Reference

Best practice to use column class in html

Always use:

<div class=’container’>

<div class=” row”>

<div class=” col-md-12”>

How to use ‘NgZone’ and what it will do.

* It’s make angular methods to avail outside of angular (it can used in pluggable app)

First import NgZone (it is available in angular/core package), call it in constructor (like: private zone: NgZone)

Routing (Navigation through one url to another)

Routing for page (to navigate from one page to another page)

{path: ‘router\_name’, component: ‘component\_name’ …}

Observable.forkJoin([

this.getData(“json\_data\_1 or api\_service\_1”),

this.getData(“json\_data\_2 or api\_service\_2”)

])

How to pass multiple query parameter with Activated Routes: **see below**

This.activatedRoute.queryParams.subscribe( params => {

const reason = params [‘reason’];

const contentkey = params [‘contentkey’];

const alertmsg = params [‘alertmsg’];

//Apply condition based on your requirement like: see below code

If(reason) { // if reason is there then pass reason to the parameter.

// write the condition

}

If (contentkey) {

// write the conditions.

}

});

Way to access the object without dot notation in javascript.

1. We can access the object through dot(.) notation.
2. We can have the another way to access the object through square brackets([]).

Way to access the object:

// Given:

var foo = {'bar': 'baz'};

// Then

var x = foo['bar'];

// vs.

var x = foo.bar;

**Square bracket notation allows the use of characters that can't be used with dot notation:**

var foo = myForm.foo[]; // incorrect syntax

var foo = myForm["foo[]"]; // correct syntax

**Secondly, square bracket notation is useful when dealing with property names which vary in a predictable way:**

for (var i = 0; i < 10; i++) {

someFunction(myForm["myControlNumber" + i]);

}

**Roundup:**

* Dot notation is faster to write and clearer to read.
* Square bracket notation allows access to properties containing special characters and selection of properties using variables

**The bracket notation allows you to access properties by name stored in a variable:**

var obj = { "abc" : "hello" };

var x = "abc";

var y = obj[x];

console.log(y); //output - hello

obj.x would not work in this case.

<http://abusanad.net/2016/08/18/angular-2-cross-modules-communication-eventaggregator-overview/>

---- Reference link for Event Aggregator service.

<https://blog.thecodecampus.de/angular-2-use-hostbindings-set-class/> ---🡪 Reference link for Host Binding in angular2

**@HostBinding()** : The @HostBinding() annotation provides the range of possibilities. The advantage is, the required logic remains in the components class and is not set in the Template File. This makes it easier to test and improves the readability of the template file.

* It is the only way to set the CSS class to the host element within the component.
* To set the class use the @HostBinding() annotation either on the Boolean property of the class or use a method returning a Boolean. The value determines whether the class is set or not if the value/return is true, the class will be applied to the component DOM Element.

EX: **Host Binding with Method** EX: **Host Binding with a Property**

Export class class\_Name implements OnInit {

@HostBinding(‘class.isActive’) isActive: boolean;

Constructor() {}

ngOnInit() {}

}

Export class class\_Name implements OnInit {

Constructor() {}

ngOnInit() {}

isActive: boolean;

@HostBinding(‘class.isActive’) isActiveiAsMethod() {

return this.isActive;

}

}

The class to be set is specified within the round brackets, prefix by class. --- it is also possible to specify several classes by combining multiple annotations.

EX: @HostBinding(‘class.isActive’)

@HostBinding(‘class.current’) isActive: boolean;

====================================================================================================================================================

Reference : How to work with Reactive Form in angular 2:

<https://alligator.io/angular/reactive-forms-formarray-dynamic-fields/>

<https://alligator.io/angular/hostbinding-hostlistener/>

Way to create route in angular 2

/\* import files…\*/

-----------

export const routes: Routes = [

{path: ‘routing\_page\_name\_Like- documents’, component: ‘component\_name’,},

{-------------------------}

---------------------------

]

@NgModule({

Imports: [RouterModule.forRoot(routes)],

Exports: [RouterModule]

})

export class AppRoutingMocule{ }

Best practice to use bootstrap classes in html.

---------------------------------------------------------------------------------

<div class=’row’>

<div class=’col-xs-12 col-md-9’>

--------//component selector define here like : <app-component></app-component>

</div>

</div>

Unit test case for iframe.Content.Document or getElementById

--------------------------------------------------------------------------------------------------------------------------------------------------

Initialize one variable inside describe like:

describe {

let testDocument;

-------------

// block of codes

------------

// Write below code inside beforeEach

testDocument = {‘contentDocument’ : ‘<html><body><p id=’iframe’>Test Data </p></body></html>}

// write below peace of code inside the it function.

spyOn(‘service\_Object\_Name’ : ‘Method\_Name\_from\_service’).and.returnValue(‘testDocument’);

}

**Session Time out (If the user is not doing any action on the current page- page is idle for some time then it’s should get idle session timed out).**

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Write the below code in component:

/\*-------imports-----\*/

// implementing this functionality by using ng-idle.

Export class…..

public isSessionTimedOut: Boolean;

public currentURL: string;

public idleTimeDuration: number;

constructor( private idle: Idle, private router: Router, …){

this. idleTimeDuration = 10; // setting default time out is 10 second.

}

ngOnInit() {

this.isSessionTimedOut = false;

this.router.events.filter(e => e instaceof NavigationEnd)

.subscribe(

(e:NavigationEnd) => {

This.currentURL = e.url;

This.idle.setIdle(this.idleTimeDuration);

This.idle.setTimeout(10); //sets time out period as 10 sec after 10 sec user will be considered timed out.

// sets the default interrupts, in this case things like: click, scroll, touches to the document

This.idle.setInterrupts(DEFAULT\_INTERRUPTSOURCES);

This.idle.onIdleStart.subscribe(()=>{

this.isSessionTimedOut = true;

});

This.idle.onIdleEnd.subscribe(()=>{

// ----

});

This.idle.onTimeout.subscribe(()=>

this.isSessionTimedOut = false;

) }

)

}

In HTML:

Put ngIf statement for displaying the model window on page idle activity

<div \*ngIf = “isSessionTimedOut”>

------------

-----------

</div>

---------------Reference from Net-------------

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

import {Idle, DEFAULT\_INTERRUPTSOURCES} from '@ng-idle/core';

import {Keepalive} from '@ng-idle/keepalive';

@Component({

selector: 'app-root',

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

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

})

export class AppComponent {

idleState = 'Not started.';

timedOut = false;

lastPing?: Date = null;

constructor(private idle: Idle, private keepalive: Keepalive) {

// sets an idle timeout of 5 seconds, for testing purposes.

idle.setIdle(5);

// sets a timeout period of 5 seconds. after 10 seconds of inactivity, the user will be considered timed out.

idle.setTimeout(5);

// sets the default interrupts, in this case, things like clicks, scrolls, touches to the document

idle.setInterrupts(DEFAULT\_INTERRUPTSOURCES);

idle.onIdleEnd.subscribe(() => this.idleState = 'No longer idle.');

idle.onTimeout.subscribe(() => {

this.idleState = 'Timed out!';

this.timedOut = true;

});

idle.onIdleStart.subscribe(() => this.idleState = 'You\'ve gone idle!');

idle.onTimeoutWarning.subscribe((countdown) => this.idleState = 'You will time out in ' + countdown + ' seconds!');

// sets the ping interval to 15 seconds

keepalive.interval(15);

keepalive.onPing.subscribe(() => this.lastPing = new Date());

this.reset();

}

reset() {

this.idle.watch();

this.idleState = 'Started.';

this.timedOut = false;

}

}

<http://www.concretepage.com/angular-2/angular-2-routing-and-navigation-example>

<http://www.discoversdk.com/blog/writing-unit-tests-in-angular-2> ------- Reference for Unit testing

<https://angular.io/guide/styleguide>

<https://stackoverflow.com/questions/42412110/angular-2-how-to-pass-an-array-to-the-router-using-queryparams>

<https://www.youtube.com/watch?v=M0X634LyQ1k&index=3&list=PLC3y8-rFHvwg5gEu2KF4sbGvpUqMRSBSW>

<https://www.w3schools.com/icons/tryit.asp?filename=tryicons_awesome_intro_larger>

<http://sass-lang.com/documentation/file.SASS_REFERENCE.html#Operations> ------ SASS reference

<https://www.code-sample.com/2016/06/angular-2-interview-questions-and.html> ----- interview Questions

<https://alligator.io/angular/route-guards/>

<https://alligator.io/angular/angular-router-child-routes/>

<https://toddmotto.com/angular-2-forms-reactive> ----- Reactive Form creation in angular 2

<https://semaphoreci.com/community/tutorials/testing-angular-2-http-services-with-jasmine> ---- unit test for Http

<http://api.jquery.com/category/miscellaneous/collection-manipulation/> ------- Jquery

<http://www.concretepage.com/angular-2/angular-2-input-and-output-example> ----- Angular 2 Input and Output example

<https://www.npmjs.com/package/angular-idle-service> ----- ngIdle service

<https://keyholesoftware.com/services/education/angular-2-course/>

<http://marclloyd.co.uk/uncategorized/spying-on-localstorage-in-angular-2-unit-tests/>

<https://www.wisdomjobs.com/e-university/angular-2-interview-questions.html> --- interview question

---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Way to Write Unit test case for component and service files:

--------------------------------------------------------------------------------

/\*\* imports all required dependencies \*/

**function initComponent(): ComponentFixture<Component\_Name>{**

**const = fixture TestBed.createComponent(Component\_Name);**

**fixture.detectChanges();**

**return fixture;**

**}**

describe(‘Component\_Name’, ()=>{

// variable declaration like: let var\_name;

beforeEach(async( () => {

TestBed.configureTestingModule({

Imports: [ //Define all dependent Module like: RouterTestingModule, etc…],

declarations: [ //declare component and u can declare dependent component as well ],

providers: [ //declare all the service file name like: LoginService, LoaderService, CookiesService … ],

})

}));

beforeEach(inject([ LoaderService //dependent service injection], (LoaderService) => {

loaderService = LoaderService;

…………………..

……………………

})) ;

It (‘component should get created ’, () => {

fixture = initComponent();

Component = fixture.componentInstance;

Expect(compoent).toBeTruthy();

})

// u can write multiple it functions based on your component

// u can test your component function that are appear in your component by calling the component method through the component.

It (‘Method should get call and defined’, () => {

fixture = initComponent;

component = fixture.componentInstance;

spyOn(service\_Object Name like: loaderService, ‘setTitle’)

expect(component.getProviderUrl).toBeDefined(); 🡪 // will check component method is defined or not. We can call the component method by component instance.

Component.Method\_Name();

})

})

Way to write Unit test cases for service file in angular 2

====================================================================================================================================================================

/\* Imports all required dependencies\*/

describe(‘Service\_name’, ()=>{

// variable declaration like: let var\_name;

beforeEach(async( () => {

TestBed.configureTestingModule({

Imports: [ //Define all dependent Module like: RouterTestingModule, etc…],

providers: [ //declare service here like: LoginService, LoaderService … {

provider: Http, HttpService,

useFactory: function (backendInstance: CoonectionBackend, defaultOptions: BaseRequestOptions){

return new Http (backendInstance, defaultOptions ) ;

},

Deps: [MockBackend, BaseRequestOptions],

} ];

});

}));

beforeEach (inject ( [ LoginService, MockBackend ], (loginService: LoginService, mockBackend: MockBackend ) => {

service = loginService;

backend = mockBackend;

…………………..

……………………

})) ;

// define all response json data.

It (‘service should get created’, () => {

expect (service).toBeTruthy();

});

// To check service method (get service will call here)

It (‘should get data from the service request’, (done) => {

backend.connection.subscribe ((connection) => {

expect(connection.request.method).toBe(RequestMethod.Get);

expect(connection.request.url).toBe(‘URL’);

connection.mockRespond(new Response(new ResponseOptions( {

body: Service\_Data;

status: 200;

})));

})

// you can call service subscribe methods like below

service.getEntitementData().subscribe((entitlementResp) =>{

expect(entitlementResp).not.toBe(null);

expect(entitlementResp.\_body.userAttribute).not.toBe(null);

done();

}, done.fail );

});

})

<https://toddmotto.com/component-events-event-emitter-output-angular-2>

 canactivate route: <https://blog.thoughtram.io/angular/2016/07/18/guards-in-angular-2.html>

Angular Cli config : <https://github.com/angular/angular-cli/wiki/angular-cli>

another one : <https://github.com/angular/angular-cli/Wiki>

Basic installation Steps : <https://github.com/angular/angular-cli/issues/2843> & <https://www.npmjs.com/package/angular-cli> & <https://angular.io/guide/quickstart>

Data flow : <http://www.sparkbit.pl/data-flow-angular-2-applications/> & <https://angular.io/guide/architecture>

Npm install : <https://docs.npmjs.com/all> & <https://docs.npmjs.com/cli/install>

Npm init command : It will create a package.json file where it will include the most common item.

To download latest npm package: npm -g install npm@latest

Anuglar 2 Life Cycle hooks : <https://angular.io/guide/lifecycle-hooks>

Angular 2 Reactive Forms Example: <https://embed.plnkr.co/?show=preview>

1. what is difference b/w <span> and <div> ?

2. what is position properties in CSS and explain difference type of position ?

3. How we can host the variable in javascript(variable Hosting in JS)?

**Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution.**

Ans: We expected the result of the log to be: ReferenceError: hoist is not defined, but instead, its output is undefined. JavaScript has hoisted the variable declaration. This is what the code above looks like to the i

1. console.log(hoist); //output: undefined
2. var hoist = “The variables have been hoisted.”

JavaScript has hoisted the variable declaration. This is what the code above looks like to the interpreter: 1. Var hoist;

2. console.log(hoist);

3. hoist = “The variables have been hoisted.”

Eg: Reference Error : b is not defined.

Function hoist() {

a= 20;

Var b = 100;

}

Console.log(a)// print output as it was defined globally by javascript

Console.log(b)// give referenceerror: b is not defined.

Hoisting can be avoided if we use “use strict” mode at the top of the javascript.

JavaScript known as strict-mode, we can be more careful about how we declare our variables. By enabling [*strict mode*](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode), we opt into a restricted variant of JavaScript that will not tolerate the usage of variables before they are declared.

<https://scotch.io/tutorials/understanding-hoisting-in-javascript>

**Let** are block scoped and not function scoped. we should *declare* then *assign* our variables to a value before using them.

The **const** keyword was introduced in es6 to allow *immutable variables*.

4. What is diff b/w var and let?

Ans The scope of a variable defined with var is function scope or declared outside any function, global.The scope of a variable defined with let is block scope.

The difference is scoping. **var** is scoped to the nearest function block and **let** is scoped to the nearest *enclosing* block, which can be smaller than a function block. Both are global if outside any block.

Let in the loop can re-binds it to each iteration of the loop, making sure to re-assign it the value from the end of the previous loop iteration, so it can be used to avoid issue with closures.

<http://www.jstips.co/en/javascript/keyword-var-vs-let/>

5. How to read JSON data in angular2?

6. what is Arrow function in angular2 and how we can use it explain with syntax ?

<https://ng2.codecraft.tv/es6-typescript/arrow/>

Arrow functions capture the this where the function is created rather than where it is invoked:

7. Difference Between normal function and arrow function in angular2

8. Inheritance in javascripts.

<https://stackoverflow.com/questions/16261044/why-javascript-does-not-support-inheritance-by-default>

9. What is promise in javascript. Differentiate it with observable.

JavaScript functions can be loosely classified as the following:

1. Function declarations: These are of the following form and are hoisted completely to the top. Now, we can understand why JavaScript enable us to invoke a function seemingly before declaring it.

hoisted();

Function hoisted() { console.log(“The function declaration has been hoisted”); }

1. Function expressions: Function expressions, however are not hoisted.

expression();

var expression = function() { console.log(“This function expression will not hoisted”) }

Order of precedence: It's important to keep a few things in mind when declaring JavaScript functions and variables.

1. Variable assignment takes precedence over function declaration

Var double = 22;

Function double(num) { return (num\*2) }

Console.log( typeod double ); // Output : number

1. Function declarations take precedence over variable declarations

Var double;

Function double(num) { return (num\*2) }

Console.log( typeod double ); // Output : function

12. What is the diff b/w ng-if and ng-hide?

<https://stackoverflow.com/questions/19177732/what-is-the-difference-between-ng-if-and-ng-show-ng-hide>

13. What is the diff b/w ng-bind and ng-model?

14. two-way binding in angular 1 how we can implement it?

15. predefined directives in angular1?

16. How to read query data from the data base?

17. what is @Input and @Output in angular2?

18. what is use of $scope in angular1?

19. what is promise how u can read the huge data from the backend?

20. difference b/w promise and observable?

Ans: **Promise**:A Promise handles a **single event** when an async operation completes or fails.

**Observable:** An Observable is like a Stream (in many languages) and allows to pass zero or more events where the callback is called for each event.

Often Observable is preferred over Promise because it provides the features of Promise and more. With Observable it doesn't matter if you want to handle 0, 1, or multiple events. You can utilize the same API in each case.

Observable also has the advantage over Promise to be **cancelable**. If the result of an HTTP request to a server or some other expensive async operation isn't needed anymore, the Subscription of an Observable allows to cancel the subscription, while a Promise will eventually call the success or failed callback even when you don't need the notification or the result it provides anymore.

Observable provides **operators** like map, forEach, reduce, ... similar to an array

21. have u developed any directive?

22. what is root directive in angular 1, how u will initialize the angular1?

23. what is the use of ng-repeat in angular1 explain?

<http://www.c-sharpcorner.com/article/top-50-angularjs-interview-questions-and-answers/>

24. how we can read the array of data through the service?

25. Diff between component and directivesss ?

<https://stackoverflow.com/questions/34613065/what-is-the-difference-between-component-and-directive>

26. What is Callback help in angularJs ?

27. what is difference between config and run in angularjs ?

28. what is data type in angular 2 ?

29. what is the replacement of $apply in angular2 ?

Ans: Zone.js (We have to import Ngzone for the same to use it )

30. Difference between synchronous and asynchronous in angularjs ?

Ans : To make sure the second calls are executed after the first one is finished, put the second call within then of the first call. To make multiple 'second' calls depending on the number of results of the first call, use $q.all

31. Promises in angularJs ?

<http://www.dwmkerr.com/promises-in-angularjs-the-definitive-guide/#whatarepromises>

<https://chariotsolutions.com/blog/post/angularjs-corner-using-promises-q-handle-asynchronous-calls/>

32. What makes the angular.copy() method so powerful?

A deep copy of a variable means it doesn’t point to the same memory reference as that variable. Usually assigning one variable to another creates a “shallow copy”, which makes the two variables point to the same memory reference. Therefore if one is changed, the other changes as well.

33. How would you make an Angular service return a promise? Write a code snippet as an example

Ans : To add promise functionality to a service, we inject the “$q” dependency in the service, and then use it .The $q library is a helper provider that implements promises and deferred objects to enable asynchronous functionality.

34. When creating a directive, it can be used in several different ways in the view. Which ways for using a directive do you know? How do you define the way your directive will be used?

Ans: When you create a directive, it can be used as an attribute, element or class name. To define which way to use, you need to set the restrict option in your directive declaration.The restrict option is typically set to: ‘A’ – only matches attribute name ‘E’ – only matches element name  
‘C’ – only matches class name, ‘AEC’ – matches either attribute or element or class name

Angular 2 Component Lifecycle:

A component has a lifecycle managed by Angular itself. Angular manages creation, rendering, data-bound properties etc. It also offers hooks that allow us to respond to key lifecycle events.

Here is the complete lifecycle hook interface inventory:

* ngOnChanges - called when an input binding value changes
* ngOnInit - after the first ngOnChanges
* ngDoCheck - after every run of change detection
* ngAfterContentInit - after component content initialized
* ngAfterContentChecked - after every check of component content
* ngAfterViewInit - after component's view(s) are initialized
* ngAfterViewChecked - after every check of a component's view(s)
* ngOnDestroy - just before the component is destroyed

The @ViewChild and @ViewChildren decorators provide access to the class of child component from the containing component.

The @ViewChild is a decorator function that takes the name of a component class as its input and finds its selector in the template of the containing component to bind to. @ViewChild can also be passed a template reference variable.

<https://angular.io/guide/component-interaction>

Remember that ngOnChanges method is called when there is a change on any Input property of a component

HTML DOM: The HTML DOM can be accessed with JavaScript (and with other programming languages).In the DOM, all HTML elements are defined as objects.

The programming interface is the properties and methods of each object.A property is a value that you can get or set (like changing the content of an HTML element).A method is an action you can do (like add or deleting an HTML element).

document.getElementById("demo").innerHTML = "Hello World!"; or

window.document.getElementById("demo").innerHTML = "Hello World!"; both are same. In the example above, getElementById is a method, while innerHTML is a property.

The document object represents your web page.If you want to access any element in an HTML page, you always start with accessing the document object.Below are some examples of how you can use the document object to access and manipulate HTML.

Finding HTML Element by Id: document.getElementById("intro");

## Finding HTML Elements by Tag Name: document.getElementsByTagName("p")

## Finding HTML Elements by Class Name: document.getElementsByClassName("intro");

## Finding HTML Elements by CSS Selectors: document.querySelectorAll("p.intro");

## Changing HTML Elements:

|  |  |
| --- | --- |
| Method | Description |
| element.innerHTML =  new html content | Change the inner HTML of an element |
| element.attribute = new value | Change the attribute value of an HTML element |
| element.setAttribute(attribute, value) | Change the attribute value of an HTML element |
| element.style.property = new style | Change the style of an HTML element |

Adding & Deleting Elements:

|  |  |
| --- | --- |
| Method | Description |
| document.createElement(*element*) | Create an HTML element |
| document.removeChild(*element*) | Remove an HTML element |
| document.appendChild(*element*) | Add an HTML element |
| document.replaceChild(*element*) | Replace an HTML element |
| document.write(*text*) | Write into the HTML output stream |
| document.getElementById(*id*).onclick = function(){*code*} | Adding event handler code to an onclick event |

Call a function on click event in javascript: object.onclick = function(){myScript};

Call a add event listener function in javascript: object.addEventListener("click", myScript);

<https://www.w3schools.com/angular/tryit.asp?filename=try_ng_filters_input> ( For searching item from the list of items through input field.

<https://www.w3schools.com/angular/tryit.asp?filename=try_ng_validate> (Validation on input fields and provide necessary error in front of it.

Following are the key differences between $apply() and $digest().

* Its use is to update the model properties forcibly.
* The $digest() method evaluates the watchers for the current scope. However, the $apply() method is used to evaluate watchers for root scope, that means it’s for all scopes.

$emit, $broadcast and $on:

Using $scope.$emit will fire an event up the $scope. Using $scope.$broadcast will fire an event down the $scope. Using $scope.$on is how we listen for these events.

### Explain $q service, deferred and promises.

* ‘Promises’ are post processing logics which are executed after some operation/action is completed whereas ‘deferred’ is used to control how and when those promise logics will execute.
* We can think about promises as “WHAT” we want to fire after an operation is completed while deferred controls “WHEN” and “HOW” those promises will execute.
* “$q” is the angular service which provides promises and deferred functionality.

# Explain difference between Factory vs Service vs Provider: ?

# Ans : Factory : When you’re using a Factory you create an object, add properties to it, then return that same object. When you pass this service into your controller, those properties on the object will now be available in that controller through your factory.

Service: When you’re using Service, it’s instantiated with the ‘new’ keyword. Because of that, you’ll add properties to ‘this’ and the service will return ‘this’. When you pass the service into your controller, those properties on ‘this’ will now be available on that controller through your service.

Providers are the only service you can pass into your .config() function. Use a provider when you want to provide module-wide configuration for your service object before making it available.

**Scope in Javascipt:**

Lexical Scope : Whenever you see a function within another function, the inner function has access to the scope in the outer function, this is called Lexical Scope or Closure - also referred to as Static Scope.

Closure: A better example of how the *closure*side of things works, can be seen when returning a *function reference* - a more practical usage. Inside our scope, we can return things so that they’re available in the parent scope:

Eg: var sayHello = function (name) {  
 var text = 'Hello, ' + name;  
 return function () {  
 console.log(text);  
 };  
};

The closure concept we’ve used here makes our scope inside sayHello inaccessible to the public scope. Calling the function alone will do nothing as it *returns* a function:

sayHello('Todd'); // nothing happens, no errors, just silence...

The function returns a function, which means it needs assignment, and *then* calling:

var helloTodd = sayHello('Todd');  
helloTodd(); // will call the closure and log 'Hello, Todd'

Okay, I lied, you *can* call it, and you may have seen functions like this, but this will call your closure:

sayHello('Bob')(); // calls the returned function without assignment

AngularJS uses the above technique for its **$compile** method, where you pass the current scope reference into the closure:

$compile(template)(scope);

What is .call(), .apply() and .bind() function in javascript ?

We can use either .call () or .apply() to change the scope, but any further arguments are where the two differ: .call(scope, arg1, arg2, arg3) takes individual arguments, comma separated, whereas .apply(scope, [arg1, arg2]) takes an Array of arguments.

In order to understand this, you just need two more pieces of information. First, arguments is an Array-like object that represents all of the arguments passed into a function. Second, the apply method works exactly like the call primitive, except that it takes an Array-like object instead of listing the arguments out one at a time.

Our bind method simply returns a new function. When it is invoked, our new function simply invokes the original function that was passed in, setting the original value as this. It also passes through the arguments.

JavaScript primitive types like string and number are immutable by definition

<http://blog.ninja-squad.com/2017/03/24/what-is-new-angular-4/> how angular2 different from angular4

What is optional parameter in angular2/4 ?

Ans: <https://www.typescriptlang.org/docs/handbook/functions.html>

Explain difference between promise and callback function in anguarlJs ?

What will be output in javascript , and how to display all the loop values ?

For(var i=0;i<5;i++){ setTimeout(function() { console.log(i); },1000 ); } : // Output 6

Or use this option to display proper answer

for (let i = 0; i < 5; i++) { setTimeout(function() { console.log(i); }, 1000 ); }

function doSetTimeout(i) {

setTimeout(function() { alert(i); }, 100);

}

for (var i = 0; i <= 5; ++i)

doSetTimeout(i); // Output 0,1,2,3,4,5

Self-invoking function in simple form:

for (var i = 0; i < 5; i++) { y(i);

function y(x){ setTimeout(function() { console.log(i); }, 1000 ); }

}

for (var i = 0; i < 5; i++) { (function(x) {

setTimeout(function() { console.log(x); }, x \* 1000 );

})(i); }

**Both are same function and it will execute and return same output**

**Angular from Scratch**:

**Decorator**: Angular has many decorators that attach metadata to classes so that it knows what those classes mean and how they should work.

**Template**:  A template is a form of HTML that tells Angular how to render the component.

**Metadata**: To tell Angular that HeroListComponent is a component, attach metadata to the class.In TypeScript, you attach metadata by using a decorator. Here's some metadata for HeroListComponent

Data binding plays an important role in communication between a template and its component. Data binding is also important for communication between parent and child components.

**Directives**: Angular templates are *dynamic*. When Angular renders them, it transforms the DOM according to the instructions given by directives. A directive is a class with a @[Directive](https://angular.io/api/core/Directive) decorator. A component is a directive-with-a-template; a @[Component](https://angular.io/api/core/Component) decorator is actually a @[Directive](https://angular.io/api/core/Directive) decorator extended with template-oriented features.

Two other kinds of directives exist: structural and attribute directives.

Structural directives alter layout by adding, removing, and replacing elements in DOM.(eg \*ngFor, \*ngIf)

Attribute directives alter the appearance or behavior of an existing element. In templates, they look like regular HTML attributes, hence the name. ( [(ngModel)], ngClass, ngStyle, ngSwitch )

*NgSwitch* is actually a set of three, cooperating directives: [NgSwitch](https://angular.io/api/common/NgSwitch), [NgSwitchCase](https://angular.io/api/common/NgSwitchCase), and [NgSwitchDefault](https://angular.io/api/common/NgSwitchDefault)

Where [NgSwitch](https://angular.io/api/common/NgSwitch) Is attribute directive without \* symbol. And Ngswitchcase and Ngswitchdefault are structural directive with \* symbol in the front.

**Dependency Injection**: *Dependency injection* is a way to supply a new instance of a class with the fully-formed dependencies it requires. Most dependencies are services. Angular uses dependency injection to provide new components with the services they need.

Template Reference Variable: A template reference variable is often a reference to a DOM element within a template. It can also be a reference to an Angular component or directive or a [web component](https://developer.mozilla.org/en-US/docs/Web/Web_Components). Use the hash symbol (#) to declare a reference variable.

The scope of a reference variable is the *entire template*. Do not define the same variable name more than once in the same template. The runtime value will be unpredictable. You can use the ref- prefix alternative to #. This example declares the fax variable as ref-fax instead of #fax.

Input & Output Property: An *Input* property is a *settable* property annotated with an @[Input](https://angular.io/api/core/Input) decorator. Values flow *into* the property when it is data bound with a [property binding](https://angular.io/guide/template-syntax#property-binding).

An *Output* property is an *observable* property annotated with an @[Output](https://angular.io/api/core/Output) decorator. The property almost always returns an Angular [EventEmitter](https://angular.io/api/core/EventEmitter). Values flow *out* of the component as events bound with an [event binding](https://angular.io/guide/template-syntax#event-binding)

**Guards**: A guard's return value controls the router's behavior: The guard can also tell the router to navigate elsewhere, effectively canceling the current navigation

* If it returns true, the navigation process continues.
* If it returns false, the navigation process stops and the user stays put.

Angular 2/4 Performance Improvement:

This used AoT ( ahead of time compiler ) as compare to the JIT ( Just in time compile ). In AoT compiler the typescript was converted into javascript at the time of build application.

Performs compilation of the templates with the Angular compiler and generates (usually) TypeScript.

Angular 2/4 also use **lazy loading** to improve the performance of the application, it can be done by segregating the application into multiple modules and load that modules as per the need of the user’s.

**Angular Lifecycle Hooks**: A component has a lifecycle managed by Angular.Angular creates it, renders it, creates and renders its children, checks it when its data-bound properties change, and destroys it before removing it from the DOM.

Directive and component instances have a lifecycle as Angular creates, updates, and destroys them. Developers can tap into key moments in that lifecycle by implementing one or more of the *lifecycle hook* interfaces in the Angular core library.Each interface has a single hook method whose name is the interface name prefixed with ng. For example, the [OnInit](https://angular.io/api/core/OnInit) interface has a hook method named ngOnInit().

**ngOnChanges**(): It was called whenever one or more input bound properties change, it was called before ngOnInit(). Angular calls its ngOnChanges() method whenever it detects changes to *input properties* of the component (or directive). Angular only calls the hook when the value of the input property changes. The value of the hero property is the *reference to the hero object*. Angular doesn't care that the hero's own name property changed.

**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(). In the above example the hero object change detection was not happening, that can be done through DoCheck().

**ngDestroy()**: 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.

The *AfterView* sample explores the [**AfterViewInit**](https://angular.io/api/core/AfterViewInit)() and [**AfterViewChecked**](https://angular.io/api/core/AfterViewChecked)() hooks that Angular calls after it creates a component's child views.

The *AfterContent* sample explores the [**AfterContentInit**](https://angular.io/api/core/AfterContentInit)() and [**AfterContentChecked**](https://angular.io/api/core/AfterContentChecked)() hooks that Angular calls *after* Angular projects external content into the component

Initialize with alias eg : @[**Input**](https://angular.io/api/core/Input)('master') masterName: string;

@**Output**: The child component exposes an [EventEmitter](https://angular.io/api/core/EventEmitter) property with which it emits events when something happens. The parent binds to that event property and reacts to those events.

@**viewchild** : First, you have to import references to the [ViewChild](https://angular.io/api/core/ViewChild) decorator and the [AfterViewInit](https://angular.io/api/core/AfterViewInit) lifecycle hook.

Initialization: @[ViewChild](https://angular.io/api/core/ViewChild)(CountdownTimerComponent) private timerComponent: CountdownTimerComponent;

**Dynamic component loading:** The hero agency is planning an ad campaign with several different ads cycling through the banner. New ad components are added frequently by several different teams. This makes it impractical to use a template with a static component structure.

Instead, you need a way to load a new component without a fixed reference to the component in the ad banner's template. Angular comes with its own API for loading components dynamically.

**Directive**: 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.

**Custom Attribute Directive**: While creating the custom attribute directive, we have to specify name in the [ ] for eg : selector: '[**appHighlight]'.** The import statement specifies an additional [**ElementRef**](https://angular.io/api/core/ElementRef) symbol from the Angular core library: You use the [**ElementRef**](https://angular.io/api/core/ElementRef) in the directive's constructor to [inject](https://angular.io/guide/dependency-injection) a reference to the host DOM element, the element to which you applied **appHighlight**. [ElementRef](https://angular.io/api/core/ElementRef) grants direct access to the host DOM element through its **nativeElement** property.

**User initiated Events**: Currently, **appHighlight** simply sets an element color. The directive could be more dynamic. It could detect when the user mouses into or out of the element and respond by setting or clearing the highlight color. Begin by adding [**HostListener**](https://angular.io/api/core/HostListener) to the list of imported symbols.

The @[**HostListener**](https://angular.io/api/core/HostListener) decorator lets you subscribe to events of the DOM element that hosts an attribute directive, the <p> in this case.

**Binding To @Input**: Notice the @[Input](https://angular.io/api/core/Input) decorator. It adds metadata to the class that makes the directive's highlightColor property available for binding.

**Structural Directive**: Structural directives are responsible for HTML layout. They shape or reshape the DOM's structure, typically by adding, removing, or manipulating elements.

[NgIf](https://angular.io/api/common/NgIf) refers to the directive class; [ngIf](https://angular.io/api/common/NgIf) refers to the directive's attribute name. When the condition is false, [NgIf](https://angular.io/api/common/NgIf) removes its host element from the DOM, detaches it from DOM events (the attachments that it made), detaches the component from Angular change detection, and destroys it.

The asterisk is "syntactic sugar" for something a bit more complicated. Internally, Angular translates the \*[ngIf](https://angular.io/api/common/NgIf)*attribute* into a <ng-template> *element*, wrapped around the host element, like this.

<ng-template [[ngIf](https://angular.io/api/common/NgIf)]="hero"> <div class="name">{{hero.name}}</div> </ng-template>

A template input variable is a variable whose value you can reference within a single instance of the template. There are several such variables in this example: hero, i, and odd. All are preceded by the keyword let.

Template input and reference variable names have their own namespaces. The hero in let hero is never the same variable as the hero declared as #hero.

NOTE : The element to which you apply a directive is its *host* element.

**Ng-container**: Use [<ng-container>](https://angular.io/guide/structural-directives#ng-container) when there's no single element to host the directive. The Angular <ng-container> is a grouping element that doesn't interfere with styles or layout because Angular doesn't put it in the DOM.

The <ng-container> is a syntax element recognized by the Angular parser. It's not a directive, component, class, or interface

**Pipes**: 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). They are all available for use in any template.

A pipe can accept any number of optional parameters to fine-tune its output. To add parameters to a pipe, follow the pipe name with a colon ( : ) and then the parameter value (such as currency:'EUR'). If the pipe accepts multiple parameters, separate the values with colons (such as slice:1:5)

The parameter value can be any valid template expression such as a string literal or a component property. There are two categories of pipes: *pure* and *impure*. Pipes are pure by default. You make a pipe impure by setting its pure flag to false.

**Pure pipes**: Angular executes a *pure pipe* only when it detects a *pure change* to the input value. A pure change is either a change to a primitive input value (String, Number, Boolean, Symbol) or a changed object reference (Date, Array, Function, Object).

**Impure Pipes**: Angular executes an *impure pipe* during every component change detection cycle. An impure pipe is called often, as often as every keystroke or mouse-move.

**Users Input**: User actions such as clicking a link, pushing a button, and entering text raise DOM events. You can use [Angular event bindings](https://angular.io/guide/template-syntax#event-binding) to respond to any [DOM event](https://developer.mozilla.org/en-US/docs/Web/Events). Many DOM events are triggered by user input. Binding to these events provides a way to get input from the user.

<https://angular-2-training-book.rangle.io/handout/observables/cold_vs_hot_observables.html>

<https://angular-2-training-book.rangle.io/handout/observables/observables_array_operations.html>

**Change Detection in Anuglar2**: Angular will create special classes that are called *change detectors*, one for every component that we have defined.

changeDetection: ChangeDetectionStrategy.Default (he **default strategy** for the change detection is **to traverse all the components of the tree** even if they do not seem to have been modified.)

To improve performace we will create a new object every time any of its properties changes, otherwise we keep the same object. we will use the OnPush change detection strategy on the MovieComponent.

**Zones**: [Zone.js](https://github.com/angular/zone.js) provides a mechanism, called zones, for encapsulating and intercepting asynchronous activities in the browser (e.g. setTimeout, , promises). These zones are *execution contexts* that allow Angular to track the start and completion of asynchronous activities and perform tasks as required (e.g. change detection).

The **NgZone** service provides us with a number of Observables and methods for determining the state of Angular's zone and to execute code in different ways inside and outside Angular's zone.

* onUnstable – Notifies when code has entered and is executing within the Angular zone.
* onMicrotaskEmpty - Notifies when no more microtasks are queued for execution. Angular subscribes to this internally to signal that it should run change detection.
* onStable – Notifies when the last onMicroTaskEmpty has run, implying that all tasks have completed and change detection has occurred.
* onError – Notifies when an error has occurred. Angular subscribes to this internally to send uncaught errors to its own error handler, i.e. the errors you see in your console prefixed with 'EXCEPTION:'

## Get user input from the $event object: bind to the keyup event of an input box to get the user's input after each keystroke. Eg : <input (keyup)="onKey($event)"> When a user presses and releases a key, the keyup event occurs, and Angular provides a corresponding DOM event object in the $event variable which this code passes as a parameter to the component's onKey() method.

## All [standard DOM event objects](https://developer.mozilla.org/en-US/docs/Web/API/Event) have a target property, a reference to the element that raised the event. In this case, target refers to the [<input> element](https://developer.mozilla.org/en-US/docs/Web/API/HTMLInputElement) and event.target.value returns the current contents of that element.

## Angular [template reference variables](https://angular.io/guide/template-syntax#ref-vars). These variables provide direct access to an element from within the template. To declare a template reference variable, precede an identifier with a hash (or pound) character (#). Eg: <input #box (keyup)="onKey(box.value)">

## Forms: If a component, directive, or pipe belongs to a module in the imports array, ​*don't*​ re-declare it in the declarations array. If you wrote it and it should belong to this module, ​*do*​ declare it in the declarations array.

## Tempalte Driven Forms: In template driven forms, if you've imported [FormsModule](https://angular.io/api/forms/FormsModule), you don't have to do anything to the <form> tag in order to make use of [FormsModule](https://angular.io/api/forms/FormsModule)

## The [NgForm](https://angular.io/api/forms/NgForm) directive supplements the form element with additional features. It holds the controls you created for the elements with an [ngModel](https://angular.io/api/forms/NgModel) directive and name attribute, and monitors their properties, including their validity. It also has its own valid property which is true only *if every contained control* is valid.

## Notice that you also added a name attribute to the <input> tag and set it to "name", which makes sense for the hero's name. Any unique value will do, but using a descriptive name is helpful. Defining a name attribute is a requirement when using [([ngModel](https://angular.io/api/forms/NgModel))] in combination with a form. Internally, Angular creates [FormControl](https://angular.io/api/forms/FormControl) instances and registers them with an [NgForm](https://angular.io/api/forms/NgForm) directive that Angular attached to the <form> tag. Each [FormControl](https://angular.io/api/forms/FormControl) is registered under the name you assigned to the name attribute.

**NOTE:**

* Each input element has an id property that is used by the label element's for attribute to match the label to its input control.
* Each input element has a name property that is required by Angular forms to register the control with the form.

**Track control state and validity with ngModel**: Using [ngModel](https://angular.io/api/forms/NgModel) in a form gives you more than just two-way data binding. It also tells you if the user touched the control, if the value changed, or if the value became invalid.The NgModel directive doesn't just track state; it updates the control with special Angular CSS classes that reflect the state. You can leverage those class names to change the appearance of the control.

|  |  |  |
| --- | --- | --- |
| **State** | **Class If True** | **Class If False** |
| The control has been visited | Ng-touched | Ng-untouched |
| The control’s value has changed | Ng-dirty | Ng-pristine |
| The control’s value is valid | Ng-valid | Ng-invalid ( This was true when there is no value in input box) |

You can mark required fields and invalid data at the same time with a colored bar on the left of the input box:

.ng-valid[required], .ng-valid.required { border-left: 5px solid #42A948; /\* green \*/ }

.ng-invalid:not(form) { border-left: 5px solid #a94442; /\* red \*/ }

Leverage the control's state to reveal a helpful message. You need a template reference variable to access the input box's Angular control from within the template. Here you created a variable called name and gave it the value "ngModel".

Why "ngModel"? A directive's [exportAs](https://angular.io/api/core/Directive) property tells Angular how to link the reference variable to the directive. You set name to [ngModel](https://angular.io/api/forms/NgModel) because the [ngModel](https://angular.io/api/forms/NgModel) directive's exportAs property happens to be "ngModel".

In this example, you hide the message when the control is valid or pristine; "pristine" means the user hasn't changed the value since it was displayed in this form. This user experience is the developer's choice. Some developers want the message to display at all times. If you ignore the pristine state, you would hide the message only when the value is valid. If you arrive in this component with a new (blank) hero or an invalid hero, you'll see the error message immediately, before you've done anything.

Some developers want the message to display only when the user makes an invalid change. Hiding the message while the control is "pristine" achieves that goal. You'll see the significance of this choice when you add a new hero to the form.

Enter a name and click *New Hero* again. The app displays a *Name is required* error message. You don't want error messages when you create a new (empty) hero. Why are you getting one now? You have to clear all of the flags imperatively, which you can do by calling the form's reset() method after calling the newHero() method. Eg : heroForm.reset()

Forms Validator:

1. The <input> element carries the HTML validation attributes: required and [minlength](https://angular.io/api/forms/MinLengthValidator)
2. #name="[ngModel](https://angular.io/api/forms/NgModel)" exports [NgModel](https://angular.io/api/forms/NgModel) into a local variable called name. [NgModel](https://angular.io/api/forms/NgModel) mirrors many of the properties of its underlying [FormControl](https://angular.io/api/forms/FormControl) instance, so you can use this in the template to check for control states such as valid and dirty
3. The \*[ngIf](https://angular.io/api/common/NgIf) on the <div> element reveals a set of nested message divs but only if the name is invalid and the control is either dirty or touched.
4. Each nested <div> can present a custom message for one of the possible validation errors. There are messages for required, [minlength](https://angular.io/api/forms/MinLengthValidator), and forbiddenName

**Reative Forms**: Angular offers two form-building technologies: *reactive* forms and *template-driven* forms. The two technologies belong to the @angular/forms library and share a common set of form control classes. They even have their own modules: the [ReactiveFormsModule](https://angular.io/api/forms/ReactiveFormsModule) and the [FormsModule](https://angular.io/api/forms/FormsModule).

Angular *reactive* forms facilitate a *reactive style* of programming that favors explicit management of the data flowing between a non-UI *data model* (typically retrieved from a server) and a UI-oriented *form model* that retains the states and values of the HTML controls on screen. Reactive forms offer the ease of using reactive patterns, testing, and validation.

With *reactive* forms, you create a tree of Angular form control objects in the component class and bind them to native form control elements in the component template. You create and manipulate form control objects directly in the component class. As the component class has immediate access to both the data model and the form control structure, you can push data model values into the form controls and pull user-changed values back out. The component can observe changes in form control state and react to those changes

One advantage of working with form control objects directly is that value and validity updates are [always synchronous and under your control](https://angular.io/guide/reactive-forms#async-vs-sync).

with the reactive forms, the component preserves the immutability of the *data model*, treating it as a pure source of original values.

#### **Why check dirty and touched**?

#### You may not want your application to display errors before the user has a chance to edit the form. The checks for dirty and touched prevent errors from showing until the user does one of two things: changes the value, turning the control dirty; or blurs the form control element, setting the control to touched.

**Reactive Forms Validation**: In a reactive form, the source of truth is the component class. Instead of adding validators through attributes in the template, you add validator functions directly to the form control model in the component class. Angular then calls these functions whenever the value of the control changes.

**Validator Functions**: There are two types of validator functions: sync validators and async validators.

* Sync validators: functions that take a control instance and immediately return either a set of validation errors or null. You can pass these in as the second argument when you instantiate a [FormControl](https://angular.io/api/forms/FormControl).
* Async validators: functions that take a control instance and return a Promise or Observable that later emits a set of validation errors or null. You can pass these in as the third argument when you instantiate a [FormControl](https://angular.io/api/forms/FormControl).

EG : this.heroForm = new [FormGroup](https://angular.io/api/forms/FormGroup)({ 'name': new [FormControl](https://angular.io/api/forms/FormControl)(this.hero.name, [ [Validators](https://angular.io/api/forms/Validators).required, [Validators](https://angular.io/api/forms/Validators).minLength(4), forbiddenNameValidator(/bob/i) // <-- Here's how you pass in the custom validator. ]), 'alterEgo': new [FormControl](https://angular.io/api/forms/FormControl)(this.hero.alterEgo), 'power': new [FormControl](https://angular.io/api/forms/FormControl)(this.hero.power, [Validators](https://angular.io/api/forms/Validators).required) });

1. The name control sets up two built-in validators [Validators](https://angular.io/api/forms/Validators).required

and [Validators](https://angular.io/api/forms/Validators).minLength(4)—and one custom validator, forbiddenNameValidator

1. As these validators are all sync validators, you pass them in as the second argument.
2. Support multiple validators by passing the functions in as an array.
3. This example adds a few getter methods. In a reactive form, you can always access any form control through the get method on its parent group, but sometimes it's useful to define getters as shorthands for the template

Async VS Sync :

1. Reactive forms are synchronous. Template-driven forms are asynchronous. It's a difference that matters.
2. In reactive forms, you create the entire form control tree in code. You can immediately update a value or drill down through the descendents of the parent form because all controls are always available. And Template-driven forms delegate creation of their form controls to directives. To avoid "*changed after checked*" errors, these directives take more than one cycle to build the entire control tree. That means you must wait a tick before manipulating any of the controls from within the component class.
3. The asynchrony of template-driven forms also complicates unit testing.
4. For example, if you inject the form control with a @[ViewChild](https://angular.io/api/core/ViewChild)([NgForm](https://angular.io/api/forms/NgForm)) query and examine it in the[ngAfterViewInit lifecycle hook](https://angular.io/guide/lifecycle-hooks#afterview), you'll discover that it has no children. You must wait a tick, using setTimeout, before you can extract a value from a control, test its validity, or set it to a new value.

**Reactive Forms**:

[FormControl](https://angular.io/api/forms/FormControl) is a directive that allows you to create and manage a [FormControl](https://angular.io/api/forms/FormControl) instance directly.

name = new [FormControl](https://angular.io/api/forms/FormControl)(); (Here you are creating a [FormControl](https://angular.io/api/forms/FormControl) called name. It will be bound in the template to an HTML input box for the hero name.)

A [FormControl](https://angular.io/api/forms/FormControl) constructor accepts three, optional arguments: the initial data value, an array of validators, and an array of async validators.

To let Angular know that this is the input that you want to associate to the name [FormControl](https://angular.io/api/forms/FormControl) in the class, you need [formControl]="name" in the template on the <input>.

### **Essential form classes:**

1. [***AbstractControl***](https://angular.io/api/forms/AbstractControl) is the abstract base class for the three concrete form control classes: [FormControl](https://angular.io/api/forms/FormControl), [FormGroup](https://angular.io/api/forms/FormGroup), and [FormArray](https://angular.io/api/forms/FormArray). It provides their common behaviors and properties, some of which are *observable*.
2. [***FormControl***](https://angular.io/api/forms/FormControl) tracks the value and validity status of an *individual* form control. It corresponds to an HTML form control such as an input box or selector.
3. [***FormGroup***](https://angular.io/api/forms/FormGroup) tracks the value and validity state of a *group* of [AbstractControl](https://angular.io/api/forms/AbstractControl) instances. The group's properties include its child controls. The top-level form in your component is a [FormGroup](https://angular.io/api/forms/FormGroup).
4. [***FormArray***](https://angular.io/api/forms/FormArray) tracks the value and validity state of a numerically indexed *array* of [AbstractControl](https://angular.io/api/forms/AbstractControl) instances.

The novalidate attribute in the <form> element prevents the browser from attempting native HTML validations.  Without a parent [FormGroup](https://angular.io/api/forms/FormGroup), [formControl]="name"worked earlier because that directive can stand alone, that is, it works without being in a [FormGroup](https://angular.io/api/forms/FormGroup). With a parent [FormGroup](https://angular.io/api/forms/FormGroup), the name input needs the syntax [formControlName](https://angular.io/api/forms/FormControlName)=name in order to be associated with the correct [FormControl](https://angular.io/api/forms/FormControl) in the class. This syntax tells Angular to look for the parent [FormGroup](https://angular.io/api/forms/FormGroup), in this case heroForm, and then *inside* that group to look for a [FormControl](https://angular.io/api/forms/FormControl) called name.

**FormBuilder**: The [FormBuilder](https://angular.io/api/forms/FormBuilder) class helps reduce repetition and clutter by handling details of control creation for you.

[FormBuilder](https://angular.io/api/forms/FormBuilder).group is a factory method that creates a [FormGroup](https://angular.io/api/forms/FormGroup).   [FormBuilder](https://angular.io/api/forms/FormBuilder).group takes an object whose keys and values are [FormControl](https://angular.io/api/forms/FormControl) names and their definitions.

The [formGroupName](https://angular.io/api/forms/FormGroupName) and [formControlName](https://angular.io/api/forms/FormControlName) attributes. They are the Angular directives that bind the HTML controls to the Angular [FormGroup](https://angular.io/api/forms/FormGroup) and [FormControl](https://angular.io/api/forms/FormControl) properties in the component class.

|  |  |
| --- | --- |
| **Property** | **Description** |
| myControl.value | the value of a [FormControl](https://angular.io/api/forms/FormControl). |
| myControl.status | the validity of a [FormControl](https://angular.io/api/forms/FormControl). Possible values: VALID, INVALID, PENDING, or DISABLED. |
| myControl.pristine | true if the user has *not* changed the value in the UI. Its opposite is myControl.dirty. |
| myControl.untouched | true if the control user has not yet entered the HTML control and triggered its blur event. Its opposite is myControl.touched. |

**Setvalue**: With setValue, you assign *every* form control value *at once* by passing in a data object whose properties exactly match the *form model* behind the [FormGroup](https://angular.io/api/forms/FormGroup).

The setValue method checks the data object thoroughly before assigning any form control values.It will not accept a data object that doesn't match the FormGroup structure or is missing values for any control in the group. This way, it can return helpful error messages if you have a typo or if you've nested controls incorrectly.patchValue will fail silently.On the other hand,setValue will catch the error and report it clearly.

Eg: this.heroForm.setValue({ name: this.hero.name, address: this.hero.addresses[0] || new Address() });

**patchValue**: With patchValue, you can assign values to specific controls in a [FormGroup](https://angular.io/api/forms/FormGroup) by supplying an object of key/value pairs for just the controls of interest.

Eg : this.heroForm.patchValue({ name: this.hero.name });

### **When to set form model values (***ngOnChanges***)**: In this approach, the value of hero in the HeroDetailComponent changes every time the user selects a new hero. You should call *setValue* in the [ngOnChanges](https://angular.io/guide/lifecycle-hooks#onchanges) hook, which Angular calls whenever the input hero property changes.

Eg : ngOnChanges() this.heroForm.setValue({ name: this.hero.name, address: this.hero.addresses[0] || new Address() }); }

**Reset Form**: You should reset the form when the hero changes so that control values from the previous hero are cleared and status flags are restored to the *pristine* state. The reset method has an optional state value so you can reset the flags and the control values at the same time. Internally, reset passes the argument to setValue.

Eg : ngOnChanges() { this.heroForm.reset({ name: this.hero.name, address: this.hero.addresses[0] || new Address() }); }

## *FormArray* to present an array of *FormGroups*: The Hero.addresses property is an array of Address instances. An address [FormGroup](https://angular.io/api/forms/FormGroup) can display one Address. An Angular [FormArray](https://angular.io/api/forms/FormArray) can display an array of address FormGroups.