

# Angular 7

Rinkesh Patel (15013)

# Agenda

01. Introduction	12. Component Interaction via Decorators
02. Basic development flow of Angular app	13. Pipes and Custom Pipes
03. Main building blocks of Angular app	14. Directives and Custom Directives
04. Architecture of Angular	15. Routing
05. Angular CLI and Webpack	16. Modules loading
06. Setup of Angular application	17. Services and Dependency Injection
07. Modules	18. Component Interaction via service
08. Components	19. Forms
09. Data binding	20. HTTPClient
10. Decorators	21. RxJS
11. Component Lifecycle hooks	

# Prerequisite

- What you should know
  - HTML
  - CSS
  - JavaScript
  - TypeScript

# What is Angular

- Angular is a framework for building client-side applications.
- It is TypeScript-based open-source framework.
- Angular itself is written TypeScript.
- Support Cross-Platform (Web, Mobile and Desktop)
- Focused on Single Page Application (SPA) development.
- Maintained by the Angular Team at Google and by a community of individuals.

# Versions

Version	Release Year	Description
AngularJS 1.0	June 2012	AngularJS was originally developed in 2009 and version 1.0 launched in 2012.
Angular 2	Sep 2016	Complete rewrite of AngularJS.
Angular 4	Mar 2017	<p>Every 6 month, the Angular Team release a new version by adding some new features, improve performance, compilation and optimization.</p> <p>The motive behind each release is to make it:</p> <ul style="list-style-type: none"><li>• Smaller,</li><li>• Faster and</li><li>• Easier to use for developer.</li></ul>
Angular 5	Nov 2017	
Angular 6	May 2018	
Angular 7	Oct 2018	
Angular 8	May 2019	

# Basic development flow of Angular application

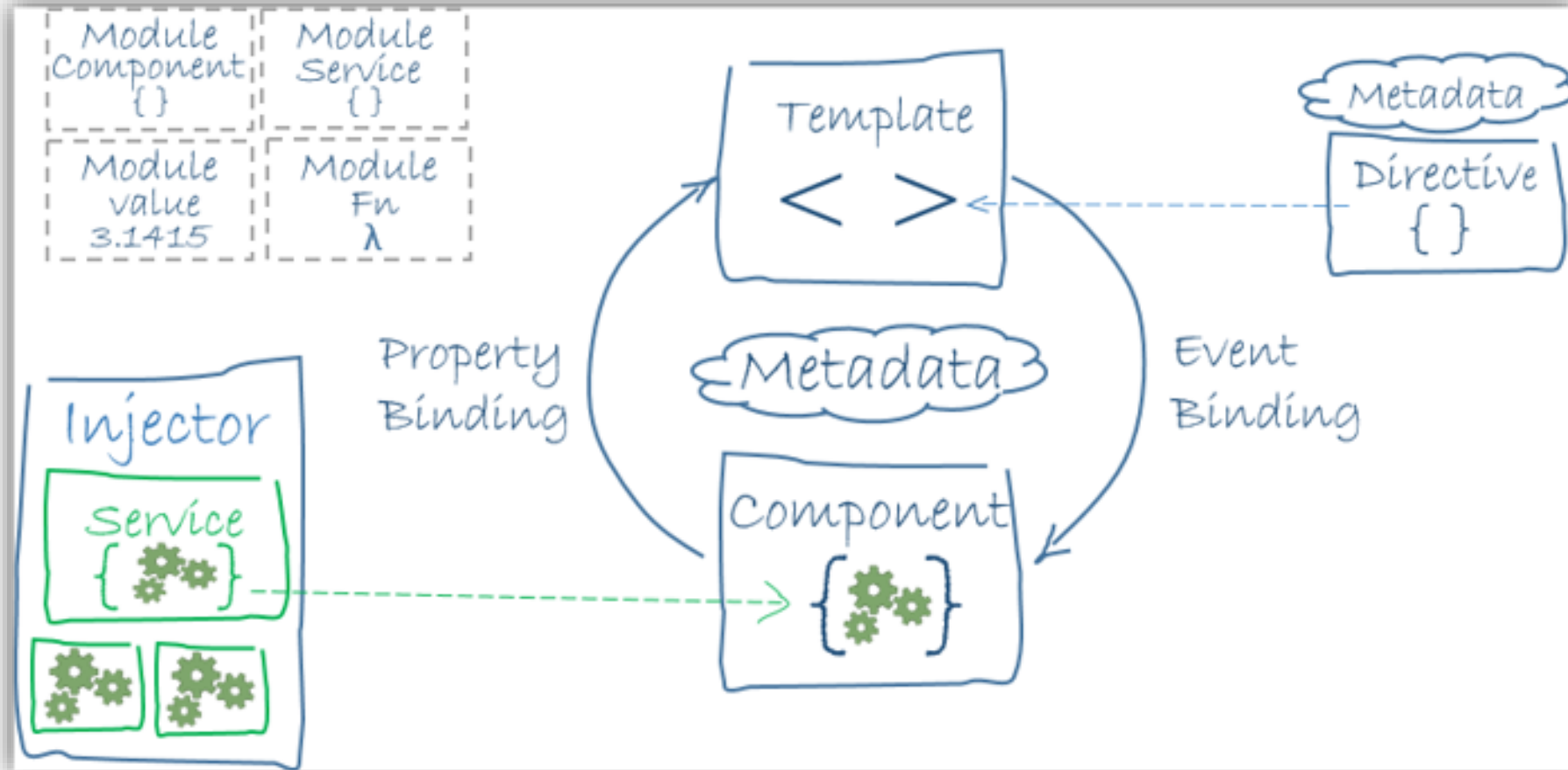
1. Set up the Development Environment using angular CLI.
2. Templates: Write HTML Templates for view.
3. Components: Write Components to manage those Templates.
4. Services: Write application logic in Services.
5. Modules: Wrap Components and services in Modules.
6. Root Module: Launch the app by bootstrapping the Root Module.
7. Now, Angular will present the application content in a browser and respond to user interactions according to the instructions provided.

# Main building blocks of Angular application

1. **Templates:** HTML views
2. **Components:** Special kind of directives with a Template.
3. **Services:** Share reusable application logic across the application.
4. **Directives:** Extended HTML attributes – change the DOM layout.
5. **Modules:** Groups of related components, services, directives etc.
6. **Metadata:** Tells Angular how to process a class (using decorator).
7. **Data Binding:** Communication between components & templates.
8. **Dependency injection:** A way to supply a new instance of one class to another class.

# Architecture:

How main building blocks related





# Angular CLI (command-line interface)

- The Angular CLI is a command-line interface tool that is used to initialize and maintain Angular applications.
- We can use the tool directly in a command shell, or indirectly through an interactive UI such as Angular Console.
- Usage:
  - Create new Angular application
  - Code scaffolding: Creating new Modules, Components, Pipes, Directives, Services etc. in existing Angular application by resolving dependencies.
  - Configure features like Routing and CSS preprocessors.
  - Run application's unit tests
  - Run application's end-to-end tests
  - Build application for both Development and Production environment.
  - Configure webpack

# Webpack

- By using Angular-CLI we don't need to bother about webpack. The Angular CLI hides all that webpack complexity.
- **Webpack** is a tool for bundling application source code that is loaded from server into a browser.
- **Bundle** is a JavaScript file that includes all application assets together. It is served from server as a response of single file request into the client.
- Webpack can preprocess and minify different non-JavaScript files such as TypeScript, SASS, and LESS files.

# Setup

- **Prerequisite**

- Node.js
  - Install Node.js from [www.nodejs.org](http://www.nodejs.org)
- CLI (Command line interface is a tool that allows us to create a project, build and run it on development server directly using command line.
  - `npm i -g @angular/cli`

- **If CLI is already installed**

- To check the version of Node.js and CLI
  - `ng -v`
- To update the version
  - `ng update`

- **Create a skeleton of Angular application:**

- `ng new PROJECT-NAME`
- `cd PROJECT-NAME`
- `ng serve -o`

# Angular.json file

- Configure the application:
  - **“root” : “src”** – Root directory of the app.
  - **“outDir”: “dist”** – Output directory of build result.
  - **“main”: “main.ts”** - The name of the main entry-point file.
  - **“index”: “index.htm”** – Name of the start HTML file (First view while bootstrapping).
  - **“styles” : [“styles.css”]** – Global styles to be included in the build.
  - **“assets” : [“src/assets“]** – List of application assets.
  - **"polyfills": "src/polyfills.ts"** - Includes configuration for browser compatibility.

# Package.json file

- Includes dependencies at both “prod” and “dev” environment.
- Includes list of modules which are required in app.
- When run command “npm install” all dependencies will be installed.
- Name and version of project:
  - "name": “DemoApp“
  - "version": "0.0.0"
- Define CLI commands:

```
"scripts": {
  - "ng": "ng",
  - "start": "ng serve",
  - "build": "ng build",
  - "lint": "ng lint",
  - "test": "jest"}
```

# How App start

- **main.ts:** (entry point of the angular app)
  - **app.module.ts:**
    - main.ts main.ts file bootstrap the app.module.ts
  - **app.component.ts:**
    - app.module.ts file bootstrap the app.component.ts.
  - **app.component.html:**
    - app.component.ts file render the app.component.html.

# Angular CLI commands

- Create Component
  - `ng g c component-name`
  - likewise create Module | Service | Pipe | Directive | Interface | Enum etc.
- Build
  - `ng b` (Development build)
  - `ng b --prod` (Production build)
- Serve (run the app)
  - `ng s`
  - `ng s -o` (open in browser)
  - `ng s --o --port 4500` (run with port 4500)
  - `ng s --watch false` (stop auto compile and refresh on source code change)
- Run **ng help** for more commands

# Dev Build and Prod Build

On execution of **ng b** command, a build version is created inside a “**dist**” folder.

## Dev Build

1. Includes **Source Maps** by default that helps to debug the code even after the files are compressed and combined.
- We can set/reset the source maps flag in both dev and prod: **ng b --sm false**

## Prod Build

- Prod build is **Minified, Uglified and TreeShaked**
1. **Minification:** Process of removing whitespace, comments and optional tokens like semicolons, curly brackets.
  2. **Uglification:** Process of transforming variable and function names with short names.
  3. **Tree Shaking:** Process of removing the unnecessary code which is not actually used in the application. (i.e. If we have created service and component but not imported or used anywhere then that service and component will be removed from the final bundle **to reduce the size**)
  4. **AOT:** Ahead-of-Time compilation



# JIT and AOT compilation

- **JIT****- Just-In-Time Compilation**

- Compiles the code during **run time**.
- The code is compiled when it is loaded on browser.
- When application runs on browser:
  1. The browser downloads application assets including HTML, CSS and JavaScript
  2. Then, Angular bootstraps the application
  3. Then, Angular goes through JIT compilation process
  4. Then, Application gets rendered on browser screen.

- **AOT****- Ahead-Of-Time Compilation**

- Compiles the code during **build time**
- Browser loads compiled and executable code so it renders faster than JIT.
- Smaller in size to load on browser.
- Detects errors at build time rather than run time

# Modules

- Angular has its own modularity system called NgModule.
- Most apps have following two types of modules:
  1. At least one root module.
  2. Many more feature modules.
- NgModule can be root or feature and it is a class with @NgModule decorator.
- **Decorator:**
  - It is a function that attach metadata to the class to tell angular how that class should work.
  - Its declaration starts with '@'
  - It is declared just before a class declaration.

# Sample code of NgModule

TS *app.module.ts* ✕

```
1  import { BrowserModule } from '@angular/platform-browser';
2  import { NgModule } from '@angular/core';
3  import { AppComponent } from './app.component';
4  import { DemoModule } from './demo/demo.module';
5  @NgModule({
6    declarations: [ AppComponent ],
7    imports: [ BrowserModule, DemoModule ],
8    providers: [],
9    bootstrap: [AppComponent]
10 })
11 export class AppModule { }
```

# Properties of NgModule:

1. **Imports:** contains other modules whose exported classes are needed by the classes declared in this module.
2. **providers:** services that this module contributes to the global collection of services; they become accessible in all parts of the app
3. **declarations:** Three kinds of view classes:
  1. Components, 2. Directives, 3. Pipes
4. **exports:** contains set of classes declared in this module which can be accessed in other modules.
5. **bootstrap:** the main application view, called the root component, that hosts all other app views. Only the root module should set this bootstrap property.

# Angular libraries

- Angular provides a collection of JavaScript modules. Which are known as library modules.
- Each Angular library name begins with the **@angular** prefix.
- We can install them with the **npm** package manager and import parts of them with JavaScript **import** statements.
- For example:
  - `npm install`
  - `import { Component } from '@angular/core';`
  - `import { BrowserModule } from '@angular/platform-browser';`

# Components

- Controls and interact with view.
- Contains application logic inside class.
- Angular creates, updates, and destroys component instance as the user moves through the application.

## **@Component decorator :**

- It is a function that attach metadata to the class to tell angular how that class should work.

# Metadata - Configuration options

1. **selector :**
    - Name of element that is written in HTML template to load this component.
  2. **templateUrl / template :**
    - address of this component's HTML template / HTML code
  3. **styleUrls / styles:**
    - address of this component's style file.
- The Template, Metadata and Component together describe a **view**.

# Sample code of component

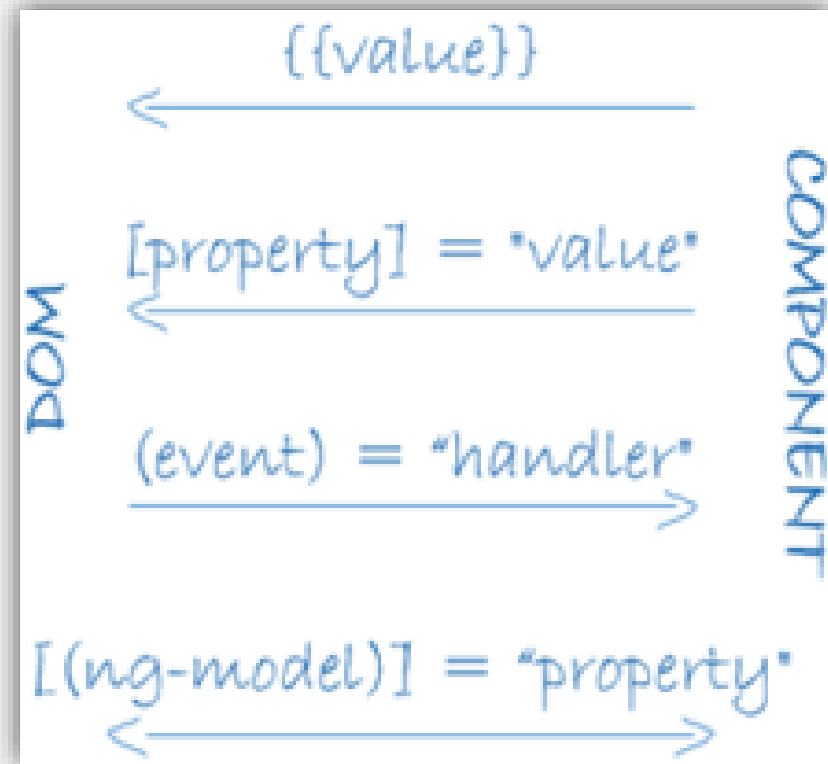
TS demo-parent.component.ts ✕

```
1  import { Component } from '@angular/core';
2  @Component({
3    selector: 'app-demo-parent',
4    templateUrl: './demo-parent.component.html',
5    styleUrls: ['./demo-parent.component.css']
6  })
7  export class DemoParentComponent {
8    public hasChild:boolean = false;
9  }
```



# Data binding

- Communication between component and View (template).
- Bind data from DOM to Component or from Component to DOM or both.
- Four types of Data binding:
  1. Interpolation
    - use `{{ }}`
  2. Property binding
    - use `[]`
  3. Event binding
    - use `()`
  4. Two way binding
    - (combination of Property and event binding)
    - use `[]()`



# Demo

- Demo for databinding

# Decorators

- Decorator is a feature that allows us to supply metadata that tells Angular how the Class | Method | Property | Parameter should work.
- Types of Decorators:
  1. **Class** example: `@Component`, `@NgModule`
  2. **Method** example: `@HostListener`
  3. **Property** example: `@Input`, `@Output`
  4. **Parameter** example: `@Inject`
  5. **Custom**

## (1) Class Decorators:

```
@DecoratorName({...})  
export class DemoClass {  
  |  .....  
}
```

# Decorators

## (2) Method decorators:

```
@HostListener('click', ['$event'])
onHostClick(event: Event) {
  this.count++;
  console.log(`Event ${this.count}: ${event.target['innerText']}`);
}
```

## (3) Property decorators:

```
export class ContactUpdateComponent {
  @Input() contacts: any;
  @Output() onUpdated = new EventEmitter<any>();
}
```

# Decorators

## (4) Parameter decorator:

```
export class DemoComponent {  
  constructor(@Inject(DemoService) private demoService) { }
```

## (5) Custom decorator:

```
function customDecorator(value) {  
  console.log(`Custom Decorator value: ${value}`);  
  return function decorator(componentInstance) {  
    console.log(componentInstance);  
  }  
}
```

```
@customDecorator(true)  
export class AppComponent {
```

# Component Lifecycle hooks

- A component has a lifecycle managed by Angular.
- Angular
  - creates it,
  - renders it,
  - creates and renders its children,
  - checks it when its data-bound properties change, and
  - destroys it before removing it from the DOM.
- During above three stages of component, Angular offers **lifecycle hook interfaces** that provide visibility into these key life moments.
- Each interface provides a **lifecycle hook method** to act when these key life moments occur.
- For example: The **OnInit** interface has a hook method named **ngOnInit()** that Angular calls shortly after creating the component

# Component lifecycle hooks sequence:

1. **ngOnChanges()** : Called when input property changed.
2. **ngOnInit()** : Called on initialization of the component.
3. **ngDoCheck()** : Called at every change detection cycle.
4. **ngAfterContentInit()** : Called after content initialized.
5. **ngAfterContentChecked()** : Called after every check of component content.
6. **ngAfterViewInit()** : called after component's view initialized.
7. **ngAfterViewChecked()** : Called after every check of a component's view.
8. **ngOnDestroy()** : Called just before the component is destroyed.

# Demo

- Demo for Component lifecycle hooks



# Component Interaction

- Ways of communication between components:
  1. Pass data from parent component to child via @input.
  2. Bind events from child component to parent component via @output
  3. Access child component properties from parent component's view via template reference variable.
  4. Inject child component into parent component via @ViewChild.
  5. Component interaction via services.

# @Input decorator

- We can pass parent component's property to child component via @Input decorator.

## @Input:

- It is a property decorator that marks a class field as an input property and supplies configuration metadata.
- The input property is bound to a DOM property in the template.
- During change detection, Angular automatically updates the data property with the DOM property's value.

## Dependency:

- `import { Input } from '@angular/core';`

# @Output decorator

- We can listen child component's event to parent component via @Output decorator.

## @Output:

- It is property decorator that marks a class field as an output property and supplies configuration metadata.
- The DOM property bound to the output property is automatically updated during change detection.

## Dependency:

- `import { Output, EventEmitter } from '@angular/core';`
- **EventEmitter** is a class used in directives and components to emit custom events and register handlers for those events by subscribing to an instance.

# Demo

- Demo for Component interaction via @Input and @Output
- Demo for Component interaction – ngOnChange
  - Detect lifecycle hook when input value changed.

# Template reference variable

- We can use child component's properties/method in parent by creating a template reference variable.
- Create creating a template (html) reference variable for the child element and then refer that variable in parent template (html).
- Example:

```
<> demo-parent.component.html x
1  <h4>This is Parent component.</h4>
2
3  <p>Click below button to access child component method and property:</p>
4
5  <button (click)="demoReference.alertFromChild()">Call child component method</button>
6
7  <app-demo-child #demoReference></app-demo-child>
8
9
```

# Demo

- Demo for component interaction via template reference variable.

# @ViewChild

- It is property decorator that configures a view query.
- We can access a child component, directive or a DOM element from a parent component via @ViewChild decorator.

Example:

```
1 import { Component, ViewChild } from '@angular/core';
2 import { DemoChildComponent } from '../demo-child/demo-child.component';
3
```

```
export class DemoParentComponent {
  // declare childComponent (create instance of DemoChildComponent)
  @ViewChild(DemoChildComponent) private childComponent: DemoChildComponent;
  public showAlert(): void {
    this.childComponent.alertFromChild();
  }
}
```

# Demo

- Demo for component interaction via @ViewChild



# Pipes

- Angular pipes (|) are used to transform data into desired output and display it on the browser.
- Pipes (|) take an input data and transform it to desired output.
- **Syntax:** {{ data | pipeName }}
- **Example:**
  - Display the data in uppercase: {{ data | uppercase }}
- **Types of Pipes:**
  1. Built-in Pipes
    - Parameterized Pipes
    - Chaining Pipes
  2. Custom Pipes

# Parameterized Pipes

- We can pass any number of parameters to the pipe using colon (:)
- **Syntax:**
  - `{{ data | pipName : parameter1 : parameter2 :.....: parameterN }}`
- **Example:**
  - `{{ currentDate | date: "dd/MM/yy" }}`
  - `{{ currentDate | date: "mediumTime" }}`
  - `{{ currentDate | date: "mediumTime" : "UTC" }}`

# Chaining Pipes

- We can use multiple pipes with the same data at the same time.
- **Syntax:**
  - `{{ data | pipe1 | pipe2 | pipe3 | ..... | pipeN }}`
- **Example:**
  - `{{ currentDate | date: "fullDate" | uppercase }}`

# Demo

- Demo for built-in pipes
  - Parameterized pipes
  - Chaining pipes

# Custom Pipes

- We can create custom pipes and apply logic to transform data as per our requirements.
- The **@Pipe** decorator allows us to define Pipe name and use it within template.
- **CLI command to generate Pipe:**
  - `ng g p pipeName`
- **Syntax:**
  - `@Pipe({ name: 'pipename' })`
- **Dependency:**
  - `import { Pipe } from '@angular/core';`

# Custom Pipes

- The transform() method of PipeTransform interface allows us to transform our data into desired output by accepting parameters and returning output.

TS custom-pipe.pipe.ts ✕

```
1  import { Pipe, PipeTransform } from '@angular/core';
2
3  @Pipe({ name: 'customPipe'})
4
5  export class CustomPipePipe implements PipeTransform {
6      transform(value: string, gender: string): string {
7          if(gender.toLowerCase() == "male"){
8              return "Mr. " + value;
9          }else{
10             return "Ms. " + value
11          }
12      }
13  }
```

# Demo

- Demo for custom Pipes.

# Routing

- Routing is a mechanism for navigating between pages and display the appropriate view of component on browser.
- Angular is focused on SPA (Single Page Application).
- It loads single full HTML page and inside that page, dynamically loads or updates partial pages as per user request.
- The above mechanism can be achieved with the help of Angular routing.



# Important concepts in Routing

1. routes array
2. router-outlet component
3. Navigation:
  1. routerLink
  2. navigate()
4. redirectTo property in routes array
5. Wildcard route
6. Child route
7. Route Guards:
  1. CanActivate
  2. CanDeactivate
  3. Resolve
  4. CanLoad
  5. CanActivateChild
8. Module Loading:
  1. Eager Loading
  2. Lazy Loading
  3. Pre Loading

# Router imports

- The Angular Router is an optional service that presents a particular component view for a given URL.
- It is not part of the Angular core. Import it from @angular/router.

```
import { Routes, RouterModule } from '@angular/router';
```

- When creating the App in CLI, add Angular routing:

```
E:\>ng new demo1  
? Would you like to add Angular routing? (y/N) y
```

# Configuration

- As we add routing while creating Angular app from CLI, Angular create app-routing file and configure RouterModule.forRoot() method for routing.
- We have to add routes in the routes array passed in forRoot() method.
- The forRoot() method supplies the service providers and directives needed for routing, and performs the initial navigation based on the current browser URL

```
const routes: Routes = [];  
  
@NgModule({  
  imports: [RouterModule.forRoot(routes)],  
  exports: [RouterModule]  
})  
export class AppRoutingModule { }
```

# (1) Routes array

- The routes array describes how to navigate.
- We have to add path and component reference to the routes array.
- The router parse this array and navigate to matching path.

```
import { Page1Component } from './page1/page1.component';
import { Page2Component } from './page2/page2.component';
import { Page3Component } from './page3/page3.component';

const routes: Routes = [
  {path: 'page1', component:Page1Component},
  {path: 'page2', component:Page2Component},
  {path: 'page3', component:Page3Component}
];
```

## (2) router-outlet Component

- Router Outlet is a dynamic component that the router uses to display views based on navigations.
- **Syntax:**
  - `<router-outlet></router-outlet>`
- `<router-outlet>` tells the router where to display the views of components.

```
<a [routerLink]="['/page1']">Page 1</a> |  
<a [routerLink]="['/page2']">Page 2</a> |  
<a [routerLink]="['/page3']">Page 3</a>  
  
<router-outlet></router-outlet>
```

## (3.1) Navigate using routerLink

- **routerLink** is Angular's built-in directive used to link routes of our application from the HTML template.
- The routerLink directive turns user click into navigation based on the path assigned to it.

 app.component.html ✕

```
2
3  <a [routerLink]="['/page1']">Page 1</a> |
4  <a [routerLink]="['/page2']">Page 2</a> |
5  <a [routerLink]="['/page3']">Page 3</a>
6
7  <router-outlet></router-outlet>
```

## (3.1) Navigate using **navigate()** method

- Router service provides `navigate()` method for navigation.
- In component, inject Router service and use its method `navigate()`

```
import { Router } from '@angular/router';
```

```
constructor(private router: Router){}
```

```
public navigatePage1(): void{  
  this.router.navigate(['/page1']);  
}
```

## (4) Redirecting routes using redirectTo

- A redirect route translates initial URL (‘ ‘) to desired default path.
- When application starts it navigates to empty route by default. We can configure it to specific route.

```
import { Routes, RouterModule } from '@angular/router';  
import { HomeComponent } from '../home/home.component';
```

```
const routes: Routes = [  
  {path: '', redirectTo:'home', pathMatch:'full'},  
  {path: 'home', component:HomeComponent}  
];
```



## (5) Wildcard route

- Wildcard route is used to intercept invalid routes and handle them by redirecting to specific route.
- A wildcard route has path consisting of two asterisks (\*\*).
- Router matches every path in routes array with URL, if no path match then it redirects to wildcard route.

```
const routes: Routes = [  
  {path: '', redirectTo:'home', pathMatch:'full'},  
  {path: 'home', component:HomeComponent},  
  {path: 'page1', component:Page1Component},  
  {path: 'page2', component:Page2Component},  
  {path: 'page3', component:Page3Component},  
  {path: '**', component:PageNotFoundComponent}  
];
```

## (6) Child route using children

- If a component has multiple children components and if we want to set route for each child component, then we can configure them as child route.

```
const routes: Routes = [  
  {path: '', redirectTo:'home', pathMatch:'full'},  
  {path: 'home', component:HomeComponent},  
  {  
    path: 'admin',  
    children:  
    [  
      {path: '', component:AdminComponent},  
      {path: 'dashboard', component:DashboardComponent},  
      {path: 'manage-users', component:ManageUsersComponent},  
      {path: 'manage-employees', component:ManageEmployeesComponent}  
    ],  
  },  
];
```

# Demo

Demo for:

- Routing using router-links
- Routing using router.navigate() method
- Redirecting routes using “redirectTo”
- Wildcard route
- Child route.

# (7) Route Guards

- Route guards are interfaces with methods that allow us to apply logic when user navigates between routes. We can provide or remove the access of particular route path using route guards.
- Allowing the user to navigate all parts of the application is not a good idea. We need to restrict the user until the user performs specific actions like login. Angular provides the Route Guards for this purpose.
- Use of Route Guards:
  - Allow access after login
  - Allow access to certain parts of the application to specific users
  - Get confirmation from user before leaving the current page.
  - Validating the route parameters before navigating to the route
  - Fetching required data from API before navigating and loading the component.
- Types of Route Guards
  1. CanActivate
  2. CanDeactivate
  3. Resolve
  4. CanLoad
  5. CanActivateChild

# Route Guards

## (1) CanActivate

- This guard is used to allow access of route based on logic and condition. If the guard returns true then it allow user to navigate otherwise it will not.

## (2) CanDeactivate

- This guard is useful where the user might have some pending changes, which was not saved and if user navigates to other page. The CanDeactivate route guard allows us to ask user confirmation before leaving the component.

## (3) Resolve

- This guard delays the activation of the route until some tasks are complete. We can use the guard to pre-fetch the data from the API, before activating the route.

# Route Guards

## (4) CanLoad

- The CanLoad Guard prevents the loading of the Lazy Loaded Module. We generally use this guard when we do not want to unauthorized user to be able to even see the source code of the module.
- This guard works similar to CanActivate guard with one difference. The CanActivate guard prevents a particular route being accessed. The CanLoad prevents entire lazy loaded module from being downloaded.

## (5) CanActivateChild

- This guard determines whether a child route can be activated. This guard is very similar to CanActivateGuard. We apply this guard to the parent route. The Angular invokes this guard whenever the user tries to navigate to any of its child route. This allows us to check some condition and decide whether to proceed with the navigation or cancel it.

# Steps to implement Route Guards

1. Build the Guard as Service.

```
export class CanactivateGuard implements CanActivate {
```

2. Implement the Guard Method in the Service.

```
canActivate(): boolean {  
  let token = sessionStorage.getItem('authToken');  
  if(!token){  
    alert('Your not authorized, Please login');  
    return false;  
  }  
  return true;  
}
```

# Steps to implement Route Guards

(3) Update the Routes array to use the guards.

```
export const routes: Routes = [
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent },
  { path: 'login', component: LoginComponent },
  { path: 'register', component: RegisterComponent, canActivate: [DeactivateGuardService] },
  { path: 'contact', component: ContactComponent },
  { path: 'members', component: MemebersComponent, canActivate : [CanactivateGuard] },
  { path: 'admin', loadChildren: './admin/admin.module#AdminModule', canLoad: [CanloadGuardServ
];
```



# Demo

- Demo for Route guards:
  - canActivate
  - canDeactivate
  - canLoad

## (8) Module loading

- An effective module loading strategy is key to a successful single page application.
- A module can be loaded in three different ways:
  1. Eager loading
    - All modules loaded on application start.
  2. Lazy loading
    - Each module loaded only when it is required.
    - Used in large application where large amount of data loaded in each module.
  3. Preloading
    - Only required module is loaded on application start and other modules are loaded in background after application start.

# Eager Loading Module

- For eager loading we need to import all modules in `app.module`.
- When module loaded, it loads all imported modules, components, services and pipes of that module.
- Eager loading strategy is good for small applications where small amount of data loaded in each module.
- As all modules are loaded on application start, the access of application and navigation will be faster.

# Eager Loading Module : Demo Setup

- Create new demo app for eager loading
  - `ng new eagerLoadingDemo --routing`
- Generate modules with routing
  - `ng g m module1 --routing`
  - `ng g m module2 --routing`
  - `ng g m module3 --routing`
- Generate components in each modules
  - `ng g c module1/component1 --flat`
  - `ng g c module2/component2 --flat`
  - `ng g c module3/component3 --flat`

# Eager Loading Module: app-routing module configuration

TS app-routing.module.ts x

```
1  import { NgModule } from '@angular/core';
2  import { Routes, RouterModule } from '@angular/router';
3  import { Component1Component } from '../module1/component1/component1.component';
4  import { Component2Component } from '../module2/component2/component2.component';
5  import { Component3Component } from '../module3/component3/component3.component';
6
7  const routes: Routes = [
8    { path: 'page1', component: Component1Component },
9    { path: 'page2', component: Component2Component },
10   { path: 'page3', component: Component3Component },
11   { path: '**', redirectTo: '/page1', pathMatch: 'full' }
12 ];
13
14 @NgModule({
15   imports: [RouterModule.forRoot(routes)],
16   exports: [RouterModule]
17 })
18 export class AppRoutingModule { }
19
```

# Eager Loading Module: module-routing configuration

TS module1-routing.module.ts ✕

```
1  import { NgModule } from '@angular/core';
2  import { Routes, RouterModule } from '@angular/router';
3  import { Component1Component } from '../component1/component1.component';
4
5  const routes: Routes = [
6    { path: 'page1', component: Component1Component }
7  ];
8
9  @NgModule({
10    imports: [RouterModule.forChild(routes)],
11    exports: [RouterModule]
12  })
13  export class Module1RoutingModule { }
14
```

# Lazy Loading Module

- In lazy loading strategy Modules are loaded on demand.
- It helps to decrease the application startup time.
- Our application doesn't need to load everything at once, it needs to load only what we expect to load at default route on application startup.
- When we navigate to other routes the related modules with that route will be loaded.
- In large applications where each module has large amount of data, all modules loaded on application startup and it takes more time to start the application.
  - **So as a solution we can use lazy loading module strategy to reduce the application startup time.**

# Lazy Loading Module : Demo Setup

- Create new demo app for lazy loading
  - `ng new lazyLoadingDemo --routing`
- Generate modules with routing
  - `ng g m module1 --routing`
  - `ng g m module2 --routing`
  - `ng g m module3 --routing`
- Generate components in each modules
  - `ng g c module1/component1 --flat`
  - `ng g c module2/component2 --flat`
  - `ng g c module3/component3 --flat`



# Lazy Loading Module: app-routing module configuration

TS app-routing.module.ts ✕

```
1  import { NgModule } from '@angular/core';
2  import { Routes, RouterModule } from '@angular/router';
3
4  const routes: Routes = [
5    { path: 'page1', loadChildren: './module1/module1.module#Module1Module' },
6    { path: 'page2', loadChildren: './module2/module2.module#Module2Module' },
7    { path: 'page3', loadChildren: './module3/module3.module#Module3Module' },
8    { path: '**', redirectTo: '/page1', pathMatch: 'full' }
9  ];
10
11  @NgModule({
12    imports: [RouterModule.forRoot(routes)],
13    exports: [RouterModule]
14  })
15  export class AppRoutingModule { }
```

# Lazy Loading Module: module-routing configuration

TS module1-routing.module.ts ✕

```
1  import { NgModule } from '@angular/core';
2  import { Routes, RouterModule } from '@angular/router';
3  import { Component1Component } from '../component1/component1.component';
4
5  const routes: Routes = [
6    { path: '', component: Component1Component }
7  ];
8
9  @NgModule({
10   imports: [RouterModule.forChild(routes)],
11   exports: [RouterModule]
12 })
13 export class Module1RoutingModule { }
```

# Preloading Module

- In Preloading module strategy, like lazy loading only required module is loaded on application startup and once application startup the rest of the modules are loaded in background without navigation to their route.
- In lazy loading module we can reduce application startup time however when we navigate to other module routes it takes time to load that particular module on demand.
  - **As a solution we can use preloading module strategy so that when we navigate to different modules they are already loaded in background.**

# Preloading Module Configuration:

- For Preloading module setup is same as lazy loading module setup,
  - just we have to add **preloadingStrategy** property in RouterModule.forRoot()

```
@NgModule({  
  imports: [RouterModule.forRoot(routes, { preloadingStrategy: PreloadAllModules })],  
  exports: [RouterModule]  
})
```

# Demo

- Demo for:
  1. Eager loading modules
  2. Lazy loading modules
  3. Preloading modules

# Directives

- Angular directive help us to manipulate the DOM.
- Using directives we can change the layout or behavior of the DOM.
- Types of Directives:
  1. Structural Directives
    - Change DOM by adding / removing HTML elements.
  2. Attribute Directives
    - Change appearance or behavior of the DOM
  3. Component
    - Component is also one type of directive having its own template.
    - Component add DOM element with behavior.

# Structural Directives

- Structural directives can change the DOM layout by adding and removing HTML elements as per the expressions we pass to it.

## Examples:

- `*ngIf`
- `*ngFor`
- `*ngSwitch`

# \*ngIf

- It conditionally includes a template based on the boolean value of the expression passed to it.
- Shorthand Syntax:

```
<div *ngIf="condition">
```

Content to render when condition is true.

```
</div>
```



# \*ngFor

- It repeats the HTML element for each value of an array passed to it.
- Syntax:

```
<div *ngFor="let item of collection">  
    {{item}}  
</div>
```
- **collection** is the property that holds collection of data.
- **item** is the iterative element value of each iteration of the collection.

# \*ngFor – local variables

- Index
  - Contains the index of current element while iteration.
- First
  - Boolean value, true if the current element is the first while iteration.
- Last
  - Boolean value, true if the item is the last item in the iteration.
- Odd
  - Boolean value, true if the item is the odd-numbered item in the iteration.
- Even
  - Boolean value, true if the item is the even-numbered item in the iteration.

# \*ngSwitch

- Used to show/hide HTML element depending on the matching expression.
- It is combination of two types directives:
  - Attribute directive ([ngSwitch])
  - Structural directive ( \*ngSwitchCase and \*ngSwitchDefault)
- In [ngSwitch] we have to pass the desired value to match.
- In each \*ngSwitchCase we match the expression, the directive will show only matching expression element on the DOM.
- If no expression matched then we can set default section to display as default.

## \*ngSwitch example

```
<select (change)="onSelectDay($event)">
  <option value="">Select Day</option>
  <option value="Friday">Friday</option>
  <option value="Saturday">Saturday</option>
  <option value="Sunday">Sunday</option>
</select>
<div [ngSwitch]="selectedDay">
  <p *ngSwitchCase="'Friday'"> Friday selected</p>
  <p *ngSwitchCase="'Saturday'"> Saturday selected</p>
  <p *ngSwitchCase="'Sunday'"> Sunday selected</p>
  <p *ngSwitchDefault> No Day selected</p>
</div>
```

# Demo

- Demo for structural directives:
  - \*ngIf
  - \*ngFor
  - \*ngSwitch

# Attribute Directives

- An Attribute directive can change the appearance or behavior of an element.

## Examples:

- ngModel
- ngClass
- ngStyle

# ngClass

- ngClass directive allows us to add or remove CSS classes to an HTML element dynamically.
- It takes conditional expression or Boolean value and set to add or remove CSS classes to an HTML element accordingly.

## Syntax:

```
<div [ngClass]="{'cssClass1': expression1, 'cssClass2': expression2}'">
    .....
    // content of this element
    .....
</div>
```

# ngStyle

- ngStyle directive allows us to set the inline CSS style of the HTML element using an expression.
- It takes value of CSS style in expression and apply inline style to the HTML element.

## Syntax:

```
<div [ngStyle]="{'cssStyle1': expression1, 'cssStyle2': expression2}">  
    .....  
    // content  
    .....  
</div>
```



# Demo

- Demo for attribute directives:
  - `ngClass`
  - `ngStyle`

# Built-in ng elements

1. `ng-template`
2. `ng-container`
3. `ng-content`

# (1) ng-template

- `<ng-template>` is a template element that Angular uses with structural directives like `*ngIf` (if then else), `*ngFor`, `[ngSwitch]` and custom directives.
- These template elements only work in the presence of structural directives. Angular wraps the host element (to which the directive is applied) inside `<ng-template>` and consumes the `<ng-template>` in the finished DOM by replacing it with diagnostic comments.

```
<div *ngIf="isValid; then validBlock else invalidBlock"></div>  
✓ <ng-template #validBlock>  
  | This is valid block.  
  </ng-template>  
✓ <ng-template #invalidBlock>  
  | This is invalid block.  
  </ng-template>
```

## (2) ng-container

- Two structural directives can not be used at the same HTML element.
- As a solution we can use `<ng-container>` without break of existing layout of HTML page.

```
<div [ngSwitch]="selectedDay">
  <ng-container *ngFor="let day of dayList">
    <p *ngSwitchCase="day"> {{day}} selected</p>
  </ng-container>
</div>
```

## (3) ng-content

- Using `<ng-content>` we can add HTML content dynamically where we want to place it.
- For that we have to use the `<ng-content></ng-content>` tag as a placeholder for that dynamic content, then when the template is parsed Angular will replace that placeholder tag with our content.
- Using `<ng-content>` we can access Parent component's HTML templates from Child component's HTML template by passing Parent component templates inside Child component tags.

# Demo

- Demo for:
  - ng-template
  - ng-container
  - ng-content

# Custom Directive

- We can create our own custom directive and apply logic to manipulate the DOM as per our requirements.
- CLI command to generate directive: **ng g d directiveName**

```
TS country-code.directive.ts ✕  
  
1  import { Directive } from '@angular/core';  
2  @Directive({  
3    selector: '[appCountryCode]'  
4  })  
5  export class CountryCodeDirective {  
6    constructor() { }  
7  }
```

# Declare custom directive in Module

- When we create directive in Angular CLI, the Angular automatically declare the directive inside its Angular Module. (here we have created CountryCodeDirective inside DemoModule so it is automatically added in it.

TS demo.module.ts ✕

```
1  import { NgModule } from '@angular/core';
2  import { CommonModule } from '@angular/common';
3  import { DemoComponent } from './demo.component';
4  import { CountryCodeDirective } from './country-code.directive';
5
6  @NgModule({
7    declarations: [DemoComponent, CountryCodeDirective],
8    imports: [CommonModule]
9  })
10 export class DemoModule { }
```



# Demo

- Demo for custom directive.

# Services

- Service is a class of reusable code with well-defined purpose.
- Contains specific values and functions that application needs.

## Use of Services:

- Can be shared across the application.
- Components are big consumers of services.
- Can be used for external interactions like data access from Web API.

## Advantages of Services:

- Easier to test.
- Easier to debug.
- Reusable

# Dependency injection

- Dependency injection is a way to supply a new instance of one class to another class.
- Angular can tell which services a component needs.



A handwritten diagram in blue ink on a white background. It shows a rectangular box containing the text "Component" followed by a line of code "{Constructor(service)}". The word "service" is written in green ink and is underlined with three green lines.

```
import { DemoService } from '../services/demo.service';
```

```
export class DemoComponent {  
  constructor(private demoService: DemoService) { }
```

# How to create service

CLI command: `ng generate service serviceName` or `ng g s serviceName`

```
demo.service.ts ✕  
  
1  import { Injectable } from '@angular/core';  
2  
3  @Injectable({  
4    providedIn: 'root'  
5  })  
6  export class DemoService {  
7    constructor() { }  
8  }
```

# Mark Service as injectable

- When we create Service from angular CLI, by default the newly created service is marked as injectable at root (application) level.
- **@Injectable** decorator tells Angular that this Service is injectable.

```
TS demo.service.ts x
1  import { Injectable } from '@angular/core';
2
3  @Injectable({
4    | providedIn: 'root'
5  })
```

- We can also mark service as injectable at Module level and Component level.

# Inject Service at Module level

TS demo.module.ts ✕

```
1  import { NgModule } from '@angular/core';
2  import { CommonModule } from '@angular/common';
3  import { DemoComponent } from './demo.component';
4  import { DemoService } from '../services/demo.service';
5
6  @NgModule({
7    declarations: [DemoComponent],
8    providers: [DemoService],
9    imports: [CommonModule]
10 })
```

# Inject Service at Component level

TS demo.component.ts ✕

```
1  import { Component, OnInit } from '@angular/core';
2  import { DemoService } from '../services/demo.service';
3
4  @Component({
5    selector: 'app-demo',
6    templateUrl: './demo.component.html',
7    styleUrls: ['./demo.component.less'],
8    providers: [DemoService]
9  })
```

# Demo

- Demo to create a custom service.



# Forms

- Forms are used to collect the data from the user.
- Using Angular forms we can:
  - Capture user input events from the view,
  - Validate the user input,
  - Create a form model and data model to update,
  - Track changes in input data.
- Angular provides two different approaches to handling user input through forms:
  1. **Template-driven forms** approach
  2. Model-driven forms approach (**Reactive forms**)

# Template-driven forms

- All things are defined in Templates hence very little code is required in the component class.
- Easy to use.
- Suitable for small and very basic forms.
- Two-way data binding (using ngModel directive).
- Unit testing is challenging as compared with Reactive forms.

# Template-driven forms

- Prerequisite :
  - We need to import FormsModule in our application's module.

```
TS demo.module.ts ✕  
  
1  import { NgModule } from '@angular/core';  
2  import { CommonModule } from '@angular/common';  
3  import { DemoComponent } from './demo.component';  
4  import { FormsModule } from '@angular/forms'  
5  @NgModule({  
6    declarations: [DemoComponent],  
7    exports: [DemoComponent],  
8    imports: [ CommonModule, FormsModule]  
9  })  
10 export class DemoModule { }
```

# Steps for Template-driven forms

- Create form element with local template-reference variable.
- Bind form-data with ngModel directive.
- Pass template-reference variable on submit the form to component.

demo.component.html ✕

```
3  <div class="form">
4    <form #regForm='ngForm' (ngSubmit)="onClickSubmit(regForm)">
5      <div><input type="text" placeholder="First name" name="fname" ngModel></div>
6      <div><input type="text" placeholder="Last name" name="lname" ngModel></div>
7      <div><input type="text" placeholder="Email ID" name="email" ngModel></div>
8      <div><button type="submit">Submit</button></div>
9    </form>
10 </div>
```

# Demo

- Demo for template-driven forms.

# Validation - Template-driven from

- To add validation to a template-driven form, we need to add the same validation attributes as we use with native HTML form validation.
- Angular uses directives to match these attributes with validator functions in the framework.
- Every time the value of a form control changes, Angular runs validation and generates either a list of validation errors, which results in an INVALID status, or null, which results in a VALID status.
- We can then inspect the control's state by exporting **ngModel** to **local template variable**.

```
<input type="text" placeholder="First name" name="fname" [(ngModel)]="formModel.fname" #fname=ngModel required>  
<span *ngIf="fname.invalid && fname.touched">This field is mandatory</span>
```

# Demo

- Demo for Validation of Template-driven forms.

# Model-driven (Reactive) forms

- In Model-driven forms, a Model created in .ts file is responsible for handling all user interactions and validations.
- For that, we need to create a model using Angular's built-in classes like
  - **formGroup** and **formControl**
- With this approach, we create the tree of form controls and bind them into HTML form controls.
- It is More flexible.
- More component side code and less HTML code.
- It is Easier for unit testing.



# Building blocks of Reactive forms

- There are three main building blocks to create Model-driven (Reactive) forms.
  1. FormControl class
  2. FormGroup class
  3. FormArray class

# Form Control

- The **FormControl** is an object that encapsulates all the information related to the single input element.
- It Tracks the value and validation status of each input element of the form.
- Import and define a new instance of **FormControl** in **component**:

```
import { FormControl } from '@angular/forms';
```

```
name = new FormControl('');
```

- Use **[formControl]** directive in HTML input element to bind newly defined formControl so that we can track/set/validate input value of **name**:

```
<input type="text" [formControl]="name">
```

# Form Control

- We can use FormControl to set the value of the Form input field.

```
this.name.setValue('Rinkesh');
```

- We can also track the value of input field:

```
this.name.value      // returns value of the input field
this.name.errors      // returns the list of errors
this.name.dirty       // true if the value has changed (dirty)
this.name.touched     // true if input field is touched
this.name.untouched   // true if input field is untouched
this.name.valid        // true if the input value passed the validation
this.name.invalid     // true if the input value failed the validation
```

# FormGroup

- FormGroup is a **group** of **FormControl** instances.
- Generally forms have more than one field.
- It is helpful to have a simple way to manage the FormControls together.
- **For example:** We have three input fields: Street, City and State. So we can group these controls together and name it as “Address”:

```
Street: <input type="text" formControlName="street">  
City: <input type="text" formControlName="city">  
State: <input type="text" formControlName="state">
```

- **formControlName** directive is used with **FormGroup** to bind the input.

# FormGroup

- In component, we have to import FormGroup and create an instance of it by passing multiple FormControl into it:

```
import { FormGroup } from '@angular/forms';
```

```
address = new FormGroup({  
  street : new FormControl(""),  
  city : new FormControl(""),  
  pinCode : new FormControl("")  
})
```

```
this.address.value;  
// returns object {street:'', city:'', state:''}
```

# FormArray

- **FormArray** is an array of **FormControls**.
- It is similar to **FormGroup** but the difference is that we can dynamically add or remove **FormControls** from the array of **FormControls**.
- It is useful when the number of input fields may vary.
- **For example:**
  - We have list of checkbox fields to select color of the product.
  - one product is available in Black and green color while another product is available in Black, green and red color.
  - So we can add/remove red color option in the list of checkbox fields.

```
<input type="checkbox" formControlName="black"> Black  
<input type="checkbox" formControlName="green"> green
```

# FormBuilder class

- The FormBuilder provides mechanism that shortens creating instances of a FormControl, FormGroup, or FormArray.

```
import { FormBuilder } from '@angular/forms';
```

```
constructor(private formBuilder: FormBuilder) { }
```

```
address = this.formBuilder.group({  
  |   | street: [''],  
  |   | city: [''],  
  |   | state: ['']  
  |   |});
```

# Which one is better?

- Which one is better Template-driven forms or reactive forms?
  - It depends on our requirement.
  - Both are different approaches for different requirements.
  - We can use any of them as per our requirements.
  - Even we can use both of them in one application.
    - For example:
      - Use Template-driven form for login form
      - Use Reactive form for user registration form.



# Demo

- Demo for:
  - Reactive forms
  - Reactive forms with validation
  - Reactive forms with custom validation
  - Reactive forms setValue, patchValue and reset

# RxJS (Reactive Extensions for JavaScript)

- **RxJS** is a library for programming using observables that makes it easier to compose asynchronous and event based programming code.
- The essential concepts in RxJS :
  1. Observable
  2. Observer
  3. Subscription
  4. Operators
  5. Subject
  6. Schedulers
- RxJS official site: <https://rxjs-dev.firebaseapp.com/guide/overview>

# (1) Observable

- An **Observable** is basically a function that can return a stream of values to an observer over time, this can either be synchronously or asynchronously.
- Observables provide support for passing messages between **publishers** and **subscribers** in the application.
- Publishers return Observable along with data and subscribers receive the observable using subscribe() method.
- Observables offer significant benefits over other techniques for event handling, asynchronous programming, and handling multiple values.

## (2) Observer

- Observer is a consumer of values those are delivered by Observable.
- Observer is an object with three callback methods:
  1. **next** - called when new value captured successfully
  2. **error** - called when error occurred
  3. **complete** - called when publisher have no more values to send

## (3) Subscription

- Subscription is an object that represent the execution of Observable.
- Observable method executed only if it is subscribed by subscription.
- Subscription has two methods:
  1. **subscribe()** - to get values from observable
  2. **unsubscribe()** - to release resources or cancel Observable executions

## (4) Operators

- Operators are methods used to modify or filter the observable values and return the updated values as a new observable.
- For example in Observable method we get values from Web API through http methods and if we want to filter or transform the API response before using it in our application then we can use operators to modify the response and create a new observable with updated API response that can be subscribed.

# Types of Operators

1. Filtering operators
2. Transforming operators
3. Error handling operators
4. Conditional and Boolean Operators
5. Mathematical and Aggregate Operators
6. Creating operators
7. Utility operators
8. Combination operators

Visit below link for more details:

<https://rxjs-dev.firebaseapp.com/guide/operators>

## (5) Subjects

- Subject is a special type of Observable that allows values to be multicast to many Observers.
- Every instance of Subject is an Observer. It is an object with the methods `next(v)`, `error(e)`, and `complete()`. To feed a new value to the Subject, just call `next(theValue)`, and it will be multicast to the Observers registered to listen to the Subject.



## (6) Schedulers

- A scheduler **controls** when a subscription starts and when notifications are delivered.
- Using Scheduler we can store tasks in queue and execute them based on the priority and other criteria.
- Reference:
- <https://rxjs-dev.firebaseapp.com/guide/scheduler>
- Example:
- <https://stackblitz.com/edit/typescript-jexdny?file=index.ts>

# HttpClient Module

- Most front-end applications communicate with backend services over the HTTP protocol.
- Angular provides HttpClient module to communicate external web services or api by through HTTP protocol.
- We can execute HTTP methods like GET, POST, PUT, DELETE etc. to get data from server side application, create new instance of data, update and delete the data.
- Steps to use HttpClient:
  - Import HttpClientModule in our app module or feature module.
  - Inject HttpClient in our application service.
  - Use http methods of HttpClient in our service.

# Import HttpClientModule

TS app.module.ts ✕

```
1  import { BrowserModule } from '@angular/platform-browser';
2  import { NgModule } from '@angular/core';
3  import { AppComponent } from './app.component';
4  import { DemoModule } from './demo/demo.module';
5  import { HttpClientModule } from '@angular/common/http';
6
7  @NgModule({
8    declarations: [ AppComponent ],
9    imports: [ BrowserModule, DemoModule, HttpClientModule ],
10   providers: [],
11   bootstrap: [AppComponent]
12 })
13 export class AppModule { }
14
```

# Inject HttpClient and use its http method

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

```
constructor(private httpClient: HttpClient) { }
```

```
public getData( ): any{  
    return this.httpClient.get(this.apiUrl);  
}
```

# Observables

- Observables provide support for passing messages between publishers and subscribers in your application.
- Observables are declarative that is, you define a function for publishing values, but it is not executed until a consumer subscribes to it.
- The subscribed consumer then receives notifications until the function completes.
- Dependency:
  - Import **Observables** from **rxjs** library.
  - **RxJS** (Reactive Extensions for JavaScript) is a library for reactive programming using **observables** that makes it easier to compose asynchronous or callback-based code.

# Observables in Angular

- Angular makes use of observables as an interface to handle a variety of common asynchronous operations.
- For example:
  - The EventEmitter class extends Observable.
  - The HTTP module uses observables to handle AJAX requests and responses.
  - The Router and Forms modules use observables to listen for and respond to user-input events.


# Observables setup for http call

TS demo.service.ts ✕

```
4 import { Observable } from 'rxjs';  
5 import { map } from 'rxjs/operators';
```

```
public getEmployeeList(): Observable<Employee>{  
  console.log("Inside method that returns Observable");  
  let httpGetUrl = this.config["WEB_API_URL"] + "/Employee";  
  return this.http.get(httpGetUrl).pipe(map((response:Employee ) => response));  
}
```

TS demo.component.ts ✕

```
13  public getEmployeeList(): void {  
14     this.demoService.getEmployeeList()  
15     .subscribe(data => {this.employeeList = data;},  
16     error =>{alert("Error: " + error);});  
17 }
```

# Error handling

TS demo.component.ts ✕

```
9   export class DemoComponent {
10     public employeeList: any;
11     constructor(private demoService: DemoService){ }
12
13     public getEmployeeList(): void {
14       this.demoService.getEmployeeList()
15         .subscribe(data => {
16           this.employeeList = data;
17         },
18         error => {
19           alert("Error: " + error);
20         });
21     }
```



# Demo

- Demo for HTTP calls with Observables

# forkJoin()

- forkJoin() method helps us to call multiple asynchronous calls in parallel and get response of all calls combined in single array.

```
private joinCall() {  
    const getEmployees = this.empService.getEmployees();  
    const getUsers = this.empService.getUsers();  
    forkJoin(getEmployees, getUsers)  
        .subscribe(responseArray => {  
            this.employeeList = responseArray[0];  
            this.userList = responseArray[1];  
            this.isLoading = false;  
        });  
}
```

# Demo

- Demo for `forkJoin()` method.

# HttpInterceptor

- Angular provides HttpInterceptor interface to intercept all http calls of application.
- For example:
  1. If we have to pass some parameter in request header (like API Authorization token) then instead of passing it in every http methods we can pass in interceptor at generic level only once.
  2. We can handle generic errors of http calls (like 404 page not found error) in interceptor only once instead of handling error in every http call methods.

# Setup for HttpInterceptor in app.module

TS app.module.ts x

```
6 import { HttpClientModule, HTTP_INTERCEPTORS } from '@angular/common/http';
7 import { HeaderInterceptorService } from '../services/header-interceptor.service';
8 import { ErrorInterceptorService } from '../services/error-interceptor.service';
```

```
providers: [
  {
    provide: HTTP_INTERCEPTORS,
    useClass: HeaderInterceptorService,
    multi: true
  },
  {
    provide: HTTP_INTERCEPTORS,
    useClass: ErrorInterceptorService,
    multi: true
  }
],
```

# Example of HttpInterceptor

TS header-interceptor.service.ts ✕

```
1  import { Injectable } from '@angular/core';
2  import { HttpRequest, HttpHandler, HttpEvent, HttpInterceptor } from '@angular/common/http';
3  import { Observable } from 'rxjs';
4  @Injectable({
5    providedIn: 'root'
6  })
7  export class HeaderInterceptorService implements HttpInterceptor{
8    constructor() { }
9    intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
10      request = request.clone({
11        setHeaders: { Authorization: `TestToken` }
12      });
13      return next.handle(request);
14    }
15  }
```

# Demo

- Demo for `HttpInterceptor`

# References:

- <https://angular.io/>
- <https://blog.angular.io/>
- <https://update.angular.io/>
- [https://en.wikipedia.org/wiki/Angular\\_\(web\\_framework\)](https://en.wikipedia.org/wiki/Angular_(web_framework))
- [https://en.wikipedia.org/wiki/Tree\\_shaking](https://en.wikipedia.org/wiki/Tree_shaking)
- [https://en.wikipedia.org/wiki/Dead\\_code\\_elimination](https://en.wikipedia.org/wiki/Dead_code_elimination)
- <https://jsonplaceholder.typicode.com/>
- <http://dummy.restapiexample.com/>



# Thanks you

- You can find and download all demo applications from below GitHub link:
  - <https://github.com/rinkeshpatel22?tab=repositories>
- Rinkesh Patel
- Ex.: 5216-C
- Email: [rinkeshp@cybage.com](mailto:rinkeshp@cybage.com)