# Data Binding

Sending data from ts → html & html → ts

* One way
  + Component class → View Tempate
    - String interpolation
    - Property binding
  + View Tempate → Component class
    - Event binding
* Two way
  + Component → View Template, View Tempate → Component
    - [(ngModel)]

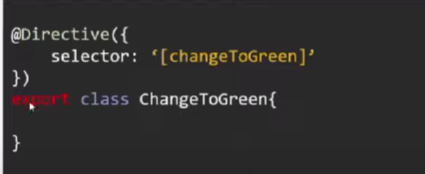
More Info

* Property binding
  + there are some attributes in html tags that we cannot use for property binding. In such cases we use attr.<attribute>
    - Ex: [**attr.**aria-hidden]=” ”
* Event binding
  + With each event, there is an event object which contains information about the target html element. We can access that object using **$event**
  + The value of the html element can be accessed using **$event.target.value**
* Two way binding
  + It is a combination of Property binding and Event binding
  + It can show the value from component class to view template and if we make changes in html, it will send that value back to component class
  + It is done using **[(ngModel)]**
  + ngModel needs to be imported in app module by importing **FormsModule**

# Directives

A directive is an instruction to the DOM. We use them to manipulate DOM, change behaviour, Add/Remove DOM Elements

Just like Component, Directive is also a TS class



* Component Directive
  + It is a directive with a template
  + When we use a component selector in other html files, we are instructing angular to add the contents of the components view template to that file. Hence we can call it a directive
* Attribute Directive
  + We can change the appearance and behavior of a DOM element
    - **ngStyle, ngClass**
    - Custom Ex: <div changeToGreen> Some Content <div>
* Structural Directive
  + Used to add or remove DOM elements on the WebPage
    - **ngIf, ngFor, ngSwitch**
    - Ex: <div \*ngIf=””> Some Content <div>

More Info

* ngStyle
  + It allows us to set many inline styles of a HTML element using expression
  + With ngStyle, we can assign a CSS property value dynamically

<div [ngStyle]="{ color : product.is\_in\_inventory? 'Green':'Red'}">

      {{ product.is\_in\_inventory? 'Available in Stock': 'Out of Stock'}}

</div>

* ngClass
  + Used to set a CSS class dynamically using ts expression
  + ngClass should be used in the following way: every class should have a class name and Boolean value assigned to it. We can manipulate the Boolean with ts.
  + If Boolean value is not given, then any type will be fore-casted to Boolean as shown in the example. We used string in place of Boolean which is later casted to Boolean.

<button [ngClass]="{'btn':true, 'btn-search': searchText, 'btn-search-disabled': !searchText}" [disabled]="!searchText">Search</button>

String is a true value, empty string is a false value if used as a Boolean

Falsey values: false, ‘ ’, null, undefined, 0

Remaining all are truthy values

# Communication between related components

# Custom Property Binding

Parent -> Child Component

*Done using assign value to custom attribute*

@Input()

* With @Input() decorator, we can make a custom attribute to the selector of that component.
* We can pass the values from that custom attribute to the variable in the child component.

Parent component having child selector

    <app-product \*ngFor="let prod of products" [product]="prod"></app-product>

Child component variable that receives value from parent

  @Input()

  product:{

      id: number,

      name: string,

      description: string,

      brand: string,

      gender: string,

      category: string,

      size: number[],

      color: string[],

      price: number,

      discountPrice?:number,

      is\_in\_inventory: boolean,

      items\_left: number,

      imageURL: string,

      slug: string

  };

In an variable of particular custom type, we can add optional attributes.

In below example, discountPrice?:number is optional, so it may or may not exist while assigning a product object to this variable but the remaining attributes should be there for sure

  product:{

      id: number,

      name: string,

      description: string,

      brand: string,

      discountPrice? : number,

  };

# Custom Event Binding

Child -> Parent Component

*Done by sending values using custom event*

@Output()

* With Output() decorator, we can make custom events to the selector of that component.
* Using this event, we can pass values from child to parent component.

In this example, we created custom event using **EventEmitter.** This event will emit a string of selected radio button using **emit()** method. We use @Output to send this event to parent.

In Parent, we use this event and receive the value using $event.

Parent

<app-filter

    (selectedFilterRadioButtonChanged)="onFilterChanged($event)">

</app-filter>

Child

<div class="filter-container">

    <span>Filter: </span>

    <input type="radio" name="filter" value="all"

        [(ngModel)]="selectedFilterRadioButton"

        (change)="OnselectedFilterRadioButtonChanged()"/>

    <span>{{'All ('+all+')'}}</span>

    <input type="radio" name="filter" value="true"

        [(ngModel)]="selectedFilterRadioButton"

        (change)="OnselectedFilterRadioButtonChanged()"/>

    <span>{{'In Stock (' + inStock + ')'}}</span>

    <input type="radio" name="filter" value="false"

        [(ngModel)]="selectedFilterRadioButton"

        (change)="OnselectedFilterRadioButtonChanged()"/>

    <span>{{'Out of Stock (' + outOfStock + ')'}}</span>

</div>

 @Output()

  selectedFilterRadioButtonChanged : EventEmitter<string> = new EventEmitter<string>();

  selectedFilterRadioButton: string = 'all';

  OnselectedFilterRadioButtonChanged(){

   this.selectedFilterRadioButtonChanged.emit(this.selectedFilterRadioButton);

  }

We can created Custom event using EventEmitter module and its constructor

Here we import **EventEmitter** module, use **EventEmitter<type>()** constructor

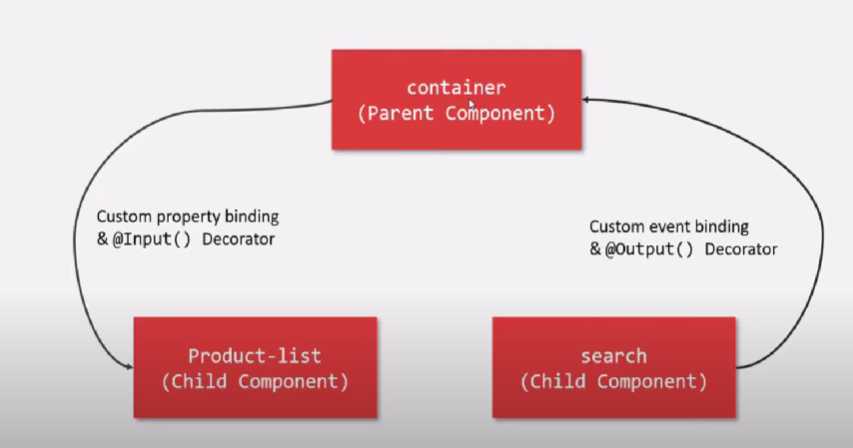
selectedFilterRadioButtonChanged : EventEmitter<string> = new EventEmitter<string>();

To send the value from that event we use **emit()** method

this.selectedFilterRadioButtonChanged.emit(this.selectedFilterRadioButton);

# Communication between sibling components

* To communicate with sibling components, we need to use both techniques in communication between related components.
* First we send the data from a Component A to its Parent. Then we send that data from Parent to Component B.



# Template Reference Variable

* It is a variable which stores a reference to DOM element, Component or Directive
* We can then access all the properties of that DOM element or Component or Directive using that variable
* Done by giving **#\_name** to a DOM element and using the name to access it

Reference to DOM element

<input class="ekart-search-product-input" #searchInput>

<button class="btn btn-search" (click)="updateSearchText(searchInput)"> Search </button>

updateSearchText(inputEl: HTMLInputElement){

    this.searchText = inputEl.value;

  }

Reference to Component

Here we are using product list component as reference and accessing it in product detail component

<product-list [searchText]="searchText" #productListComponent></product-list>

<product-detail \*ngIf="productListComponent.selectedProduct"></product-detail>

@ViewChild()

* It is used to query and get a reference to DOM element in the component without an event.
* It returns the first matching element
* **@ViewChild** has 2 arguments (2nd argument is optional)
  + **1st argument: @ViewChild(‘selector’)**
    - here selector is the #name of that element
    - if the child component is present only once in parent component, we can directly use the component class name instead of selector

For selector

@ViewChild(‘productListComponent’) searchInputEl: ElementRef;

For direct component

@ViewChild(ProductListComponent) searchInputEl: ElementRef;

* + **2nd argument: @ViewChild(‘selector’, {read: \_\_ ,static: true/false’})**
    - 2nd argument has 2 properties: read and static
    - read is use to read the different token from queried elements
    - static tells when the query should be resolved
      * by default it is true, so immediately when the component is initialized, the variable is assigned the reference
      * if set to false, when a change detection cycle runs, it will be assigned
* @ViewChild variable should be of type **ElementRef**
* To access the value of the variable, use variable**.nativeElement.value**

<div class="ekart--search--product">

    <input class="ekart-search-product-input" #searchInput>

</div>

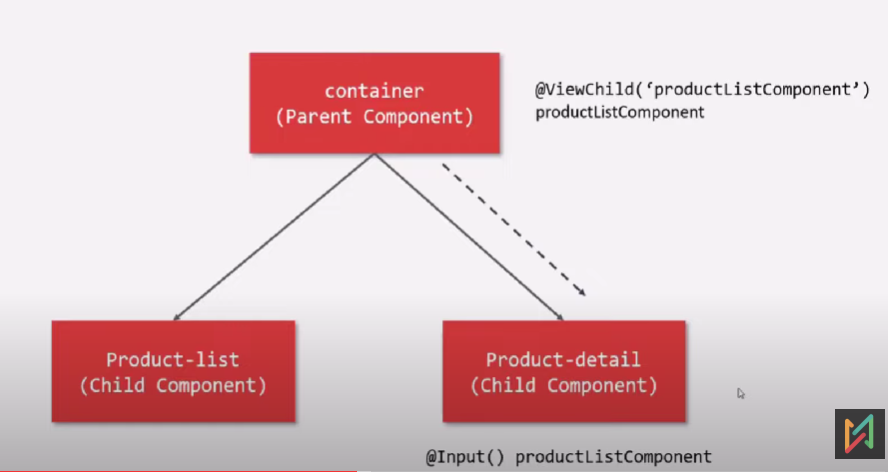
  @ViewChild('searchInput') searchInputEl: ElementRef;

  updateSearchText(){

    this.searchText = this.searchInputEl.nativeElement.value;

  }

Now by using the @ViewChild, we can send data from 2 different components



Here we have 2 child’s product-list and product-detail.

We are taking product-list reference into parent by @ViewChild and sending that variable to product-detail using custom attribute binding @Input()

Parent.html

<div class="ekart--product--container">

    <product-list [searchText]="searchText" #productListComponent></product-list>

    <product-detail \*ngIf="productListComponent.selectedProduct" [productListComp]="productListComponent"></product-detail>

</div>

Parent.ts

@ViewChild(ProductListComponent) productListComponent: ElementRef;

Product-detail.ts

@Input()

productListComp: ProductListComponent = undefined;

@ViewChildren()

* It is used to get the reference to all the list of DOM element from the view template of component class.
* It returns all the matching elements
* Its usage is similar to @ViewChild
* The type of value it returns is **QueryList<ElementRef>**
* To access all the DOM elements in that list, we ca use **.forEach()**
* **@ViewChildren** don’t have static property as ViewChild. It only runs after a change detection cycle runs

Here all the DOM elements with same selector name are returned as a list by using @ViewChildren

<div class="viewChildrenTest">

    <input type="text" placeholder="first name" #inputEl>

    <br><br>

    <input type="text" placeholder="middle name" #inputEl>

    <br><br>

    <input type="text" placeholder="last name" #inputEl>

    <br><br>

    <button type="button" class="btn btn-primary" (click)="showChildren()"> Submit </button>

    <br><br>

</div>

 @ViewChildren('inputEl')

  inputElements: QueryList<ElementRef>;

  showChildren(){

    this.inputElements.forEach( ele => console.log(ele.nativeElement));

  }

<ng-template>

* ng-template help to define a template that can be reused multiple times within an Angular component.
* It is an angular element used to define a template of html elements
* This template wont render in web page unless you use \***ngTemplateOutlet** Directive to show the template

Example 1

Here the code inside <ng-template> wont render if the div below it is not present.

We should give a reference name to template and pass it to \*ngTemplateOutlet to show the template.

<div class="ngTemplateTest">

    <h5>Testing ng-template</h5>

    <ng-template #myTestTemplate>

        <h5>Inside Template</h5>

        <p>Para in template</p>

    </ng-template>

    <!-- ngTempalteOutlet Directive -->

    <div \*ngTemplateOutlet="myTestTemplate"></div>

</div>

Example 2

Here we defined a template and we are showing it if the ngIf condition fails.

<div class="ekart-product-detail-addtoCart">

        <button class="btn-add-to-cart" \*ngIf="product.is\_in\_inventory; else notifyMe">ADD TO CART</button>

        <ng-template #notifyMe>

            <button class="btn-add-to-cart">NOTIFY ME</button>

        </ng-template>

    </div>

<ng-container>

* ng-container when we need to group elements without adding additional nodes to the DOM.
* It is mostly used along with \*ngIf
* Main use of ng-container is that, we can use multiple structural directive at a time using this.

Example 1

<div class="ngContainerTest t-center">

    <br><br>

    <h5>Testing ng-container</h5>

    <ng-container \*ngIf="toggle; else toggleOff">

        <p>Toggle On</p>

    </ng-container>

    <ng-template #toggleOff>

        <p>Toggle Off</p>

    </ng-template>

    <button class="btn btn-primary" (click)="onToggle()">Toggle</button>

</div>

Example 2

Here first we used div to use ngFor because we cannot use more than one structural directive in same element. But div is useless.

Hence we can use ng-container which will not create any extra element for using multiple structural directives

(Wrong approach)

    <div \*ngFor="let prod of products" (click)="selectedProduct = prod">

        <app-product [product]="prod" \*ngIf="searchText == '' || prod.name.toLowerCase().includes(searchText)"></app-product>

    </div>

(Correct approach)

 <ng-container \*ngFor="let prod of products" (click)="selectedProduct = prod">

        <app-product [product]="prod" \*ngIf="searchText == '' || prod.name.toLowerCase().includes(searchText)"></app-product>

  </ng-container>

<ng-content>

* **ng-content** is used to render content dynamically that is passed in from the parent component.
* When we use a component selector multiple times, but want different in all of them, we can use ng-content to pass where ever we want to change the content

Here we are using the featured-brands component in parent multiple times. We use ng-content in child component and use **select** attribute to give the class of which html element needs to be rendered in that particular ng-content.

Child Component (featured-brands)

<div class="ekart-featured-product-item">

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

    <p class="description">

        Lorem Ipsum is simply dummy text of the printing and typesetting industry.

        Lorem Ipsum has been the industry's standard dummy text ever since the 1500s.

    </p>

    <ng-content select=".call-to-action"></ng-content>

</div>

Parent

<featured-brands>

      <h3 class="title">New Arrivals in Nike</h3>

      <button class="call-to-action">Learn More About Nike</button>

</featured-brands>

<featured-brands>

      <h3 class="title">New Arrivals in Adidas</h3>

      <button class="call-to-action">Learn More About Adidas</button>

</featured-brands>

<featured-brands>

      <h3 class="title">New Arrivals in Puma</h3>

      <button class="call-to-action">Learn More About Puma</button>

</featured-brands>

While using ng-content, we can also use another component tag inside a component tag.

<featured-brands>

      <h3 class="title">New Arrivals in Puma</h3>

      <button class="call-to-action">Learn More About Puma</button>

<app-child> </app-child>

</featured-brands>

@ContentChild

* When we use ng-content, the child component does not have access to whatever is being rendered in the ng-content tag. All that content belongs to parent component.
* To get access to such content from child component we use **@ContentChild**

Here the Parent contains child tag, in which we are rendering some content along with child 2.

We can access both the content(para) and child2 properties using @ContentChild and providing the respective selectors like below

@ContentChild('para') paraEl : ElementRef;

@ContentChild(TestChild2Component) testChild2Ele: TestChild2Component;

Parent

<div class="ContentChildTest t-center">

    <br><br>

    <test-child>

        <p #para>Body in test child from ng content</p>

        <test-child2></test-child2>

    </test-child>

</div>

Child

<div class="childMain">

    <h3>Test Child</h3>

    <hr>

    <ng-content></ng-content>

    <hr>

    <button class="btn btn-primary" (click)="showContentChild()"> Submit</button>

    <button class="btn btn-primary" (click)="showChild2()"> Submit 2</button>

    <hr>

    <p>{{contentChildOutput}}</p>

    <br><br>

</div>

contentChildOutput: string='';

  //accesing ng content para

  @ContentChild('para') paraEl : ElementRef;

  showContentChild(){

    this.contentChildOutput = '{'+ this.paraEl.nativeElement.innerHTML + '} is present in ContentChild';

  }

  @ContentChild(TestChild2Component) testChild2Ele: TestChild2Component;

  showChild2(){

    this.contentChildOutput = '{'+ this.testChild2Ele.name + '} is name property of Test Child 2';

  }

Child 2

<h4>Name of this element is {{name}}</h4>

  name: string = 'Test Child 2';

@ContentChildren

* It is similar to @ContentChild, but we can get all the references of that particular selector
* In **@ContentChildren** we use **QueryList<ElementRef>** to receive all the references
* We can use @ContentChildren for multiple component as well, same as we used in @ContentChild

Here using @ContentChildren(‘para’) variable: QueryList<ElementRef>; we are gaining access to all the ‘para’ selector children. We use **.forEach()** to iterate and use all values of children.

Parent

<div class="ContentChildrenTest t-center">

    <br><br>

    <test-child>

        <p #para>Body in test child from ng content</p>

        <p #para>Content Children para 2</p>

        <p #para>Content Children para 3</p>

        <test-child2 [name]="'Rahul'"></test-child2>

        <test-child2 [name]="'Varma'"></test-child2>

        <test-child2 [name]="'Sujatha'"></test-child2>

    </test-child>

</div>

Child

<div class="childMain">

    <h3>Test Child</h3>

    <hr>

    <ng-content></ng-content>

    <hr>

    <button class="btn btn-primary" (click)="showContentChild()">Submit</button>

    <button class="btn btn-primary" (click)="showChild2()">Submit 2</button>

    <button class="btn btn-primary" (click)="showParaChildren()">Show all para children</button>

    <button class="btn btn-primary" (click)="showComponentChildren()">Show all component children</button>

    <hr>

    <p>{{contentChildOutput}}</p>

    <br>

</div>

@ContentChildren('para') paraEls: QueryList<ElementRef>;

  showParaChildren(){

    this.contentChildOutput = '';

    this.paraEls.forEach(

      (ele) => {

        this.contentChildOutput = this.contentChildOutput + ele.nativeElement.innerHTML+ '/ ';

      }

    )

  }

  @ContentChildren(TestChild2Component) componentEls: QueryList<TestChild2Component>;

  showComponentChildren(){

    this.contentChildOutput =  "check Console";

    this.componentEls.forEach(

      (ele) => {

        console.log(ele);

      }

    )

  }

# Life Cycle Hooks

* When Angular Application starts, it first creates and renders root component.
* Then it creates and renders its children and their children. In this way, it forms a tree of components.
* Once Angular loads the component, it starts the process of rendering the view. For that, it checks the input properties, evaluate the data binding & expressions, render the projected content etc.
* Angular also removes the components from DOM when it is no longer needed
* Angular lets us know when these events happen, using **Angular lifecycle hooks**
  + They are the methods that angular invokes on a directive or a component, as it created, changes and destroys them.
  + **ngOnChanges, ngOnInit, ngDoCheck, ngAfterContentInit, ngAfterContentChecked, ngAfterViewInit, ngAfterViewChecked, ngDestroy**
* Lifecycle begins when a component is created and ends when a component is destoryed

Change detection

* It is a mechanism by which Angular keeps the view template in sync with the component class.
* Angular runs a change detection cycle when ever a change happens
  + When @Input property of component changes
  + When DOM event happens, ex: click, change
  + When a timer event happens using setTimeOut()/ setInterval()
  + When http request is made

Component Initialization

* When ever a component is initialized, its constructor() is called first
* When a constructor is called, none of its @Input properties are updated and available to use
  + Even if we bind that attribute if @Input and send the value from parent, it wont store that value before the constructor
* Projected contents are not available by the time the constructor is called
  + When we use ng-content, we project some elements there. But they are not projected before constructor

ngOnChanges

* **ngOnChanges** hook gets executed at the start, when a new component is created and its, input bound properties are updated
* It gets called every time the input bound properties of component changes
  + It is called when ever the value of **@Input** bound property of a component will change
* It wont run when there is no change in the actual value of that @Input property
* ngOnChanges has a property to get the previous value of the change using **SimpleChanges**

Here it is showing the variable name ‘message’, and showing the previous and current values

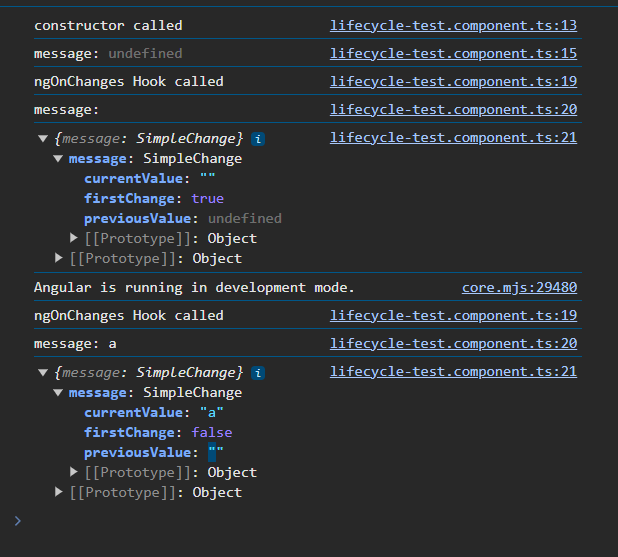
  ngOnChanges(changes: SimpleChanges){

    console.log("ngOnChanges Hook called");

    console.log('message:',this.message)

    console.log(changes);

  }



* When an array value changes, even if it is an input property, the ngOnChanges hook will not be called.
* This is because, the array is stored in the form of reference. When it is initialized, a reference to that variable is created. Even if the values are added to array or changed, the initial reference is the same. Hence this hook is not called

Here, we used message as array and an input property.

Child

<p #para \*ngFor="let msg of message">Message: {{msg}}</p>

@Input() message: string[];

ngOnChanges(changes: SimpleChanges){

    console.log("ngOnChanges Hook called");

    console.log('message:',this.message)

    console.log(changes);

  }

Parent

<div class="ngOnChangesTest t-center">

    <h3>Life Cycle Hooks</h3>

    <input type="text" #inputVal>

    <button class="btn btn-primary" (click)="onSubmit(inputVal)">Submit</button>

    <lifecycle-test [message]="inputValue"></lifecycle-test>

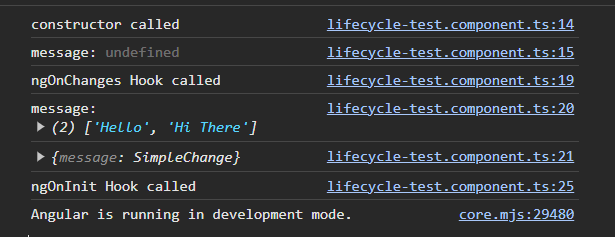
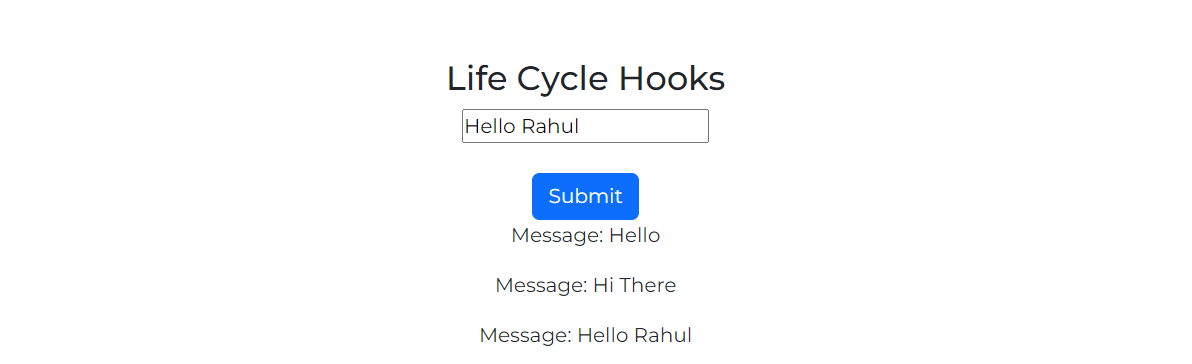
</div>

inputValue:string[] = ['Hello','Hi There'];

onSubmit(input: HTMLInputElement){

  this.inputValue.push(input.value);

}



ngOnInit

* Angular raises ngOnInit hook after it creates the component and update its input properties.
* This hook is raised after ngOnChanges
* This hook is fired only once i.e. during the first change detection cycle. After that, even if the input property changes, this hook doesn’t get called.
* By the time ngOnInit gets called, none of the child components or projected contents or view are available at this point. Hence properties decorated with @ViewChild, @ViewChildren, @ContentChild, @ContentChildren will not be available to use

<p #para>Message: {{message}}</p>

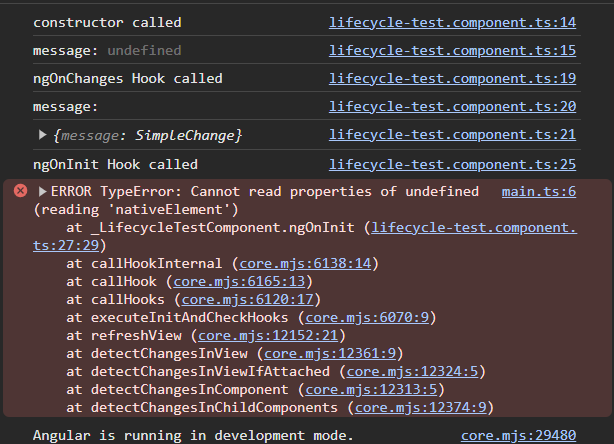
 ngOnInit(){

    console.log("ngOnInit Hook called");

    // This below code is not possible because paraEl is not yet available

    console.log(this.paraEl.nativeElement.innerHTML);

  }



ngDoCheck

* Angular invokes ngDoCheck hook during every change detection cycle.
* It is invoked even if there is no change in input bound properties.
* For example: this hook will run when a button is clicked on a webpage which doesn’t do anything. But still it is an event, so the change detection cycle will run and execute ngDoCheck hook
* Angular invokes ngDoCheck after ngOnChanges and ngOnInit
* It can be used to implement a custom change detection, whenever angular fails to detect any change made to input bound properties
* It is useful when you want to execute some code on every change detection cycle

Here on click of submit or focusing on the input, the ngDoCheck is called everytime.

Parent

<div class="ngOnChangesTest t-center">

    <h3>Life Cycle Hooks</h3>

    <input type="text" #inputVal (focus)="0">

    <button class="btn btn-primary" (click)="onSubmit(inputVal)">Submit</button>

    <lifecycle-test [message]="inputValue"></lifecycle-test>

</div>

 inputValue:string;

  // inputValue:string[] = ['Hello','Hi There'];

  onSubmit(input: HTMLInputElement){

    this.inputValue = input.value;

    // this.inputValue.push(input.value);

  }

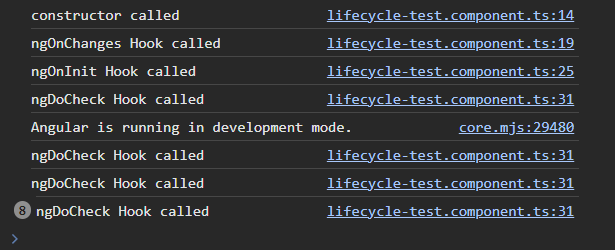
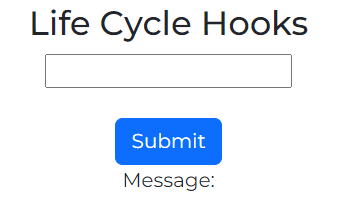
Child

<p #para>Message: {{message}}</p>

  ngDoCheck(){

    console.log("ngDoCheck Hook called");

  }



ngAfterContentInit

* This hook is called after the component’s **projected content** has been fully initialized
* Even if we don’t have a projected content, this hook is still called
* Angular updates the properties decorated with @**ContentChild** & @**ContentChildren** decorator just before this hook is raised
* This hook gets called only once during the first change detection cycle.

Here, ngAfterContentInit hook is called after @ContentChild is initialized. Hence paraContent is undefined during ngDoCheck and value is assigned during ngAfterContentInit.

Parent

<lifecycle-test [message]="inputValue">

    <p #temppara>This is projected content</p>

</lifecycle-test>

Child

<p #para>Message: {{message}}</p>

<ng-content></ng-content>

  ngDoCheck(){

    console.log("ngDoCheck Hook called");

    console.log(this.paraContent)

  }

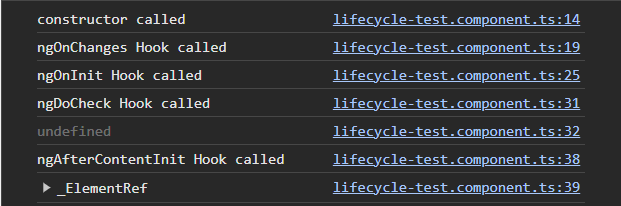
  @ContentChild('temppara') paraContent: ElementRef;

  ngAfterContentInit(){

    console.log("ngAfterContentInit Hook called");

    console.log(this.paraContent)

  }



ngAfterContentChecked

* This hook is called when the projected content changes
* This hook is called after every change detection cycle same like ngDoCheck
* Angular updates the variables decorated with @ContentChild & @ContentChildren decorator, before raising ngAfterContentChecked hook
* Even if we don’t have a projected content, this hook is still called
* ngAfterContentInit is called after projected content is initialized, but ngAfterContentChecked hook is called whenprojected content is initialized, checked, updated

Here when ever inputValue in para changes, this hook will be called. Also if there is any other event, this hook is again called with change detection aswell

Parent

<lifecycle-test [message]="inputValue">

        <p #temppara>User has entered: {{inputValue}}</p>

</lifecycle-test>

Child

<p #para>Message: {{message}}</p>

<ng-content></ng-content>

 ngAfterContentChecked(){

    console.log("ngAfterContentChecked Hook called");

  }

ngAfterContentInit and ngAfterContentChecked are component only hooks. They don’t apply for directives

ngAfterViewInit

* This hook is called after the components View template and all its child components view template are fully initialized
* This hook is only called once during the first change detection cycle, when Angular initializes the View for the first time
* Angular updates @ViewChild & @ViewChildren decorated variables before calling this hook
* By the time this hook gests called, all lifecycle hook methods of child components and directives are completely processed and child components are completely ready
* Even if there are no such variables, this hook is called once

Here in we are using @ViewChild and checking when the paraEl is initialized.

<p #para>Message: {{message}}</p>

<ng-content></ng-content>

  @ViewChild('para') paraEl: ElementRef;

  ngAfterContentChecked(){

    console.log("ngAfterContentChecked Hook called");

    console.log("In ngAfterContentChecked:",this.paraEl);

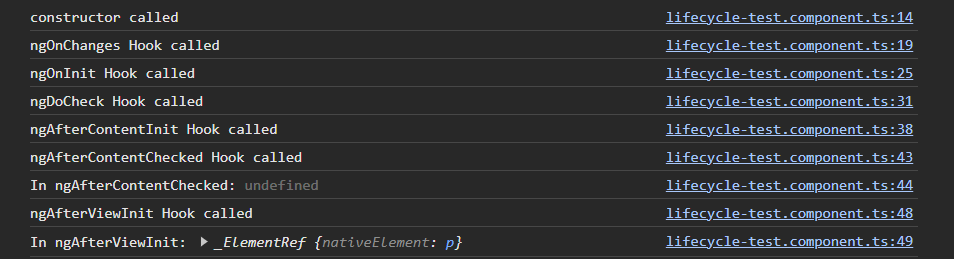
  }

  ngAfterViewInit(){

    console.log("ngAfterViewInit Hook called");

    console.log("In ngAfterViewInit:",this.paraEl);

  }



All the lifecycle hooks of child components are called first, before the lifecycle hooks of parent

ngAfterViewChecked

* This hook is called after it checks and updates the components View template and all its child components view templates
* This hook is called during first change detection cycle, after nfAfterViewInti hook. After that this hook is called during every change detection cycle

ngAfterViewChecked(){

  console.log("ngAfterViewChecked Hook called");

}

ngAfterViewChecked is component only hooks. They don’t apply for directives

ngOnDestroy

* This hook is fired just before the component or the directive id destroyed, i.e. removed from the DOM
* This hook is a great place to do some cleanup work like unsubscribe from an observable or detach event handler etc. as this hook is right before component is destroyed
* This hook is the last lifecycle hook of a component & directive

Here the child is lifecycle-test component. We are using \*ngIf to destroy the component by clicking a button. Hence while !toDestory is false, the ngOnDestroy hook is called.

When we unhide the component, it is created again and all the life cycle hooks gets called again.

Parent

<div class="ngOnChangesTest t-center">

    <br>

    <h3>Life Cycle Hooks</h3>

    <input type="text" #inputVal (focus)="0">

    <br><br>

    <button class="btn btn-primary" (click)="onSubmit(inputVal)">Submit</button>

    <lifecycle-test [message]="inputValue" \*ngIf="!toDestroy">

        <p #temppara>User has entered: {{inputValue}}</p>

    </lifecycle-test>

    <button class="btn btn-primary" (click)="DestroyComponent()">Show/Hide</button>

</div>

  //ngOnDestroy

  toDestroy: boolean = false;

  DestroyComponent(){

    this.toDestroy = !this.toDestroy;

  }

Child (lifecycle-test)

<p #para>Message: {{message}}</p>

<!-- <p #para \*ngFor="let msg of message">Message: {{msg}}</p> -->

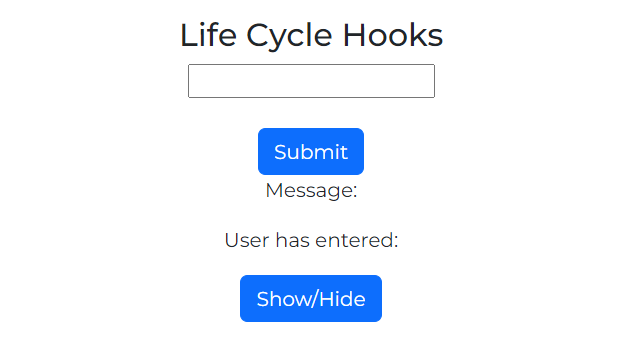
<ng-content></ng-content>

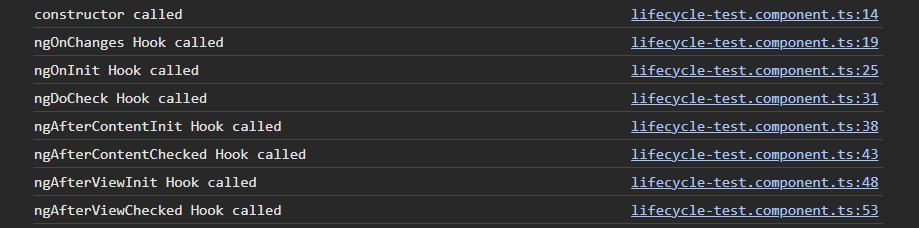
  ngOnDestroy(){

    console.log("ngOnDestroy Hook called");

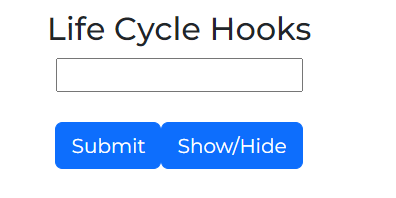
  }

Initially



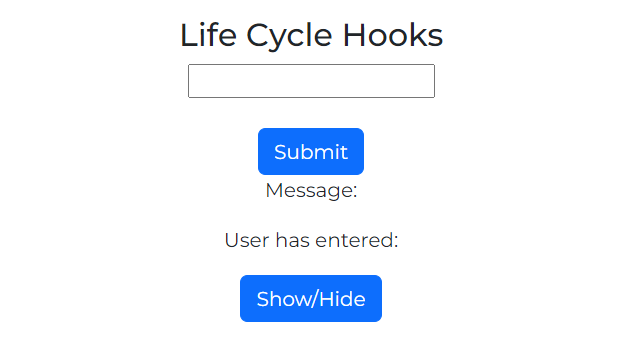


After Hide





After Unhiding the component again





# Custom Attribute Directive

Custom Style Directive

* We create a custom style directive to give the style described in that directive to any element that is using this directive
* To use this directive to any element, we should declare this directive in app module in declarations

This is a custom directive SetBackground.directive.ts

In this we create a class and use @Directive decorator with a selector as shown below. In selector we use ‘[ \_name]‘ for the directives. We can create directive automatically using **ng g d ‘directive\_name’**

Now we create a constructor and mention a variable of type ElementRef. Angular injects the HTML element (in which this directive is used), to that variable automatically. Now we use this variable to access the HTML element and set styles as shown below.

Here, we can declare that ElementRef variable outside the constructor and assign the value inside as shown in commented code. But we can also just mention private keyword inside constructor brackets. This will create a variable and auto assign that ElementRef to that variable. Now we can directly assess that variable outside constructor aswell

Now we use ngOnInit to assess that ElementRef variable and change the style properties as shown below. We do this in ngOnInit because the variables are not initialized by the time the constructor runs.

Custom Directive

import { Directive, ElementRef } from "@angular/core";

@Directive({

    selector: '[setBackground]'

})

export class SetBackground{

    // private element: ElementRef;

    constructor(private element: ElementRef){

        // element.nativeElement.style.backgroundColor = '#36454F';

        // element.nativeElement.style.color = 'white';

        // this.element = element;

    }

    ngOnInit(){

        this.element.nativeElement.style.backgroundColor = '#36454F';

        this.element.nativeElement.style.color = 'white';

    }

}

Usage of Custom Directive

 <div class="ekart-product-detail-gbc">

            <span setBackground>{{product.gender}}</span>

            <span setBackground>{{product.brand}}</span>

            <span setBackground>{{product.category}}</span>

 </div>

<div class="ekart-product-detail-price" setBackground>

            <p>

                <b>Price Each Pair:</b>

                <span style="background-color: red; color:white"><s>{{'$'+product.price}}</s></span>

                <span style="background-color: green; color:white">{{'$'+product.discountPrice}}</span> (Inclusive of all taxes)

            </p>

        </div>

App Module

@NgModule({

  declarations: [

    AppComponent,

    HeaderComponent,

    SetBackground

  ],

  imports: [

    BrowserModule,

    AppRoutingModule,

    FormsModule

  ],

  providers: [],

  bootstrap: [AppComponent]

})

export class AppModule { }

Renderer2

* In the custom style attribute, we are using ElementRef’s nativeElement property to change the styles. By doing this, we are directly accessing the DOM element which is not a good practice
* This is because
  + Angular keeps component and view in sync using templates, data binding and change detection etc. All of them are bypassed when we update the DOM directly
  + DOM manipulation works only in browsers. We cannot use the app in other platforms like web workers, servers for server-side rendering, desktop or mobile apps etc. where there is no browser
  + The DOM APS’s does not sanitize the data. Hence it is possible to inject a script, thereby, opening our app an easy target for the XSS injection attacks
* Render2 allows us to manipulate the DOM without accessing the DOM elements directly, by providing a layer of abstraction between the DOM element and the component code

Here we using Renderer2 class by injecting it into a variable in the constructor.

We now use the variable to set the style as shown below. We are using .setStyle which has 3 main arguments and 1 optional argument. The main arguments are:

* DOM element
* Which style to set
* Value of the style

There are various other methods the we can use to set different properties like setAttribute, addClass, removeClass etc.

@Directive({

    selector: '[setBackground]'

})

export class SetBackground{

    // private element: ElementRef;

    constructor(private element: ElementRef, private renderer: Renderer2){

        // element.nativeElement.style.backgroundColor = '#36454F';

        // element.nativeElement.style.color = 'white';

        // this.element = element;

    }

    ngOnInit(){

        // this.element.nativeElement.style.backgroundColor = '#36454F';

        // this.element.nativeElement.style.color = 'white';

        this.renderer.setStyle(this.element.nativeElement, 'backgroundColor', '#36454F');

        this.renderer.setStyle(this.element.nativeElement, 'color', 'white');

        // this.renderer.setAttribute(this.element.nativeElement, 'title', 'This is example title');

    }

}

@HostListener

* @HostListener decorator listens to a DOM event on the host element and it reacts to that event by executing an event handler method

Here, we created a custom directive called Highlight Directive. We created some methods. We are using @HostListener to execute these methods when the event mentioned inside HostListener occurs.

To create this, use @HostListerer to a method and mention the name of the event inside that HostListener.

Inside these methods, we access the DOM element from constructor and use renderer2 to add class when and event happens and remove it when another event happens.

In this way, we are describing many events and how the DOM element has to react to those events in a custom directive.

Custom Directive

import { Directive, ElementRef, HostListener, Renderer2 } from '@angular/core';

@Directive({

  selector: '[appHighlight]'

})

export class HighlightDirective {

  constructor(private element: ElementRef, private renderer: Renderer2) {

  }

  ngOnInit(){

  }

  @HostListener('mouseenter')

  OnMouseEnter(){

    this.renderer.addClass(this.element.nativeElement, 'highlight-product');

  }

  @HostListener('mouseleave')

  OnMouseOut(){

    this.renderer.removeClass(this.element.nativeElement, 'highlight-product');

  }

}

The DOM element where we used the custom directive

<div class="ekart--product--item" appHighlight>

    <div class="ekart--product--in--stock" [ngStyle]="{fontWeight : 'bold', color : product.is\_in\_inventory? 'Green':'Red'}">

        {{ product.is\_in\_inventory? 'Available in Stock': 'Out of Stock'}}

    </div>

</div>

The css class that we added to the element. This should be present is the same component where we are using the custom directive.

.highlight-product{

    -webkit-box-shadow: 0 0 1px #000 inset;

    -moz-box-shadow: 0 0 1px 3px #000 inset;

    border: #efefef 2px solid;

    box-shadow: 10px 10px 18px #cccccc;

    transform: scale(1.05);

}

@HostBinding

* @HostBinding decorator binds a host elements property to a property of a directive or a component class

Here we are using this Custom Directive AppHover on a button element. So the button element becomes the host. We use @HostBinding and bind the properties of that host element to this AppHover properties.

To use HostBinding, we decorate a property of directive with @HostBinding and mention the property of that host to be binded.

Now we can use this directive property to manipulate the host elements property.

Custom Directive

import { Directive, ElementRef, HostBinding, HostListener, Renderer2 } from '@angular/core';

@Directive({

  selector: '[appHover]'

})

export class AppHoverDirective {

  constructor(private element: ElementRef, private renderer: Renderer2) {

  }

  @HostBinding('style.backgroundColor') backgroundColor: string = '#28282B';

  @HostBinding('style.border') border: string = 'none';

  @HostBinding('style.color') textColor: string = 'white';

  @HostListener('mouseenter')

  OnMouseEnter(){

    this.backgroundColor = 'white';

    this.border = '#28282B 2px solid'; ;

    this.textColor = '#28282B';

  }

  @HostListener('mouseleave')

  OnMouseLeave(){

    this.backgroundColor = '#28282B';

    this.border = 'none'; ;

    this.textColor = 'white';

  }

}

Host element (here button)

<div class="ekart-product-detail-addtoCart">

        <button class="btn-add-to-cart" \*ngIf="product.is\_in\_inventory; else notifyMe" appHover>ADD TO CART</button>

 </div>

Property Binding in Directives

* We can bind the properties of the directive to a DOM element by first using the custom directive on that element and accessing it in the directive.
* Then we manipulate the properties of that DOM element in the directive.
* Here while passing the value to these properties that we manipulate, we can use @Input decorator to those properties and pass the values from the DOM element using property binding.

Here we are using a custom directive setBackground on span elements. We are accessing the span elements in the directive and setting their backgound color and text color.

To do this we are creating some variables in directive. We are decorating them with @Input. Now we can pass values to these properties from span using property binding as shown

DOM element

 <div class="ekart-product-detail-gbc">

            <span setBackground [backColor]="'red'" [textColor]="'white'">{{product.gender}}</span>

            <span setBackground [backColor]="'yellow'" [textColor]="'black'">{{product.brand}}</span>

            <span setBackground [backColor]="'black'" [textColor]="'white'">{{product.category}}</span>

        </div>

Custom Directive

import { Directive, ElementRef, Input, Renderer2 } from "@angular/core";

@Directive({

    selector: '[setBackground]'

})

export class SetBackground{

    @Input() backColor: string = '#36454F';

    @Input() textColor: string = 'White';

    constructor(private element: ElementRef, private renderer: Renderer2){

    }

    ngOnInit(){

        this.renderer.setStyle(this.element.nativeElement, 'backgroundColor', this.backColor);

        this.renderer.setStyle(this.element.nativeElement, 'color', this.textColor);

    }

}

Now if we want to set the values of the properties by directly using the attribute directive, we can do it by giving an alias name to that @Input(‘alias\_name’). Now we can use custom directive in square brackets and specify the value to pass to that property

<div class="ekart-product-detail-gbc">

            <span [setBackground]="'red'" [textColor]="'white'">{{product.gender}}</span>

            <span [setBackground]="'yellow'" [textColor]="'black'">{{product.brand}}</span>

            <span [setBackground]="'black'" [textColor]="'white'">{{product.category}}</span>

        </div>

import { Directive, ElementRef, Input, Renderer2 } from "@angular/core";

@Directive({

    selector: '[setBackground]'

})

export class SetBackground{

    @Input('setBackground') backColor: string = '#36454F';

    @Input() textColor: string = 'White';

    constructor(private element: ElementRef, private renderer: Renderer2){

    }

    ngOnInit(){

        this.renderer.setStyle(this.element.nativeElement, 'backgroundColor', this.backColor);

        this.renderer.setStyle(this.element.nativeElement, 'color', this.textColor);

    }

}

If we have a property in custom directive that has the same name as a property that already exist in the DOM element, then Angular first checks the custom directive for the properties and gives the preference to that directive property value.

Ex: We can have a property like title and title is an in build property of all DOM elements. But it takes the value of directive as main preference.

We can also give multiple inputs to the custom directive to set its properties by using JSON object as input. We create a property of type JSON with specific values that we want. Then we decorate is with @Input(‘\_custom\_directive\_selecor\_name’)

Now we pass the values as a JSON to that attribute property.

        <div class="ekart-product-detail-gbc">

            <span [setBackground]="{backColor:'red', textColor: 'white'}" [title]="'Example title'">{{product.gender}}</span>

            <span [setBackground]="{backColor:'black', textColor: 'white'}">{{product.brand}}</span>

            <span [setBackground]="{backColor:'blue', textColor: 'white'}">{{product.category}}</span>

        </div>

import { Directive, ElementRef, Input, Renderer2 } from "@angular/core";

@Directive({

    selector: '[setBackground]'

})

export class SetBackground{

    // @Input('setBackground') backColor: string = '#36454F';

    // @Input() textColor: string = 'White';

    // @Input() title: string = 'Example Title from Directive';

    @Input('setBackground') changeTextAndBackColor: {backColor: string, textColor: string};

    constructor(private element: ElementRef, private renderer: Renderer2){

    }

    ngOnInit(){

        this.renderer.setStyle(this.element.nativeElement, 'backgroundColor', this.changeTextAndBackColor.backColor);

        this.renderer.setStyle(this.element.nativeElement, 'color', this.changeTextAndBackColor.textColor);

    }

}

Conditional Attribute Directive

* We can apply a directive to a DOM element conditionally
* To do this, we use the directive name and create a variable and use @Input on it
* Then we use a **set keyword on that variable to create a setter method**. By using this set keyword, we can use this property as a method.
* Now to receive the value sent form DOM element, we specify an argument to store the value there. Now we use this argument value to write logic inside setter method and manipulate the DOM element accordingly.
* By this, we are able to change the DOM element properties only when certain conditions are met.

Here we are creating a custom directive DisableProduct. We are creating a property disableProduct with same name as selector.

Now we are using @Input to send values for DOM element

We are using set keyword to make this property a setter method. Now we use argument disable to store the sent value. We use this argument value to decide whether to add a class to that element or not using the logic inside set method.

In this way we are using this directive conditionally

Custom Conditional Directive

import { Directive, ElementRef, Input, Renderer2 } from '@angular/core';

@Directive({

  selector: '[disableProduct]'

})

export class DisableProductDirective {

  constructor(private element: ElementRef, private renderer: Renderer2) {

  }

  @Input() set disableProduct(disable: boolean){

    if(disable){

      this.renderer.addClass(this.element.nativeElement, 'disable-out-of-stock-product');

    }

  }

}

DOM element

<div class="ekart--product--item" appHighlight [disableProduct]="!product.is\_in\_inventory">

    <div class="ekart-wishlist-sale-container">

        <div class="ekart--add--to--wishlist">

            <i class="fa fa-heart-o" aria-hidden="true"></i>

        </div>

    </div>

    <div class="ekart--product--image">

        <img [src]="product.imageURL" class="ekart--product--image">

    </div>

    <div class="ekart--product--in--stock" [ngStyle]="{fontWeight : 'bold', color : product.is\_in\_inventory? 'Green':'Red'}">

        {{ product.is\_in\_inventory? 'Available in Stock': 'Out of Stock'}}

    </div>

</div>

.disable-out-of-stock-product{

    /\* background-color: #28282B; \*/

    background-color: rgb(228, 227, 227);

    opacity: 0.5;

}

CSS class (in the same component as DOM element)

.disable-out-of-stock-product{

    /\* background-color: #28282B; \*/

    background-color: rgb(228, 227, 227);

    opacity: 0.5;

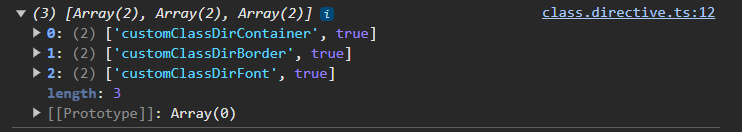
}

Creating a Custom Class Directive

* We have attribute directives like ngClass. We can create such directives by ourselves.
* We don’t usually get the need to create the our custom ngClass but we might need it in some cases
* To do this we create a custom directive. We create a property in that directive and decorate it with @Input to receive values. We use set keyword to make it a setter method. Same as in ngClass, we pass an object with {'\_className’: Boolean, ..} to the argument in setter. We use this argument of type Object and pass it to Object.entries(\_name of argument that has the object).
* Object.entries() method gives us an array of array in which each array has 2 values, one is property name and other is its value
* We iterate this array and check the value according to which we add the class it the value is true.

Here we crated a custom attrubite cirective ClassDirective. We created display property with @Input and set to make it setter. We now pass the Object of {‘class’:Boolean,..} to the argument in setter.

We used Object.entries(value) to get array of arrays of property value pair. We used it to add the classes as shown in the code.



Here we can also use let[value1\_var \_name, value2\_var\_name] = array\_name; Bydoing this we are assigning the values in array to those variables. This is done better name and identify values.

import { Directive, ElementRef, Input, Renderer2 } from '@angular/core';

@Directive({

  selector: '[appClass]'

})

export class ClassDirective {

  constructor(private element: ElementRef, private renderer: Renderer2) { }

  @Input('appClass') set display(value: Object){

    let entries = Object.entries(value);

    console.log(entries);

    // for(let item of entries){

    //   if(item[1]){

    //     this.renderer.addClass(this.element.nativeElement, item[0]);

    //   }

    // }

    for(let item of entries){

      let[className, condition] = item;

      if(condition){

        this.renderer.addClass(this.element.nativeElement, className);

      }

    }

  };

}

DOM element where we are using Custom Class Directive

<div class="customClassDirective t-center">

    <br>

    <h3>Custom Class Directive</h3>

    <div [appClass]="{'customClassDirContainer': true, 'customClassDirBorder': true, 'customClassDirFont': 5 > 2}">

        <p>For those who are interested in finding random paragraphs, that's exactly what this webpage provides.

            you can use this other paragraph generator. Below you can find a number of ways that this generator can be used.</p>

    </div>

    <br><br>

</div>

.customClassDirContainer{

    background-color: aliceblue;

    margin: 0% 5%;

}

.customClassDirBorder{

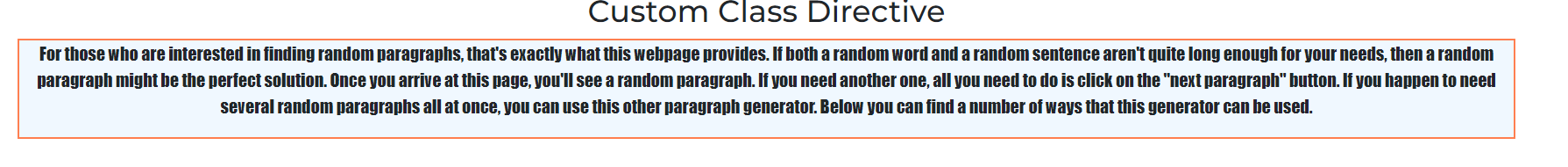
    border: 2px coral solid;

}

.customClassDirFont{

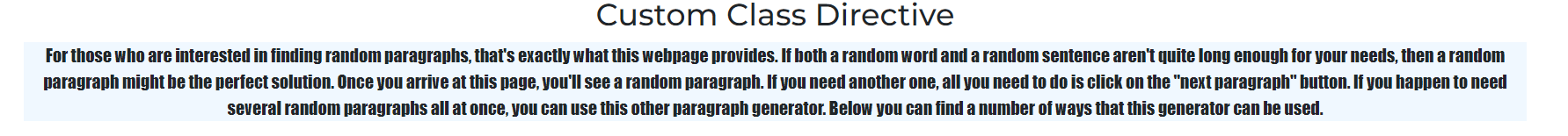
    font-family: Impact, Haettenschweiler, 'Arial Narrow Bold', sans-serif;

}



If I make a value false in object that we are passing (here we made Border class false)

    <div [appClass]="{'customClassDirContainer': true, 'customClassDirBorder': false, 'customClassDirFont': 5 > 2}">



Creating a Custom Style Directive

* Same as the above case, we create a custom Style Directive replicating ngStyle
* For this we create a custom directive. We create a property, use @Input and set on it.
* We pass the Object containing { ’css\_style\_name’: ‘style\_value’, ..} from the DOM element.
* We receive that in argument in setter method. We convert this object into Array of arrays using Object.entries(object\_name).
* We loop these arrays to set the style using Renderer.setStyle()
* Here while we pass object of values, we use ternary operator for style to determine which style to be passed according to the condition. We can directly use the ternary operator.

@Directive({

  selector: '[appStyle]'

})

export class StyleDirective {

  constructor(private element: ElementRef, private renderer: Renderer2) { }

  @Input('appStyle') set style(styles: Object){

    let styleEntries = Object.entries(styles);

    for(let item of styleEntries){

      let[cssStyle, value] = item;

      this.renderer.setStyle(this.element.nativeElement, cssStyle, value);

    }

  }

}

DOM element

<div class="customStyleDirective t-center">

    <br>

    <h3>Custom Style Directive</h3>

    <div class="customStyleDirContainer" [appStyle]="{backgroundColor: active? 'green':'red', color: active? 'white':'black'}">

    <!-- [ngStyle]="{backgroundColor: active? 'green':'red', color: active? 'white':'black'}"> -->

        <p>For those who are interested in finding random paragraphs, that's exactly what this webpage provides.

            you can use this other paragraph generator. Below you can find a number of ways that this generator can be used.</p>

    </div>

    <br><br>

</div>

Component’s Boolean value that we are using to assess the ternary operator

  active: boolean = false;

# Structural Directive

How a structural directive works

* We know structural directive like \*ngIf \*ngFor etc
* We always use \* in-front of a structural directive. Angular identifies the structural directive by that \*
* Then Angular removes the \*ngIf, and wraps the HTML content inside it with <ng-template>
* In this <ng-template> Angular adds [ngIf]. Here Angular is using the structural directive same like attribute directive

Here initially display is false. When we set it to true, the content inside that div is shown

Here Angular converts the below code as shown

HTML

<div class="structuralDirective t-center">

    <h3>Structural Directive</h3>

    <button class="btn btn-primary" (click)="showMore()">Drop</button>

    <div \*ngIf="display">

        <h4>Random Content</h4>

        <p>For those who are interested in finding random paragraphs</p>

        <h4>Random Content 2</h4>

        <p>For those who are interested in finding random paragraphs</p>

    </div>

</div>

<div class="structuralDirective t-center">

    <h3>Structural Directive</h3>

    <button class="btn btn-primary" (click)="showMore()">Drop</button>

    <ng-template [ngIf]="display">

        <div>

            <h4>Random Content</h4>

            <p>For those who are interested in finding random paragraphs </p>

            <h4>Random Content 2</h4>

            <p>For those who are interested in finding random paragraphs</p>

        </div>

    </ng-template>

</div>

TS

  display: boolean = false;

  showMore(){

    this.display = true;

  }

When we have else block in \*ngIf, then Angular converts the code as below

<div \*ngIf="display; else tempDiv">

        <h4>Random Content</h4>

        <p>For those who are interested in finding random paragraphs</p>

        <h4>Random Content 2</h4>

        <p>For those who are interested in finding random paragraphs </p>

    </div>

<ng-template #tempDiv>

        <h4>Temp Div</h4>

        <p>Temp para ....</p>

</ng-template>

<ng-template [ngIf]="display" [ngIfElse]='tempDiv'>

        <div>

            <h4>Random Content</h4>

            <p>For those who are interested in finding random paragraphs</p>

            <h4>Random Content 2</h4>

            <p>For those who are interested in finding random paragraphs</p>

        </div>

</ng-template>

<ng-template #tempDiv>

        <h4>Temp Div</h4>

        <p>Temp para ....</p>

</ng-template>

We can also directly use the converted code. It will produce the same result.

Creating a Custom Structural Directive

***Replication \*ngIf***

* We create a custom directive. We create private arguments inside constructor to inject the template and its container to those variables.
  + 1st argument is of type TemplateRef<any>. This stores the template reference to which we want to use the structural directive on.
  + 2nd argument is of type ViewContainerRef. We know that Angular will wrap the template in <ng-container>. So we will get the reference of that container here.
* We created a property in it with @Input. We use set to make it a setter.
* Now we use this directive with \* before it on a DOM element. By using \* Angular will identify that this is a structural directive and wrap it in <<ng-container>
* We pass a Boolean value from the DOM to the argument in setter method of directive
* We use this argument value and manipulate the structure.
* When argument is true, we use createEmbededView( \_template\_object) method of ViewContainerRef and pass the template object to it. This renders the template inside that container.
* It the argument is false, we use clear() method of ViewContainerRef. This will not render the template and removes it from the container

Here we created custom directive IfDirective. We created 2 objests template: TemplateRef<any> and viewContainer: ViewContainerRef as discussed above.

We created a property display decorated with @Input of alias name as attribute selector. We made it setter method using set and used an argument to receive Boolean value from DOM element. Then we used the methods of viewContainer to add or remove DOM element accordingly.

Custom If directive

import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';

@Directive({

  selector: '[appIf]'

})

export class IfDirective {

  constructor(private template: TemplateRef<any>, private viewContainer: ViewContainerRef) { }

  @Input('appIf') set display(condition: boolean){

    if(condition){

      this.viewContainer.createEmbeddedView(this.template);

    }else{

      this.viewContainer.clear();

    }

  };

}

DOM tempale

<div \*appIf="10 > 5">

            <h4>Random Content</h4>

            <p>For those who are interested in finding random paragraphs</p>

</div>

ngSwitch

* This directive can be used same like a switch statement.
* We can use a variable and switch its value. We add cases to the DOM elements.
* According to the variable value, we show the DOM elements
* We use ngSwitch as an attribute directive like [ngSwitch] = ‘\_variable’
* To specify cases, we use \*ngSwitchCase= “\_value”
* We also have \*ngSwitchDefault which by default renders that element when no cases match.

Here we are using [ngSwitch] and assigned tab variable to it. According to value of tab, we show the following div’s and the Info div is the default div when no cases match.

<div class="ngSwitch t-center">

    <div>

        <button (click)="setTabValue('info')">Info</button>

        <button (click)="setTabValue('service')">Serice</button>

        <button (click)="setTabValue('privacy')">Privacy</button>

        <button (click)="setTabValue('T&C')">T&C</button>

    </div>

    <div [ngSwitch]="tab">

        <div \*ngSwitchDefault>

            <h4>Info</h4>

            <p>Info about the ngSwitch. Info about the ngSwitch.</p>

        </div>

        <div \*ngSwitchCase="'service'">

            <h4>Serice</h4>

            <p>Serice about the ngSwitch. Serice about the ngSwitch </p>

        </div>

        <div \*ngSwitchCase="'privacy'">

            <h4>Privacy</h4>

            <p>Privacy about the ngSwitch. Privacy about the ngSwitch.</p>

        </div>

        <div \*ngSwitchCase="'T&C'">

            <h4>T&C</h4>

            <p>T&C about the ngSwitch. T&C about the ngSwitch.</p>

        </div>

    </div>

</div>

 tab: string = '';

  setTabValue(input: string){

    this.tab = input;

  }

# View Encapsulation

* It is a behaviour in Angular where component CSS styles are encapsulated into the components view and do not affect the rest of the application
* There are three types of View Encapsulation
  + ViewEncapsulation.None
  + ViewEncapsulation.Emulated
  + ViewEncapsulation.ShadowDOM
* By default Angular uses **Emulated** to achieve View Encapsulation.
* By using this, Angular creates a unique attribute to each component (ex: \_ngcontent-oni-c12). That attributes are also used to CSS classes by angular such that those classes are only applied to that components View Template.
* You can use **Null** type to remove encapsulation. This will now not restrict the css to that component and apply this css properties to all the child components present in it aswell.
* But if the child component has a specific css that overwrites that parent css, then child css will be used.
* We can change the encapsulation type in the following way

@Component({

  selector: 'test',

  templateUrl: './test.component.html',

  styleUrl: './test.component.css',

  encapsulation: ViewEncapsulation.None

})

* You can use **ShadowDOM** type when you want to create a separate own DOM of the component which rendered separately by the browser. The feature, state and style of the ShadowDOM stay private and does not get affected by main DOM.
* By doing this, any external or parent css properties will not affect the component’s css. This way we achieve true encapsulation

@Component({

  selector: 'test',

  templateUrl: './test.component.html',

  styleUrl: './test.component.css',

  encapsulation: ViewEncapsulation.ShadowDOM

})

# Service

* Services in Angular are useful in many ways
  + Allows us to re-use a piece of code in different components
  + Allows us to separate business logic from UI logic
  + We can unit test the business logic in a service class separately which makes testing and debugging easier
* In Angular, component class should only be responsible for representing UI to the user. Presentation logic must be separated from business logic to make components more maintainable. Hence we can use a Service
* A service can be used to **pass the data between 2 completely non related components**
* To crate service we use **ng g s service\_name**

import { Injectable } from '@angular/core';

@Injectable({

  providedIn: 'root'

})

export class SubscribeService {

  constructor() { }

  OnSubscribeClicked(type: string){

    alert('Thank you for your '+ type +' subscription. You can access the services now')

  }

}

We can use this service by importing it in the components and creating an instance of it as shown

  OnSubscribe() {

    let subService = new SubscribeService();

    subService.OnSubscribeClicked('monthly');

  }

<div class="header-nav">

        <a href="#" (click)="HomeClicked()">HOME</a>

        <a href="#" (click)="AdminClicked()">ADMIN</a>

        <button class="header-button" (click)="OnSubscribe()">

SUBSCRIBE

</button>

</div>

Dependency Injection

* A dependency is a relationship between two software components where one component relies on the other component to work properly.

For example, in the above section we crated an instance of service in a component

  OnSubscribe() {

    let subService = new SubscribeService();

    subService.OnSubscribeClicked('monthly');

  }

Here we made the Subscribe Service a dependency of that component. Hence, we are tightly coupling the component to that service. Tight coupling reduces flexibility and re-useability.

It also makes testing difficult because when the dependency changes, the class has to change. When class changes, the unit tests change.

To overcome this, we use dependency injection.

To use dependency injection in the above case, we create a constructor and create the parameter of what ever type you want to inject to that parameter.

For Angular to know which type of dependency it has to inject, we specify that in providers in the metadata object of @Compponent decorator

import { Component } from '@angular/core';

import { SubscribeService } from '../../../Services/subscribe.service';

@Component({

  selector: 'app-sidebar',

  templateUrl: './sidebar.component.html',

  styleUrl: './sidebar.component.css',

  providers: [SubscribeService]

})

export class SidebarComponent {

  //1. HOW TO PROVIDE DEPENDENCY

  constructor(private subService: SubscribeService) {

  }

  OnSubscribeClicked() {

    this.subService.OnSubscribeClicked('quaterly');

  }

}

* Dependency Injection is a technique using which a class receives its dependencies from an external source rather than creating them itself.
* Advantages
  + Keeps code flexible, testable and mutable
  + Classes can inherit external logic without knowing how to create it
  + It benefits components, directives and pipes

Hierarchical Dependency Injection

* + When we provide a dependency on a component, the same instance of that dependency is injected in component class and all its child components and their child components. This is called hierarchical injection.
  + When we are using ‘providers’ in all the components, each component will create a different instance of the service. If we use the providers only in the **parent component**, then all its child components will use the service instance of parent component itself.
  + But you use providers in both parent and child, then child instance overrides the parent’s service instance and using its own new service instance.
  + We can also provide a dependency on **root component** which can be injected to all components, directives and services.
  + We can also inject the service from **module class** by using the providers option available in module file. In this way, we implement singleton pattern where a single instance is shared throughout the application.

@NgModule({

  declarations: [

    AppComponent,

    HeaderComponent,

    HeroComponent,

    SidebarComponent

  ],

  imports: [

    BrowserModule,

    AppRoutingModule,

    FormsModule

  ],

  providers: [SubscribeService],

  bootstrap: [AppComponent]

})

export class AppModule { }

**Use case explaining Hierarchical Dependency Injection**

Here we have parent and child components. We are injecting an instance of UserService to both of them. Here, parent component is creating the new user. Child component is getting all the users present in users array of the service.

But we created 2 different instances of UserService to both parent and child. Hence, when we add a new user in parent component, only its instance of the UserSerivce updates its users array. We cannot access this updates users array from child because the instance of UserService that it has is different and affected by parent.

If we remove the providers in child component, then the child will use the same instance of service as the parent and can now access the updated users array.

We can also declare the UserService in the providers of root app module so that all the components present in the app will use the same instance of the service.

Parent component

@Component({

  selector: 'app-admin',

  templateUrl: './admin.component.html',

  styleUrl: './admin.component.css',

  providers: [UserService]

})

export class AdminComponent {

  constructor(private userService: UserService){

  }

  name: string = '';

  gender: string = 'Male';

  subType: string = 'Yearly';

  status: string = 'Active';

  CreateNewUser(){

    this.userService.CreateUser(this.name, this.gender, this.subType, this.status);

    console.log(this.userService.users)

  }

}

Child component

@Component({

  selector: 'app-user-list',

  templateUrl: './user-list.component.html',

  styleUrl: './user-list.component.css',

  providers: [UserService]

})

export class UserListComponent {

  constructor(private userService: UserService){

  }

  userList = this.userService.GetAllUsers();

  // ShowUserDetails(user: User){

  //   this.userService.OnShowUserDetails(user);

  // }

}

User Service

export class UserService {

  constructor() { }

  users: User[] = [

    new User('Steve Smith', 'Male', 'Monthly', 'Active'),

    new User('Mery Jane', 'Female', 'Yearly', 'Inactive'),

    new User('Mark Tyler', 'Male', 'Quaterly', 'Active')

  ];

  GetAllUsers(){

    return this.users;

  }

  CreateUser(name: string, gender: string, subType: string, status: string){

    let user = new User(name, gender, subType, status);

    this.users.push(user);

  }

}

Injecting a Service into another Service

* A service can also be injected to another service.
* To do this we need to add @Injectable decorator to a meta data
* Now we should declare the Service that we need to inject in the providers of the app module

Here we have 2 services Logger and User. We are using Logger service in the User service to log a message when a user is added.

To do this we are using @Injectable in UserService and we are creating a parameter of type LoggerService in the constructor. Now we are declaring the LoggerService in providers in app module. Angular injects the LoggerService into logger. We can use this logger to access the properties of LoggerService.

Logger Service

@Injectable({

    providedIn: 'root'

})

export class LoggerService{

    LogMessage(name: string, status: string){

        console.log(`A new user with name ${name} with status ${status} is added to user list.`)

    }

}

User Service

@Injectable({

  providedIn: 'root'

})

export class UserService {

  constructor(private logger: LoggerService) { }

  users: User[] = [

    new User('Steve Smith', 'Male', 'Monthly', 'Active'),

  ];

  GetAllUsers(){

    return this.users;

  }

  CreateUser(name: string, gender: string, subType: string, status: string){

    let user = new User(name, gender, subType, status);

    this.users.push(user);

    this.logger.LogMessage(name, status);

  }

}

App Module

@NgModule({

  declarations: [

    AppComponent,

    HeaderComponent,

    TopHeaderComponent,

  ],

  imports: [

    BrowserModule,

    AppRoutingModule,

    FormsModule

  ],

  providers: [LoggerService],

  bootstrap: [AppComponent]

})