**Angular**

Angular is a framework for building client applications in HTML and TypeScript

**Component**

A component controls a part of the web application screen. It consists of JavaScript (or precisely TypeScript) code, HTML code and CSS

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

@Component({

selector: "app-root",

templateUrl: "./app.component.html",

styleUrls: ["./app.component.css"]

})

export class AppComponent {

title = "Angular";

}

**CLI command**

ng g component MyComponent / ng g c MyComponent

**Data Binding**

data binding provide the communication between a component class and the template that its associated with

* interpolation Binding
* Property Binding

set the html element attribute value to the value of a template expression

* Event Binding

Event Binding allows our component to listen to events triggered by user actions in the view

* Two-Way Binding

The square brackets indicate Property Binding and the parenthesis indicate Event Binding to send a notification of the user entered data back to the property

**Decorators**

***Decorators*** are functions that modify JavaScript classes. Decorators are used for attaching metadata to classes, it knows the configuration of those classes and how they should work.

In TypeScript, decorators are implemented using functions. Actually it's a function that returns a function. The outside function is called as decorator factory and the inside one is called as the decorator. The decorator factory takes some configuration values and returns a decorator.

Angular uses quite a lot of decorators. There are decorators for classes, properties, methods and even parameters. Some of the important decorators are:

* NgModule
* Component
* Input
* Output

The *Component* decorator is used to decorate a class as an angular component and adds additional metadata information like the template, selector name, styles etc. to it.

The *Component* decorator does the following things.

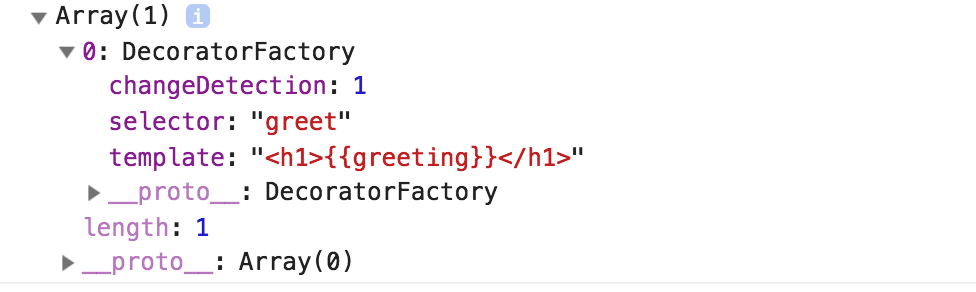
* Creates an instance of a function called *DecoratorFactory* (derived from *Directive*).
* Fills that instance with the passed arguments (selector, template etc).
* Defines a static property in the component type with name "\_\_annotations\_\_" and assign the instance to it.

To verify this, please run the below statement.

|  |  |
| --- | --- |
| 1 | console.log(GreetingComponent['\_\_annotations\_\_']); |

*Listing 5. Displaying GreetingComponent's Annotations*

You'll see the below output.

[](http://prideparrot.com/demos/hollywood/index.html)

*GreetingComponent's annotations*

**Directives**

Provide new behavior or modify existing behavior of HTML elements

There are three kinds of directives in Angular:

1. Components—directives with a template.
2. Structural directives—change the DOM layout by adding and removing DOM elements.
3. Attribute directives—change the appearance or behavior of an element, component, or another directive.

Structural directives alter layout by adding, removing, and replacing elements in DOM.

Let us briefly understand the two majorly used built-in structural directives:

<li \*ngFor="let movie of movies"></li>

<movie-detail \*ngIf="selectedMovie"></movie-detail>

Attribute directive alter the appearance or behavior of an existing element

<input [(ngModel)]="movie.name">

ngSwitch , ngStyle and ngClass

**Component Vs Directive**

**A @Component requires a view whereas a @Directive does not.**

Directives

Directives add behaviour to an existing DOM element or an existing component instance. One example use case for a directive would be to log a click on an element.

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

@Directive({

selector: "[logOnClick]",

hostListeners: {

'click': 'onClick()',

},

})

class LogOnClick {

constructor() {}

onClick() { console.log('Element clicked!'); }

}

Which would be used like so:

<button logOnClick>I log when clicked!</button>

Components

A component, rather than adding/modifying behaviour, actually creates its own view (hierarchy of DOM elements) with attached behaviour. An example use case for this might be a contact card component:

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

@Component({

selector: 'contact-card',

template: `

<div>

<h1>{{name}}</h1>

<p>{{city}}</p>

</div>

`

})

class ContactCard {

@Input() name: string

@Input() city: string

constructor() {}

}

Which would be used like so:

<contact-card [name]="'foo'" [city]="'bar'"></contact-card>

**What is a typescript?**

Typescripts is superset of javascripts its mean whatever we do in javascript it can be possible in typescript. In the end typescript code will be compiled in javascripts code (tsc)

Browser understand javascript not typescript

In Interface or class to make property as optional use ? after property name

interface Employee{

Name:string

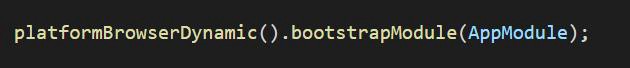
Age?:number

}

AngularJS is MVC arch but angular 2+ follow component and Module archi

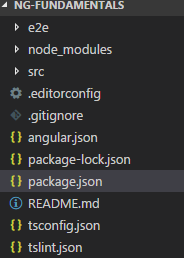
**Angular code flow**

Then entry point to every Angular application is the main.ts file which contains this last line:



In our AppModule, we then need to specify the component that will serve as the entry point component for our application. This happens in our app.module.ts file where we import the entry component (conventionally AppComponent) and supply it as the only item in our bootstrap array inside the NgModule configuration object.

https://cdn-images-1.medium.com/max/800/1*KL1JJ4iAP_it0VJyHjNH0A.jpeg





 This file holds various metadata relevant to the project. This file is used to give information to npm that allows it to identify the project as well as handle the project's dependencies

The package.json is organized into two groups of packages:

* [Dependencies](https://angular.io/guide/npm-packages#dependencies) are essential to *running* applications.
* [DevDependencies](https://angular.io/guide/npm-packages#dev-dependencies) are only necessary to *develop* applications.

**For angular.json see below link**

<https://angular.io/guide/workspace-config>

**NgModules**

An NgModule is a class marked by the @[NgModule](https://angular.io/api/core/NgModule) decorator. @[NgModule](https://angular.io/api/core/NgModule) takes a metadata object that describes how to compile a component's template and how to create an injector at runtime. It identifies the module's own components, directives, and pipes, making some of them public, through the [exports](https://angular.io/api/core/NgModule#exports) property, so that external components can use them. @[NgModule](https://angular.io/api/core/NgModule) can also add service providers to the application dependency injectors.

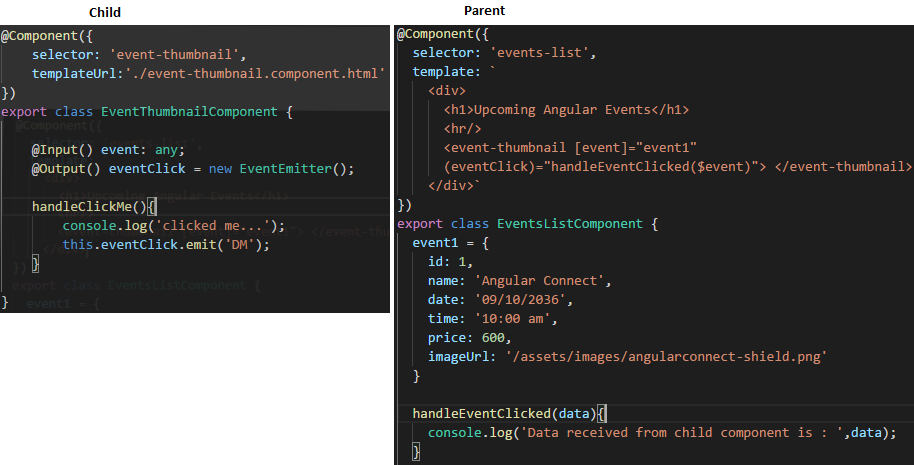
NgModule metadata does the following:

* Declares which components, directives, and pipes belong to the module.
* Makes some of those components, directives, and pipes public so that other module's component templates can use them.
* Imports other modules with the components, directives, and pipes that components in the current module need.
* Provides services that the other application components can use.



@Input

@Input decorator binds a property within one component (child component) to receive a value from another component (parent component). This is one way communication from parent to child



@Output decorator binds a property of a component to send data from one component (child component) to calling component (parent component).

## Template reference variables ( #var )

Use the hash symbol (#) to declare a reference variable. The #phone declares a phone variable on an <input>

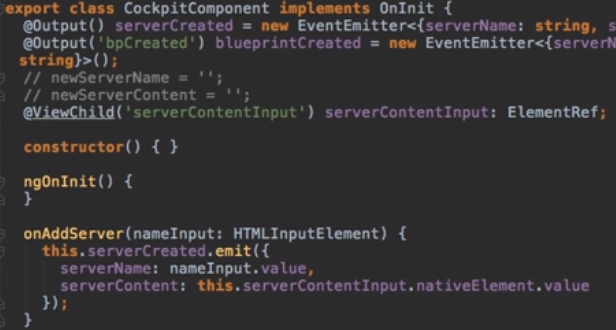
<input #phone placeholder="phone number">

You can refer to a template reference variable anywhere in the template. The phone variable declared on this <input> is consumed in a <button> on the other side of the template

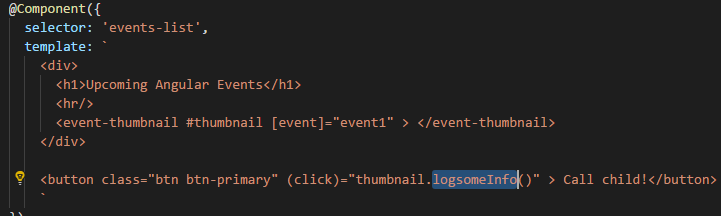
But this phone variable can not be accessible directly in component or typescript code

But we can access it by @viewChild decorator in component like below

**@ViewChild(‘phone’) myPhone: ElementRef**



<!-- phone refers to the input element; pass its `value` to an event handler --> <button (click)="callPhone(phone.value)">Call</button>



You can use the ref- prefix alternative to #. This example declares the fax variable as ref-fax instead of #fax.

<input ref-fax >

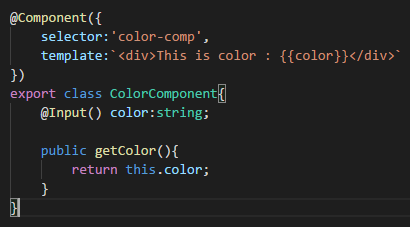
<button (click)="callFax(fax.value)">Fax</button>

Four ways we can interact with parent and child components

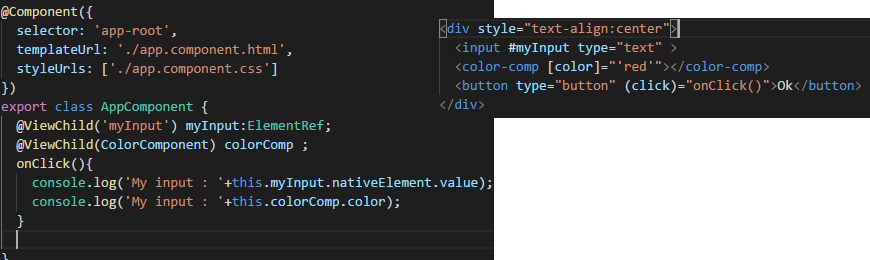
1. @input and @output
2. Template ref variable #
3. @ViewChild
4. ng-content (**content projection**)

With the help of @ViewChild we can access any reference variable(# tag)and any child component’s variable and methods declared in template inside parent component

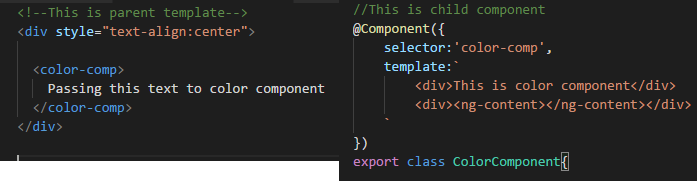
**Child component**



**Patent component**



With the help of ng-content we can pass content from parent to child component This is called **content projection**



Safe Navigation Operator (?.)

Safe navigation operator avoids exception for null and undefined values in property paths. While accessing properties from object it may throw exception if object is null or undefined. For safer side we can use safe navigation operator to access property from object and hence it will not throw exception for the scenario that object is null or undefined. If we have an object person with the properties name then using safe navigation operator we will access object property as follows.

{{personone?.name}}

{{companythree?.owner?.name}}

**Class Binding & NgClass**

If you want to bind single class to element then use below pattern

<div [class.active]="isActive">

...

</div>

If you want to bind one or more than one class then use ngClass binding

We can use ngClass with many ways

[ngClass]="text-success"

[ngClass]="getCurrentClass()"

getCurrentClass(){

return {text-success:true} **or**

return ‘text-success’ **or**

return [‘text-success’]

}

[ngClass]="{'text-success':true}"

[ngClass]="{'text-success':checkMyFunction()}"

[ngClass]="{'text-success':person.country === 'UK'}"

<li [ngClass]="{

'text-success':person.country === 'UK',

'text-primary':person.country === 'USA',

'text-danger':person.country === 'HK'

}">

**NgStyle**

<div [ngStyle]="{'background-color':'green'}"></<div>

<div [ngStyle]="{'background-color':person.country === 'UK' ? 'green' : 'red' }"></<div>

[ngStyle]="{'color':getColor(person.country)}"

[style.font-size.**px**]="24"

<div [style.color]="getColor(person.country)"> </<div>

**Dependency Injection**

* DI is a coding pattern in which a class asks for dependencies from external sources rather than creating them itself
* A component can get services from its own injector, from the injectors of its component ancestors, from the injector of its parent NgModule, or from the root injector.
* The @Injectable() decorator has the providedIn metadata option, where you can specify the provider of the decorated service class with the root injector, or with the injector for a specific NgModule
* The @NgModule() and @Component() decorators have the providers metadata option, where you can configure providers for NgModule-level or component-level injectors.
* Services are singletons within the scope of an injector. That is, there is at most one instance of a service in a given injector.
* Whenever Angular creates a new instance of a component that has providers specified in @Component(), it also creates a new child injector for that instance
* Child modules and component injectors are independent of each other, and create their own separate instances of the provided services
* When Angular destroys an NgModule or component instance, it also destroys that injector and that injector's service instances.

**Optional dependencies**

HeroService requires a logger, but what if it could get by without one?

When a component or service declares a dependency, the class constructor takes that dependency as a parameter. You can tell Angular that the dependency is optional by annotating the constructor parameter with @Optional().

When using @[Optional](https://angular.io/api/core/Optional)(), your code must be prepared for a null value. If you don't register a logger provider anywhere, the injector sets the value of logger to null.

constructor(@Optional() private logger: Logger) {

if (this.logger) {

this.logger.log(some\_message);

}

}

@[Inject](https://angular.io/api/core/Inject)() and @[Optional](https://angular.io/api/core/Optional)() are parameter decorators.

Angular 6 brought us new better providedIn syntax for registration of services into Angular dependency injection mechanism

component providers are made available to the component and all its view AND content child components

View providers are only made available for the component and its view child components. They are declared by the viewProviders option in the component decorator



The RandomService is registered in the providers: [] of the RandomComponent so we will get different random number every time we use <random></random> component in our template.

Before  providedIn, libraries had to provide all heir publicly available services in the providers: []field of the main module

In providedIn metadata we can not use component name like below



# Why do we have to use @Injectable()

@Component({

selector: 'ponyracer-app',

template: '<h1>PonyRacer</h1>'

})

export class PonyRacerAppComponent {

constructor(private appService: AppService) {

console.log(appService);

}

}

export class AppService {

constructor() {

console.log('new app service');

}

}

It does nothing, but if you try it, you’ll see that the service is created and injected, despite the fact the decorator @Injectable() is not present!

But let’s say that now, our AppService has a dependency itself:

export class AppService {

constructor(http: HttpService) {

console.log(http);

}

}

If we launch our app again, we’ll now have an error:

Error: Can't resolve all parameters for AppService: (?).

If we add the @Injectable() decorator, the app works again

So add the @Injectable() decorator on a service if this service has some dependencies itself!

The best practice is to add it on every service, even if it doesn’t have any dependencies on its own.

**Lifecycle Hooks**

You don't have to add the lifecycle hook interfaces to directives and components to benefit from the hooks themselve

Angular instead inspects directive and component classes and calls the hook methods if they are defined. Angular finds and calls methods like ngOnInit(), with or without the interfaces.

it's good practice to add interfaces to TypeScript directive classes in order to benefit from strong typing and editor tooling.

1. **ngOnChanges()**

* Used in pretty much any component that has an input.
* Called whenever an input value changes
* Is called the first time before ngOnInit

1. **ngOnInit()**

* Used to initialize data in a component.
* Called after input values are set when a component is initialized.
* Added to every component by default by the Angular CLI.
* Called only once

1. **ngDoCheck()**

* Called during all change detection runs
* A run through the view by Angular to update/detect changes

1. **ngAfterContentInit()**

* Called only once after first ngDoCheck()
* Called after the first run through of initializing content

1. **ngAfterContentChecked()**

* Called after every ngDoCheck()
* Waits till after ngAfterContentInit() on first run through

1. **ngAfterViewInit()**

* Called after Angular initializes component and child component content.
* Called only once after view is initialized

1. **ngAfterViewChecked()**

* Called after all the content is initialized and checked. (Component and child components).
* First call is after ngAfterViewInit()
* Called after every ngAfterContentChecked() call is completed

1. **ngOnDestroy()**

* Used to clean up any necessary code when a component is removed from the DOM.
* Fairly often used to unsubscribe from things like services.
* Called only once just before component is removed from the DOM.

In my experience as an Angular developer, I primarily use only four of these hooks. Mostly because I don’t want to do something to a component after the content has already been checked.

ngOnChanges()

ngOnInit()

ngAfterViewInit()

ngOnDestory()

The first two I use fairly frequently. They are very useful when dealing with input values or setting your component state based on outside data. The other two are very use case specific. If for some reason you need to do something after your component content has been set, use ngAfterViewInit. As I stated above, clean up your component with ngOnDestory().

**User Define Attribute directive or Custom Directive**

@Directive({

selector: '[appBasicHighlight]'

})

export class BasicHighlightDirective implements OnInit{

constructor(private elementRef:ElementRef) { }

ngOnInit(){

this.elementRef.nativeElement.style.backgroundColor="green";

}

}

<div style="text-align:center">

<p appBasicHighlight>Highlight me...</p>

</div>

Stop manipulating DOM with ElementRef

Relying on direct DOM access creates tight coupling between your application and rendering layers which will make it impossible to separate the two and deploy your application into a web worker.

Stop using ElementRef and use Renderer only

@Directive({

selector: '[appBetterHighlight]'

})

export class BetterHighlightDirective implements OnInit {

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

ngOnInit() {

this.renderer.setStyle(this.elementRef.nativeElement,'background-color','blue');

}

}

**@HostListener** is a function decorator that accepts an event name as an argument. When that event gets fired on the host element it calls the associated function.



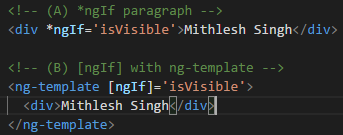
We can access the host element in directive with below ways

1. ElementRef
2. Renderer2
3. @HostBinding

**@HostBinding** directive can change the properties of the host element, such as the list of classes that are set on the host element as well as a number of other properties.



If we don’t want to use \* in ngIf then we have to use ng-template as shown below



Both ways will generate same output in DOM

**Custom Structural Directive**

I want to create my own custom structural directive like \*ngIf



**What is Shadow DOM ?**

It allows us to scope styles to a specific component without affecting the outer world.

Shadow DOM allows us to hide DOM logic behind other elements

In Shadow DOM, Angular write all the styles into the shadowRoot which will enable style encapsulation.

The Shadow DOM is part of Web Components. now angular has its own way to use shadow dom to scope element styles

with **ViewEncapsulation.Emulated** (default)

## V[iew Encapsulation Types](https://blog.thoughtram.io/angular/2015/06/29/shadow-dom-strategies-in-angular2.html#view-encapsulation-types)

* **ViewEncapsulation.None** - No Shadow DOM at all. Therefore, also no style encapsulation.
* **ViewEncapsulation.Emulated** - No Shadow DOM but style encapsulation emulation.
* **ViewEncapsulation.Native** - Native Shadow DOM with all it’s goodness.

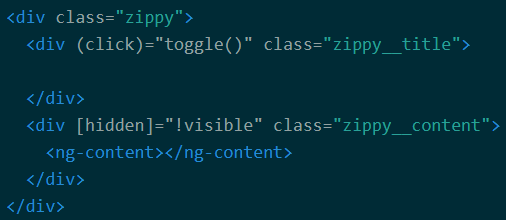
**ViewEncapsulation.None**

Angular doesn’t use Shadow DOM at all. Styles applied to our component are written to the document head Or in other words, a component could overwrite styles from another component because its styles are applied to the document head later.

That’s why this is the **unscoped** strategy



And template



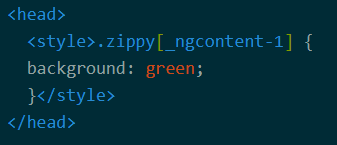
Will make Angular creating a DOM like this:

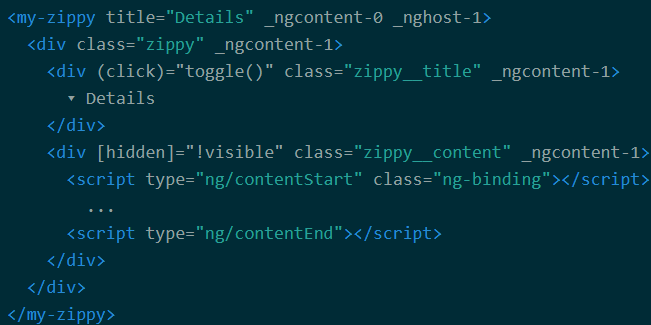


**ViewEncapsulation.Emulated**

This view encapsulation is used by default. ViewEncapsulation.Emulated emulates style encapsulation, even if no Shadow DOM is available. This is a very powerful feature in case you want to use a third-party component that comes with styles that might affect your application

Here’s what the head looks like with the exact same component but different strategy:





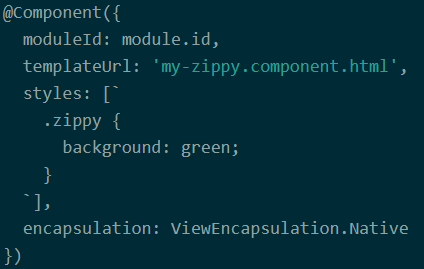
Angular added some attributes to our component’s template as well! We see the \_ngcontent-1 attribute which is also used in our rewritten CSS. So what the hell is going on there?

Actually it’s quite simple. We want scoped styles without Shadow DOM right?

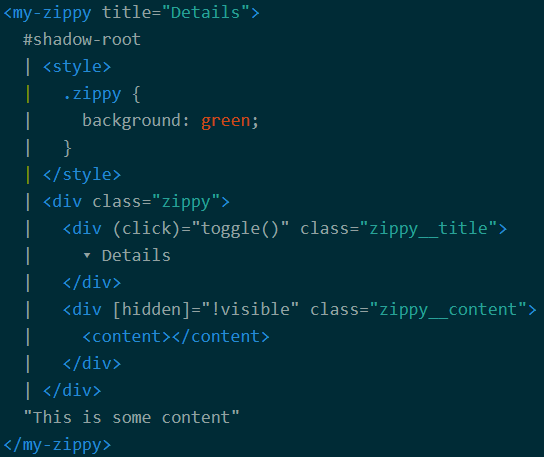
But in order to enable scoped styles, Angular has to make sure that the component’s style selectors only match this particlar component and nothing else on the page. That’s why it extends the CSS selectors, so they have a higher specificity and don’t collide with other selectors defined before at the same. And of course, to make those selectors actually match, the elements in the template need to be extended as well. That’s why we see all those \_ngcontent-\* and \_nghost-\* attributes.

**ViewEncapsulation.Native**

Last but not least, we have the native Shadow DOM view encapsulation. This one is super simple to understand since it basically just makes Angular using native Shadow DOM. We can activate it the same way we did with the other types. Here’s what that looks like:

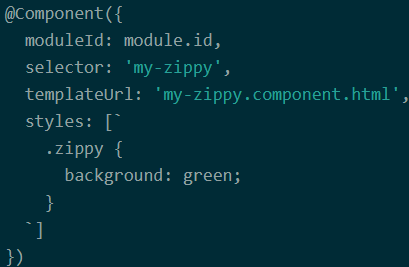


However, styles do now end up in the component’s template inside the shadow root. Here’s what that looks like:

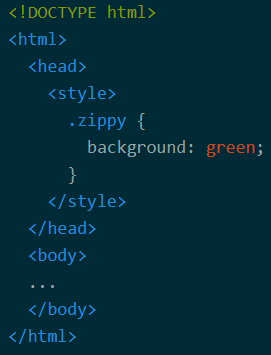


Styling Type

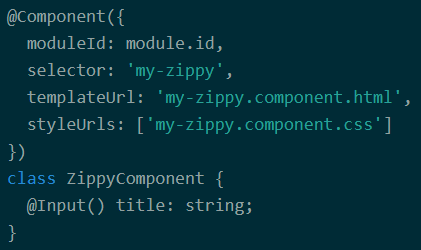
## [Component inline styles](https://blog.thoughtram.io/angular/2015/06/25/styling-angular-2-components.html#component-inline-styles)



Here’s what that looks like after rendering



## [Styles urls](https://blog.thoughtram.io/angular/2015/06/25/styling-angular-2-components.html#styles-urls)



Angular append styleUrls text in head after component inline styles so styleUrls has higher priority then component inline style

## [Template inline styles](https://blog.thoughtram.io/angular/2015/06/25/styling-angular-2-components.html#template-inline-styles)



Template inline styles always have the highest priority than other two

# Observable

An observable is a Producer of multiple values which pushing values to subscribers.

Handle a variety of common asynchronous operations. For example:

* The [EventEmitter](https://angular.io/api/core/EventEmitter) class extends Observable.
* The HTTP module uses observables to handle AJAX requests and responses.
* The Router and Forms modules use observables to listen for and respond to user-input events.

Consumers that subscribes to the Observable keeps receiving values until the Observable is completed or the consumer unsubscribes from the observable.

1. const myObservable = of(1, 2, 3);
2. const myObserver = {
3. next: x => console.log('Observer got [a](https://angular.io/api/router/RouterLinkWithHref) next value: ' + x),
4. error: err => console.error('Observer got an error: ' + err),
5. complete: () => console.log('Observer got [a](https://angular.io/api/router/RouterLinkWithHref) complete notification'),
6. };
7. myObservable.subscribe(myObserver);

|  |
| --- |
| // create observable |
|  | const simpleObservable = new Observable((observer) => { |
|  |  |
|  | // observable execution |
|  | observer.next("bla bla bla") |
|  | observer.complete() |
|  | }) |
|  |  |
|  | // subscribe to the observable |
|  | simpleObservable.subscribe() |
|  |  |
|  | // dispose the observable |
|  | simpleObservable.unsubscribe() |

There are three functions available to send data to the subscribers of the observable:

* “next”: sends any value such as Numbers, Arrays or objects to it’s subscribers.
* “error”: sends a Javascript error or exception
* “complete”: does not send any value.

In angular there are two ways to subscribe observable.

1. Using async pipe -> Angular deals with subscriptions during the lifecycle of a component. Angular maintain subscription and un-subscription by itself.

**public** getApiData(): Observable<UserApiInterface[]> {   
**this.apiData =** **this**.http.get<UserApiInterface[(‘REST\_URL');   
}

<section>   
 <ul>   
 <li \*ngFor=”let item of apiData | async” >   
 <p>{{item.title}}</p>   
 </li>   
 </ul>   
 </section>

2. subscribe to the observable ourselves using the actual subscribe()method. This is much better choice necessary to do something with the data before displaying it. The downside is that, have to manage the subscription by yourself.

**this**.observableService.getApiData().subscribe(   
 res => {console.log(res);   
 this.apiData = res; },   
 );

**Forms**

Two ways of working with forms

1. Template driven form

2. Model driven forms (Reactive Forms)

<form #userlogin = "ngForm" (ngSubmit) = "onClickSubmit(userlogin.value)" >

<input type = "text" name = "emailid" placeholder = "emailid" ngModel>

<br/>

<input type = "password" name = "passwd" placeholder = "passwd" ngModel>

<br/>

<input type = "submit" value = "submit">

</form>

onClickSubmit(data) {

alert("Entered Email id : " + data.emailid);

}

## Model Driven Form

We need to import the ReactiveFormsModule from @angular/forms in app module

In **app.component.ts**, we need to import a few modules for the model driven form. For example, **import { FormGroup, FormControl } from '@angular/forms'**.

<div>

<form [formGroup]="formdata" (ngSubmit) = "onClickSubmit(formdata.value)" >

<input type="text" class="fortextbox" name="emailid" placeholder="emailid"

formControlName="emailid">

<br/>

<input type="password" class="fortextbox" name="passwd"

placeholder="passwd" formControlName="passwd">

<br/>

<input type="submit" class="forsubmit" value="Log In">

</form>

</div>

ngOnInit() {

this.todaydate = this.myservice.showTodayDate();

this.formdata = new FormGroup({

emailid: new FormControl("angular@gmail.com"),

passwd: new FormControl("abcd1234")

});

}

onClickSubmit(data) {this.emailid = data.emailid;}

## Form Validation

import { FormGroup, FormControl, Validators} from '@angular/forms'

this.formdata = new FormGroup({

emailid: new FormControl("", Validators.compose([

Validators.required,

Validators.pattern("[^ @]\*@[^ @]\*")

])),

passwd: new FormControl("")

});

1. **this**.profileForm = **new** FormGroup({
3. FirstName: **new** FormControl(''),
4. LastName: **new** FormControl(''),
5. UserName: **new** FormControl(''),
6. Email: **new** FormControl(''),
7. MobileNumber: **new** FormControl('')

**setValue()**

1. profileForm.setValue({
2. "FirstName":"Ajay",
3. "LastName":"Panda",
4. "UserName":"ajay33",
5. "Email":"ajay.panda@gmail.com",
6. "MobileNumber":"8745212589"
7. });

**patchValue()**

1. profileForm.patchValue({
2. "FirstName":"Ajay",
3. "LastName":"Panda"
4. });

Setvalue to set all form control values

Pathvaue to set single or multiple values /// used to override form values

**Pipes**

Use pipe to transform input values to desired output

Angular comes with a stock of pipes such as [DatePipe](https://angular.io/api/common/DatePipe), [UpperCasePipe](https://angular.io/api/common/UpperCasePipe), [LowerCasePipe](https://angular.io/api/common/LowerCasePipe), [CurrencyPipe](https://angular.io/api/common/CurrencyPipe), and [PercentPipe](https://angular.io/api/common/PercentPipe).

**Format** : <someValue> | <PipeName> : <Param1>,<Param2>…

