**Forms:**

There are two types of forms in angular,

1. Template Driven Form:

The template driven forms are heavily dependent on HTML templates. They are easy to understand and use to create simple forms. It's not easy to do unit tests in template form.

1. Reactive Form ( Model Driven Form ):

The reactive form used to build form via code. It can add form input dynamically and adjust validations on run time based on the decision made in the code. It’s easy to unit test, the most of the form logic is in the component class.

Classes for creating form control tree:

1. FormGroup:

The form group warps a collection of form control, with each control name as key value.

1. FormControl:

The form control is used to get, set values and validate the form control such as input, select tags. To access values,

| employeeForm.controls.fullName.value  employeeFrom.get('fullName').value |
| --- |

employeeForm.setValue({ fullName: 'karthik')

patchValue - used to update a subset of form data.

Nested Form Group:

| Import { FormGroup, FormControl } from '@angular/forms';  Class Employee implements OnInit{  employeeForm: FormGroup //type as form group  ngOnInit(){  this.employeeForm = new FormGroup({  fullName : new FormControl(),  email: new FormControl(),  skills: new FormGroup({  skillName: new FormControl(),  experience: new FormControl()  })  })  } } |
| --- |

| <**form** [formGroup]="employeeForm" (ngSubmit)="onFormSubmit()">  <**input** type="text" formControlName="fullName"/>  <**input** type="email" formControlName="email"/>  <**div** formGroupName="skills">  <**input** type="text" formControlName="skillName"/>  <**input** type="radio" formControlName="experience"/>Beginner <**input** type="radio" formControlName="experience"/>Intermediate <**input** type="radio" formControlName="experience"/>Experience  </**div**>  </form> |
| --- |

Form Builder:

The form builder is a helper class and predefined service, it has 3 methods - group(), control(), array()

Validator:

The validator is a function provided by Validator class. Some predefined validations are,

1. required - It validates a required fields
2. requiredTrue - It commonly used on a required checkbox
3. email - It ensure, a control pattern is a valid email address
4. pattern - It ensure, a field value matches the specified regular expression pattern
5. min - It validates a field value is greater than or equal to
6. max - It validates a field value is less than or equal to
7. minLength - It validates a number of character in a field is greater than or equal to
8. maxLength - It validates a number of character in a field is less than or equal to

Form Control valueChanges (Observable):

The valueChanges emit an event every time the value of a form group and form control is changed.

| this.employeeForm.get('fullName').valueChange.subscribe( (value) => {  console.log(value) }); |
| --- |

Custom Validator

functions, they take as input a FormControl instance and returns either null if it’s valid or an error object if it’s not.

**Form Array:**

The form array holds the array of form group, form control and nested form group.

Form Array Properties:

Touched, Untouched, Valid, Invalid, Dirty, Pristine

Form Array Methods:

1. Push - Insert the control at end of the array
2. Insert - Insert the control at the specific index of the array
3. removeAt - Removes the control at the specific index of the array
4. setControl - Replace an existing control at the specific index
5. At - Return the control at the specific index of the array

Read value from URL:

To read id value from the route url, need to import ActivatedRoute

| this.route.paramMap.subscribe((param) => {  const userId = param.get('id') }) |
| --- |

**Angular Module:**

An angular module is a class, decorated with @ngModule Decorator. It is a mechanism to group components, directive, pipes and services that are a feature area of our application.

Types of Angular Module:

1. Root Module:

All angular application have one module as by default, which is the AppModule

1. Feature Module:

To create Feature Module

| ng g m <moduleName> --routing -m app |
| --- |
|  |

To create Component inside Feature Module

| ng g c <componentName> -m <ModuleName> |
| --- |

1. Core Module:

Module that is only imported once in the AppModule and never again in the other modules

1. Shared Module:

Module which is imported in every feature module that needs some shared components. It re-export other common angular modules (FormsModule, CommonModule, etc)

| ng g m <moduleName> --flat |
| --- |

1. Routing Module

Advantage of splitting application into Modules:

Better code organisation, Code reuse, Code maintenance, Performance

**Module Loading Strategy:**

1. Eager Loading

By default, all modules are eagerly loaded. The root module is always eagerly loaded. This is best for smaller applications. It will load all the module at once

1. Lazy Loading

Modules are loaded on demand ( Asynchronous routing ), when the user navigates to the route in those respective modules. This can significantly reduce the initial load time of our application.

| {  path: 'library',   loadChildren: () => import('./feature-modules/library/library.module').then( module => module.LibraryModule) } |
| --- |

1. Pre Loading or Eager Lazy Loading:

In pre-loading, it will first load the module to be bootstrapped, then load the eager modules as background. Once the application is up and running, it will load in the background.

Types of Pre Loading Strategy:

1. No Pre Loading:

Default strategy in angular, which provides no preloading for any module.

| import { NoPreloading } from '@angular/router';  @NgModule({  imports: [RouterModule.forRoot(routes, { preloadingStrategy: NoPreloading })],  exports: [RouterModule] }) |
| --- |

1. Pre Load All Modules:

Preload all the lazy-loading modules as quickly as possible.

| import { PreloadAllModules } from '@angular/router';  @NgModule({  imports: [RouterModule.forRoot(routes, { preloadingStrategy: PreloadAllModules })],  exports: [RouterModule] }) |
| --- |

1. Custom Pre Loading:

To set pre loading form custom modules. Need to create a custom preloading service.

| import { PreloadingStrategy, Route } from '@angular/router';  import { Observable, of } from 'rxjs';  export class CustomPreloadingStrategy implements PreloadingStrategy {  preload(route: Route, load: Function): Observable<any> {  return route.data && route.data.preload ? load() : of(null);  } } |
| --- |

| {  path: 'login',  loadChildren: () => import('./feature-modules/login/login.module').then( module => module.LoginModule),  data: { preload: true } } |
| --- |

Import the custom preloading ts file to app.module.ts and inject it in Provider.

| @NgModule({  imports: [RouterModule.forRoot(routes, { preloadingStrategy: CustomPreloadingStrategy })],  exports: [RouterModule] }) |
| --- |

**Route Guard:**

The Angular router’s navigation guards allow to grant or remove access to certain parts of the navigation

Types of Route Guard:

1. CanActivate:

It controls, route can be activated.

1. CanActivateChild:

It controls, children of the route can be activated.

1. CanLoad:

It controls, If a route can even be loaded. This becomes useful for feature modules that are lazy loaded. They won’t even load if the guard returns false.

1. CanDeactivate:

It controls , If the user can leave a route. Note that this guard doesn’t prevent the user from closing the browser tab or navigating to a different address. It only prevents actions from within the application itself.

To use Route Guard:

We need to create a route guard service,

| import {  Router, CanActivate, CanActivateChild, ActivatedRouteSnapshot,  RouterStateSnapshot, CanLoad, CanDeactivate } from '@angular/router'; |
| --- |

| canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean { } |
| --- |

| canActivateChild(childRoute: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean { } |
| --- |

| canLoad(): boolean { } |
| --- |

| canDeactivate(component: AddComponent): boolean { } |
| --- |

Then, configure the Route Guards to the app-routing module.

| {  path: 'library',  canLoad: [AuthGuard] //route name : [routeServiceName] } |
| --- |

**Angular Pipes:**

The pipes are used to transform data before display. when we *only* need that data transformed in a template.

Built-in pipes include lowercase, uppercase, decimal, date, percentage, currency, jsonPipe.

We can also chain pipe using colon : and also pass parameter

| <h1> {{employee.dob | date: 'fullDate'}} </h1> |
| --- |

Custom Pipe:

We can create custom pipe, using Pipe and PipeTransform

| ng g p <pipeName> |
| --- |

Custom Pipe by extending pipeTransform,

| transform(value: any, ...args: any[]): any {  //value - pipe value to be transform  //...arg - pipe arguments } |
| --- |

Template Driven Form:

Here most of work are done in the form template, Need to import FormModule,

| <form #userlogin="ngForm" (ngSubmit)="onClickSubmit(userlogin)" >   <**input** type="text" name="email-id" ngModel>  <**input** type="submit" value="submit">  </**form**> |
| --- |

**Promises vs Observable:**

To handle Asynchronous data,

| Promises | Observable |
| --- | --- |
| Emits a single value | Emits multiple value over a period of time |
| It use two methods as parameter, onFulfilled( ) and onRejected( ) | It use three methods as parameter,  next( ), error( ) and complete( ) |
| Not Lazy | Lazy. An observable is not called until we use subscribe method to that observable. |
| Cannot be Cancelled | Can be Cancelled, by using the unsubscribe( ) method |
|  | Observable provides many powerful filters like map, forEach, filter, reduce, retry, retryWhen, etc. |

Observable Retry on Error:

To retry an observable import retryWhen, scan, delay from RxJS.

| this.getEmployee(url).retryWhen((err) => {  return err.scan((retryCount) => {  retryCount + = 1;  if(retryCount > 6){  this.message = 'Retrying attempt ' + retryCount;  return retryCount;  }else{  throw(err)  }  }, 0).delay(1000)  .subscribe((response) => {  },  (error) => {  } }); |
| --- |

Cancel an Observable:

We can cancel an observable using the unsubscribe method.

1. Using UnSubscribe Method:

| this.subscription = this.getEmployee(url).subscribe((response) => {}) this.subscription.unsubscribe() |
| --- |

1. Using Async | Pipe
2. Using RxJs take\* Operator
3. Using Rxjs first Operator
4. Use decorator to automate UnSubscription
5. Use tsLint

**Angular Compilation:**

In angular, there are 2 types of compilation

1. Just-in-Time Compilation ( JIT ) - (Run Time Compilation)

JIT compiles the application just in time, in the browser at run time.

By default the application is downloaded with the angular compiler code

In vendor.bundle.js - it has the half size of code for the compiler.

1. Ahead-of-Time Compilation ( AOT ) - (Pre-Compiled)

In AOT, it compiles the application at Build time. By default the application is pre compiled, so the browser loads the executable code and it will render the application. So there is no need to download the angular compiler code.

It also use Tree shaking, Minification, Uglification

In vendor.bundle.js - Half of size is reduced, No compiler code

# Source Map Explorer tool helps to analysis the build files

Development Build vs Production Build:

1. Source Map:

It helps us to easily debug our application even after the files are compressed and combined.

Dev Build - Yes (Source map will be included)

| ng build --dev -sm false //To prevent source map |
| --- |

Prod Build - No (Source map will not be included)

1. Extract CSS:

Dev Build - Global styles are extracted to .js file

Prod Build - Global styles are extracted to .css file

| ng build --dev --ec true //extract styles to css file |
| --- |

1. Minification & Uglification:

The minification is a process of removing extra white space, comments, optional tokens like curly brackets and semicolons.

The uglification is a process of transforming code to use short variable and function names.

Dev Build - No

Prod Build - Yes

1. TreeShaking:

It is a process of removing any code that we are not actually using in our application from the final bundle.

Dev Build - No

Prod Build - Yes

1. AOT Compilation:

The angular component templates are pre-compiled so need of downloading the angular compiler code in bundle file

Dev Build - No

Prod Build - Yes

**RxJS:**

The RxJS is a library for reactive programming using Observables, to make it easier to compose Asynchronous or callback based code.

RxJS Operator:

1. Finalize:

It returns an observable, that mirrors the source observable. It will call a specific function when the source is terminated on complete or error.

| this.getEmployee(url).pipe(  finalize(() => { //it will call after the success or error, to stop loading  }) ).subscribe((response) ={   //success logic }, (error) => {  //error logic }) |
| --- |

**Component Life Cycle:**

1. Constructor: It is the default method of the class that is executed when the class is initialized.
2. ngOnInit: It will invoked, when a component is initialized
3. ngOnChanges: It will invoked, when there is a change in one of the input properties of the component (parent to child)
4. ngDoCheck: It will invoked, when the change detector of the component is invoked
5. ngAfterContentInit: It will invoked, when the content is projected into the component view
6. ngAfterContentChecked: It will be invoked, each time when the given component has been checked by the change detection mechanism.
7. ngAfterViewInit: It will be invoked when the component view has been fully initialized.
8. ngAfterViewChecked: It will be invoked each time when the component has been checked by the change detection mechanism.
9. ngOnDestroy: It will be invoked, when the angular component is destroyed.

**Sharing data between Components:**

1. Parent to Child: Sharing data via Input
2. Child to Parent: Sharing data via Output & EventEmitter
3. Child to Parent: Sharing data via ViewChild
4. Unrelated Component: Sharing data via shared service & Subject Behaviour

Parent to Child via Input:

@Input( ) - Decorator used to share data from parent to child via template.

| //parent component - html <app-child-component [attribute]="sharedData"></**app-child-component**> |
| --- |

| //child component - ts @Input(‘attribute’) message: any; |
| --- |

Child to Parent via Output and EventEmitter:

The child component can emit events using @Output( ) decorator and Event Emitter method.

| //child component - ts import { Output, EventEmitter } from '@angular/core';  @Output() messageEvent = new EventEmitter();  sendMsg(){  this.messageEvent.emit(data) } |
| --- |

| //parent component - html <app-child-component (messageEvent)=”method()”> </**app-child-component**> |
| --- |

Child to Parent via ViewChild:

The viewchild allows one component to be injected to another component and gives parent access to its attributes and functions.

| //parent component - ts import { ViewChild } from '@angular/core'; import { ChildComponent } from '../child-component';  @ViewChild(ChildComponent, {static: false}) private childReference; //this.childReference - used to access child attributes and functions. |
| --- |

Unrelated Component via Subject Behaviour and Shared Service:

| //shared service - ts import { BehaviourSubject } from 'rxjs';  public sharedData = new BehaviourSubject({}); public sharedInfo = this.sharedData.asObservable();  //method to update the value updatedContent(message){  this.sharedData.next(message); } |
| --- |
|  |

| //component one - To update the shared value this.dashboardAPI.updateContent(data); |
| --- |

| //component two - To get the shared value this.dashboardAPI.sharedInfo.subscribe((data) => { }) |
| --- |

**Directive:**

1. Component Directive:

The main class has the details of how the component to be processed, instantiated and used at runtime.

1. Structural Directive:

It deals with manipulating the dom elements. Adding, removing, inserting the content of DOM elements.

1. Attribute Directive:

It deals with changing the look and behaviour of the DOM, altering or adding properties to the DOM element.

1. Custom Directive:

ng g d <directiveName> //To create custom directive

| import { ElementRef, Renderer2, HostListner, HostBinding } |
| --- |

| //To set property on a DOM element renderer.setElementStyle(el.nativeElement, 'background', 'red');  //To select a DOM element this.el.nativeElement.querySelector('#content') |
| --- |

@HostListner - It will listen to the event emitted by the host element.

| @HostListner(event) functionName(){  //code logic to handle the target element } |
| --- |

@HostBinding - It will bind property to the host element

| @HostBinding(<Attribute>) <varibaleName> : any |
| --- |

**Routing:**

In Routing, we need to define the route path and configure the route to corresponding components in the app.routing.module.ts

| { path: 'path-name', component: <**ComponentName**> } |
| --- |

| Add the router outlet component, where the page is going to display. <router-outlet></**router-outlet**> |
| --- |

In order to active the route, need to do it via HTML or Code

1. Activate via HTML:

| [routerLink] = "['/path-name']" |
| --- |

1. Activate via Code

| import { Router } from '@angular/router';  constructor(private router : Router)  this.router.navigationByUrl('/path-name'); |
| --- |

To show active route selection in HTML

| [routerLinkActive] = "['css-classname']" [routerLinkActiveOptions] = "{exact: true}" |
| --- |

To read value from browser URL,

| import { ActivatedRoute } from 'angular/router'; constructor(private route: ActivatedRoute) this.route.paraMap.subscribe((route) => {  //route.params - can read the id value from url }) |
| --- |

**Service:**

The services are used to share code logic across different components. It is also used to access the data connection.

| ng g s ServiceName //To generate a service |
| --- |

Then, we need to import and inject that service on app.module.ts

| @NgModule({  providers: [ServiceName] }) |
| --- |

Then, the component again needs to inject it at the constructor.

**HTTP Communication:**

1. Import the **HttpClientModule** at the app.module.ts

| import { HttpClientModule } from "@angular/common/http"; |
| --- |

1. Inject that HttpClientModule to the imports part of @NgModule
2. Create a shared service and import **HttpClient**

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

1. Inject that HttpClient to the constructor part of the service
2. Use the REST method, from the reference of HttpClient
3. Inject the shared service to the component, which requires the data to be fetched.
4. Inject the shared service to the component constructor
5. Use the shared function to access the data from the backend server.

Install JS and CSS Plugin:

In angular.json, add the JS and CSS plugin library under the build object.

**Data Binding:**

1. Component to DOM
   1. Interpolation: {{ VariableName }}
   2. Property Binding: [Property] = "VariableValue"
2. DOM to Component
   1. Event Binding: (event)="functionName( );"
   2. TwoWay Binding:

| import { FormsModule } from '@angular/forms' [(ngModel)] = "VariableName" |
| --- |

Routing:

1. Navigating from one page to another page
2. In navigation, there are 3 ways of approach
   1. From one entire screen to another entire screen
   2. Changing content inside one page (gmail) - Nested Routing
   3. Module to Module navigation (feature module)