**Angular 8 directives can be classified in 3 categories based on how they behave:**

* Component Directives
* Structural Directives
* Attribute Directives

**Component Directives:** Component directives are used in main class. They contain the detail of how the component should be processed, instantiated and used at runtime.

**Structural Directives:** Structural directives start with a \* sign. These directives are used to manipulate and change the structure of the DOM elements. For example, \*ngIf directive, \*ngSwitch directive, and \*ngFor directive.

* **\*ngIf Directive:** The ngIf allows us to Add/Remove DOM Element.
* **\*ngSwitch Directive:** The \*ngSwitch allows us to Add/Remove DOM Element. It is similar to switch statement of C#.
* **\*ngFor Directive:** The \*ngFor directive is used to repeat a portion of HTML template once per each item from an iterable list (Collection).

**Attribute Directives:** Attribute directives are used to change the look and behavior of the DOM elements. For example: ngClass directive, and ngStyle directive etc.

* **ngClass Directive:** The ngClass directive is used to add or remove CSS classes to an HTML element.
* **ngStyle Directive:** The ngStyle directive facilitates you to modify the style of an HTML element using the expression. You can also use ngStyle directive to dynamically change the style of your HTML element.
* Below is the example of ngIf. ServerCreated will be resolve to true or false if it’s false then noServer ng-template will get called.

<p \*ngIf="serverCreated else noServer"> server was created.

Name is {{ serverName }}</p>

<ng-template #noServer>

<P>No Server was created!</P>

</ng-template>

* Unlike Structural directive attribute directive don’t add or remove the elements they only change the element they were placed
* **@Input & @Output** used for component interaction, from parent to child component if we want to pass something then we can use @Input and if we want to pass something from child to parent then we can use @Output directive.
* *you can pass data from child component to parent component using @Output()*
* *you can pass data from parent component to child component using @Input()*
* In the app-cockpit components
* <button
* class="btn btn-primary"
* (click)="onAddServer(serverNameInput)">Add Server</button>
* <button
* class="btn btn-primary"
* (click)="onAddBlueprint(serverNameInput)">Add Server Blueprint</button>
* @Output() serverCreated = new EventEmitter<{serverName: string, serverContent: string}>();
* @Output('bpCreated') blueprintCreated = new EventEmitter<{serverName: string, serverContent: string}>();
* onAddServer(nameInput: HTMLInputElement) {
* this.serverCreated.emit({
* serverName: nameInput.value,
* serverContent: this.serverContentInput.nativeElement.value
* });
* }
* onAddBlueprint(nameInput: HTMLInputElement) {
* this.blueprintCreated.emit({
* serverName: nameInput.value,
* serverContent: this.serverContentInput.nativeElement.value
* });
* }
* In the app-componenets
* <app-cockpit
* (serverCreated)="onServerAdded($event)"
* (bpCreated)="onBlueprintAdded($event)"
* ></app-cockpit>
* onServerAdded(serverData: {serverName: string, serverContent: string}) {
* this.serverElements.push({
* type: 'server',
* name: serverData.serverName,
* content: serverData.serverContent
* });
* }
* onBlueprintAdded(blueprintData: {serverName: string, serverContent: string}) {
* this.serverElements.push({
* type: 'blueprint',
* name: blueprintData.serverName,
* content: blueprintData.serverContent
* });

In the above example we are passing data from child cockpit component to app-components using eventEmitter we are emitting the value and listening to that in app-component

* **Encapsulation**: ViewEncapsulation by default it’s value is Emulated this is used when we want to use css to the other component also then we can use this.Emulated property will apply all our css written in that component only.
* encapsulation: ViewEncapsulation.Emulated // None, Native
* Local reference template we can use **#anyName** in the template to access the value of input html element we can use this is html component only we can’t use in typescript but we can access is ts also by passing it in method.
* <input type="text" class="form-control" #serverNameInput>
* <button class="btn btn-primary"
* (click)="onAddServer(serverNameInput)">Add Server</button>
* <button
* onAddServer(nameInput: HTMLInputElement) {
* this.serverCreated.emit({
* serverName: nameInput.value,
* **@ViewChild** this is used when you want to access local reference template of child in parent. If you want to access following inside the Parent Component, use @ViewChild decorator of Angular.
* Child Component
* Directive
* DOM Element

ViewChild returns the first element that matches the selecto

* Any component, directive, or element which is part of a template is ViewChild and any component or element which is projected in the template is ContentChild.
* **<ng-content></ng-content>** use to display the html content of parent element written in child selector
* @Component({
* selector: "app-child",
* template: `
* <p class='hand'>
* child body!
* </p>
* <p class='foot'>
* child foot!
* </p>
* `
* })
* export class ChildComponent {
* public name: string = "get child name";
* }
* <app-parent>
* <app-child #target></app-child>
* <app-child class="hand"></app-child>
* </app-parent>

Below is the main app-component

* @Component({
* selector: "app-parent",
* template: `
* <p \*ngIf="childCmp">
* {{childCmp.name}}
* </p>
* <ng-content select='.hand'></ng-content>
* `
* })
* export class ParentComponent {
* @ContentChild("target") childCmp: ChildComponent;
* ngAfterContentInit() {
* console.dir(this.childCmp);
* }
* }

You use the <ng-content></ng-content> tag as a placeholder for that dynamic content, then when the template is parsed Angular will replace that placeholder tag with your content. Think of it like curly brace interpolation, but on a bigger scale. The technical term for this is “content projection" because you are projecting content from the parent component into the designated child component.

<app-server-element> this is written in app-componenet

…

….

</app-server-element>

Whatever written inside this element will be projected to <ng-content> in placed in the app server element

* **LifeCycle hooks**

ngOnChanges()🡪 Respond when Angular (re)sets data-bound input properties. The method receives a SimpleChanges object of current and previous property values.

Called before ngOnInit() and whenever one or more data-bound input properties change.

ngOnInit()🡪 Initialize the directive/component after Angular first displays the data-bound properties and sets the directive/component's input properties.

Called once, after the first ngOnChanges().

ngDoCheck()🡪 Detect and act upon changes that Angular can't or won't detect on its own.

Called during every change detection run, immediately after ngOnChanges() and ngOnInit().

ngAfterContentInit()🡪 Respond after Angular projects external content into the component's view / the view that a directive is in.

Called once after the first ngDoCheck().

ngAfterContentChecked()🡪 Respond after Angular checks the content projected into the directive/component.

Called after the ngAfterContentInit() and every subsequent ngDoCheck().

ngAfterViewInit()🡪 Respond after Angular initializes the component's views and child views / the view that a directive is in.

Called once after the first ngAfterContentChecked().

ngAfterViewChecked()🡪 Respond after Angular checks the component's views and child views / the view that a directive is in.

Called after the ngAfterViewInit() and every subsequent ngAfterContentChecked().

ngOnDestroy()🡪 Cleanup just before Angular destroys the directive/component. Unsubscribe Observables and detach event handlers to avoid memory leaks.

Called just before Angular destroys the directive/component.

* **@Directive** is used at class level to create our own directive like giving some css property etc.
  + We can use ElementRef of angular core to set the css property.
* @Directive({
* selector: '[appBasicHighlight]'
* })
* export class BasicHighlightDirective implements OnInit {
* constructor(private elementRef: ElementRef) {
* }
* ngOnInit() {
* this.elementRef.nativeElement.style.backgroundColor = 'green';
* }
* }

@HostListener and @HostBinding directive are use to bind some property to the html dom.

@Directive({

selector: '[appBetterHighlight]'

})

export class BetterHighlightDirective implements OnInit {

@Input() defaultColor: string = 'transparent';

@Input('appBetterHighlight') highlightColor: string = 'blue';

@HostBinding('style.backgroundColor') backgroundColor: string;

@HostListener('mouseenter') mouseover(eventData: Event) {

this.backgroundColor = this.highlightColor;

}

@HostListener('mouseleave') mouseleave(eventData: Event) {

this.backgroundColor = this.defaultColor;

}

* **Sevices** are use when there is a code which is common to component that we can put in the services.Services are created by giving the name as

App.service.ts it’s the simple type script class without any decorator to

But to use this service in our component we need to use Providers in @Component({

Providers: [nameOfServices]}

To Inject service into the services we need to use @Injectable() annotation on the service where we are injecting the other services.

Services need to define commonly for the the component if we define the services in app module ts it will be available for all the components. Services are define inside providers

* If we provide the service in **appModule** same instance of service is available application wide.
* If we provide service in **appComponents** same instance of service is available for all component but not for other services.
* Any other component same instance of service is available for the component and all its child components.

**Routers:**

We need to import router from angular router to use it in our application.

routerLink is use to route to specific path we should provide relative path to router link

we can navigate from on component to another by clicking the button.

* The builtin matching strategies are prefix (the default) and full.
* When the matching strategy of a route is prefix, the router will simply check if the start of the browser’s URL is prefixed with the route’s path. If that’s the case, it will render the related component.

constructor(private serversService: ServersService, private router: Router) { }

ngOnInit() {

this.servers = this.serversService.getServers();

}

onReload() {

this.router.navigate(['/servers']);

}

Router.navigate we can provide relative path as shown above or absolute path it won’t give any error.But for routerLink we need to provide the relativepath

<div class="col-xs-12 col-sm-10 col-md-8 col-sm-offset-1 col-md-offset-2">

<ul class="nav nav-tabs">

<li role="presentation" routerLinkActive="active" [routerLinkActiveOptions]="{exact: true}">

<a routerLink="/">Home</a>

</li>

<li role="presentation" routerLinkActive="active" ><a routerLink="/servers">Servers</a></li>

<li role="presentation" routerLinkActive="active"><a [routerLink]="[ '/users']">Users</a></li>

</ul>

</div>

</div>

<div class="row">

<div class="col-xs-12 col-sm-10 col-md-8 col-sm-offset-1 col-md-offset-2">

<router-outlet></router-outlet>

</div>

routerLinkActiveOptions is use to make the navigation bar active inacitive.

Whatever path we give in routerLink the corresponding component will be shown at

<Router-outlet></Router-outlet>

***ActivatedRoute***this is use to get the current activated route. Snapshot will load the parameter first time only when the component get created next time you click on link then parameters will get change but it will no listen in snapshot so we need to subscribe to params to get the updated value. This Is called observable

constructor(private route: ActivatedRoute) { }

ngOnInit() {

this.user = {

id: this.route.snapshot.params['id'],

name: this.route.snapshot.params['name']

};

this.route.params.subscribe(

(params: Params) => {

this.user.id = params['id'];

this.user.name = params['name'];

}

);

**CanActivate** and **CanDeactivate** guard in routing.

* ActivatedRoute service can be used to extract route parameters inside component

**Observable:** various data sources (User input)Events, Http Requests Triggered in code.

export class HomeComponent implements OnInit , OnDestroy {

  private firstObsSubsciption: Subscription;

  constructor() { }

  ngOnInit() {

    this.firstObsSubsciption = interval(1000).subscribe( (count) => {

      console.log(count);

    });

  }

  ngOnDestroy() {

    this.firstObsSubsciption.unsubscribe();

  }

To unsubscipe the observable we need to implement ngOnDestroy otherwise it will keep counting for all the component we need count only in homecomponent.

We can create our custom observable and subscribe to it.

 ngOnInit() {

    const customIntervalObservable = Observable.create( (observer) => {

      let count = 0;

      setInterval( () => {

        observer.next(count);

        if(count === 2) {

          observer.complete();

        }

        if(count > 3) {

          observer.error(new Error('Count is Greater then 3'));

        }

        count++;

      },1000);

    });

    this.firstObsSubsciption = customIntervalObservable.subscribe(data => {

      console.log(data);

    },error => {

      console.log(error);

      alert(error);

    }, () => {

      console.log('completed');

    });

  }

* With a Promise, you can only handle one event.
* With an Observable, you can handle multiple events.
* .subscribe () is similar to .then ().

An RxJS Subject is a special type of Observable that allows values to be multicasted to many Observers. While plain Observables are unicast (each subscribed Observer owns an independent execution of the Observable), Subjects are multicast.

* **Subject :** A subject is imported from “rxjs” package. A subject is an active listener. it triggers with next(). A subject is a special kind of observable. It maintains the stream of data which is maintained actively. A subject is an observer and observable. Event emitter built on top of the subject. It is good practice to use subject. Subscribe to consume the data. Unsubscribe in case of leaving the component to avoid the memory leak.
* Event emitter is built on top of the subject**. A subject is better to use in case of cross-component communication**.Event emitter is built on top of the subject. A subject is better to use in case of cross-component communication.
* Services are a great way to share information among classes that don’t know each other.Services are a great way to share information among classes that don’t know each other.
* Event-emitter is imported from “@anguar/core” package. It is used in directives and components to emit custom events synchronously or asynchronously and register handlers for those events by subscribing to an instance. Event emitter is a generic type, it can take any additional information.

It uses ‘@Output’ and ‘@Input’ decorators

**Observable.subscribe():** The observable subscribe method is used by angular components to subscribe to messages that are sent to an observable.

**Subject.next():** The subject next method is used to send messages to an observable which are then sent to all angular components that are subscribers (a.k.a. observers) of that observable.

The handler for receiving the observable notifications implements the Observer interface. It is an object that defines the callback methods to handle the three types of notifications that an observable can send. These are the following.

* next: Required. The handler for each delivered value called zero or more times after execution starts.
* error: Optional. The handler for error notification. The error halts the execution of the observable instance.
* complete: Optional. The handler for an execution-complete notification. The delayed values can continue to be delivered to a next handler after execution is complete.

**Angular Forms:**

there are 2 approaches to handle user's input through forms:

* Reactive forms
* Template-driven forms

Reactive Forms

* Reactive forms are more robust.
* Reactive forms are more scalable, reusable, and testable.
* They are most preferred to use if forms are a key part of your application, or your application is already built using reactive patterns. In both cases, reactive forms are best to use.

Template-driven Forms

* Template-driven forms are best if you want to add a simple form to your application. For example: email list signup form.
* Template-driven forms are easy to use in the application but they are not as scalable as Reactive forms.
* Template-driven forms are mainly used if your application's requires a very basic form and logic. It can easily be managed in a template.
* FormsModule import this in appModule for Template driven approach, ReactiveFormsModule import this for reactive forms.
* **FormControl:** It tracks the value and validation status of the individual form control.
* **FormGroup:** It tracks the same values and status for the collection of form controls.
* **FormArray:** It tracks the same values and status for the array of the form controls.
* **ControlValueAccessor:** It creates a bridge between Angular FormControl instances and native DOM elements.

**ngStyle:**

<div [ngStyle]="{'color': color, 'font-size': size, 'font-weight': 'bold'}">

style using ngStyle

</div>

ngStyle is attribute directive which use to style the html elements

**ngClass:**

The CSS classes are updated as follows, depending on the type of the expression evaluation:

* string - the CSS classes listed in the string (space delimited) are added,
* Array - the CSS classes declared as Array elements are added,
* Object - keys are CSS classes that get added when the expression given in the value evaluates to a truthy value, otherwise they are removed.

<div class="container">

<div \*ngFor="let car of cars" [ngClass]="{

'alert-primary':car.name === 'MG Hector',

'alert-secondary':car.name === 'Ford',

'alert-success':car.name === 'Kia',

'alert-danger':car.name === 'BMW',

'alert-warning':car.name === 'Jaguar',

'alert-info':car.name === 'Suzuki'

}">

{{ car.name }}

</div>

</div>

Bootstrap css classes alert-primary etc will get added based on the expression evaluation.

* **ng lint** command is used to run Static Code analysis of Angular application
* One component can be can’t declared inside more than one NgModule
* OnPush is faster in performance than Default change detection strategy