Message">
    This is the content of the message
</div>
```

rendered to:

```
<div class="ui message">
    <div class="header">
        My Message
    </div>

    <p>
        This is the content of the message
    </p>
</div>
```

We have two challenges:

- change the host element

  to add the ui and message CSS classes
- add the div's contents to a specific place in our markup

To solve the first problem we will use the

```
@HostBinding('attr.class') cssClass = 'ui message'
```

This decoration in the component tells angular that we want the value of cssClass to be kept in sync with the host's attribute class.

To solve the second problem, we will use the:

- Include the original host element children in a specific part of a view. To do that, we use the
- ng-content directive

The complete component:

```
@Component({
    selector: '[app-message]',
    template: `
        <div class="header">
            {{header}}
        </div>
        <p>
            <ng-content></ng-content>
        </p>
    `
})
export class MessageComponent implements OnInit {
    @Input() header: string;
    @HostBinding('attr.class') cssClass = 'ui message';

    ngOnInit(): void {
        console.log('header', this.header);
    }
}
```

Lifecycle Hooks

Lifecycle hooks are the way Angular allows you to add code that runs before or after each step of the directive of component lifecycle

- Most used lifecycle hooks:
  - OnInit
  - OnDestroy
  - DoCheck
  - OnChanges

Lifecycle Hooks (2)

- Other lifecycle hooks:
    - AfterContentInit
    - AfterContentChecked
    - AfterViewInit
    - AfterViewChecked

OnInit and OnDestroy hook

- The OnInit hook is called when your directive properties have initialized, and before any of the child directive propeorties are initialized
- The OnDestroy hook is called when the directive instance is destroyed

OnChanges hook

- The OnChanges hook is called after one or more of our component properties have been changed.
- The ngOnChanges method receives a parameter which tells which properties have changed.

DoCheck hook

- in order to evaluate what changed, ANgular provides differs. Differs will evaluate a given property of your directive to determine what changed.

There are two types of built-in differs: iterable differs and key-value differs:

Iterable differs

- Iterable differs should be used when we have a list-like structure and we're only interested in knowing things that were added or removed from that list

Key-value differs

- Key-value differs should be used for dictionary-like structure, and work at the key level. This differ will identitfy changes whan a new key is added, when a key removed and when the value of a key changed.

AfterContentInit, AfterViewinit, AfterContentChecked and AfterViewChecked

- The AfterContentInit hook is called after OnINit, right after the initialization of the content of the component or directive has finished
- The AfterContentChecked works siomilarly, but it's called after the directive check has finished. The check, in this contenxt , is the change detection system check
- AfterViewInit and AftwareViewChecked are triggered right after the content ones above, right after the view has been full initialized. Those two hooks are only applicable to components, and not to directives

Advanced Templates

- Template elements are special elements used to create views that can be dynamically manipulated
- In order to make working with templates ismoler, Angular provides some sytactic sugar to create templates, so we often don't create them by hand

Change Detection

- As a user interacts with our app, data (state) changes and our app needs to respond accordingly.
- In order to make the view react to changes to components state, Angular uses change detection

What trigger changes in a component's state?

- User interaction
- HTTP request
- Timer

But, how does Angular become aware of these changes?

Component Change Detector

- Each cmoponent gets a change detector
- Every Angular app is made of a component tree. For each component on our tree, a chenage detector is created and so we end up with a tree of change detectors
- When one of the components change, no matter where it is in the tree, a change detection pass is triggered for the whole tree.
- Due to a number of optimizations, change detection process is surprisingly fast.

Customizing Change Detection

- There are times that the built-in or default change detection mechanism may be overkill.
- In these cases, Angular provides mechanisms for configuring the change detection system so that you get very fast performance.