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 …

CSS

- The CSS standard was originally focused around separation of concerns
  - Content → HTML
  - Logic → JavaScript
  - Styling → CSS
- Having different programming language for each concern is great
- However, the standard original MOO contradicts the component state of mind

Styling Components

- A component is an isolated unit of UI
- Styling inside parent component should not break a child
- On the other hand, a parent may need to customize "a bit" the appearance of a child component
- However, CSS is global by nature. It usually "cascades" more than we need

Be aware of Cascading

- Assume the following ContactListComponent

parent.component.html

```
<ul>
    <li *ngFor="let contact of contacts">
        <span>{{contact.name}}</span>
        <button>Delete</button>
    </li>
</ul>
```

- And the following parent component

contactList.component.html

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

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

How easily can we style each button differently ?

Be aware of Cascading

- If we want to style the button element inside the parent, we can use the following

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

- However it means that every descendant button of my-app is effected
- Do we want to change the styling of contact-list component too ?

Be Specific

- We can move to a more specific definition

```
my-app > button {
   background-color: red;
}
```

- It means that only direct child of my-app is effected
- However, this is too strict definition
- Any time we move the button inside its containing component the CSS definition must be fixed

CSS Modules

- A CSS Module is a CSS file in which all class names are scoped locally by default
- See more details at https://github.com/css-modules/css-modules
- The trick is to change the class name into something unique

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

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

- And then somehow use the unique name inside HTML/code

Shadow DOM

- One of four Web Component standards
  - HTML templates
  - Shadow DOM
  - Custom Elements
  - HTML Imports
- Allows for scoped CSS (and more …)
  - Styles don't leak out
  - Page styles don't bleed in
- Most browsers do not support it ☹

Angular POV

- Angular implements parts of the Shadow DOM standard even for older browser
  - CSS encapsulation
  - :host
  - /deep/
  - template
  - content
- You can think of Angular as a way to bring the future power of web components into today SPA development

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

This is the trick

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

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/ ➔ ::ng-deep

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

Omitting :host creates a "plain" global CSS rule

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

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

ng-container

- Allows for grouping of sibling elements without introducing additional HTML element

```
<p>
    I turned the corner
    <ng-container *ngIf="hero">
        and saw {{hero.name}}. I waved
    </ng-container>
    and continued on my way.
</p>
```

Content Projection

- Previously known as transclusion (AngularJS)
- Every component may have a content that is defined by the host of the component
- For example, a dialog component

```
<h1>Hello Angular</h1>

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

This is the content

We expect that the dialog component reuses the content somewhere inside its template

Content Projection

- By default the content is not part of the DOM
- However, all components inside the content are created !!!

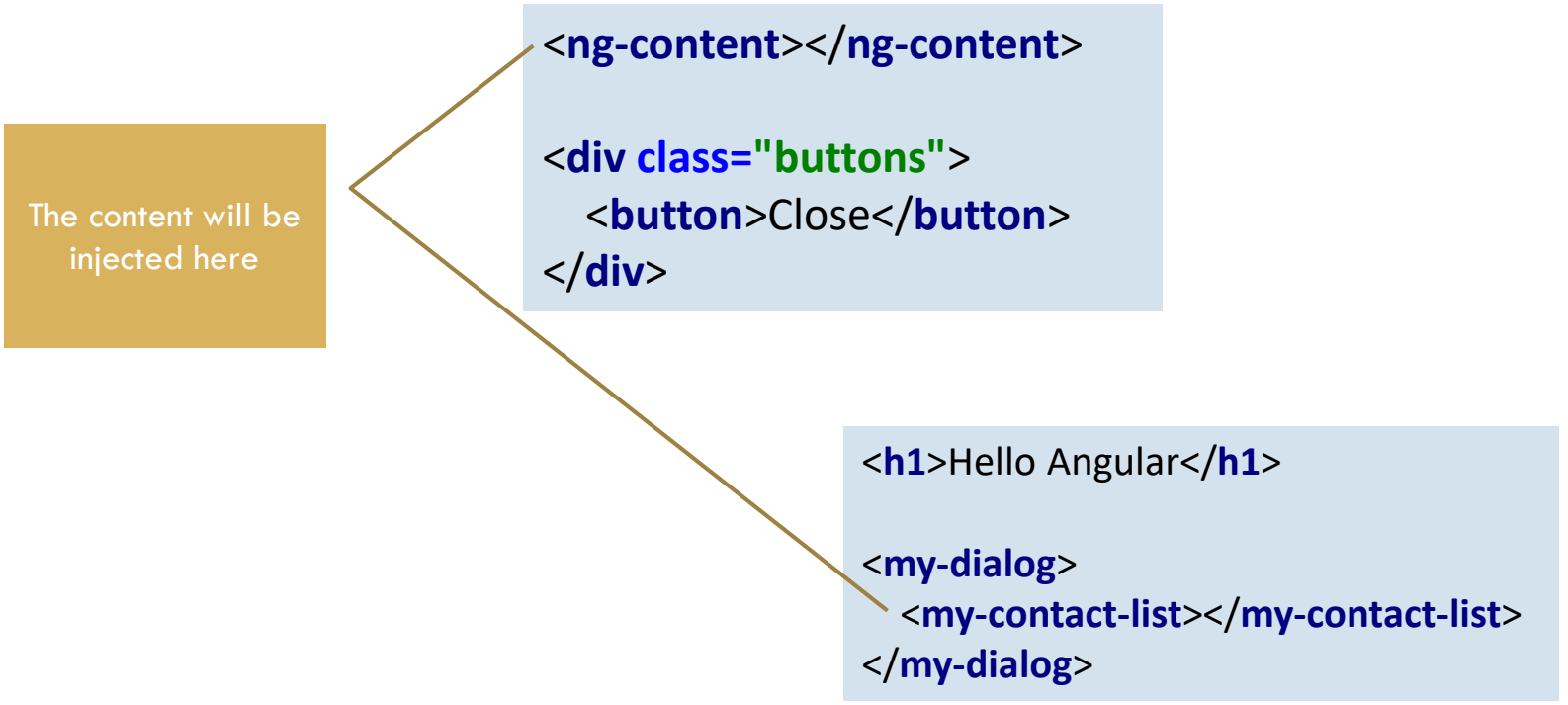
my-contact-list component is created but not display !!!!

```
<h1>Hello Angular</h1>

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

Content Projection

- A component may decide to inject the content into its template by using the ng-content marker



Content Lifetime

- The lifetime of the content is not controlled by the surrounding component
- The owner of the content is the parent of the dialog

```
<ng-content *ngIf="showDialog"></ng-content>

<div class="buttons">
   <button>Close</button>
   <button (click)="toggleDialog()">Toggle Dialog</button>
</div>
```

Content is removed from the DOM but its child components are still alive

Multi ng-content

- Use the select attribute

```
<ng-content select=".header"></ng-content>

<ng-content select=".content"></ng-content>

<ng-content select=".buttons"></ng-content>
```

- The client need to “reuse” the correct selectors

```
<my-dialog>
  <my-contact-list class="content"></my-contact-list>
</my-dialog>
```

Must be the same

Default content

- Angular does not allow default content ☹

Does not work !

```
<ng-content select=".buttons">
  <button>Close</button>
  <button (click)="toggleDialog()">Toggle Dialog</button>
</ng-content>
```

- You can simulate that using the following trick

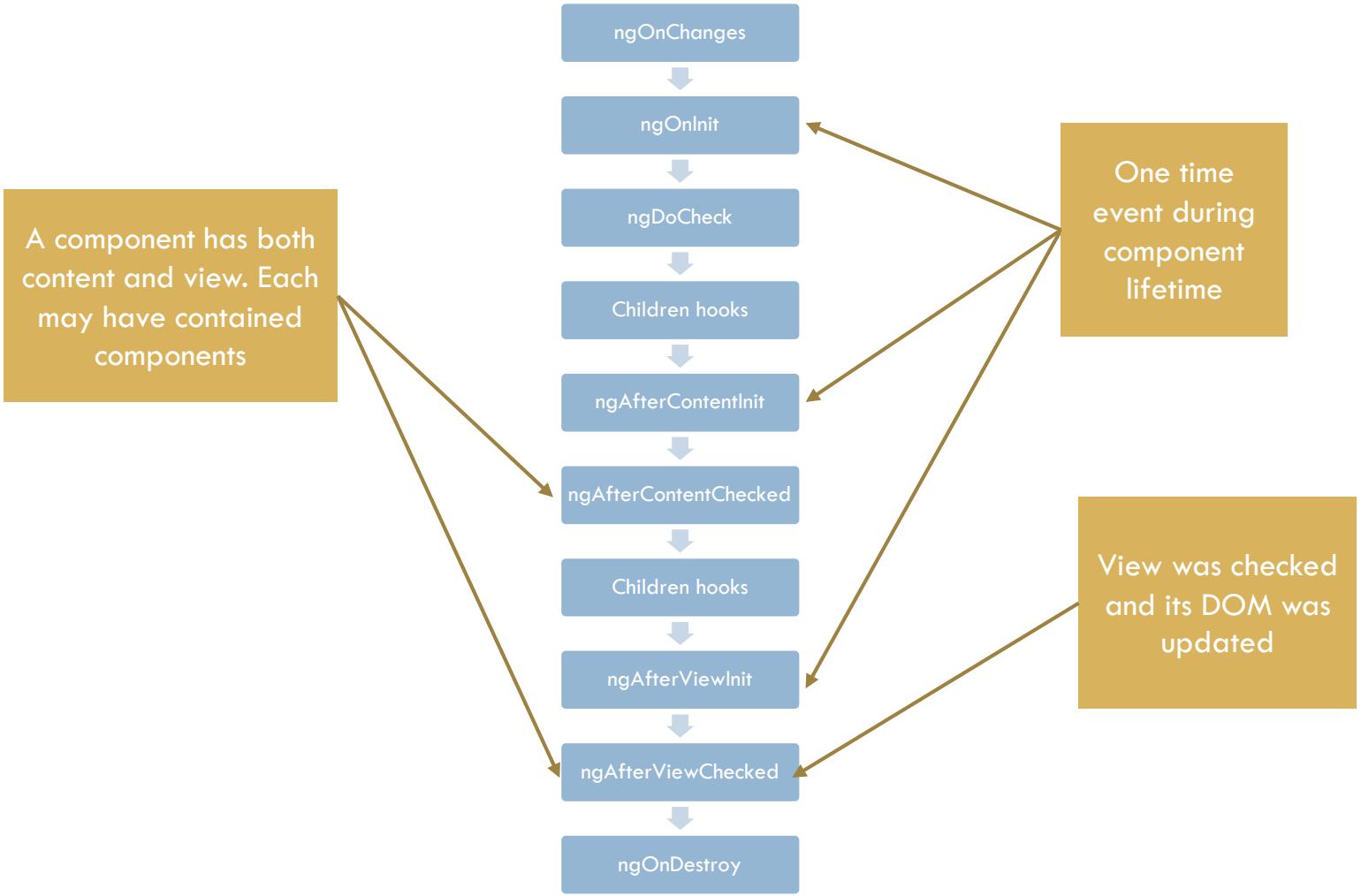
If the wrapper of ng-content has no children it means the client did not specify any content and we should use the default

```
<div #buttons>
  <ng-content select=".buttons"></ng-content>
</div>
<div *ngIf="!buttons.firstElementChild">
  <button>Close</button>
  <button (click)="toggleDialog()">Toggle Dialog</button>
</div>
```

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

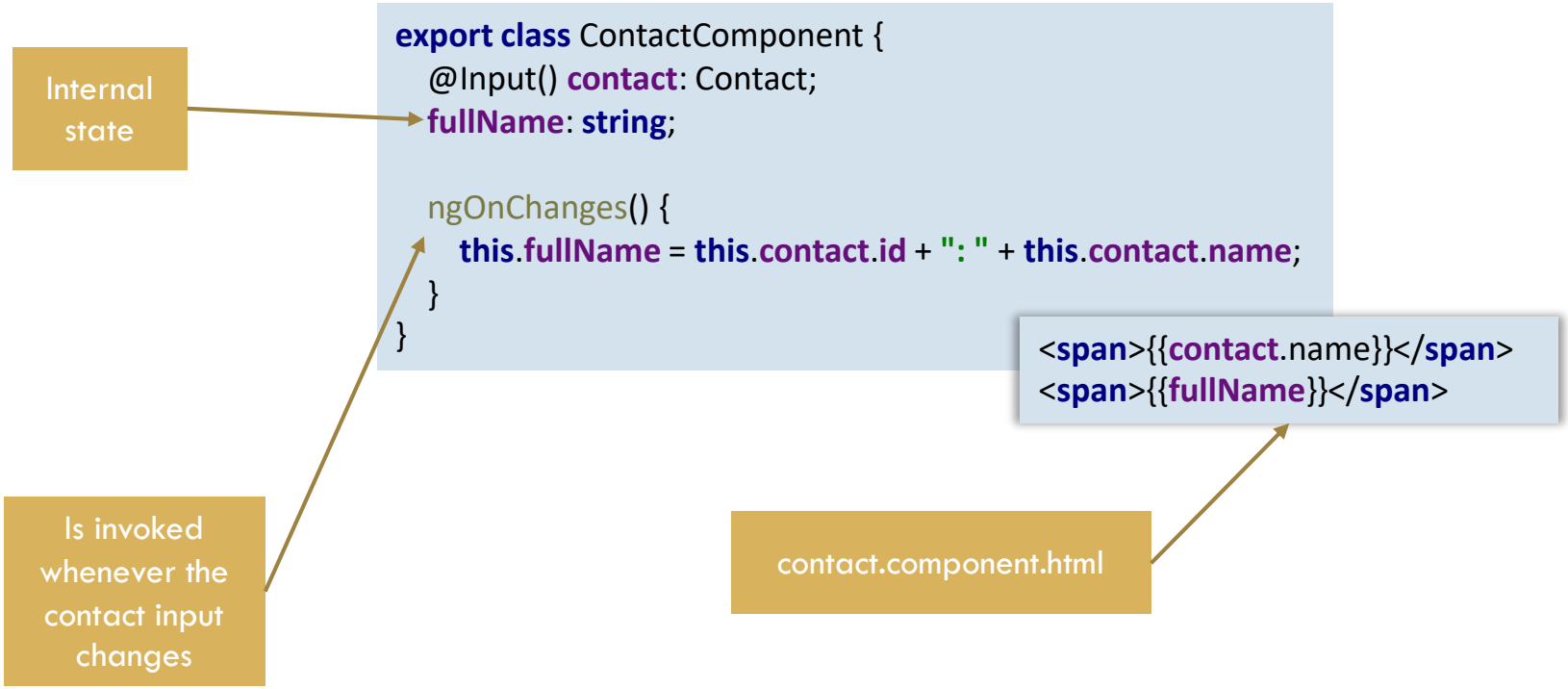
Lifecycle Hooks



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 all inputs

ngOnChanges



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 does not trigger 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"
    });
  }
}
```

Avoid internal state

- Component internal state can be extracted into external model that is sent as an input
- Upon change, the parent component clones the model and updates it
- Angular rebinds the input
- Since all state is inside the input model the component does not need to react to changes
- It just displays the input data
- A service may encapsulate the details

Avoid Internal State


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

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

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

```
export interface IContactViewModel {
   id?: number;
   name?: string;
   fullName?: string;
}

export class ContactViewModel implements IContactViewModel{
   id: number;
   name: string;
   fullName: string;

   constructor(options: IContactViewModel) {
      this.change(options);
   }

   change(options: IContactViewModel) {
      Object.assign(this, options);
      this.fullName = this.id + ": " + this.name;
   }
}
```

The change goes through a method which is responsible for “fixing” the whole model

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

Dynamic Component

- There are some cases where the component template can only be known at runtime
  - Think about a view that is defined by a database
  - Wix style
- In that case we need to dynamically compile a component
- Adding a component to an existing module is not allowed
- Therefore, you will need to compile a module first !!!

Dynamically Create a Module


```
createComponentFactory(template: string) {
    @Component({
        template: template,
    })
    class DynamicComponent {
        constructor(contactService: ContactService) {
            console.log(contactService);
        }
    }

    @NgModule({
        imports: [
            CommonModule
        ],
        declarations: [DynamicComponent],
    })
    class DynamicModule {
    }

    const moduleFactory = this.compiler.compileModuleAndAllComponentsSync(DynamicModule);
    const componentFactory = moduleFactory.componentFactories[0];

    return componentFactory;
}
```

No selector !!! Angular creates a random one

Add the component to the module

Using the Component

- Having a component factory we can inject a new component instance into a parent

```
injectTemplate() {
   const template = "<h1>{{counter}}</h1>";
   const componentFactory = createComponentFactory(template);

   const componentRef = this.marker.createComponent(componentFactory);
}
```

Add the component to the module

```
export class AppComponent {
  @ViewChild("marker", {read: ViewContainerRef}) marker: ViewContainerRef;
}
```

```
<div #marker></div>
```

Controlling Dynamic Component State

- Creating component at runtime is just a mechanism to inject a template into existing parent component
- How can we let template attach to parent state ?

```
injectTemplate() {
   const template = "<h1>{{state.counter}}</h1>";
   const componentFactory = this.createComponentFactory(template);
   const componentRef = this.marker.createComponent(componentFactory);

   componentRef.instance.state = this;
}
```

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

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

```
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);
}
```

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 too

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 permutations 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 causes 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

Heterogeneous ngFor

- Assume a polymorphic collection of items
- For each type of item we want to display different view
- According to OCP we don't want to maintain an if/else
- Instead we hold a map between items types and views
- How can we handle injection of different view depends on the item type ?

ComponentFactoryResolver

- A built-in provider that returns a component factory for a component type
- Use it to dynamically inject a component to an existing parent
- Usually you will set a marker inside the parent's view and inject the component using ViewContainerRef
- See next slide

Heterogeneous ngFor


```
export class ListItemComponent {
    @ViewChild("marker", {read: ViewContainerRef})
    marker: ViewContainerRef;

    @Input() item: any;
    @Input() itemToComponentType: any;

    componentRef: ComponentRef<any>;

    constructor(private componentFactoryResolver: ComponentFactoryResolver) {
    }

    ngOnChanges() {
        if(this.componentRef!=null) {
            this.componentRef.destroy();
            this.componentRef = null;
        }

        if(this.item != null) {
            const componentType = this.itemToComponentType(this.item);
            const componentFactory = this.componentFactoryResolver.resolveComponentFactory(componentType);
            this.componentRef = this.marker.createComponent(componentFactory);
            this.componentRef.instance["item"] = this.item;
        }
    }
}
```

```
<div #marker></div>
```

Define a marker inside the view

Dynamically create a component at after the marker

```
<ul>
  <li *ngFor="let item of items">
    <my-list-item [item]="item"
          [itemToComponentType]="itemToComponentTypeFn">
    </my-list-item>
  </li>
</ul>
```

entryComponents

- By default a component does not have a factory
  - Less CPU and code
- You need to manually ask Angular to create a factory by specifying the component under the entryComponents section

```
@NgModule({
   entryComponents: [
      ContactComponent,
      GroupComponent,
   ]
})
export class AppModule {
}
```

Filtering ngFor

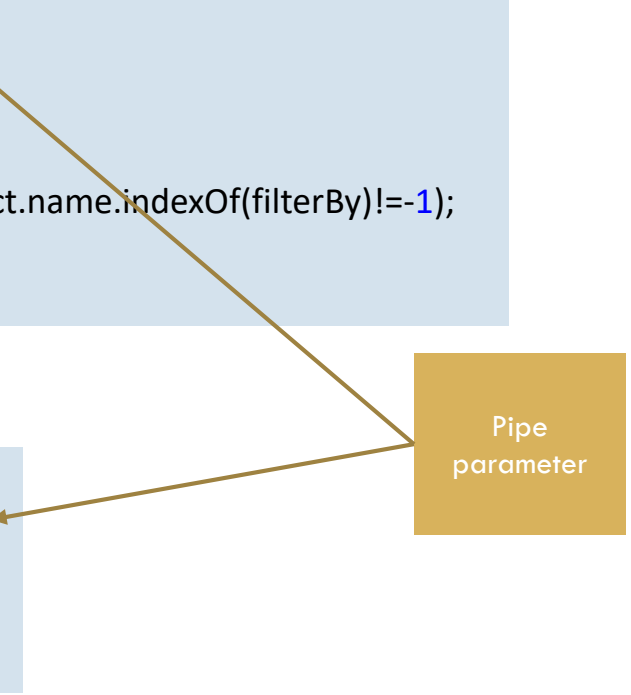
- Angular by design does not offer a pipe for filtering/sorting the collection
  - AngularJS did
- Performance is the main reason
- Past experience showed that developers do not use that capability in efficient way
- You can still define your own

Custom Pipe


```
@Pipe({name: 'filter'})
export class FilterPipe implements PipeTransform {
   transform(coll: any[], filterBy: string): any[] {
      if(filterBy === undefined) {
         return coll;
      }

      return coll.filter(contact => contact.name.indexOf(filterBy)!=-1);
   }
}
```

```
<ul>
   <li *ngFor="let contact of contacts | filter: filterBy">
      <span>{{contact.name}}</span>
   </li>
</ul>
```



Pipe Optimization

- Angular is smart enough to run the pipe only if one of its input changes
- Angular assume the pipe is a pure function
  - The output is derived only from the input
- However, for a collection this assumption is problematic. For example,
  - Collection value at index 5 changes
  - The collection reference remains the same
  - Angular does not run the filter
  - DOM is not updated

Impure Pipe

- Angular always execute the pipe even if none of its inputs changed

```
@Pipe({
   name: 'filter',
   pure: false
})
export class FilterPipe implements PipeTransform {
   transform(coll: any[], filterBy: string): any[] {
      if(filterBy === undefined) {
         return coll;
      }

      console.log("transform");

      return coll.filter(contact => contact.name.indexOf(filterBy)!=-1);
   }
}
```

Impure pipe

Be Aware

- Running filtering during every change detection is expensive
- This is the main reason Angular does not offer a filter pipe
- It is better to react to change events and only then filter the data

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 …