To create new app - ng new instagram

To run the app - ng serve --open (opens app in browser)

To generate Component - ng g c Header

ng g c Header --standalone --inline-template (create component with template property, so we can add inline HTML, --standalone reduces complexity)

**Before Using Component:**

* Add selector tag to the root component’s HTML template.
* Import component to the root component (i.e; AppComponent)

**Components:**

* Class with @Component decorator, a selector, HTML template, CSS styles is a Component.
* Components are basic building blocks of angular app’s UI.
  + **Ex:** navbar, menu, header, footer, etc
* @Component **decorator** specifies metadata for component, including HTML, CSS templates.

**File Struture:**

* index.html - main html file of app.
* main.ts - entry point of app & runs when app is launched.
* styles.css - main css file of app, which has styles for our app.

**DataBinding:**

**Interpolation:**

* Insert the value of variable into HTML.
  + {{name}}
* Also used to set values of HTML attributes.

<img src = “{{imageURL}}“ alt = ”{{altText}}>

* Also used to evaluate expressions.

<h1> Welcome, {{firstName + lastName}} </h1>

**One Way Data Binding:**

* We can bind HTML DOM properties to corresponding values.
* To bind a property, we need to enclose it in []

<img [src] = “imageUrl”>, imageUrl: string = ‘tree.jpg’

<td [colSpan] = “columnsCount”>text</td>, columnsCount: number = 7

* + In modern Angular development it is used widely
* For HMTL attributes, DOM properties don’t exist. For example, ARIA attributes.
* For these cases bind attributes using the **attr. attributeName.**

<input type = “text” [attr. Aria-label] = “inputName”>

* We can also bind classNames & styles.

<p [class.highlight] = “isHighlighted”> Text </p>, isHighlighted: boolean = true

* + Highlight - is the class defined in styles.css.
* We can bind to multiple classes,

<p [class] = “myClasses”> text </p>, myClasses = [‘highlight’, ’uppercase’]

* + Highlight, uppercase are classes defined in styles.css.

**Two Way Binding:**

<input type="text" [(ngModel)]="name" />

<p>{{name}}</p>

name = ''; //ts file

* ngModel allows us to bind input to the name property & track its value.

**Event Binding:**

* Angular allows us to bind events to methods in our component.
* It makes our responsive to clicks, touches and other interactions.
* To bind an event use ()
  + **EX:** <button (click) = “login()”>- Click Me </button>

<p [class.red]="isRed">some text for our page</p>

<button (click)="change()">Switch Background Color</button>

**Directives:**

* In web pages there is need to repeat a certain structure multiple times on the page. To accomplish this Angular has Directives.
  + **EX:** images in gallery, data in table, posts on blog etc…
* To use directives, we need to import it from @angular/common & added it as a dependency using imports property.
* **ngFor:**
  + It is used to create a list of items by repeating a given HTML node.

<div \*ngFor="let item of products">{{ item }}</div>

* **ngIf:**
  + It allows us add/remove elements based on its condition.

<div \*ngIf="loggedIn">Welcome!</div>, loggedIn = true;

**Forms:**

* **Forms** are used for user input, such as user login details, posting a comment & much more.
* **Template-Driven-Forms:**
  + One type of forms in angular is template-driven, there are defined & maintained in the template.
  + To use this template-driven-form ,we first need to import the **FormsModule** & include its dependency in our component.

   <input type="text" [(ngModel)]="name" />

<p>{{name}}</p>

name = ''; //ts file

* + ngModel directive is used to bind input to the name property of our class & track its value.
* TO handle the **Submit** event of the form, we can bind the **(ngSubmit)** event of the form.

<form (ngSubmit)="showName()">

<input type="text" [(ngModel)]="name" name="name" />

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

</form>

**Validations:**

* + We can validate the form before submitting it.
  + Angular tracks the state of the form internally. In order to check its validity, we need to access the form using a reference, defined using **#.**
  + **ngForm** Is a directive that allows us to refer to the whole form & checks its properties using its name **myForm.**
  + We can disable & enable the button based on the validity of the form.

<form (ngSubmit)="showName()" #myForm="ngForm">

  <input type="text" [(ngModel)]="name" name="name" required />

  <input type="submit" value="Submit" [disabled]="!myForm.form.valid" />

</form>

* + **ng-valid** & **ng-invalid** classes can be used to style the form controls based on their state.
* **Reactive Forms:**
  + Template-driven forms are great for building small forms, but for large/multiple forms the code becomes very large & templates would be difficult to read & maintain.
  + To solve this we have reactive-forms.
  + To create these forms, we need to import **ReactiveFormsModule.**
  + These forms are controlled from component class, instead of template.
  + To create form controls we need to import FormControl class & create a instance as follows,

name = new FormControl('james');

* + After creating the control, we need to associate it with an element in HTML template.

<form (submit)="showName()">

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

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

</form>

<p>{{name.value}}</p>

* + - **[formControl]** is used to associate the input with the control of the class.

**Grouping Controls:**

* + - Most of the time, reactive forms contain multiple controls.
    - Angular allows us to group these controls under a **FormGroup.**
    - To define **FormGroup,** first import **FormGroup** class along with **FormControl.**
    - To define FormGroup, use its constructor & pass the corresponding **FormControl.**

<form [formGroup]="loginForm" (ngSubmit)="login()">

  <p><input type="text" formControlName="username" /></p>

  <p><input type="text" formControlName="password" /></p>

  <p><input type="submit" value="Submit" /></p>

  <p><input type="button" value="Reset" (click) = “loginForm.reset()”/></p>

</form>

// TS file

loginForm = new FormGroup({

username: new FormControl(''),

    password: new FormControl(''),

   });

  login() {

     alert(this.loginForm.value.username + ' | ' + this.loginForm.value.password);

  }

**Validations:**

* + To validate the inputs import **Validators** class.
  + To make a field required, use **Validators.required** in its constructor, as the 2nd parameter.
  + We can disable & enable the button based on the validity of the form.
  + **ng-valid** & **ng-invalid** classes can be used to style the form controls based on their state.

<form [formGroup]="loginForm" (ngSubmit)="login()">

   <p><input type="text" formControlName="username" /></p>

  <p><input type="text" formControlName="password" /></p>

   <p><input type="submit" value="Submit" [disabled]="!loginForm.valid" /></p>

</form>

  loginForm = new FormGroup({

    username: new FormControl('', Validators.required),

    password: new FormControl('', Validators.required),

  });

**Routing:**

* An Angular router is used to handle the navigation from one screen to another.
* With routing we can have persistent header for all screens & change main content of screen.
* To enable routing, first we need to import **RouterModule** from **@angular/router.**
  + Next import **RouterModule** as dependency to our root component.
* If our app has a **HeaderComponent** which is same for both **HomeComponent** & **ContactsComponent.**
  + Tapping them should navigate to corresponding component. Code for this navigation is,

//app.component.ts

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

import { HeaderComponent } from '../header/header.component';

import { RouterModule } from '@angular/router';

@Component({

  selector: 'app-root',

  standalone: true,

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

  imports: [HeaderComponent, RouterModule],

})

export class AppComponent {}

//app.component.html

<app-header />

<router-outlet />

* We need to define our routing configuration in **main.ts**. this is an array, which contains the routes.
* Each route is an object in the form of,
  + **{path: ‘string’, component: ComponentClass}**
* After above config, we need to import & use **providerRouter** to provide the routes to our app when launching **main.ts**

//main.ts

//HomeComponent & ContactsComponent have no config regarding routing

const routes: Routes = [

  { path: 'home', component: HomeComponent },

  { path: 'contacts', component: ContactsComponent },

];

bootstrapApplication(AppComponent, {

  providers: [provideRouter(routes)],

});

* Now to add the links, we first need to import the **RouterModule** in corresponding component.
* Now add the routes to the navigation links in **HTML** using the **routerLink** attribute.
* To display the component that corresponds to the route, angular uses the **<router-outlet/>** element in root component’s template.

//header.component.ts

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

import { RouterModule } from '@angular/router';

@Component({

  selector: 'app-header',

  standalone: true,

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

  styleUrls: ['./header.component.css'],

  imports: [RouterModule],

})

export class HeaderComponent {}

//header.component.html

<h1>My Header</h1>

<nav>

  <a routerLink="/home">Home</a>

  <a routerLink="/contacts">Contacts</a>

</nav>

* We can define the default path, using an empty set, which means **HomeComponent** will open by default.

{ path: '', component: HomeComponent },

* Apps need to handle incorrect urls, to display a custom error page when the user attempts to navigate to an unknown path.
  + For this, angular provides **wildcard routes,** defined by **\*\*** .
  + Mention wildcards always at the end, because if url match is found then angular won’t go through remaining urls

{ path: '\*\*', component: NotFoundComponent },

* We can define page title for each route to display in browser using **title** property in router config.
  + Each screen in our app should have a **unique title** so that they can be identified in **browser history**.

{ path: '', title: ‘Home Page’, component: HomeComponent },

* We can also navigate to routes when button is clicked.
  + We used **inject()** from **@angular/core** to get router instance & used its **navigateByUrl()** to navigate to path.

export class HomeComponent {

  router = inject(Router);

  navigate() {

    this.router.navigateByUrl('/contacts');

  }

}

//home.component.html

<button (click)="navigate()">Contacts</button>

**Notes:**

* Angular CLI creates all files & folders for angular app.
* Use the **import** property to add the component to another component as a dependency.
* Use selector tag to reference the component into the HTML template.
* To use directives, we need to import it from @angular/common & added it as a dependency using imports property.
* Routing notes,
  + Import **RouterModule** from **@angular/router.**
  + Define the routes config array, in the format **path: ‘string’, component: ComponentClass.**
  + Launch the app with routes config, using **providers: [providerRouter(routes)].**
  + Define the links where applicable using the **routerLink** attribute in html.
  + Use the **<router-outlet/>** to display component that corresponds to the path.
* More on Routing notes,
  + We can set default path using set of empty quotes ‘ ’.
  + We can set page titles using **title** property.
  + We can use wildcard path **\*\*** for all URLs that are not matched.
  + **navigateByUrl()** of router is used to navigate to a path programatically.