**What is root module for Angular 2 and what are its parameters? hat are its parameters?**

**Answer**

@ngModule is the root module or class for an angular application. It is identified with decorator @ngModule and identifies the entry point of an application. It takes an object as parameter whose properties are imports, exports, declarations, providers, bootstrap.

**How does angular 2 keep track of two-way data binding?**

Angular 2 runs script for checking model stored values against displayed HTML values once per Javascript event cycle for whole application tree. If any value has changed on either side it replace the value to make them consistent. [(ngModel)]=”foo” is used as two-way binding attribute.

**What limits the number of two-way binding attributes in Angular 2?**

Too many two-way binding attributes can cause performance concerns in application. Each data binding pushes an event in Javascript event loop messages que, and pushing too many messages can block event loop and browser window.

**What is difference between component and directive in Angular 2?**

Component is a directive with template. Directive doesn’t have a template, while components have. Directives are identified with @Directive decorator while components are identified as @Component.

Components break up the application into smaller parts; whereas, Directives add behavior to an existing DOM element.

**What can be considered as replacement of $watch in Angular 2?**

ngOnChanges lifecycle hook can be considered as replacement of $watch in Angular 1. This hook is invoked every time angular identifies a change in components input model with two parameters; currentValue, oldValue. <http://www.angulartutorial.net/2017/12/watch-input-changes-angular-4.html>

# [Angular 4: How to watch an object for changes?](https://stackoverflow.com/questions/46330070/angular-4-how-to-watch-an-object-for-changes)

**I know that there are various ways to watch my form for changes. That is not what I am trying to do. As the title says, I am asking how to watch for changes to an object?**

You can't watch changes in an object. **Its not angular 1** there are no watchers here. Another solution will be via observables.

use form

<form #f="ngForm">

<input type="text" name="firstName" [(ngModel)]="customer.firstName">

<input type="text" name="favoriteColor" [(ngModel)]="customer.favoriteColor">

</form>

in code

@ViewChild('f') f;

ngAfterViewInit() {

this.f.form.valueChanges.subscribe((change) => {

console.log(change)

})

}

**What are pipes in Angular 2?**

Pipes are used in HTML templates to convert output to user friendly and readable form within interpolation braces i.e. {{ release | date }}. The ‘|’ is denoted as pipe.

Why we use eventemitter ?

Use EventEmitter only for event binding between a child and parent component.if we have the requirement to pass the data from child to parent component that time we use @output decorator and event emitter in child component.

@Component({

selector : 'child',

template : `

<button (click)="sendNotification()">Notify my parent!</button>

`

})

class Child {

@Output() notifyParent: EventEmitter<any> = new EventEmitter();

sendNotification() {

this.notifyParent.emit('Some value to send to the parent');

}

}

@Component({

selector : 'parent',

template : `

<child (notifyParent)="getNotification($event)"></child>

`

})

class Parent {

getNotification(evt) {

// Do something with the notification (evt) sent by the child!

}

}

[**https://coursetro.com/posts/code/171/Angular-7-Tutorial---Learn-Angular-7-by-Example**](https://coursetro.com/posts/code/171/Angular-7-Tutorial---Learn-Angular-7-by-Example)

## Angular 7 Templating

You may have noticed that one of the components we generated was called nav. Let's implement a header bar with a navigation in our app!

The first step is to visit the **app.component.html** file and specify the following contents:

<app-nav></app-nav>

<section>

<router-outlet></router-outlet>

</section>

So, we've removed a bunch of templating and placed in <app-nav></app-nav>, what does this do and where does it come from?

Well, if you visit **/src/app/nav/nav.component.ts** you will see that in the component decorator, there's a selector property bound to the value of app-nav. When you reference the selector of a given component in the form of a custom HTML element, it will nest that component inside of the component it's that's referencing it.

If you save the file you just updated, you will see in the browser we have a simple, nav works! And that's because the nav.component.html file consists of a simple paragraph stating as much.

At this point, let's specify the following HTML to create a simple navigation:

<header>

<div class="container">

<a routerLink="/" class="logo">apptitle</a>

<nav>

<ul>

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

<li><a routerLink="/about">About</a></li>

<li><a routerLink="/contact">Contact us</a></li>

</ul>

</nav>

</div>

</header>

The only thing that might look a little strange is **routerLink**. This is an Angular 7 specific attribute that allows you to direct the browser to different routed components. The standard href element will not work.

While we're here on the subject of templating, what if we wanted to display properties that are coming from our component? We use what's called interpolation.

Make the following adjustment to our template:

<!-- From: -->

<a routerLink="/">myapp</a>

<!-- To: -->

<a routerLink="/">{{ appTitle }}</a>

Interpolation is executed by wrapping the name of a property that's defined in the component between {{ }}.

Let's define that property in **nav.component.ts**:

export class NavComponent implements OnInit {

appTitle: string = 'myapp';

// OR (either will work)

appTitle = 'myapp';

constructor() { }

ngOnInit() {

}

}

You can use the TypeScript way of defining properties or standard JavaScript. Save the file and you will see **myapp** is back in the template.

There's a lot more to templating, but we will touch on those topics as we continue. For now, let's apply style to our header.

First, let's visit the global stylesheet by opening **/src/styles.scss** and define the following rulesets:

@import url('[**https://fonts.googleapis.com/css?family=Montserrat:400,700**](https://fonts.googleapis.com/css?family=Montserrat:400,700)');

body, html {

height: 100%;

margin: 0 auto;

}

body {

font-family: 'Montserrat';

font-size: 18px;

}

a {

text-decoration: none;

}

.container {

width: 80%;

margin: 0 auto;

padding: 1.3em;

display: grid;

grid-template-columns: 30% auto;

a {

color: white;

}

}

section {

width: 80%;

margin: 0 auto;

padding: 2em;

}

Visit **nav/component.scss** and paste the following contents:

header {

background: #7700FF;

.logo {

font-weight: bold;

}

nav {

justify-self: right;

ul {

list-style-type: none;

margin: 0; padding: 0;

li {

float: left;

a {

padding: 1.5em;

text-transform: uppercase;

font-size: .8em;

&:hover {

background: #8E2BFF;

}}}}}}

## Angular 7 Routing

Now that we have a navigation, let's make our little app actually navigation between our components as needed.

Open up **/src/app/app-routing.module.ts** and specify the following contents:

// Other imports removed for brevity

import { HomeComponent } from './home/home.component';

import { AboutComponent } from './about/about.component';

import { ContactComponent } from './contact/contact.component';

const routes: Routes = [

{ path: '', component: HomeComponent },

{ path: 'about', component: AboutComponent },

{ path: 'contact', component: ContactComponent },

];

// Other code removed for brevity

As we can see here, we're defining importing our components and defining an object for each route inside of the **routes** constant. These route objects also accept other properties, which allow you to define URL parameters, but because our app is simple, we won't be doing any of that.

Save this file and try clicking on the links above. You will see that each of the respective component's HTML templating shows up in the **<router-outlet></router-outlet>** defined in app.component.html.

## Angular 7 Event Binding

In the next several sections, we're going to use our **/src/app/home** component as a playground of sorts to learn features specific to Angular 7.

One of the most used forms of event binding is the click event. You often need to make your app respond when a user clicks something, so let's do that!

Visit the **/src/app/home/home.component.html** template file and specify the following:

<h1>Home</h1>

<button (click)="firstClick()">Click me</button>

You define an event binding by wrapping the event between (), and calling a method. You define the method in the **home.component.ts** file as such:

export class HomeComponent implements OnInit { constructor() { }

ngOnInit() {}

firstClick() {

console.log('clicked');}

You can experiment with the other event types by replacing (click) with the names below:

(focus)="myMethod()"

(blur)="myMethod()"

(submit)="myMethod()"

(scroll)="myMethod()" (cut)="myMethod()"

(copy)="myMethod()"

(paste)="myMethod()" (keydown)="myMethod()"

(keypress)="myMethod()"

(keyup)="myMethod()"

(mouseenter)="myMethod()"

(mousedown)="myMethod()"

(mouseup)="myMethod()"

(click)="myMethod()"

(dblclick)="myMethod()"

(drag)="myMethod()"

(dragover)="myMethod()"

(drop)="myMethod()"

## Angular 7 Class & Style Binding

Let's say that you want to control whether or not a CSS class is applied to a given element. Update the **h1** element in home.component.html to the following:

<h1 [class.gray]="h1Style">Home</h1>

Here, we're saying that the CSS class of .gray should only be attached to the h1 element if the property h1Style results to true. Let's define that in the home.component.ts file:

h1Style: boolean = false;

constructor() { }

ngOnInit() {

}

firstClick() {

this.h1Style = true;

}

Let's also define the .gray class in this component's scss file:

.gray {

color: gray;

}

Save it, and you can now click on the click me button to change the color of the Home title.

What if you wanted to control multiple classes on a given element? You can use ngClass. Modify the home component's template file to the following:

<h1 [ngClass]="{

'gray': h1Style,

'large': !h1Style

}">Home</h1>

Then, add the large ruleset to the .scss file:

.large {

font-size: 4em;

}

Now give it a shot in the browser. Home will appear large, but shrink down to the regular size when you click the button. Great!

You can also control appearance by changing the styles directly from within the template. Modify the template as such:

<h1 [style.color]="h1Style ? 'gray': 'black'">Home</h1>

Refresh and give this a shot by clicking the button.

Like ngClass() there's also an ngStyle() that works the same way:

<h1 [ngStyle]="{

'color': h1Style ? 'gray' : 'black',

'font-size': !h1Style ? '1em' : '4em'

}">Home</h1>

**Services:**

Services in Angular allow us to define code that's accessible and reusable throughout multiple components. A common use case for services is when you need to communicate with a backend of some sort to send and receive data.

ng generate service data

Open up the new service file **/src/app/data.service.ts** and let's create the following method:

// Other code removed for brevity

export class DataService {

constructor() { }

firstClick() {

return console.log('clicked');

}

}

To use this in a component, visit **/src/app/home/home.component.ts** and update the code to the following:

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

import { DataService } from '../data.service';

@Component({

selector: 'app-home',

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

styleUrls: ['./home.component.scss']

})

export class HomeComponent implements OnInit {

constructor(private data: DataService) { }

ngOnInit() {

}

firstClick() {

this.data.firstClick();

}

}

There are 3 things happening here:

* We're first importing the *DataService* at the top.
* We're creating an instance of it through dependency injection within the *constructor()* function.
* Then we call the method with *this.data.firstClick()* when the user clicks on the button.

## Angular 7 HTTP Client

In order to gain access to the HTTP client library, we have to visit the **/src/app/app.module.ts** file and make a couple changes. Up until this point, we haven't touched this file, but the CLI has been modifying it based on the generate commands we've issued to it.

Add the following to the imports section at the top:

// Other imports

import { HttpClientModule } from '@angular/common/http';

Next, add it to the imports array:

imports: [

BrowserModule,

AppRoutingModule,

HttpClientModule, // <-- Right here

],

Now we can use it in our **/src/app/data.service.ts** file:

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

import { HttpClient } from '@angular/common/http'; // Import it up here

@Injectable({

providedIn: 'root'

})

export class DataService {

constructor(private http: HttpClient) { }

getUsers() {

return this.http.get('[**https://reqres.in/api/users**](https://reqres.in/api/users)')

}

}

[**reqres.in**](http://reqres.in/) is a free public API that we can use to grab data.

Open up our home.component.ts file and modify the following:

export class HomeComponent implements OnInit {

users: Object;

constructor(private data: DataService) { }

ngOnInit() {

this.data.getUsers().subscribe(data => {

this.users = data

console.log(this.users);

}

);

}

}

The first thing you might notice is that we're placing the code inside of the **ngOnInit()** function, which is a lifecycle hook for Angular. Any code placed in here will run when the component is loaded.

We're defining a **users** property, and then we're calling the .getUsers() method and subscribing to it. Once the data is received, we're binding it to our users object and also console.logging it.

Give it a try in the browser and you will see the console shows an object that's returned. Let's display it on our home template!

Open up **home.component.html** and specify the following:

<h1>Users</h1>

<ul \*ngIf="users">

<li \*ngFor="let user of users.data">

<img [src]="user.avatar">

<p>{{ user.first\_name }} {{ user.last\_name }}</p>

</li>

</ul>

Great! Let's specify some CSS to make this look better in **home.component.scss**:

ul {

list-style-type: none;

margin: 0;padding: 0;

li {

background: rgb(238, 238, 238);

padding: 2em;

border-radius: 4px;

margin-bottom: 7px;

display: grid;

grid-template-columns: 60px auto;

p {

font-weight: bold;

margin-left: 20px;

}

img {

border-radius: 50%;

width: 100%;

}

}

}

## Angular 7 Forms

If you recall, we generated a component called **contact**. Let's create a contact form so that you can learn how to use forms in Angular 7.

Angular 7 provides you with two different approaches to dealing with forms: template driven and reactive forms. I'm not going to go into the differences between these two approaches, but reactive forms generally provide you with more control andform validation can be unit tested as opposed to template driven forms.

To get started, we have to visit the **app.module.ts** file and import the Reactive Forms Module:

// other imports

import { ReactiveFormsModule } from '@angular/forms';

// other code

imports: [

BrowserModule,

AppRoutingModule,

HttpClientModule,

ReactiveFormsModule // <- Add here

],

 Next, visit the **contact.component.ts** file and specify the following

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

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

@Component({

selector: 'app-contact',

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

styleUrls: ['./contact.component.scss']

})

export class ContactComponent implements OnInit {

messageForm: FormGroup;

submitted = false;

success = false;

constructor(private formBuilder: FormBuilder) { }

ngOnInit() {

this.messageForm = this.formBuilder.group({

name: ['', Validators.required],

message: ['', Validators.required]

});

}

onSubmit() {

this.submitted = true;

if (this.messageForm.invalid) {

return;

}

this.success = true;

}

}

First, we're importing FormBuilder, FormGroup, Validators from @angular/forms.

Then we're setting a few boolean properties that will help us determine when the form has been submitted and if it validation is successful.

Then we're creating an instance of the formBuilder in the constructor. We then use this form building to construct our form properties in the ngOnInit() lifecycle hook.

We have two properties, name and message.

Then we created an onSubmit() method that will be called when the user submits the form. This is typically where you would call upon a method in the service to communicate with a mail service of sorts.

Next, visit contact.component.html:

<h1>Contact us</h1>

<form [formGroup]="messageForm" (ngSubmit)="onSubmit()">

<h5 \*ngIf="success">Your form is valid!</h5>

<label>

Name:

<input type="text" formControlName="name">

<div \*ngIf="submitted && messageForm.controls.name.errors" class="error">

<div \*ngIf="messageForm.controls.name.errors.required">Your name is required</div>

</div>

</label>

<label>

Message:

<textarea formControlName="message"></textarea>

<div \*ngIf="submitted && messageForm.controls.message.errors" class="error">

<div \*ngIf="messageForm.controls.message.errors.required">A message is required</div>

</div>

</label>

<input type="submit" value="Send message" class="cta">

</form>

<div \*ngIf="submitted" class="results">

<strong>Name:</strong>

<span>{{ messageForm.controls.name.value }}</span>

<strong>Message:</strong>

<span>{{ messageForm.controls.message.value }}</span>

</div>

Baked in here is a full form with validation. It also prints out the form values beneath it when the form has been submitted.

Let's update the css for the component to make it look decent:

label {

display: block;

input, textarea {

display: block;

width: 50%;

margin-bottom: 20px;

padding: 1em;

}

.error {

margin-top: -20px;

background: yellow;

padding: .5em;

display: inline-block;

font-size: .9em;

margin-bottom: 20px;

}

}

.cta {

background: #7700FF;

border: none;

color: white;

text-transform: uppercase;

border-radius: 4px;

padding: 1em;

cursor: pointer;

font-family: 'Montserrat';

}

.results {

margin-top: 50px;

strong {

display: block;

}

span {

margin-bottom: 20px;

display: block;

}

}

1. **What will be the output of below program?**

function fun(input: boolean) {

let a = 100;

if (input) {

let b = a + 1;

return b;

}

return b;

}

a. Undefined

b. 101

c. Compilation error for a

d. Compilation error for b

ans) d

1. **What is name of a special function of class which gets called when object is created and it’s syntax in Typescript?**

constructor — constructor(){}

1. **What are the basic rules of Decorators?**

a function which has annotation of @() with optional parameters

1. **If you do not know the number of arguments to be passed to function in advance, you should use \_\_\_\_\_\_\_ parameter type.**

Rest parameter

1. **\_\_\_\_\_ keyword is used to access class’s member variables and functions inside class member function.**

**this**

**6. In Angular, you can pass data from parent component to child component using**

@Input()

in child component we mention @input in front of property

@Input() testele: {type: string, name: string, content: string};

In parent componet

**7. In Angular, you can pass data from child component to parent component using**

Ans)@Output()

**8. Write a syntax for ngFor with <li> example.**

Ans)<li \*ngFor=”let item of allItems”>

{{item}}

</li>

**9. We must import Formsmodule module to use [(ngModel)].**

**10. Import reactive FormsModule module to use reactive form.**

**11. Write an example to define custom event with Boolean argument with code and passing data to parent component.**

Ans) objEvent = new EventEmitter<boolean>();

**12. Write a syntax to bind custom CSS class (e.g. highlighted) to a <div> tag.**

Ans)<div [class]=”myclass”]

**13. You can create local HTML reference of HTML tag using variable which starts with character**

a. @

b. #

c. \*

d. &

Ans)b

**14. You can access HTML local reference alias in component’s typescript code using \_\_\_\_\_\_\_\_\_\_\_ decorator.**

**@viewchild**

Through this we can access input element from html to .ts file.

**Component.html:**

<input type="text" class="form-control" #locRef>

**Component.ts:**

@ViewChild('locRef') varViewChild: ElementRef;

console.log(this.varViewChild);

**15. In template driven form \_\_\_\_\_\_\_\_\_ object is created internally whenever we have below code**

<form #heroForm=”ngForm”>

whereas in reactive form, we have to create this object explicitly.

Ans) FormGroup

**16. Choose correct form control class name which is set to true when value is modified**

a. .ng-valid

b. .ng-invalid

c. .ng-pending

d. .ng-pristine

e. .ng-dirty

f. .ng-untouched

g. .ng-touched

Ans)e

**17. If you provide a service in two components’ “providers” section of @Component decorator, how many instances of service shall get created?**

a. 1

b. 2

c. 3

d. 4

Ans)b

**18. When you apply ‘pipe’, it changes value of underlying component’s member variable as well.**

a. True

b. False

Ans)b

**19. In routing, below tag is used to show selected route component dynamically**

a. <router></router>

b. <router-output></router-output>

c. <router-outlet></router-outlet>

d. <router-input></router-input>

Ans) c

**20. We need to call below method of RouterModule for providing all routes in AppModule**

a. RouterModule.forChild

b. RouterModule.forRoot

c. RouterModule

d. RouterModule.all

Ans)b

**1. Angular web application can only be written using Typescript as code behind language**

a. True

b. False

**2. Fill in blank keyword used to implement inheritance in typescript ?**

<derived class>\_\_\_\_\_\_\_\_\_\_\_\_<baseclass>

**3. Decorator is a class?**

a. True

b. False

**4. Optional parameter is marked using below syntax:**

a. Fun(name:string=”balram”)

b. Fun(name?:string)

c. Fun(…names:string[])

**5. We can have constructor overloading in Typescript**

a. True

b. False

ans)

1. b

2. extends

3. b

4. b

5. b

**6. In Angular, you can hide/remove HTML element using**

a. #ngIf

b. \*ngIf

c. #hide

d. \*hide

**7. There can be more than one RootModule in Angular**

a. True

b. False

**8. ngFor can be applied on any HTML tag e.g.<div> and not just <li>**

a. True

b. False

**9. We can implement two-way data binding using**

a. [ngModel]

b. [(ngModel)]

c. (ngModel)

d. ng-Model

**10. For standard HTML onclick event, we can have below Angular event**

a. (onclick)

b. (click)

c. (clickhandler)

d. [click]

ans)

6. b

7. b

8. a

9. b

10. b

**11. EventEmitter data type implementation uses generics concept of Typescript.**

a. True

b. False

**12. You can update background color of <div> HTML tag using [ngStyle]**

a. True

b. False

**13. You can have more than one #local variable for an HTML element. E.g. <input #ref1 #ref2>**

a. True

b. False

**14. Angular custom/built-in Directives cannot listen to standard HTML event raised on their host**

a. True

b. False

**15. Using Template Driven Form, you can create dynamic form layout**

a. True

b. False

ans)

11. a

12. a

13. b

14. b

15. b

**16. Choose correct form control class name which is set to true when blur event gets fired on control**

a. .ng-valid

b. .ng-invalid

c. .ng-pending

d. .ng-pristine

e. .ng-dirty

f. .ng-untouched

g. .ng-touched

**17. Angular Service can be injected at Root Module, Feature Module or Component level**

a. True

b.False

**18. You can modify DOM of target element using Pipe**

a. True

b. False

**19. You can define 404 route using below wild card route path**

a. \*\*

b. /

c. ##

d. default

**20. Below service can be injected in your component to do dynamic route navigation**

a. Router

b. Routing

c. RouterService

d. RoutingService

### Answers:

16. g

17. a

18. b

19. a

20. a

1. **Angular template can be defined inline as well as a separate html file**

A. True

B. False

**2. Angular can be used with different module bundle packages apart from WebPack like Systemjs, Commonjs etc**

A. True

B. False

**3. Below tool is used for Static Code analysis of Angular application**

A. Linting

B. Scanning

C. Profiling

D. Refactoring

**4. Below construct provide modular structure to angular application**

A. Component

B. NgModule

C. Directives

D. Pipes

**5. Pipes modify underlying value of bound expression/variables**

A. True

B. False

**6. Which decorator is used for declaring class as a Service**

A. @Service

B. @Injectable

C. @Provider

D. @DI

**7. Below module is required for building reactive forms**

A. FormsModule

B. ReactiveFormsModule

C. DynamicFormsModule

D. None of the above

**8. Directive object can be created and used in HTML without target host DOM element**

A. True

B. False

**9. Using below directive you can modify inner HTML of DOM element**

A. Attribute directive

B. Structural directive

**10. You can pass optional parameters to custom pipes**

A. True

B. False

**11. Pick incorrect type of pipe in angular**

A. Pure Pipe

B. Impure Pipe

C. Reactive Pipe

**12. If one service is provided at different NgModule and Component here will be only one instance of that service**

A. True

B. False

**13. Below life cycle hook is executed when component is destroyed**

A. destructor

B. delete

C. destroy

D. ngDelete

**14. Select type of pipe Angular executes during every change detection cycle of component**

A. Pure Pipe

B. Impure Pipe

**15. In Angular application there can be at most one <router-outlet> can be used for routing**

A. True

B. False

**16. Which router function should be called in to feed route configuration for lazy routing approach**

A. RouterModule.forRoot

B. RouterModule.forChild

**17. Which routing guard is used to check whether routing can take place or not**

A. CanActivate

B. Resolve

C. CanLoad

D. CanDeactive

**18. void => \* transition animation gets applied when**

A. Element is entering

B. Element is leaving

**19. \* => void transition animation gets applied when**

A. Element is entering

B. Element is leaving

**20. Below compilation option create smaller in size and faster in performance angular distributed package**

A. JIT

B. AOT

### Answers:

1- A

2- A

3- A

4- B

5- B

6- B

7- B

8- B

9- B

10- A

11- C

12- B

13- C

14- B

15- B

16- B

17- A

18- A

19- B

20- B

1. **In AOT compilation mode, Angular web application get shipped with angular compiler along with website content in browser.**

A. True

B. False

**2. Async Pipe subscribes to observer and update expression whenever there is data sent from observer**

A. True

B. False

**3. Below command is used to run Static Code analysis of Angular application**

A. ng build

B. ng profile

C. ng lint

D. ng serve

**4. Router service needs to be explicitly provided in angular module to use it in other component via DI**

A. True

B. False

**5. We can chain multiple pipe in a single expression along with “async” pipe**

A. True

B. False

**6. Below component represent “target/host” DOM element inside Directive’s constructor**

A. Element

B. ElementRef

C. Host

D. Target

**7. To build application in production mode, use below command**

A. ng build

B. ng serve

C. ng build — prod

D. ng lint

**8. Directive can listen to host/target events using below decoarator**

A. @Listener()

B. @HostListner()

C. @TargetListener()

D. @OnListener()

**9. Using below wild card we can define page not found route**

A. \*

B. \*\*

C. 404

D. ‘’

**10. There can be more than one <router-outlet> element in angular application**

A. True

B. False

**11. Below service can be used to extract route parameters inside component**

A. Router

B. Route

C. ActivatedRoute

**12. HTTP service’s get/put/post/delete function returns**

A. object of type “any”

B. Observable

C. Json data

D. callback function

**13. Observer sends data to multiple clients via below method**

A. send()

B. next()

C. emit()

D. publish()

**14. By default, component’s change detection strategy is set to**

A. OnPush

B. Default

**15. One component can be declared inside more than one NgModule**

A. True

B. False

**16. To use HttpClient component you need to import below module**

A. HttpModule

B. HttpClientModule

C. Http

D. None of the above

**17. Below pseudo class represent void => \* animation state**

A. :enter

B. :leave

**18. HTTP service is called even if observer doesn’t have “subscribe” method**

A. True

B. False

**19. A component can have child route defined using below property**

A. routes

B. children

C. routeCollection

D. None of the above

**20. OnPush is faster in performance than Default change detection strategy**

A. True

B. False

### Answers:

1- B

2- A

3- C

4- B

5- A

6- B

7- C

8- B

9- B

10- A

11- C

12- B

13- B

14- B

15- B

16- B

17- A

18- B

19- B

20- A

Question # 9) In Angular 2 Module consist of what arrays?

Answer # Angular 2 Module consist of Bootstrap Array, Export Array, Import Array.

Question # 10) What is Bootstrap array in Angular 2?

Answer # In [Angular 2](https://thinkster.io/tutorials/learn-angular-2), Bootstrap array is used to inform Angular JS which components need to be loaded, so that its functionality can be accessed in the application.

Question # 16) What is the meaning of Ng in Angular 2?

Answer # ng is the core module and stands for Angular. The ng module is loaded by default when an AngularJS application is started.  
The module itself contains the essential components for an AngularJS application to function.

Question # 19) What are the main components of Routing in Angualr 2?

Answer # There are four main components that can be used to configure **routing in Angular 2**:

Routes: – It describes our application’s routes.

RouterModule

In app-routing.module.ts

const approuts: Routes = [

{path: '', component: HomeComponent},

{path: 'recipes', loadChildren: () => RecipesModule},

// {path: 'recipes', loadChildren: './recipes/recipes.module#RecipesModule'},

// for lazy loding of recipes we commented recipes module in app.module and added the path above

{path: 'shopping-list' , component: ShoppingListComponent },

{path: 'forms-demo' , component: FormsDemoComponent },

{path: 'pipes' , component: FiltersComponent }

];

@NgModule({

imports: [RouterModule.forRoot(approuts)], /\*\*here we configure the routes with routermodule \*/

exports: [RouterModule] /\*\* we have to export this Router module to main module 'app.module' \*/

})

RouterOutlet: – It is a “placeholder” component that gets expanded to each route’s content.

We add it in app.module.html page

<app-header ></app-header>

<div class="container">

<div class="row">

<div class="col-md-12">

<router-outlet></router-outlet>

</div>

</div>

</div>

RouterLink: – IT is used to link to application’s routes.

RouterLinkActive:

<li routerLinkActive="active"><a routerLink="/recipes" >Recipies</a></li>

<li routerLinkActive="active"><a routerLink="/shopping-list" >Shopping List</a></li>

Question # 20) What is Tsconfig JSON in Angulr 2?

Answer # The presence of a tsconfig.json file in a directory indicates that the directory is the root of a TypeScript project.  
The tsconfig.json file specifies the root files and the compiler options required to compile the project.

**Sharing data between Cross components**

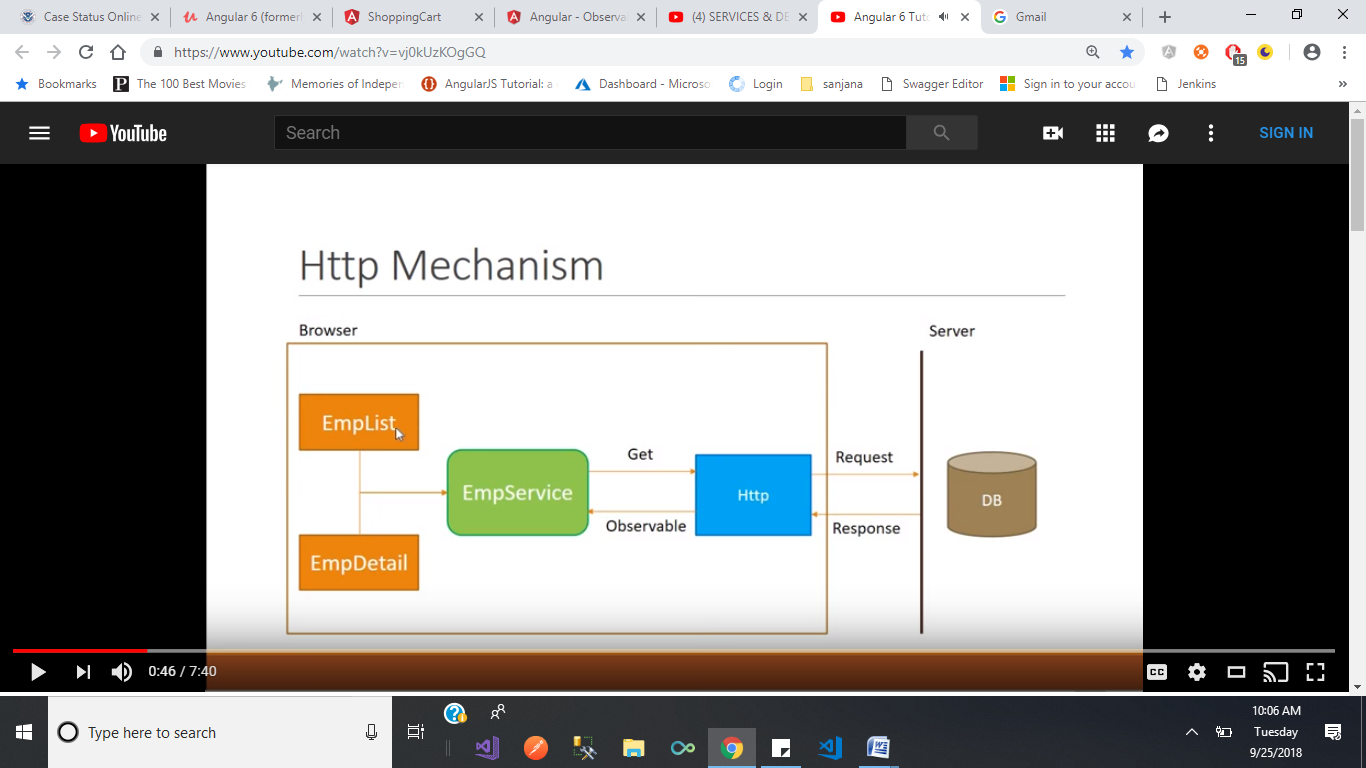
[**https://angularfirebase.com/lessons/sharing-data-between-angular-components-four-methods/**](https://angularfirebase.com/lessons/sharing-data-between-angular-components-four-methods/)

1. [**Parent to Child: Sharing Data via Input**](https://angularfirebase.com/lessons/sharing-data-between-angular-components-four-methods/#Parent-to-Child-Sharing-Data-via-Input)
   1. [parent.component.ts](https://angularfirebase.com/lessons/sharing-data-between-angular-components-four-methods/#parent-component-ts)
   2. [child.component.ts](https://angularfirebase.com/lessons/sharing-data-between-angular-components-four-methods/#child-component-ts)
2. [**Child to Parent: Sharing Data via ViewChild**](https://angularfirebase.com/lessons/sharing-data-between-angular-components-four-methods/#Child-to-Parent-Sharing-Data-via-ViewChild)
   1. [parent.component.ts](https://angularfirebase.com/lessons/sharing-data-between-angular-components-four-methods/#parent-component-ts-1)
   2. [child.component.ts](https://angularfirebase.com/lessons/sharing-data-between-angular-components-four-methods/#child-component-ts-1)
3. [**Child to Parent: Sharing Data via Output() and EventEmitter**](https://angularfirebase.com/lessons/sharing-data-between-angular-components-four-methods/#Child-to-Parent-Sharing-Data-via-Output-and-EventEmitter)
   1. [parent.component.ts](https://angularfirebase.com/lessons/sharing-data-between-angular-components-four-methods/#parent-component-ts-2)
   2. [child.component.ts](https://angularfirebase.com/lessons/sharing-data-between-angular-components-four-methods/#child-component-ts-2)
4. [**Unrelated Components: Sharing Data with a Service**](https://angularfirebase.com/lessons/sharing-data-between-angular-components-four-methods/#Unrelated-Components-Sharing-Data-with-a-Service)
   1. [data.service.ts](https://angularfirebase.com/lessons/sharing-data-between-angular-components-four-methods/#data-service-ts)
   2. [parent.component.ts](https://angularfirebase.com/lessons/sharing-data-between-angular-components-four-methods/#parent-component-ts-3)
   3. [sibling.component.ts](https://angularfirebase.com/lessons/sharing-data-between-angular-components-four-methods/#sibling-component-ts)

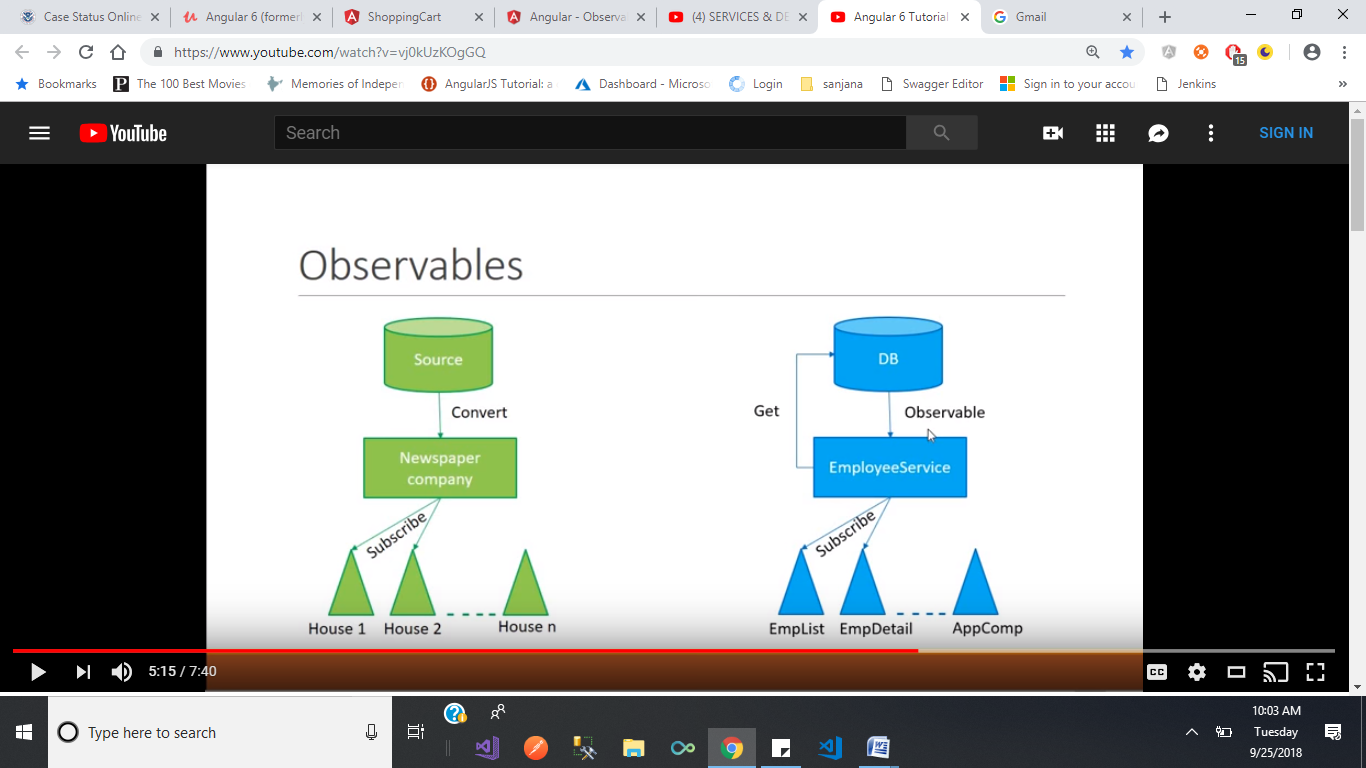
### ****What Are Observables?****

Observables are a lazy collection of multiple values, or data, over a period of time. Observables open the continuous channel of communication where multiple values are emitted over time. This allows us to determine the pattern of the data.

In a real-world example, we can say that the Internet service offered by mobile devices is an observable. It is available only to the people who have subscribed to it. We continuously receive this service from the service provider only as long as this service is on and we are subscribed to it.



In our application we have two components emplist and empDetails components. To fetch the details from the web application server we use Http call. These calls we write in employee service related to data. First we send get http request. Then it feches the data from the server and send the http response . this http response we get as observable. Employee service need to cast the observable into employee array and return the data to emplist and empdetails components.



Based on the newspaper company request source continuously send the sequence of data to the company. This news paper company format that information into particular format and print the papers. But it doesn’t send the papers to every home . who ever subscribed to that paper company for them only they send the papers.

Communicating Between Components with Observable & Subject

### Observable.subscribe()

The observable subscribe method is used to subscribe to messages that are sent to an observable.

### Subject.next()

The subject next method is used to send messages to an observable which are then sent to all subscribers of that observable.

## Angular 2/5 Message Service

The message service enables subscribing to messages and sending messages from any component in the application.

IMPORTANT: If you add a service like this to your application don't forget to add it to the providers: [...] section of your app.module.ts file, [see here](https://github.com/cornflourblue/angular2-communicating-between-components/blob/master/app/app.module.ts#L19-L21)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import { Injectable } from '@angular/core';  import { Observable } from 'rxjs';  import { Subject } from 'rxjs/Subject';  @Injectable()  export class MessageService {      private subject = new Subject<any>();      sendMessage(message: string) {          this.subject.next({ text: message });      }        clearMessage() {          this.subject.next();      }      getMessage(): Observable<any> {          return this.subject.asObservable();      }  } |

## Angular 2/5 App Component that Receives Messages

The app component uses the message service to subscribe to new messages and make them available to the app component template via the message property.

[?](http://jasonwatmore.com/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | import { Component, OnDestroy } from '@angular/core';  import { Subscription } from 'rxjs/Subscription';    import { MessageService } from './\_services/index';    @Component({      moduleId: module.id,      selector: 'app',      templateUrl: 'app.component.html'  })    export class AppComponent implements OnDestroy {      message: any;      subscription: Subscription;        constructor(private messageService: MessageService) {          // subscribe to home component messages          this.subscription = this.messageService.getMessage().subscribe(message => { this.message = message; });      }        ngOnDestroy() {          // unsubscribe to ensure no memory leaks          this.subscription.unsubscribe();      }  } |

## Angular 2/5 Home Component that Sends Messages

The home component uses the message service to send messages to the app component.

[?](http://jasonwatmore.com/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | import { Component } from '@angular/core';    import { MessageService } from '../\_services/index';    @Component({      moduleId: module.id,      templateUrl: 'home.component.html'  })    export class HomeComponent {      constructor(private messageService: MessageService) {}        sendMessage(): void {          // send message to subscribers via observable subject          this.messageService.sendMessage('Message from Home Component to App Component!');      }        clearMessage(): void {          // clear message          this.messageService.clearMessage();      }  } |

Recipeservice.ts

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

import { Subject } from 'rxjs/Subject';

import { Recipe } from '../recipes/recipe.model';

import { Ingredient } from '../shared/ingredient.model';

import { ShoppingListService } from '../shopping-list/shopping-list.service';

import { Http, Response, ResponseType } from '@angular/http';

import { AuthService } from '../auth/auth.Service';

import { HttpClient, HttpHeaders, HttpParams, HttpRequest } from '@angular/common/http';

// import 'rxjs/map';

import 'rxjs/add/operator/map';

@Injectable()

export class RecipeService {

recipesChanged = new Subject<Recipe[]>();

setRecipes(recipes: Recipe[]) {

this.recipes = recipes;

this.recipesChanged.next(this.recipes.slice());

}

addIngredientsToShoppingList(ingredients: Ingredient[]) {

this.slService.addIngredients(ingredients);

}

addRecipe(recipe: Recipe) {

this.recipes.push(recipe);

this.recipesChanged.next(this.recipes.slice());

}

updateRecipe(index: number, newRecipe: Recipe) {

this.recipes[index] = newRecipe;

this.recipesChanged.next(this.recipes.slice());

}

recipes-list.componet.ts use this recipeschanged property

ngOnInit() {

this.subscription = this.recipeservice.recipesChanged.subscribe(

(recipes: Recipe[]) => {

this.recipes = recipes;

}

);

this.recipes = this.recipeservice.getRecipes();

}

In header.component.ts we used recipeservice

onSaveData() {

this.recipeService.storeRecipes()

.subscribe(

(response) => {

console.log(response);

}, (error) => console.log(error)

);

**What Is Angular 4 or 5 or 6?**

Angular is a web development framework for developing mobile apps as well as desktop applications.

**What Is Angular Prerequisites?**

Before you can install Angular 4 or 5, you must need to have some prerequisites.

ü  You must to have **Node.js** installed.

ü  You must to have **NPM** (Node Package Manager) installed.

  node -v

**Noted Points -**

We need to setup our machine’s local environments which are the following.

ü  node.js >= 6.9.x

ü  npm >= 3.x.x

ü  Visual Studio Code or Atom

ü  And so on

# What's New In Angular 6? What Are Improvements In Angular 6?

Service level changes (the way of marking a service as global) -

In the earlier versions, if you want to provide a service to the entire application –you should add it to *providers []* in the AppModule but in the Angular 6 released you should not add in the *providers []* in the AppModule.

Example for marking a service as global -

Instead of

//my.service.ts

export class MyService { }

//In app.module.ts

//JavaScript imports services

import { MyService } from './my-serice.service';

//AppModule class with the @NgModule decorator

@NgModule({

  declarations: [],

  providers: [MyService] //My services instances are now available across the entire app.

})

export class AppModule {

   //exporting app module

}

Use with Angular 6 released-

//my.service.ts

@Injectable({providedIn: 'root'})

export class MyService { }

@NgModule({

  declarations: [],

  providers: [] // Service does not need to be added here

})

export class AppModule {}

The second one obviously saves you some lines of code as compare to previous code.

**Angular 6 introduces Angular Elements** -

**1)** Typescript 2.6.x supports

**2)** Added Angular Material and CDK Stable

**3)** Component Dev Kit (CDK) - CDK allows you to build your own library of UI components using Angular Material.

**4)** Improved decorator error messages

**5)** Fix platform-detection example for Universal

**6)** Ivy Renderer - It is a new backward compatible and main focused area - speed improvements, size reduction, and increased flexibility.

**7)** Add afterContentInit and afterContentChecked to render

**8)** Added to supports of nativeElement

**9)** Added Optional generic type for ElementRef

The Example looks like -

@ViewChild('your-element') yourElement:ElementRef;

**10)** Bazel Compiler - Bazel only rebuilds what is necessary.

**11)** Added Test Comment

**12)** Add missing lifecycle tests for projected components

**13)** Closure Compiler - Closure Compiler consistently generates smaller bundles.

**14)** Rename QueryPredicate to LQuery and LQuery to LQueries

**15)** Service Worker - Service worker is a script that runs in the web browser. It also manages caching for an application.

**16)** Added multiple validators for array method of FormBuilder

**17)** Handle string with and without line boundary - Now Handle string with and without line boundary (^ & $) on pattern validators. Previously, it works with string not boundaries.

**18)** AbstractControl statusChanges - Previous version, not emits an event when you called “markAsPending” but now emits an event of "PENDING" when we call AbstractControl markAsPending.

**19)** Updates on NgModelChange - Now emitted after value and validity is updated on its control. Previously, it was emitted before updated.

**20)** Allow HttpInterceptors to inject HttpClient –

Previously, an interceptor attempting to inject HttpClient directly would receive a circular dependency error, as HttpClient was constructed via a factory which injected the interceptor instances. Users want to inject HttpClient into interceptors to make supporting.

Either HttpClient or the user has to deal specially with the circular Dependency. This change moves that responsibility into HttpClient itself. By utilizing a new class HttpInterceptingHandler which lazily Loads the set of interceptors at request time, it's possible to inject HttpClient directly into interceptors as construction of HttpClient no longer requires the interceptor chain to be constructed.

**21)** Add navigationSource and restoredState to NavigationStart – Currently, NavigationStart there is no way to know if navigation was triggered imperatively or via the location change. These two use cases should be handled differently for a variety of use cases (e.g., scroll position restoration). This PR adds a navigation source field and restored navigation id (passed to navigations triggered by a URL change).

**22)** Add type and hooks to directive def

**23)** Enable size tracking of a minimal CLI render3 application

**24)** Add canonical view query

**25)** Language Service – The 2.6 version of Typescript’s “resolveModuleName” started to require paths passed to be separated by '/' instead of being able to handle '\'.

**What Is a Template Reference Variable?**

A template reference variable is a way of capturing a reference to a specific element, component, directive, and pipe so that it can be used someplace in the same template HTML.

You should declare a reference variable using the hash symbol (#).

Template Reference Variable Syntax –

You can use a template reference variable by two ways.

1.      Using hash symbol (#)

2.      Using reference symbol (ref-)

The following examples of specifying a template reference variable using Input Text Box –

I have declared a reference variable “cellnumber” using the **hash symbol (#)** and **reference symbol (ref-)**.

<input type="text" ref-cellnumber> //cellnumber will be a template reference variable.

And

<input #cellnumber placeholder="Cell number"> //cellnumber will be a template reference variable.

I have created a reference to the input element that can be used later on in my template and the scope for “**cellnumber**” variable is the entire HTML template in which the reference is defined.

Here is how I could use that reference to get the value of the input for instance –

//cellnumber refers to the input element

<button (click)="show(cellnumber)">click to see</button>

In the below line of code, the variable “cellnumber” refer to the HTMLElement object instance for the input -

show(cellnumber: HTMLInputElement){

  console.log(cellnumber.value);

}

You can use the ViewChild decorator to reference it inside your component.

import {ViewChild, ElementRef} from '@angular/core';

// Reference cellnumber variable inside Component

@ViewChild('cellnumber') cellInputRef: ElementRef;

And finally, you can use this.nameInputRef anywhere inside your component class.

show(){

  this.contactNumber = this.cellInputRef.nativeElement.value

}

Template Reference Variable with NgForm –

Here we will discuss about how to access NgForm directive using template reference variable.

<form (ngSubmit)="onSubmitEmployee(empForm)" #empForm="ngForm">

  <label>F-Name </label><input name="f-name" required [(ngModel)]="employee.fname">

  <label>L-Name </label><input name="l-name" required [(ngModel)]="employee.lname">

  <label>Age    </label><input name="age" required [(ngModel)]="employee.age">

  <button type="submit" [disabled]="!empForm.form.valid">Submit</button>

</form>

In the above NgForm example contains an ngSubmit event and form directive.

The ngSubmit – The ngSubmit directive specifies a function to run when the form is submitted. Here on form submit onSubmitEmployee component method will be called.

The NgForm - It is nestable alias of form directive. The main purpose of ngForm is to group the controls, but not a replacement of <form> tag.

As you know, the HTML does not allow nesting of form elements. It is very useful to nest forms.

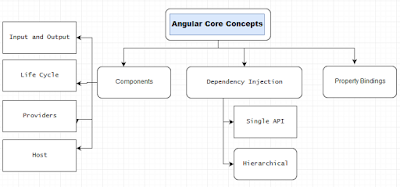
**What are components?**

Components are the most basic building block of a UI in Angular applications and it controls views (HTML/CSS). They also communicate with other components and services to bring functionality to your applications.

Technically components are basically [TypeScript](https://www.code-sample.com/2017/06/typescript-interview-questions-and.html) classes that interact with the HTML files of the components, which get displayed on the browsers.

The component is the core functionality of Angular applications but you need to know to pass the data into the components to configure them.

Angular applications must have a root component that contains all other components.

[](https://3.bp.blogspot.com/-SG-9KGnLYE4/WvUtnVQkn-I/AAAAAAAASiE/cwlLIXKFM1siU-RZvdPxnWBPgrponMnHgCLcBGAs/s1600/Core-Concepts-Of-Angular-6-5.png)

Components are created using **@Component** decorator that is part of **@angular/core module**.

You can create your own project using Angular CLI, this command allows you to quickly create an Angular application like - generate components, services, pipes, directive, classes, and modules, and so on as per your requirements.

Create your own component (login) using below command line –

ng g component login

login.component.ts -

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

@Component({

  selector: 'app-login',

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

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

})

export class LoginComponent implements OnInit {

  constructor() { }

  ngOnInit() {

  }

}

The above component class shows some of the most useful **@Component** configuration options –

1.      Selector

2.      TemplateUrl

3.      StyleURLs

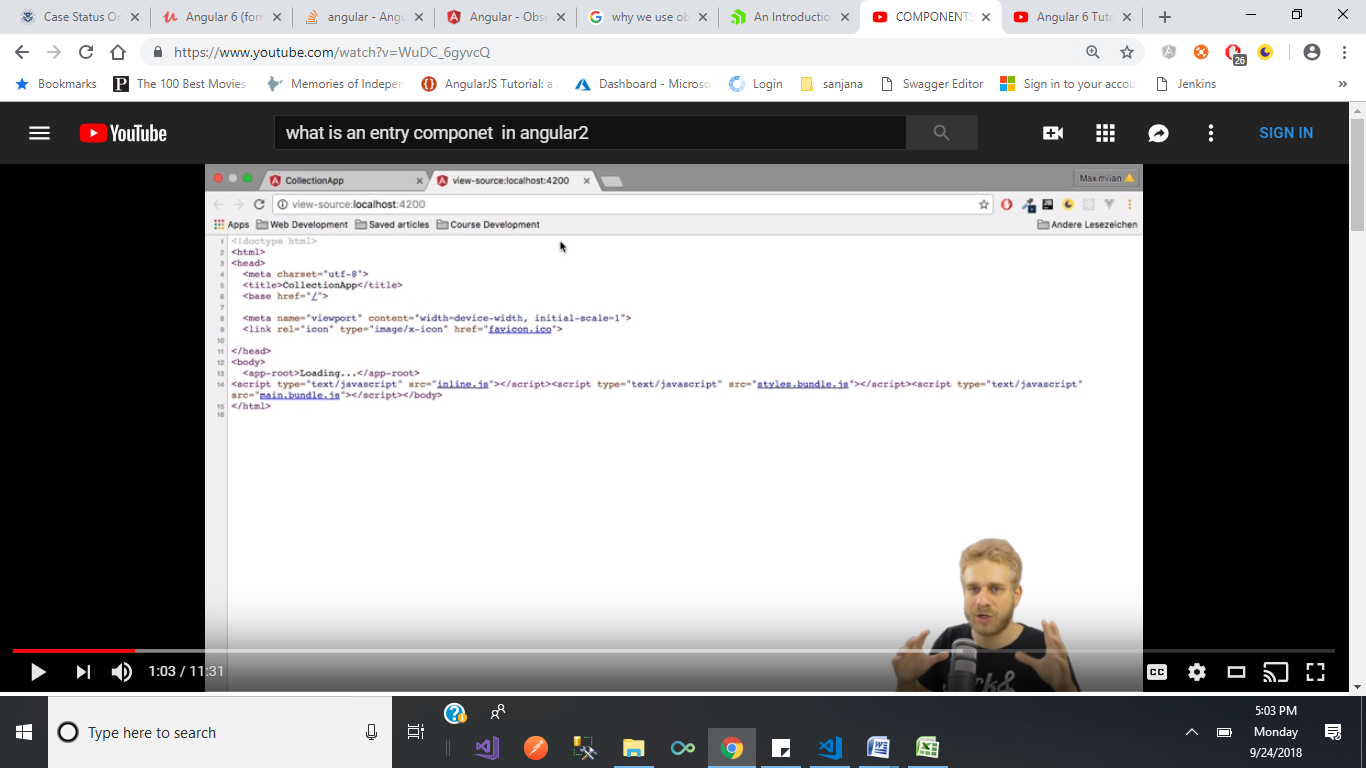
The selector – It is a CSS selector that tells Angular to create an instance of this component wherever it finds the corresponding tag in template HTML. For example, it is - <app-login></app-login>

The templateUrl – It is the module-relative address of this component's HTML template and you can also provide the inline HTML template.

The styleUrls - It can be used for CSS rules and it will affect the style of the template elements and you can also provide the inline style CSS.

**What Is an EntryComponent in Angular 5, and 4?**

When we create the angular application. If we run the application

we can see inline,style.bundle and main.bundle.js files . main.bundle contains

<body>

<app-root></app-root>

</body>

Here <app-root></app-root> indicates app.component.ts file in that file selector we have selector metadata as selector: “app-root” and template-url : “app.component.html ”

So it renders app.component and displays on the browser.

But by default angular doesn’t check all this process.

The first thing execute is main.ts file , it contains bootstrap code, which gets run first if we run the application.

platformBrowserDynamic().bootstrapModule(AppModule)

.catch(err => console.log(err));

It takes us to app.module.

Appmodule contains @ngmodule which contains bootstrap:[] array . whatever the component we specify here it considered as root component. This component displays when the application get started.

You specify an entry component by bootstrapping in the **Angular module** or you specify an entry component by routing definition.

There are two main kinds of entry components which are following -

1.      The bootstrapped root component

2.      A component you specify in a route

The bootstrapped entry component -

A bootstrapped component is an entry component that Angular loads into DOM at the application launch and the other root components loaded dynamically into entry components through routers.

The angular loads a root dynamically because it is bootstrapped in the Angular Module. In the below example, AppComponent is a root component so that angular loads dynamically.

Example –

import { BrowserModule } from '@angular/platform-browser';

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

import { AppComponent } from './app.component';

import { LoginComponent } from './login/login.component';

@NgModule({

  declarations: [

    AppComponent,

    LoginComponent

  ],

  imports: [

    BrowserModule

  ],

  providers: [],

  bootstrap: [AppComponent] // bootstrapped entry component

})

export class AppModule { }

A Routed entry component -

All router components must be entry components because the component would require you to add in two places.

1.      Router and

2.      EntryComponents

The Angular compiler is so smarter and it is recognizing that this is a router component and it automatically added router components into entry components.

The route definition refers to components by its type i.e.

1.      LoginComponent

2.      DasboardComponent

There are two components one is Login and another one is Dashboard. These components have the

ability to navigate between the login and dashboard views if passed the authentication and authorization of this app.

Example -

const routes: Routes = [

    { path: '', redirectTo: 'home', pathMatch: 'full'},

    { path: 'login', component: LoginComponent },

    { path: 'dashboard ', component: DasboardComponent },

    { path: '\*\*', redirectTo: 'home' }

];

# Why does Angular need entryComponents?

The entry components improve the performance, smallest, fastest and reusable code of your production apps.  
  
For example, if you want to load the smallest, fastest and reusable code in your production apps. These codes contain only the classes that you actually need and it should exclude the components that are never used, whether or not those components are declared in the apps.  
  
As you know, many libraries declare and [export components](https://www.code-sample.com/2018/04/what-are-components-in-angular-6-54-and.html) you will never use in your app. If you do not reference them, the tree shaker drops these libraries and components from the final code package.

@NgModule({

  declarations: [

    AppComponent

  ],

  imports: [BrowserModule],

  providers: [],

  bootstrap: [AppComponent]  // bootstrapped entry component

})

export class AppModule { }

If a component is not in an entry component, the compiler skips compiling for this component

What Is Bootstrapping (bootstrap) in Angular?

The Bootstrap is the **root** AppComponent that Angular creates and inserts into the “**index.html**” host web page.

<body>

  <app-root></app-root>

</body>

**index.html -**

<!doctype html>

<html lang="en">

<head>

  <meta charset="utf-8">

  <title>MyApp</title>

  <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">

  <link rel="icon" type="image/x-icon" href="favicon.ico">

</head>

<body>

  <app-root></app-root>

</body>

</html>

You can put more than one component tree on a host web page, that's not typical. Most of the applications have only one component tree and they bootstrap a single root component and you can call the one root component anything you want but most developers call it AppComponent.

The bootstrapping process creates the components listed in the bootstrap array and inserts each one into the browser (**DOM**).

The Angular Module ([**NgModules**](https://www.code-sample.com/2017/04/angular-4-interview-questions-and.html)) helps us to organize an application into connected blocks of functionality.

The NgModule properties for the minimum “**AppModule**” generated by the CLI which are follow as -

ü  **Declarations** — Use to declare the application components.

ü  **Imports** —Every application must import BrowserModule to run the app in a browser.

ü  **Providers** — There are none to start.

ü  **Bootstrap** — This is a root AppComponent that Angular creates and inserts into the index.html host web page.

**app.module.ts -**

import { BrowserModule } from '@angular/platform-browser';

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

import { AppComponent } from './app.component';

import { LoginComponent } from './login/login.component';

import { SignupComponent } from './signup/signup.component';

@NgModule({

  declarations: [

    AppComponent,

    LoginComponent,

    SignupComponent

  ],

  imports: [

    BrowserModule

  ],

  providers: [],

  bootstrap: [AppComponent]

})

export class AppModule { }

By default Bootstrap file is created in the folder “**src/main.ts**” and “**main.ts**” file is very stable. Once you have set it up, you may never change it again and its looks like -

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

import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';

import { environment } from './environments/environment';

if (environment.production) {

  enableProdMode();

}

platformBrowserDynamic().bootstrapModule(AppModule)

  .catch(err => console.log(err));

**MODULDS**

**What Is Modules (@NgModule decorator)?**

The NgModule is a TypeScript class marked by the @NgModule decorator.

The NgModule is a TypeScript class and work with the @NgModule decorator function and also takes a metadata object that tells Angular how to compile and run module code.

The Angular Module ([**NgModules**](https://www.code-sample.com/2017/04/angular-4-interview-questions-and.html)) helps us to organize an application into connected blocks of functionality.

An angular module represents a core concept and plays a fundamental role in structuring Angular applications.

The NgModule is used to simplify the ways you define and manage the dependencies in your applications and also you can consolidate different components and services into associative blocks of functionality.

Every Angular application should have at least one module and it contains the components, service providers, pipes and other code files whose scope is defined by the containing NgModule.

The purpose of the module is to declare everything you create in Angular and group them together.

Every application has at least one Angular module, the root module that you bootstrap to launch the application. The Angular root module is called AppModule.

The module is a way to organize your dependencies for

1.          Compiler

2.          Dependency Injection

A module can import other modules and can expose its functionality to other modules. The modules can be loaded eagerly when the application starts or lazy loaded asynchronously by the router.

The angular loads a root dynamically because it is bootstrapped in the Angular Module.

An Angular app needs at least one module that serves as the root module.

**Why use multiple NgModules?**

Multiple NgModules provides some potential benefits.

Actually, the modules help you to organize an application into associative blocks of functionality.

First one is organizing an application code. If you are putting around 99 resource files in the default app module and see the happing.

And the second one is - It opens the possibility of lazy loading via the router.

**What Are the Purpose of @NgModule?**

The NgModule is used to simplify the ways you define and manage the dependencies in your applications and also you can consolidate different components and services into cohesive blocks of functionality.

The @NgModule metadata divided into three categories as follows.

1.          Static

2.          Runtime

3.          Composability/Grouping

Static – It is compiler configuration and configured via the declarations array.

Runtime - It is injector configuration and configured via the provider’s array.

Composability/Grouping – Introducing NgModules together and configured via the imports and exports arrays.

The following is an example of specifying a NgModule metadata -

@NgModule({

  // Static, This is the compiler configuration

  declarations: [], //declarations is used for configure the selectors.

  entryComponents: [], //entryComponents is used to generate the host factory.

  //Runtime or injector configuration

  providers: [], // providers is used for runtime injector configuration.

  //Composability and Grouping

  imports: [], // imports used for composing NgModules together.

  exports: [] //A list of declarations components, directives, and pipes classes that an importing module can use.

})

**What Types of NgModules?**

There are four types of NgModules –

1.          Features Module

2.          Routing Module

3.          Service Module

4.          Widget Module

5.          Shared Module

Features Module – The feature modules are NgModules for the purpose of organizing an application code.

Routing Module – The Routing is used to manage routes and also enables navigation from one view to another view as users perform application tasks.

Service Module – The modules that only contain services and providers. It provides utility services such as data access and messaging. The root AppModule is the only module that should import service modules. The HttpClientModule is a good example of a service.

Widget Module - The third party UI component libraries are widget modules.

Shared Module – The shared module allows you to organize your application code. You can put your commonly used components, directives, and pipes into the one module and use whenever required to this module.

**What Is Feature Modules?**

**What Are the Types of Feature Modules?**

The feature modules are modules that goal of organizing an application code. It also helps you partition the app into focused areas when you can do everything within the root module.

There are five types of feature modules which are the following-

1.          Domain Feature Modules

2.          Routed Feature Modules

3.          Routing Modules

4.          Service Feature Modules

5.          Widget Feature Modules

Routed Feature Module - Routed feature modules are domain feature modules that components targets of router navigation routes.

A lazy-loaded routed feature module should not be imported by any module.

Routed feature modules do not export anything because their components never appear in the template of an external component.

Routing Module - A routing module provides routing configuration for another module and the routing module focus on the following.

1.          Defines Routes

2.          Adds Router Configuration to the module's imports

3.          Adds service providers to the module's providers

4.          A routing module doesn’t have its own declarations. The components, directives, and pipes are the responsibility of the feature module and not the routing module.

A routing module should only be imported by its companion module.

Service Feature Module - Service modules provide utility services and used to communicate with the server. The HttpClientModule is a great example of a service module.

The root AppModule is the single module that should import service modules.

Domain Feature Module - Domain feature modules deliver a user experience dedicated to a special application domain as like editing a customer and so on.

Widget Feature Module - A widget module makes components, directives, and pipes available to external modules.

The third party UI components and libraries are widget modules.

Import widget modules in any module whose component templates need the widgets.

**Why you use BrowserModule, CommonModule, FormsModule, RouterModule, and HttpClientModule?**

BrowserModule – The browser module is imported from @angular/platform-browser and it is used when you want to run your application in a browser.

CommonModule – The common module is imported from @angular/common and it is used when you want to use directives - NgIf, NgFor and so on.

FormsModule – The forms module is imported from @angular/forms and it is used when you build template driven forms.

RouterModule – The router module is imported from @angular/router and is used for routing RouterLink, forRoot, and forChild.

HttpClientModule –The HttpClientModule is imported from @angular/common/http and it used to initiate HTTP request and responses in angular apps. The HttpClient is more modern and easy to use the alternative of HTTP.

**What are the differences in NgModules and JavaScript Modules?**

NgModules vs. JavaScript Modules -

The NgModule is a TypeScript class decorated with @NgModule Decorator - is a fundamental feature of Angular.

JavaScript also has its own module system for managing collections of JavaScript objects. It is completely different from the NgModule system.

In JavaScript, each file is a module and all objects defined in the file belong to that module. The module declares some objects to be public by marking them with the export keyword.

Other JavaScript modules use import statements to access public objects from other modules.

The following is an example of specifying an export and import statements -

export class AppComponent {

   //...

}

After export your class, you can import that file code in another file.

import { AppComponent } from './app.component';

#### [Angular 5 Directives and Decorators](javascript:void(0);)

**What Are Angular Directives?**

Angular Directive is a TypeScript class which is declared as a [@directive decorator](https://www.code-sample.com/2018/05/angular-5-6-7-directives-decorator.html). The directives allow you to attach behavior to DOM elements and the @directive decorator provide you an additional metadata that determines how directives should be processed, instantiated, and used at run-time.

**What Are decorators?**

The Decorators are functions that modify JavaScript classes and it also used for attaching metadata to classes.

Directive decorator and metadata Properties -

@Directive({

    selector?: string

    inputs?: string[]

    outputs?: string[]

    host?: {...}

    providers?: Provider[]

    exportAs?: string

    queries?: {...}

})

**Selector**– It is a CSS selector that tells Angular to create an instance of this component wherever it finds the corresponding tag in template HTML.

For example, it is – <app-login></app-login>

CSS selector also triggers the instantiation of a directive.

The selector may be declared by element name, class name, attribute name, and attribute name & value.

Suppose we have a directive with an *<input type="checkbox">* selector and the HTML looks like this.

<form>

 <label>Name -</label> <input type="text">

 <label>Are you agree? </label> <input type="checkbox">

<form>

The directive will only be instantiated on the *<input type="checkbox">* element.

**Inputs**– The list of class property names to data-bind as component inputs

**Outputs**- The list of class property names that expose output events that others can subscribe too

**Host**– These properties use to map the class property to host element bindings for properties, events, actions, and attributes.

The host looks like this.

@Directive({

  selector: 'button',

  host: {'(click)': 'onClick($event.target)'}

})

**Providers**- list of providers available to this component and its children

**Queries**– To configure queries that can be injected into the component

**We have 3 types of Directives in Angular** -

1.      Component

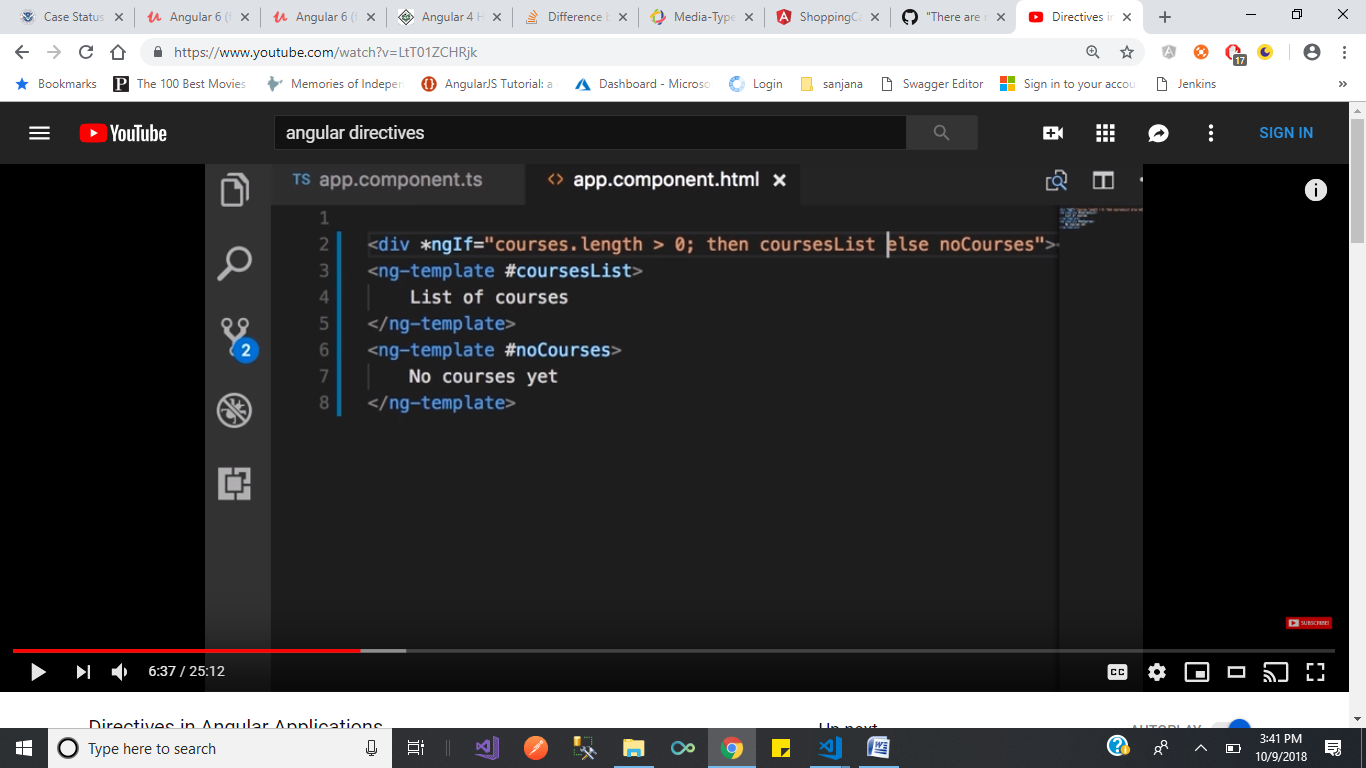
2.      Attribute Directives

3.      Structural Directives

**Components** - The [component is a directive](https://www.code-sample.com/2018/04/what-are-components-in-angular-6-54-and.html) with their own templates and it is responsible for how a component should be processed, instantiated and used at run-time.

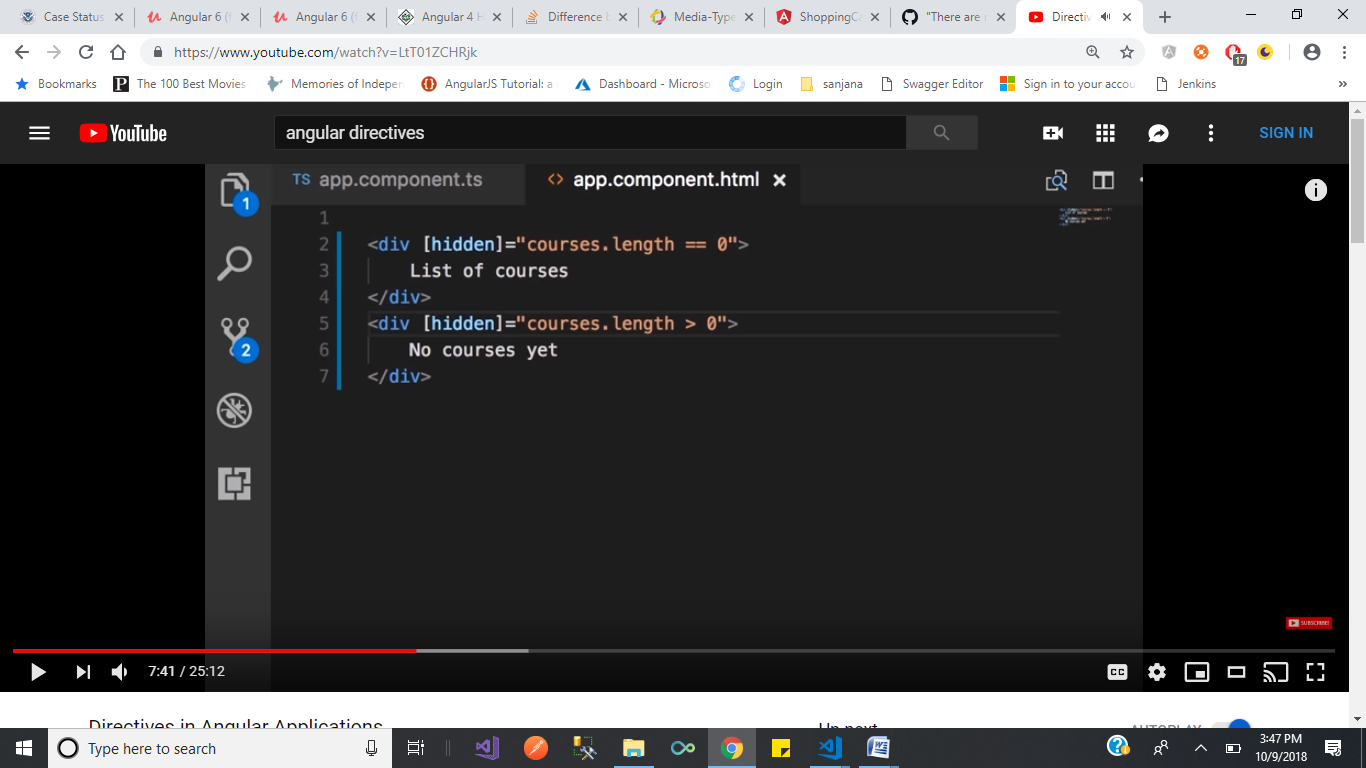
**Structural Directives** - The structural directive is a directive and it is responsible for change the DOM layout by adding, removing, and manipulating elements.

The most of the common built-in structural directives are NgIf, NgFor, and NgSwitch.



As shown in above pic , Based on the condition it displays only one template in DOM when we inspect the code we can see that.

In other way we can do this by using hidden property.like below



But if we use hidden property we can see both the(div’s) conditions in the DOM

Difference between ngif and hidden.

<div \*ngIf="expression">{{val}}</div>

<div [hidden]="!expression">{{val}}</div>

[ngIf is a structural directive](https://angular.io/docs/ts/latest/guide/structural-directives.html), it creates/destroys content inside the DOM. The second statement just hides/shows the content with css, i.e. adding/removing display:none to the element's style.

### What are structural directives?

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

In the first case if expression is false then div and it's content won't be created. In the second case div and content are always created but they are not visible if the expression is false.

in Angular, we can build rich applications some of the components may use too many resources. And even though the component is hidden, the component will be attached to its DOM element. It will keep on listening for events. Angular keep on checking for changes related to data bindings. The component behavior still exists even though it is hidden.

The component and its children components will be tie up resources. Memory burden might be high which results in poor performance, responsiveness can degrade and the user has no knowledge, why the application is slow.

So it’s better to add or remove elements to the component element to the HTML DOM rather than hiding or showing them.

if the component is simple it is better to hide or show it, because component reinitialization operation could be expensive.

**Attribute Directives** - The Attribute directive is a directive and it is responsible for change the behavior of a specified element or component.

**What are the differences between @Component and @Directive?**

The components are used, when you want to create new elements in the DOM with their own HTML template.

The attribute directives are used, when you want to change or update the existing elements in the DOM.

**How to Create Custom Directives?**

Let's start to create a simple directive.

I assuming you have installed the Angular CLI and all the necessary configurations are running in your app. Now, go to your project directory and execute the below CLI command for creating your custom directive –

ng g directive myCustom

After execute the above CLI command, created two files in the project - src/app folder

1.      src/app/my-custom.directive.spec.ts

2.      src/app/my-custom.directive.ts

And update files reference automatically in your project module – “src/app/app.module.ts”

Lest see in the code-sample, how it look like-

my-custom.directive.ts –

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

@Directive({

  selector: '[appMyCustom]'

})

export class MyCustomDirective {

  constructor() { }

}

And app.module.ts –

import { MyCustomDirective } from './my-custom.directive'

//AppModule class with @NgModule decorator

@NgModule({

  //Static, this is the compiler configuration

  //declarations is used for configure the selectors.

  declarations: [

    AppComponent,

    MyCustomDirective,

  ],

  //Composability and Grouping

  //imports used for composing NgModules together.

  imports: [

    BrowserModule

  ],

  //Runtime or injector configuration

  //providers is used for runtime injector configuration.

  providers: [],

  //bootstrapped entry component

  bootstrap: [AppComponent]

})

export class AppModule { }

### 1. What’s new in Angular 5?

Certain tools are optimized in the new version of [Angular](https://www.greycampus.com/angularjs-training-instructor-led), let us see what the tools are:

* Angular 5 supports Typescript version 2.4
* Angular 5 supports RxJS 5.5 which has new features like Pipeable Operators
* A build tool to make the js bundles (files) lighter
* Ahead of Time (AOT) is updated to be on by default
* Events like ActivationStart and ActivationEnd are introduced in Router

### 2. Name the building blocks of Angular.

The Angular application is made using the following:

Modules

Component

Template

Directives

Data Binding

Services

Dependency Injection

Routing

### 3. What is Transpiling in Angular?

Transpiling is the process of converting the typescript into javascript (using Traceur, a JS compiler). Though typescript is used to write code in the Angular applications, the code is internally transpiled into javascript.

### 4. Which of the Angular life cycle component execution happens when a data-bound input value updates?

ngOnChanges is the life cycle hook that gets executed whenever a change happens to the data that was bound to an input.

### 5. Differentiate between Components and Directives in Angular 5.

Components break up the application into smaller parts; whereas, Directives add behavior to an existing DOM element.

### 6. What is the use of @Input and @Output?

When it comes to the communication of Angular Components, which are in Parent-Child Relationship; we use @Input in Child Component when we are passing data from Parent to Child Component and @Output is used in Child Component to receive an event from Child to Parent Component.

### 7. What is ng-content Directive?

The HTML elements like p (paragraph) or h1 (heading) have some content between the tags. For example, <p>this is a paragraph</p> and <h1>this is a heading</h1>. Now, similar to this, what if we want to have some custom text or content between the angular tags like  <app-tax>some tax-related content</app-tax> This will not work the way it worked for HTML elements.  Now, in such cases, the <ng-content> tag directive is used.

### 8. What does a router.navigate do?

When we want to route to a component we use router.navigate.  Syntax: this.router.navigate([‘/component\_name’]);

### 9. What is ViewEncapsulation?

In Angular, the style applies to the component whereever the component is declared. In order to share the style across all the components by using **ViewEncapsulation**.

Declare styles in app.component.css

Add the following code in app.component.ts

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

@Component({

encapsulation: ViewEncapsulation.None

})

ViewEncapsulation decides whether the styles defined in a component can affect the entire application or not. There are three ways to do this in Angular:

Emulated: styles from other HTML spread to the component.

Native: styles from other HTML do not spread to the component.

None: styles defined in a component are visible to all components.

### 10. What are Services in Angular and what command is used to create a service?

Services help us in not repeating the code. With the creation of services, we can use the same code from different components. Here is the command to create a service in angular, ng g service User (a UserService is created when this command is used).

### 11. What is Dependency Injection in Angular 4?

When a component is dependent on another component the dependency is injected/provided during runtime.

### 12. What is Routing in Angular 5?

Routing helps a user in navigating to different pages using links.

### 13. How to handle Events in Angular 5?

Any activity (button click, mouse click, mouse hover, mouse move, etc) of a user on a frontend/web screen is termed as an event. Such events are passed from the view (.HTML) page to a typescript component (.ts).

### 14. What is a RouterOutlet?

RouterOutlet is a substitution for templates rendering the components. In other words, it represents or renders the components(ts) on a template(html) at a particular location.

### 15. Explain the usage of {{}}?

The set of brackets {{}} when used with an HTML tag, represent data from a component. For example, on a HTML page which has <h1>{{variableName}}</h1>, here the ‘variableName’ is actually typescript (component) data representing its value on the template; i.e., HTML. This entire concept is called String Interpolation.

### 16. In how many ways the Data Binding can be done?

Data Binding happens between the HTML (template) and typescript (component). Data binding can be done in 3 ways:

(i) Property Binding (ii) Event Binding (iii) Two-Way Data Binding.

### 17. What is the sequence of Angular Lifecycle Hooks?

OnChange()  -  OnInit()  -  DoCheck()  -  AfterContentInit()  -  AfterContentChecked()  -  AfterViewInit()  -  AfterViewChecked()  -  OnDestroy().

### 18. What is the purpose of using package.json in the angular project?

With the existence of package.json, it will be easy to manage the dependencies of the project. If we are using typescript in the angular project then we can mention the typescript package and version of typescript in package.json.

### 19. How is SPA (Single Page Application) technology different from the traditional web technology?

In traditional web technology, the client requests for a web page (HTML/JSP/asp) and the server sends the resource (or HTML page), and the client again requests for another page and the server responds with another resource. The problem here is a lot of time is consumed in the requesting/responding or due to a lot of reloading. Whereas, in the SPA technology, we maintain only one page (index.HTML) even though the URL keeps on changing.

### 20. What is Component in Angular Terminology?

A web page in Angular has many components involved in it. A Component is basically a block in which the data can be displayed on HTML using some logic usually written in typescript.

### 21. What are ngModel and how do we represent it?

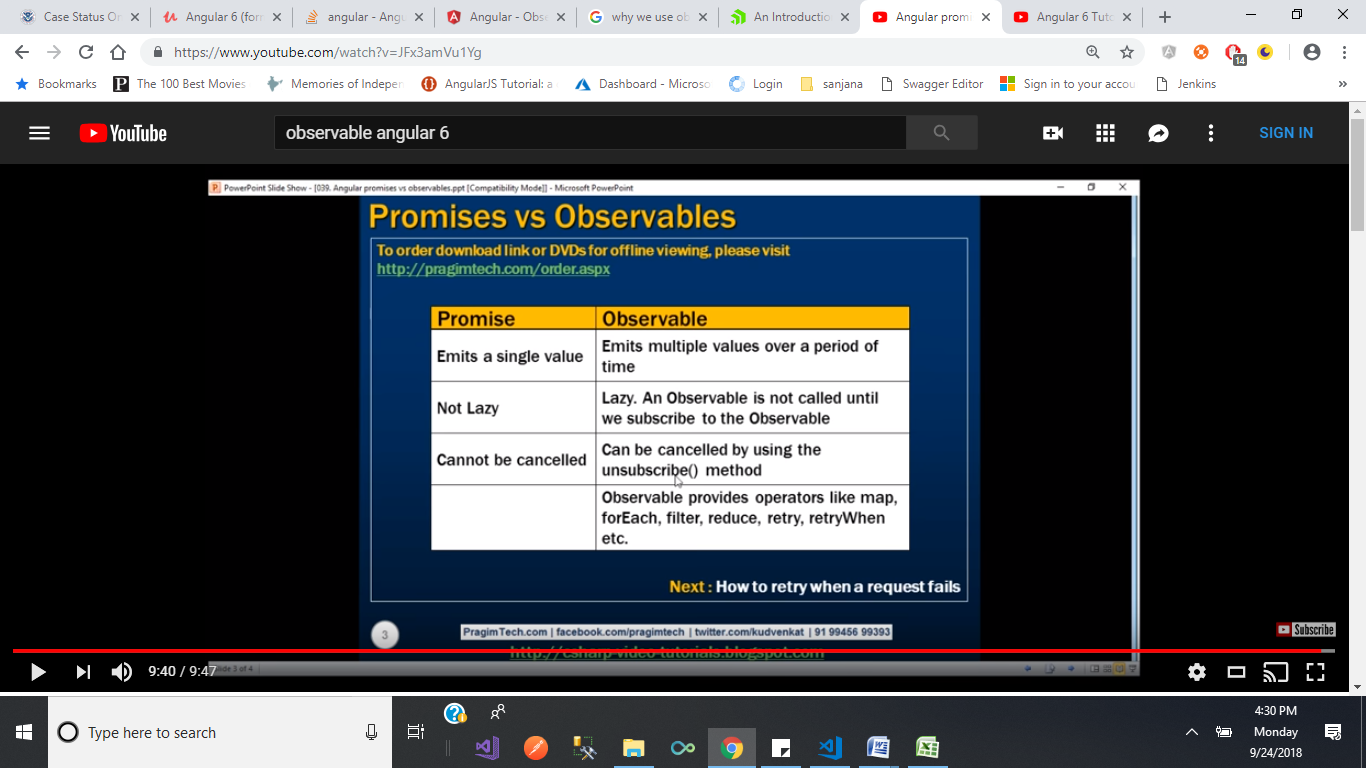
ngModel is a directive which can be applied on a text field. This a two-way data binding. ngModel is represented by [()]

### 22. What does a Subscribe method do in Angular 4?

It is a method which is subscribed to an observable. Whenever the subscribe method is called, an independent execution of the observable happens.

### 23. Differentiate between Observables and Promises.

Observables are lazy, which means nothing happens until a subscription is made. Whereas Promises are eager; which means as soon as a promise is created, the execution takes place. Observable is a stream in which passing of zero or more events is possible and the callback is called for each event. Whereas, promise handles a single event.



### 24. What is an AsyncPipe in Angular?

When an observable or promise returns something, we use a temporary property to hold the content. Later, we bind the same content to the template. With the usage of AsyncPipe, the promise or observable can be directly used in a template and a temporary property is not required.

### 25. Explain Authentication and Authorization.

Authentication: The user login credentials are passed to an authenticate API (on the server). On the server side validation of the credentials happens and a JSON Web Token (JWT) is returned. JWT is a JSON object that has some information or attributes about the current user.  Once the JWT is given to the client, the client or the user will be identified with that JWT.

Authorization: After logging in successfully, the authenticated or genuine user does not have access to everything. The user is not authorized to access someone else’s data,  he/she is authorized to access some data.

### 26. What is AOT Compilation?

Every angular application gets compiled internally. The angular compiler takes javascript code, compiles it and produces javascript code again. Ahead-of-Time Compilation does not happen every time or for every user, as is the case with Just-In-Time (JIT) Compilation.

### 27. What is Redux?

It is a library which helps us maintain the state of the application. Redux is not required in applications that are simple with the simple data flow, it is used in Single Page Applications that have complex data flow.

### 28. What are Pipes?

This feature is used to change the output on the template; something like changing the string into uppercase and displaying it on the template. It can also change Date format.

### 30. Why Typescript with Angular?

Typescript is a superset of Javascript. Earlier, Javascript was the only client side language supported by all browsers. But, the problem with Javascript is, it is not a pure Object Oriented Programming Language. The code written in JS without following patterns like Prototype Pattern, becomes messy and finally leading to difficulties in maintainability and reusability. Instead of learning concepts (like patterns) to maintain code, programmers prefer to maintain the code in a OOP approach and is made avilable with a programming language like Typescript was thus developed by Microsoft in a way that it can work as Javascript and also offer what javascript cannot ie;

* pure OOPS as Typescript offers concepts like Generics, Interfaces and Types (a Static Typed Language) which makes it is easier to catch incorrect data types passing to variables.
* TS provides flexibility to programmers experienced in java, .net as it offers encapsulation through classes and interfaces.
* JS version ES5 offers features like Constructor Function, Dynamic Types, Prototypes. The next version of Javascript ie ES6 introduced new feature like Class keyword but not supported by many browsers.
* TS offers Arrow Functions (=>) which is an ES6 feature not supported by many browsers directly but when used in TS, gets compiled into JS ES5 and runs in any browser.
* TS is not the only alternative to JS, we have CoffeScript, Dart(Google).
* Finally, it is like, TS makes life easier when compared to JS.

What are @HostBinding() and @HostListener() in Angular?

#### While creating custom directive we use these

In html of recipes

<div class="btn-group" appDropdown>

<button type="button"

class="btn btn-primary dropdown-toggle">

Manage Recipe <span class="caret"></span>

</button>

<ul class="dropdown-menu">

<li><a (click)= "onAddToShopingList()" style="cursor: pointer" >Add to Shoppinglist</a></li>

<li><a style="cursor: pointer" (click) ="onEditRecipe()" >Edit Recipe</a></li>

<li><a style="cursor: pointer"(click) ="onDeleteRecipe()">Delete Recipe</a></li>

</ul>

</div>

#### @HostListener() decorator

In Angular, the @HostListener() function decorator allows you to handle events of the host element in the directive class.

To understand **@HostListener()** in a better way, consider another simple scenario: on the click of the host element, you want to show dropdown. To do this in the directive class, add @HostListener() and pass the event ‘click’ to it. Also, associate a function to raise an alert as shown in the listing below:

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

@Directive({

selector: '[appDropdown]'

})

export class DropdownDirective {

@HostBinding('class.open') isopen = false;

@HostListener('click') toggleOpen() {

this.isopen = !this.isopen;

}

constructor() { }

}

**@HostBinding() decorator**

In Angular, the @HostBinding() function decorator allows us to set the properties of the host element from the directive class.

The @HostBinding() decorator takes one parameter, the name of the host element property which value we want to assign in the directive.

**Observable and Promises?**

### Observables in Angular

If you start using Angular you will probably encounter observables when setting up your HTTP requests. So let’s start there.

|  |
| --- |
| import { Observable } from "rxjs/Rx" |
|  | import { Injectable } from "@angular/core" |
|  | import { Http, Response } from "@angular/http" |
|  |  |
|  | @Injectable() |
|  | export class HttpClient { |
|  |  |
|  | constructor( |
|  | public http: Http |
|  | ) {} |
|  |  |
|  | public fetchUsers() { |
|  | return this.http.get("/api/users").map((res: Response) => res.json()) |
|  | } |
|  | } |

We have now created a simple HttpClient with a fetchUsers method that returns an observable. We probably like to display the users in some sort of list, so let’s do something with this method. Since this method returns an observable we have to subscribe to it. In Angular we can subscribe to an observable in two ways:

**Manner 1:**  
We subscribe to an observable in our template using the async pipe. The benefit of this is that Angular deals with your subscription during the lifecycle of a component. Angular will automatically subscribe and unsubscribe for you. Don’t forget to import the “CommonModule” into your module, as the async pipe will be exposed from that.

|  |
| --- |
| import { Component } from "@angular/core" |
|  | import { Observable } from "rxjs/Rx" |
|  |  |
|  | // client |
|  | import { HttpClient } from "../services/client" |
|  |  |
|  | // interface |
|  | import { IUser } from "../services/interfaces" |
|  |  |
|  | @Component({ |
|  | selector: "user-list", |
|  | templateUrl: "./template.html", |
|  | }) |
|  | export class UserList { |
|  |  |
|  | public users$: Observable<IUser[]> |
|  |  |
|  | constructor( |
|  | public client: HttpClient, |
|  | ) {} |
|  |  |
|  | // do a call to fetch the users on init of component |
|  | // the fetchUsers method returns an observable |
|  | // which we assign to the users$ property of our class |
|  | public ngOnInit() { |
|  | this.users$ = this.client.fetchUsers() |
|  | } |
|  | } |

<!-- We use the async pipe to automatically subscribe/unsubscribe to our observable -->

<ul class="user\_\_list" \*ngIf="(users$ | async).length">

<li class="user" \*ngFor="let user of users$ | async">

{{ user.name }} - {{ user.birth\_date }}

</li>

</ul>

Note: Please note the dollar sign. Using the dollar sign in the name of a variable that is an observable, is considered best practice. This way it’s easy to identify if your variable is an observable or not.

**Manner 2:**  
We subscribe to the observable ourselves using the actual subscribe()method. This can be handy if you would first like to do something with the data before displaying it. The downside is that you have to manage the subscription yourself.

component

|  |
| --- |
| import { Component } from "@angular/core" |
|  |  | |
|  | // client | |
|  | import { HttpClient } from "../services/client" | |
|  |  | |
|  | // interface | |
|  | import { IUser } from "../services/interfaces" | |
|  |  | |
|  | @Component({ | |
|  | selector: "user-list", | |
|  | templateUrl: "./template.html", | |
|  | }) | |
|  | export class UserList { | |
|  |  | |
|  | public users: IUser[] | |
|  |  | |
|  | constructor( | |
|  | public client: HttpClient, | |
|  | ) {} | |
|  |  | |
|  | // do a call to fetch the users on init of component | |
|  | // we manually subscribe to this method and take the users | |
|  | // in our callback | |
|  | public ngOnInit() { | |
|  | this.client.fetchUsers().subscribe((users: IUser[]) => { | |
|  |  | |
|  | // do stuff with our data here. | |
|  | // .... | |
|  |  | |
|  | // asign data to our class property in the end | |
|  | // so it will be available to our template | |
|  | this.users = users | |
|  | }) | |
|  | } | |
|  | } | |
| Template:  <ul class="user\_\_list" \*ngIf="users.length"> | |
|  | | <li class="user" \*ngFor="let user of users"> |
|  | | {{ user.name }} - {{ user.birth\_date }} |
|  | | </li> |
|  | | </ul> |

[**view raw**](https://gist.github.com/luukgruijs/35abddbf81f8f726ebb06adc25938b3e/raw/6a3b87fc2b53d9eea32d17e72af930de71ec9e28/template.html)[**template.html**](https://gist.github.com/luukgruijs/35abddbf81f8f726ebb06adc25938b3e#file-template-html) hosted with ❤ by [**GitHub**](https://github.com/)

As you can see the template logic is quite similar, the component logic can actually become much different en more complex if you go for manner 2. In general i would recommend to choose manner 1. As this is the most easy and you don’t have to manually manage your subscriptions. Keeping your subscriptions open while not using them is a memory leak and therefore not good.

### Creating an observable yourself

Now that you know how to deal with common observables that are given to you by Angular, it’s good to know how you create an observable yourself. The simplest version looks like this:

|  |
| --- |
| import { Observable } from "rxjs/Observable" |
|  |  |
|  | // 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() |

As you can see in the example observables are **created** by using the new Observable() call, then **subscribed** to by an observer, **executed** by calling the next() and **disposed** by calling unsubscribe().

**Creating observables**Creating observables is easy, just call the new Observable() and pass along one argument which represents the observer. Therefore i usually call it “observer” as well.

**Subscribing to observables**Remember, observables are lazy. If you don’t subscribe nothing is going to happen. It’s good to know that when you subscribe to an observer, each call of subscribe() will trigger it’s own independent execution for that given observer. Subscribe calls are not shared among multiple subscribers to the same observable.

**Executing observables**The code inside an observables represents the execution of the observables. On the parameter that was given when creating the observable 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.

Calls of the next are the most common as they actually deliver the data to it’s subscribers. During observable execution there can be an infinite calls to the observer.next(), however when observer.error() or observer.complete()is called, the execution stops and no more data will be delivered to the subscribers.

**Disposing observables**Because observable execution can run for an infinite amount of time, we need a way to stop it from executing. Since each execution is run for every subscriber it’s important to not keep subscriptions open for subscribers that don’t need data anymore, as that would mean a waste of memory and computing power.

When you subscribe to an observable, you get back a subscription, which represents the ongoing execution. Just call unsubscribe()to cancel the execution.

**Angular constructor versus ngOnInit**

The constructor method on Type Script is a feature of a class itself, rather than an Angular feature.

The constructor method is not actually an Angular 2 method. It is a predefined method in a TypeScript class which is called when the class is instantiated. The constructor’s purpose is to help prepare the creation of a new instance of the class. In the context of Angular 2 it can be used to properly initialize fields. Angular 2’s DI (dependency injection) also tries to find providers that match the types of the constructor’s parameters, resolves them, and passes them to the constructor as arguments.

The ngOnInit method runs after the constructor method, meaning that all of the injected dependencies will be resolved and all of the class members will be defined. This makes it the perfect place to do any of the initialization work/logic for the component.

**Difference of promise and Observable**

Promise

1.Promises work with asynchronous operations and they either return us a single value (i.e. the promise resolves) or an error message (i.e. the promise rejects).

2.  a request initiated from a promise is not cancellable

3. Since we can't cancel a promise, an HTTP request that does a search for example on keyup would be executed as many times as we press the key.

Here's a very simple example demonstrating that:

<!-- app.component.html -->

<div>

<h2>Star Wars Character Search</h2>

<input #term type="text" (keyup)="search(term.value)">

<hr>

<div \*ngIf="results.length > 0">

<li \*ngFor="let result of results">{{ result.name }} is a character with a height of {{ result.height }}.</li>

</div>

</div>

// app.component.ts

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

import { HttpClient } from '@angular/common/http';

@Component({

selector: 'app-root',

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

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

})

export class AppComponent {

private results = [];

constructor(private http: HttpClient) { }

private search(term) {

console.log(term);

this.http.get(`https://swapi.co/api/people/?search=${term}`).toPromise()

.then((data: any) => {

/\* tslint:disable:no-console \*/

console.time('request-length');

console.log(data);

console.timeEnd('request-length');

this.results = data.results;

});

}

}

The console statements are left in there deliberately. This is how the browser console would look like:

[Log] l

[Log] lu

[Log] luk

[Log] luke

[Log] {count: 2, next: null, previous: null, results: Array}

[Debug] request-length: 0.590ms

[Log] {count: 37, next: "https://swapi.co/api/people/?page=2&search=l", previous: null, results: Array}

[Debug] request-length: 0.281ms

[Log] {count: 1, next: null, previous: null, results: Array}

[Debug] request-length: 0.377ms

[Log] {count: 1, next: null, previous: null, results: Array}

[Debug] request-length: 0.263ms

Notice, for each keypress, there's a request going out - so for the search term 'luke' we have excatly 4 requests. This is really not ideal.

Observable:

An observable is essentially a stream (a stream of events, or data) and compared to a Promise, an Observable can be cancelled. It out of the box supports operators such as map() and filter().

For the above example we can do it through observables. By using this observables it sends only one outgoing request instead of four.

Observables are often compared to promises. Here are some key differences:

* Observables are declarative; computation does not start until subscription. Promises execute immediately on creation. So we can subscribe observables, whenever we need the result.
* Observables provide many values. Promises provide one. This makes observables useful for getting multiple values over time.
* Observables differentiate between chaining and subscription. Promises only have .then() clauses. This makes observables useful for creating complex transformation recipes to be used by other part of the system, without causing the work to be executed.
* Observables subscribe() is responsible for handling errors. Promises push errors to the child promises. This makes observables useful for centralized and predictable error handling.

#### How to cache an observable data in Angular 2?

Caching of an observable data is done with the help of “observable.cache”. We can use caching in order to cache the response in the memory and then, on the next subscription, instead of requesting the remote server again. This operator is used at the end of the string. Caching is important for the performance, especially on bandwidth restricted devices and slow networks. You should have a good understanding of caching while working with promises but while translating it to observable, it is a bit difficult. Therefore, when interacting with observables, we typically set up a subscription on the consumer side and react to values coming through the pipe. We can easily add caching to the observables by adding publishReplay(1) and refCount.

**What is shared module?**

 You can put commonly used directives, pipes, and components into one module and then import just that module wherever you need it in other parts of your app.

Shared.module.ts

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

import { DropdownDirective } from './dropdown.directive.ts.directive';

import { CommonModule } from '@angular/common';

@NgModule({

declarations: [

DropdownDirective

],

exports: [

CommonModule,

DropdownDirective

]

})

export class Sharedmodule {

}

**In recipes.module.ts**

@NgModule({

declarations: [

RecipeEditComponent,

RecipesComponent,

RecipeItemComponent,

RecipesListComponent,

RecipeStartComponent,

RecipesDetailComponent,

RecipesListDirective,

ShortenPipe,

RecipesearchPipe

],

imports: [

CommonModule, // it gives access to the common directives(ngclass,ngif , ngfor...) in the application

ReactiveFormsModule, // we used this at only in recipes forms

RecipesRoutingModule,

Sharedmodule

]

**Template variable? How is it used?**

You declare a reference variable by using the hash symbol (#). The **#firstNameInput** declares a **firstNameInput**variable on an <input>element.

<input type="text" #firstNameInput>

<input type="text" #lastNameInput>

After that you can access the variable anywhere inside the template. For example, I pass the variable as an parameter on an event.

<button (click)="show(lastNameInput)">Show</button>

Remember that the lastNameInput belongs to HTMLInputElement type.

show(lastName: HTMLInputElement){  
 console.log(lastName.value);  
}

Usually, the reference variable can only be accessed inside the template. However, you can use @**ViewChild**decorator to reference it inside your component.

import {ViewChild, ElementRef} from '[@angular/core](http://twitter.com/angular/core)';

// Reference firstNameInput variable inside Component  
[@ViewChild](http://twitter.com/ViewChild)('firstNameInput') nameInputRef: ElementRef;

After that, you can use **this.nameInputRef** anywhere inside your Component.

show(lastName: HTMLInputElement){  
 this.fullName = this.nameInputRef.nativeElement.value + ' ' + lastName.value;  
}

#### Explain the concept of lazy loading in Angular .

Lazy loading is a module which is used to decrease the start-up time. When lazy is used, then our system application does not need to load everything at once. It only needs to load what the user expects to see when the application first loads. The modules which are lazily loaded will only be loaded when the user navigates to their routes. Lazy loading improves the performance of our system applications. It keeps the initial payload small and these smaller payloads lead to faster download speeds. It helps in lowering the resource cost especially on mobile networks. If a user doesn’t visit a section of the application, they won’t ever download those resources. Lazy loading is setup in the main routing file. Lazy loading overcomes the problem of slow loading of applications in their own way which hence improves the loading time of the application.

Lazy loading can be done only in four steps: –

1. Update your route file
2. Install angular-router-loader and add the loader to your webpack configuration file.
3. Define the lazy routes
4. Import the routes to the module.

In recipes-routing.module.ts file

const recipesRoutes: Routes = [

// {path: 'recipes' , component: RecipesComponent , children: [

{path: '' , component: RecipesComponent , children: [

{path: '', component: RecipeStartComponent},

{path: 'new', component: RecipeEditComponent, canActivate: [AuthGuard]},

{path: ':id', component: RecipesDetailComponent},

{path: ':id/edit', component: RecipeEditComponent, canActivate: [AuthGuard]}

]

},

];

@NgModule({

imports: [

RouterModule.forChild(recipesRoutes)

],

exports: [RouterModule]

})

Instead of specifiy the recipes module in app.module (what ever we import the modules in appmodule file those modules bind at the time of loading the website). If we place all the modules in app.module then it increase the page loading time.

In app-routing.module

const approuts: Routes = [

/\*\* what ever the paths we want to mention in the page we specify here \*/

// {path: '', redirectTo: '/recipes', pathMatch: 'full'},

{path: '', component: HomeComponent},

{path: 'recipes', loadChildren: () => RecipesModule},

// {path: 'recipes', loadChildren: './recipes/recipes.module#RecipesModule'},

// for lazy loding of recipes we commented recipes module in app.module and added the path above

{path: 'shopping-list' , component: ShoppingListComponent },

{path: 'forms-demo' , component: FormsDemoComponent },

{path: 'pipes' , component: FiltersComponent }

];

@NgModule({

imports: [RouterModule.forRoot(approuts)], /\*\*here we configure the routes with routermodule \*/

exports: [RouterModule] /\*\* we have to export this Router module to main module 'app.module' \*/

})

export class AppRoutingModule {

in app.module.ts

imports: [

BrowserModule, // here we dont have a commonmodule. since this browsermodule contains all the featers of

// commonmodule and some extra featers to need to start the application.

// so we use browser module here and commonmodule in othermoduls.

// RecipesModule, // for eager loding we have to write here.but for lazyloding we have to mention in aap.routingmodule

ShoppingListModule,

FormsModule,

HttpModule, // to add httpclient we commented this(to get the response or to send the request to backend server)

HttpClientModule,

AppRoutingModule,

Sharedmodule,

AuthModule,

CoreModule

],

#### What is the meaning of component life cycle in Angular ?

Angular manages the life cycle of a component. Angular creates it, renders it. It can also create and render its children. It also checks when its data bound properties change. It can even destroy it before removing it from the DOM.

1.Constructor(){}

2.ngOnChanges-- Invoked **every** time there is a change in one of the input properties of the component.(@input decorator properties receive new value)

3ngOnInit -- Invoked when given component has been initialized.  
This hook is only called **once** after the first ngOnChanges

4.ngDoCheck -- Invoked when the change detector of the given component is invoked. It allows us to implement our own change detection algorithm for the given component.

#### Important

ngDoCheck and ngOnChanges should not be implemented together on the same component.

Below four for child components except ngondestroy

**ngAfterContentInit**

Invoked after Angular performs any content projection into the components view.

What is content projection?

In Angular, content projection is used to project content in a component. Content projection allows us to insert a shadow DOM in our component. To put it simply, if you want to insert HTML elements or other components in a component, then you do that using concept of content projection. In Angular, we achieve content projection using **< ng-content >< /ng-content >.**We can make reusable components and scalable application by right use of content projection.

Ex:

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

@Component({

    selector: 'greet',

    template: `{{message}}`

})

export class GreetComponent {

    @Input() message: string;

}

Using the @Input() decorator, you can pass a simple string to the GreetComponnet, but what if you need to pass different types of data to the GreetComponent such as:

1. Inner HTML
2. HTML Elements
3. Styled HTML
4. Another Component etc.

To pass or project styled HTML or another component, content projection is used. Let us modify the GreetComponent to use content projection in this code:





**ngAfterContentChecked**

Invoked each time the content of the given component has been checked by the change detection mechanism of Angular.

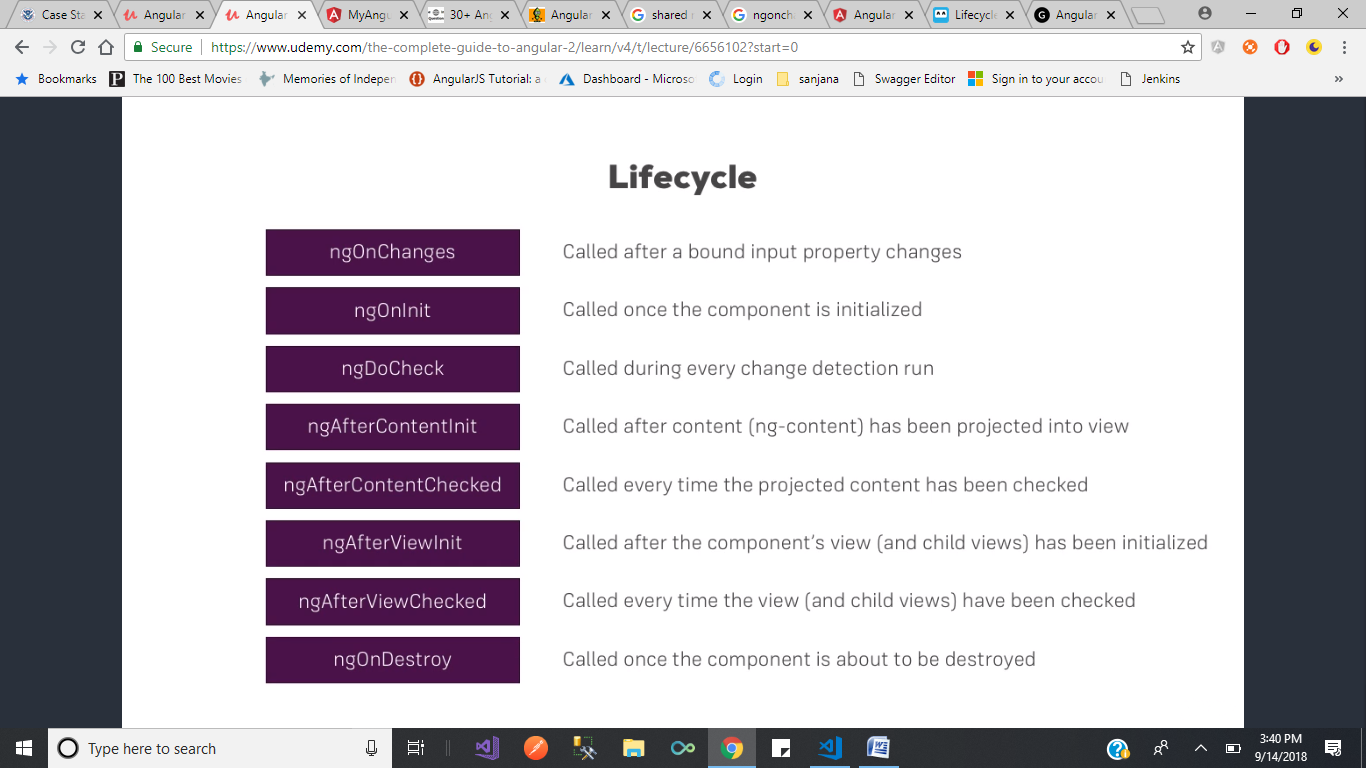
**ngAfterViewInit**

Invoked when the component’s view has been fully initialized.

**ngAfterViewChecked**

Invoked each time the view of the given component has been checked by the change detection mechanism of Angular.

5. ngOnDestroy -- This method will be invoked just before Angular destroys the component.  
Use this hook to unsubscribe observables and detach event handlers to avoid memory leaks.



What are <ng-template> <ng-content> <ng-container>? why we use those?

###### <ng-container>

<ng-container> is an Angular grouping element that is similar to <ng-template> in that it doesn’t represent a DOM element. The difference is that it will always be rendered, whereas an <ng-template> will only be rendered if it is explicitly requested. <ng-container>s are useful anywhere you need an extra container for some template elements, but don’t want to (or can’t) create a container such as a div to hold them with due to syntax or style constraints.

For example, it is not allowed in Angular to put two structural directives on the same element. If you needed to loop through an array and display a <tr> for each element, but only if a different condition was met, you may want to put both an \*ngFor and \*ngIf on the <tr> element. Angular does not allow this, however, and wrapping the <tr> in a <div> to hold one of the structural directives is not valid HTML. The utility of <ng-container> shines here, where we can use the <ng-container>to hold a structural directive and contain the <tr></span> without breaking the HTML layout.

Example:-

<div>

Pick your favorite hero

(<label><input type="checkbox" checked (change)="showSad = !showSad">show sad</label>)

</div>

<select [([ngModel](https://angular.io/api/forms/NgModel))]="hero">

<span \*[ngFor](https://angular.io/api/common/NgForOf)="let h of heroes">

<span \*[ngIf](https://angular.io/api/common/NgIf)="showSad || h.emotion !== 'sad'">

<[option](https://angular.io/api/forms/NgSelectOption) [[ngValue](https://angular.io/api/forms/NgSelectOption#ngValue)]="h">{{h.name}} ({{h.emotion}})</[option](https://angular.io/api/forms/NgSelectOption)>

</span>

</span>

</select>

the drop down is empty.

spanned options don't work

The browser won't display an <[option](https://angular.io/api/forms/NgSelectOption)> within a <span>.

### <ng-container> to the rescue

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.

Here's the conditional paragraph again, this time using <ng-container>.

Now conditionally exclude a select <[option](https://angular.io/api/forms/NgSelectOption)> with <ng-container>.

src/app/app.component.html (select-ngcontainer)

content\_copy<div>

Pick your favorite hero

(<label><input type="checkbox" checked (change)="showSad = !showSad">show sad</label>)

</div>

<select [([ngModel](https://angular.io/api/forms/NgModel))]="hero">

<ng-container \*[ngFor](https://angular.io/api/common/NgForOf)="let h of heroes">

<ng-container \*[ngIf](https://angular.io/api/common/NgIf)="showSad || h.emotion !== 'sad'">

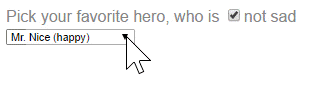
<[option](https://angular.io/api/forms/NgSelectOption) [[ngValue](https://angular.io/api/forms/NgSelectOption#ngValue)]="h">{{h.name}} ({{h.emotion}})</[option](https://angular.io/api/forms/NgSelectOption)>

</ng-container>

</ng-container>

</select>

The drop down works properly.



|  |
| --- |
| <ng-container> |
|  | You can see me! (Without any help from structural directives) |
|  | </ng-container> |
|  |  |
|  | <table> |
|  | <tbody> |
|  | <ng-container \*ngFor="let item of items"> |
|  | <tr \*ngIf="item > 1"> |
|  | <td>{{item}}</td> |
|  | </tr> |
|  | </ng-container> |
|  | </tbody> |
|  | </table> |

<ng-content>

In app.componet.html file , if I specify

<app-header ></app-header>

That means it displays header.component.html page content through app.component.html. because this <app-header></app-header> identifies in header.componet.ts file

In header.component.html contains

<p> header works here </p>

When we run the application it goes to header.component.ts file there we specified

@Component({

selector: 'app-header',

templateUrl: './header.component.html'

})

In template url we gave the path of html file so it goes to header.component.html page. And displays the content.

For suppose if add any extra content in between <app-header></app-header> of app.componet.html like below

<app-header>new content (or) some text </app-header>

Now if we run this application , we cannot see the newly added content in the browser.

To see that we need to add <ng-content></ng-content> in header.componet.html page

NOTE:

When using an asterisk (\*) in our templates, we are informing Angular we’re using a structural directive, which is also sugar syntax (a nice short hand) for using the <ng-template> element.

##### What is AOT compilation?

An Angular application consists mainly of components and their HTML templates. Because the components and templates provided by Angular cannot be understood by the browser directly, Angular applications require a compilation process before they can run in a browser.

The Angular Ahead-of-Time (AOT) compiler converts your Angular HTML and TypeScript code into efficient JavaScript code during the build phase before the browser downloads and runs that code. Compiling your application during the build process provides a faster rendering in the browser.

Angular offers two ways to compile your application:

1. *Just-in-Time* (JIT), which compiles your app in the browser at runtime.
2. *Ahead-of-Time* (AOT), which compiles your app at build time.

JIT compilation is the default when you run the *build-only* or the *build-and-serve-locally* CLI commands:

ng build

ng serve

For AOT compilation, append the --aot flags to the *build-only* or the *build-and-serve-locally* CLI commands:

ng build --aot

ng serve --aot

The ng build command with the --prod meta-flag (ng build --prod) compiles with AOT by default

## Why compile with AOT?

1. Faster rendering

With AOT, the browser downloads a pre-compiled version of the application. The browser loads executable code so it can render the application immediately, without waiting to compile the app first.

1. Fewer asynchronous requests

The compiler inlines external HTML templates and CSS style sheets within the application JavaScript, eliminating separate ajax requests for those source files.

1. Smaller Angular framework download size

There's no need to download the Angular compiler if the app is already compiled. The compiler is roughly half of Angular itself, so omitting it dramatically reduces the application payload.

1. Detect template errors earlier

The AOT compiler detects and reports template binding errors during the build step before users can see them.

1. Better security

AOT compiles HTML templates and components into JavaScript files long before they are served to the client. With no templates to read and no risky client-side HTML or JavaScript evaluation, there are fewer opportunities for injection attacks.

.

##### Q27. What are Event emitters?

An Event emitter is a class defined in core module that can be used by components and directives to emit custom events.

##### Q28. What is Angular @ RouteParams?

The RouteParams are used to map the given URL’s based on the route URLs and they become optional parameters for that route.

##### Q29. Explain Angular 2 hidden property?

The hidden property in Angular 2 is a special case.

* The property is more powerful and is used to bind any property of the elements.
* It is considered the closest cousin of **ngshow** and **nghide**.
* It sets the display property “display: none”.

##### Q30. Why are decorators used in Angular 2?

In Angular 2, decorators are used as an identifier of class or type of the object that is created by the TypeScript.The Angular 2 identifies the class below decorator call as the definition of the class and extends the decorator specific properties with the class definition.

##### Q31. Explain host decorator in Angular 2?

The host decorators in Angular 2 bind the properties of components with UI element values.The properties inside a component class definition which are decorated with @HostBinding are accessed in a template from the assigned property that is @HostBinding()title=’Our title'( whatever the title is).

##### Q32. What are Pipes in Angular 2?

Pipes in Angular 2 are used in templates in order to convert them into a content that is user-friendly and readable one within the interpolation braces that is {{release| date}}, here the symbol  
“|” denotes the pipe.

##### Q33. How can you handle errors in Angular 2 Applications?

The Angular 2 Applications provide with the option of error handling.  
The errors in Angular 2 can be handled by including the ReactJS catch library and later using the catch function.

* The catch function, which is used after adding the catch library contains the link to the Error handler function.
* And in this error, handler function, the errors are sent to the error console, and also the errors are thrown back to continue the execution.
* So, whenever an error occurs it will be redirected to the error console of the web.

##### Q34. Can you automate porting Angular 1 code to Angular 2?

No, currently we are not provided with any such tool that ports the Angular 1 code to the Angular 2 code.  
In the process of porting, the Angular 1 code to Angular 2, the side by side manual conversion of Angular 1 directives to the Angular 2 components takes place because they are two different frameworks and hence requires different approaches to solve the same problem.

35. to bind the data from class(ts file) to template(html)

We use {{}} interpolation

In ts file

Public name=”sanajana”

In html

<h2>Welcome {{name}}</h2>

These things we can do by using interpolation

<h2>{{2+3}}</h2>

<h2>{{“welcome:” + name}}</h2>

<h2>{{name.length}}</h2>

<h2>{{name.touppercase()}}</h2>

<h2>{{greetuser()}}</h2>

Greetuser(){

Return “hello “ +this.name;

}

We cannot use assignment operator in {{}} interpolation <h2>{{s= 2+3}}</h2>// wrong

We canot bind the site address directly in the html template.

<h2>{{window.location.href}}</h2>// wrong

We can do this through class

In class (.ts file)

Public siteurl = window.location.href;

Now we can bind this value to html

<h2>{{siteurl}}</h2>

38. html input value is attribute

The input value in DOM is property

**37. class binding, style binding**

* [**ng-class**](http://docs.angularjs.org/api/ng.directive%3angClass) - use when the set of CSS styles is static/known ahead of time
* [**ng-style**](http://docs.angularjs.org/api/ng.directive%3angStyle) - use when you can't define a CSS class because the style values may change dynamically. Think programmable control of the style values.

class binding: we bind the css class to html

If we have the css file with classes success , danger, and special

Styles: [‘

.text-success {color : green; }

.text-danger { color: red;}

.text-special { font-style: italic;}

’]

<https://www.youtube.com/watch?v=Y6OP-lPJxgs&index=7&list=PLC3y8-rFHvwhBRAgFinJR8KHIrCdTkZcZ>

the [ngStyle] property directly, which allows us to pass objects into it. for example [ngStyle]="{'color':'white', 'font-size':'17px'}" which also allows us to set those styles dynamically.

* for state styles that don't have a pseudo-class selector linked to it, its best to go with ngClass
* if the ngClass expressions get too big, it's a good idea to move the calculation of the styles to the component class
* only for situations where we have a dynamically calculated embedded style should we use ngStyle, this should be rarely needed

**what is local reference and when and why we use that?**

Instead of two-way binding, we can easily fetch a value of any input through**local references in Angular**. Local references can be fetched directly from the component template and into the component typescript class. Lets check out how do we use local references.

First, lets create **#giftname** as a local reference, and pass the created reference as parameter to some javascript function, as in the below snippet.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <div class="col-md-9">  <input type="text" class="form-control" trim #giftName />  </div>  <div class="row">  <button class="btn" (click)="onAddGift(giftName)">Add Gift</button>  </div> |

To retrieve the input field value through local reference in typescript file, we can do it using ***<param>*.value** notation.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | import { Component, OnInit, ViewChild, ElementRef } from '@angular/core';  import { Gift } from '../gift/gift.component';    @Component({  selector: 'app-user',  templateUrl: './user.component.html',  styleUrls: ['./user.component.css']  })  export class UserComponent implements OnInit {  userNickName: String = '';    ..    onAddGift(giftName: HTMLInputElement) {  this.userGiftName **= giftName.value;**  **..**  ..  }    } |

**There is another way to do this using @ViewChild decorator.**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <div class="col-md-9">  <input type="text" class="form-control" trim #nickName />  </div>  <div class="row">  <button class="btn" (click)="onAddGift(nickName)">Add Gift</button>  </div> |

Then in the Typescript file, use the same reference name (#nickName) to do the following.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | import { Component, OnInit, ViewChild, ElementRef } from '@angular/core';  import { Gift } from '../gift/gift.component';    @Component({  selector: 'app-user',  templateUrl: './user.component.html',  styleUrls: ['./user.component.css']  })  export class UserComponent implements OnInit {  @ViewChild('nickName') nickName : ElementRef;  userNickName: String = '';    ..    onAddGift(nickName: HTMLInputElement) {  ..  this.userNickName = nickName.nativeElement.value;  ..  }    } |