ADVANCED COMPONENTS

Ori Calvo, 2017

oric@trainologic.com

https://trainologic.com

Objectives

- Review advanced details related to building components
- Dynamic component creation
- Lifecycle hooks
- Content projection
- Accessing the DOM
- More …

View Encapsulation

- For every component Angular is aware of its template + styling
- Thus, Angular is capable of “fixing” both and make them more encapsulated
- The effective CSS + HTML is a bit different than the one you write
- Be prepared for performance penalty since Angular needs to parse both CSS & HTML → Use AOT

View Encapsulation

- Angular implementation for a Shadow DOM way of thinking

```
<div class="buttons">
    <button>Refresh</button>
</div>

<my-contact-list></my-contact-list>
```

```
<div _ngcontent-c0="" class="buttons">
    <button _ngcontent-c0="">Refresh</button>
</div>

<my-contact-list _ngcontent-c0="" _nghost-c1="">
</my-contact-list>
```

```
button {
   background-color: red;
}
```

```
button[_ngcontent-c0] {
   background-color: red;
}
```

Every element is attached with unique attribute and CSS is fixed with the same unique name

Styling the host element

- Assuming a component named my-app
- The following definition does not work

```
my-app {
    display: flex;
    flex-direction: row;
}
```

- my-app is considered a child element not the host element itself

Styling the host element

- We can use the standard :host CSS syntax

```
:host {
   display: flex;
   flex-direction: row;
}
```

- Angular transforms it to the following definition

```
[_nghost-c0] {
   display: flex;
   flex-direction: row;
}
```

Adding CSS class to host element

- There are cases where :host is not enough
  - For example, attaching 3rd party CSS class
- There is no way to do that through the HTML ☹
- Use @HostBinding instead

```
export class AppComponent {
   @HostBinding("class.external") external: boolean = true;
}
```

```
:host(.external) {
   background-color: red;
}
```

Must be true, else, the CSS class is not injected

ViewEncapsulation.None

- No CSS encapsulation
- Angular just injects the CSS into the head
- You cannot use :host

```
@Component({
  selector: "my-app",
  templateUrl: "./app.component.html",
  styleUrls: ["./app.component.css"],
  moduleId: module.id,
  encapsulation: ViewEncapsulation.None,
})
export class AppComponent {
}
```

This is the trick

ViewEncapsulation.Native

- Makes Angular use the browser's native support
- Has poor browser support (Mostly Chrome)
- No styles are written to the document head
- Styles reside inside the component template

No transformation over the CSS since :host is assumed to be natively supported by the browser

```
<my-app>
  <#shadow-root>
    <style>
      :host {
        display: flex;
        flex-direction: column;
      }
    </style>
    <h1>Hello Angular</h1>
  </#shadow-root>
</my-app>
```

/deep/

- A parent component may want to override some default stylings for its child component
- CSS encapsulation prevent that by default
- Use /deep/ syntax

Omitting :host creates a "plain" global CSS rule

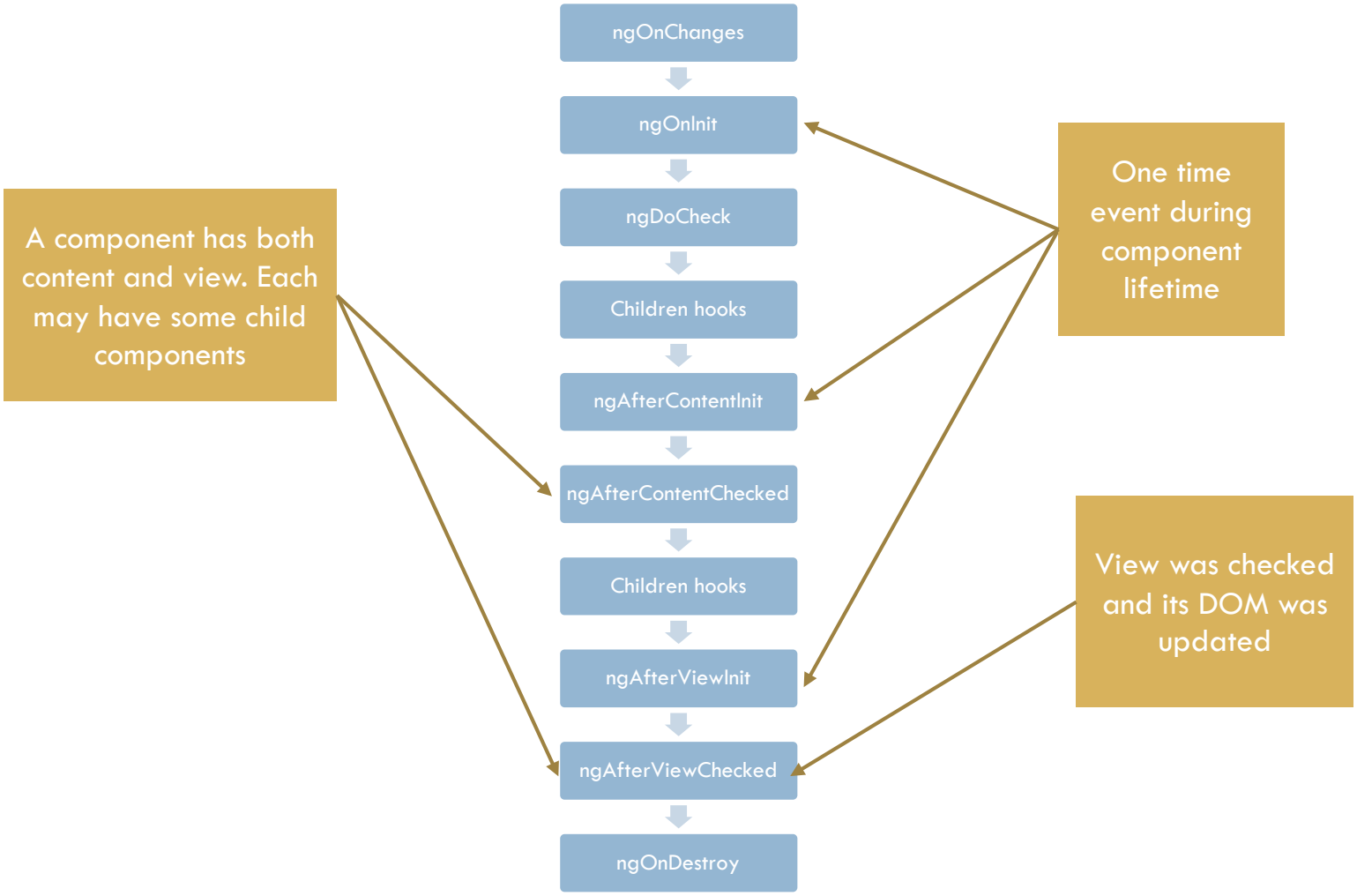
```
:host {
    display: flex;
    flex-direction: column;
}

:host /deep/ button {
    background-color: red;
}
```

Lifecycle Hooks

- Just like ASP.NET …
- Each component is notified several times by Angular during its lifetime
- We use the lifecycle hooks/events to customize component default behavior

Lifecycle Hooks



ngOnChanges

- Angular invoke this function only when one of the component's inputs changed
- The hook is not executed per input but rather after all inputs were updated by Angular
- A good place to update internal state that is derived from the inputs

ngOnChanges

Internal state

```
export class ContactComponent {
  @Input() contact: Contact;
  fullName: string;

  ngOnChanges() {
    this.fullName = this.contact.id + ": " + this.contact.name;
  }
}
```

```
<span>{{contact.name}}</span>
<span>{{fullName}}</span>
```

Is invoked whenever the contact input changes

contact.component.html

ngOnChanges – Be aware

- ngOnChanges is invoked as part of Angular change detection
- Angular executes simple change detection comparison
- The input "shallow" value is compared. Whether it's a value type or a reference type
- It means that a deep change inside an input <u>does not trigger</u> ngOnChanges

React to deep change - ngDoCheck

- ngDoCheck is always executed
- Even if no input was changed
- Use the method to update internal state
- Must be super efficient implementation
  - At your own risk …

```
export class ContactComponent {
  @Input() contact: Contact;
  fullName: string;

  ngDoCheck() {
    this.fullName = this.contact.id + ": " + this.contact.name;
  }
}
```

```
<span>{{contact.name}}</span>
<span>{{fullName}}</span>
```

React to deep change - Getter

- If you are willing to execute a calculation "all the time" then you may just use ES5 getter

```
export class ContactComponent {
  @Input() contact: Contact;

  get fullName(): string {
    return this.contact.id + ": " + this.contact.name;
  }
}
```

```
<span>{{contact.name}}</span>
<span>{{fullName}}</span>
```

Must be super efficient, else, you might hurt the performance of the whole application

React to deep change - Immutability

- Clone the whole input before changing it
- Thus the reference changes → Angular detects the change easily → ngOnChanges is invoked → Internal state can be updated

```
export class ContactListComponent {
  contacts: Contact[];

  constructor() {
    this.contacts = [
      {"id": 1, "name": "Ori"},
      {"id": 2, "name": "Roni"},
    ];
  }

  change() {
    this.contacts[1] = Object.assign({}, this.contacts[1], {
      name: this.contacts[1] + "X"
    });
  }
}
```

ngAfterContentChecked & ngAfterViewChecked

- Querying to the DOM is always tricky inside Angular
- You must query the DOM after it was updated
- ngAfterContentChecked – The content was dirty checked and its DOM was updated
- ngAfterViewChecked – The same logic but this time for the view

Accessing the DOM

- Usually there is no need to access the DOM directly when implementing components
- In case you still need it you may inject an ElementRef

```
constructor(private elementRef: ElementRef) {
  const dom = elementRef.nativeElement;

  this.button = document.createElement("button");
  this.button.innerText = "Click me";
  this.onClickHandler = this.onClick.bind(this);
  this.button.addEventListener("click", this.onClickHandler);

  dom.append(this.button);
}
```

Are you sure you want to go back to those ugly days ?

Accessing the DOM

- Angular can be executed under NodeJS or under web worker
- In that case ElementRef.nativeElement is undefined
- You should write your code with special care and guard against non browser platforms

```
export class AppComponent {
    constructor(@Inject(PLATFORM_ID) private platformId) {
        if(isPlatformBrowser(this.platformId)) {
            console.log("Running under browser");
        }
    }
}
```

Accessing Child Component


```
export class AppComponent {
   @ViewChild("clock1") clock1: ClockComponent;
   @ViewChild("clock2") clock2: ClockComponent;
   showClocks: boolean;

   toggle() {
      this.showClocks = !this.showClocks;
   }

   ngAfterViewChecked() {
      console.log("ngAfterViewChecked");

      console.log(this.clock1);
      console.log(this.clock2);
   }
}
```

clock1 & clock2 automatically change when toggling showClocks

```
<div *ngIf="showClocks">
   <my-clock #clock1></my-clock>
   <my-clock #clock2></my-clock>
</div>
```

Accessing Child Component

- You may access child component according to its Type

```
export class AppComponent {
   @ViewChild(ClockComponent) clock: ClockComponent;
   showClock: boolean;

   toggle() {
      this.showClock = !this.showClock;
   }

   ngAfterViewChecked() {
      console.log("ngAfterViewChecked");

      console.log(this.clock);
   }
}
```

- Angular also supports @ContentChild

Accessing a List of child components

- Use @ViewChlidren/@ContentChildren

```
export class AppComponent {
  @ViewChildren(ClockComponent) clocks: QueryList<ClockComponent>;

  showClock: boolean;
  counter: number;

  ngAfterViewInit() {
    this.clocks.changes.subscribe((clocks: QueryList<ClockComponent>) => {
      console.log("Change", clocks);
    });
  }

  toggle() {
    this.showClock = !this.showClock;
  }
}
```

This a live collection and is automatically updated with any view change

You should not change UI state here since the notification is executed after Angular already checked the component

```
<div *ngIf="showClock">
  <my-clock></my-clock>
  <my-clock></my-clock>
</div>
```

ngFor Analysis

- In some cases ngFor is the root cause for performance issue
- Since ngFor produces significant amount of DOM Angular puts much effort trying to optimize it
- However, the developer is still responsible for keeping it truly optimized
- Lets see …

Swapping Items

- We use ngFor to display a list of contacts
- What happens if we swap two items inside the list

```
const tmp = this.contacts[0];
this.contacts[0] = this.contacts[1];
this.contacts[1] = tmp;
```

- Angular is smart enough to swap the DOM elements

Angular just swaps the two elements

```
<ul>
  <li>
    <my-contact><span>Roni</span></my-contact>
  </li>
  <li>
    <my-contact><span>Ori</span></my-contact>
  </li>
</ul>
```

Identity

- To detect a permutation Angular by default uses the object identity (address) of each item
- Replacing an existing item with a new object but with exactly the same fields will cause DOM recreation

```
<ul>
  <li *ngFor="let contact of contacts">
    <my-contact [contact]="contact"></my-contact>
  </li>
</ul>
```

```
this.contacts = [
  {"id": 1, "name": "Ori"},
  {"id": 2, "name": "Roni"},
];
```

Executing this code twice causes DOM recreation !!!

Customizing Identity

- Use trackBy syntax to change the identity algorithm of ngFor

```
<ul>
  <li *ngFor="let contact of contacts; trackBy: trackByFn">
    <my-contact [contact]="contact"></my-contact>
  </li>
</ul>
```

```
trackByFn(index, item) {
  return item.id;
}
```

```
trackByFn(index, item) {
  return index;
}
```

Angular reuses the DOM whatever the value of item

For read-only list which need to be refreshed this is the preferred way

Summary

- There are many aspects to consider when implementing a component
- Most of the time Angular's defaults are good enough
- You may want to customize
  - View Encapsulation
  - Lifecyle Hooks
  - Pipes
  - More …