**Dec 14**

Please explain child routers again

this.router.navigate(['success', username]);

active = inject(ActivatedRoute);

Why did you put /login and not just login?

Create components

Add ROUTERS to App-routing-module.ts

Add Forms Module and ReactiveFormsModule to app-module.ts

You also import the routing module from import { AppRoutingModule } from './app-routing-module';

Every component will have the main files: html and ts

ts will have the form builder that will allow you to programmatically control on the client side. It can have the functions for events and validations that can be called from html.

The highlighted portions shows the components in ts file connect to the items in html file.

registerForm = this.\_builder.group({

    firstname: [''], lastname: [''], phone: ['']

This creates the form with the fields and default values.

<form [formGroup]="registerForm" (ngSubmit)="handleRegister()">

<input type="text" formControlName="firstname" placeholder="Enter username"

            class='form-control form-control-lg'><br />

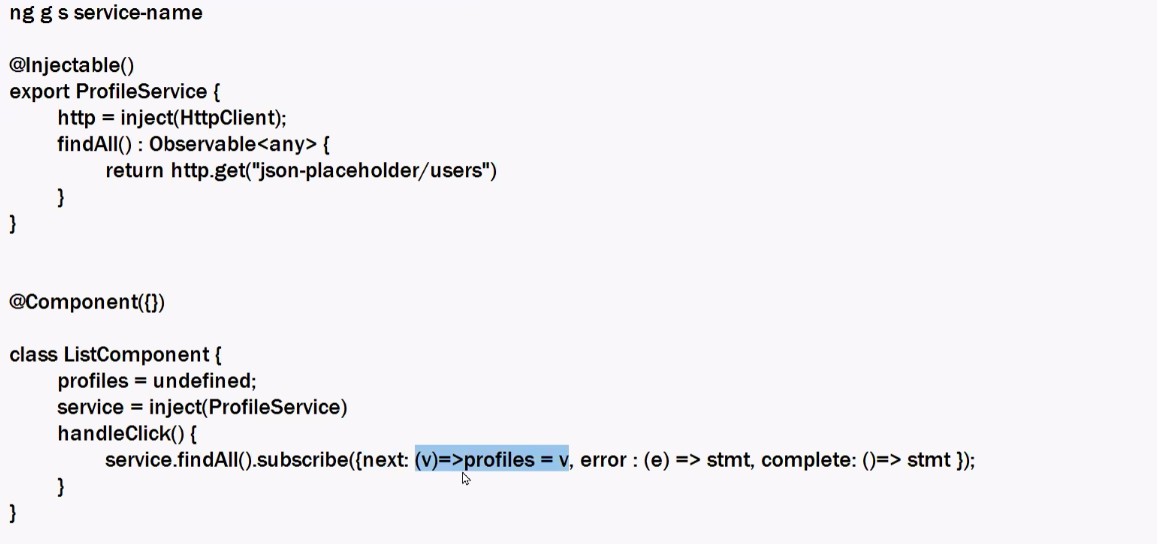
Html = you can use bootstrap to call submit, display options of buttons, text etc

Child Routing or Nested Routing

RouterLink aand routerobject(used inside code to manage programmatically) – It takes arrays.

Routers, router linkls, router outlets

Observables can be used only when they are subscribed by component class.



The colon in success/:user controls the dynamic path. User controls the dynamic path and accepts various values.

Activated Router can read router parameters like success/:user, that :

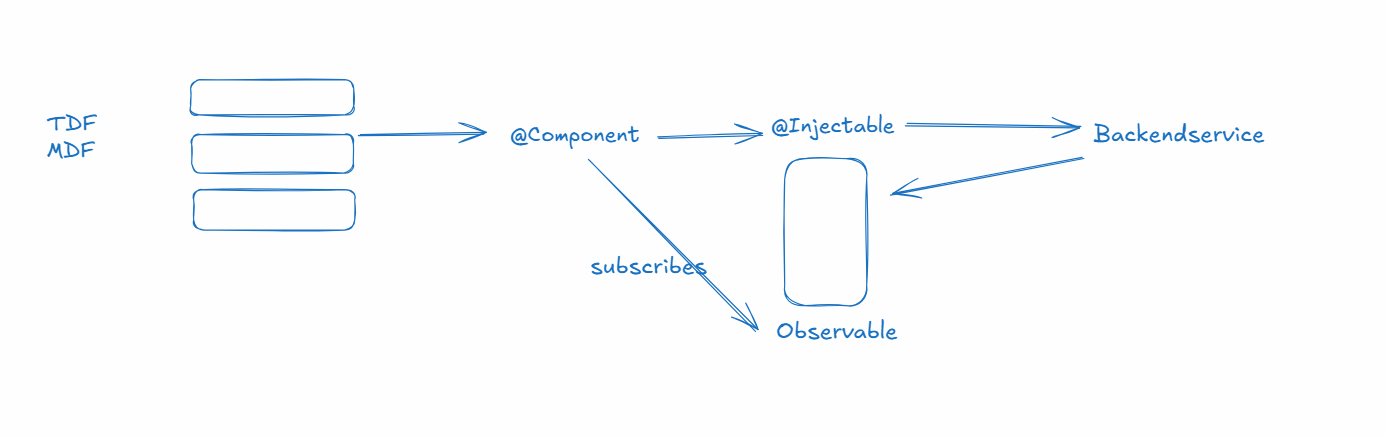
This.active.params is an observable object

**Session 18**

Session 18

14-12-2025

Flow of component to service to backend service





Angular Routers

It allows you to create a single page that can navigate from one component to another component without reloading the entire page, it uses a placeholder that can add components dynamically

router-outlet: It is a selector angular provides to load components dynamically.

Angular Router uses one of the file called app-routing-module.ts that has a routes variable which will have path & component configuration

routes : Routes = [   
 { path = “login”, component : Login}, { path = “register”, component: Register},  
 {path = “search”, component : Search }  
];

app.html

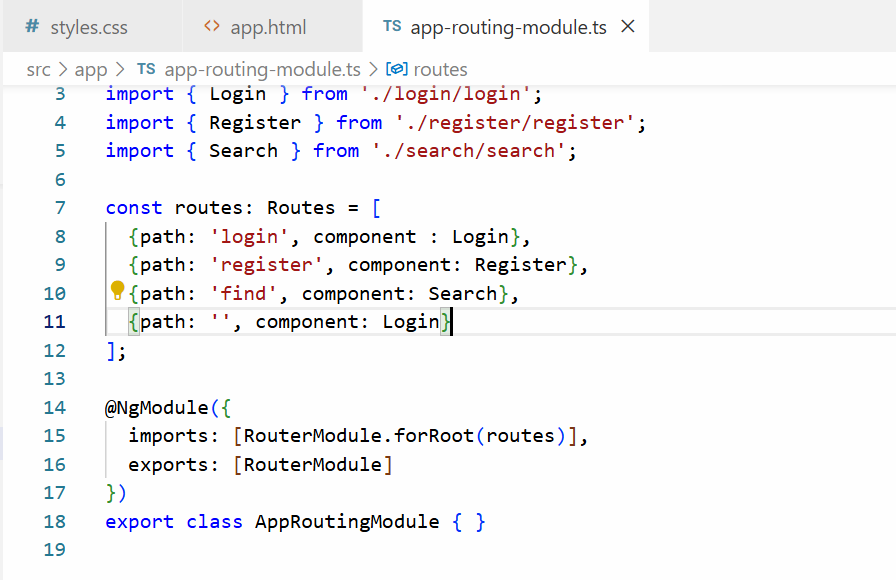
<a routerLink = “login”>Sign in</a>  
<a routerLink = “register”>Create</a>  
<a routerLink = “search”>Find</a>

<router-outlet></router-outlet>

Create 3 components

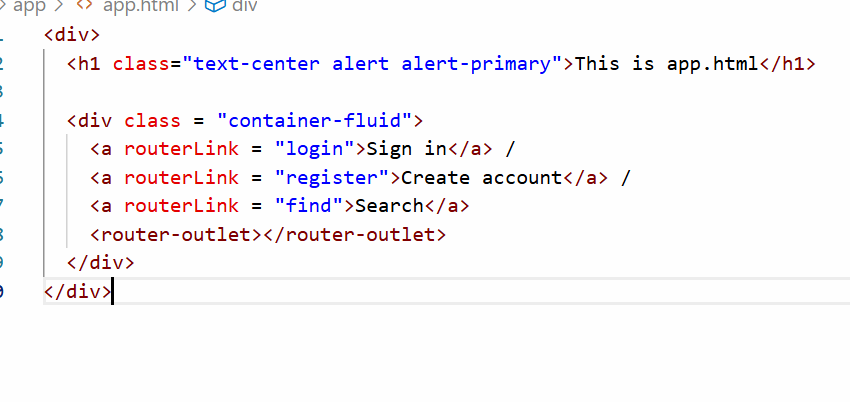
1. login
2. register
3. search

configure the path for the components in app-routing-module.ts

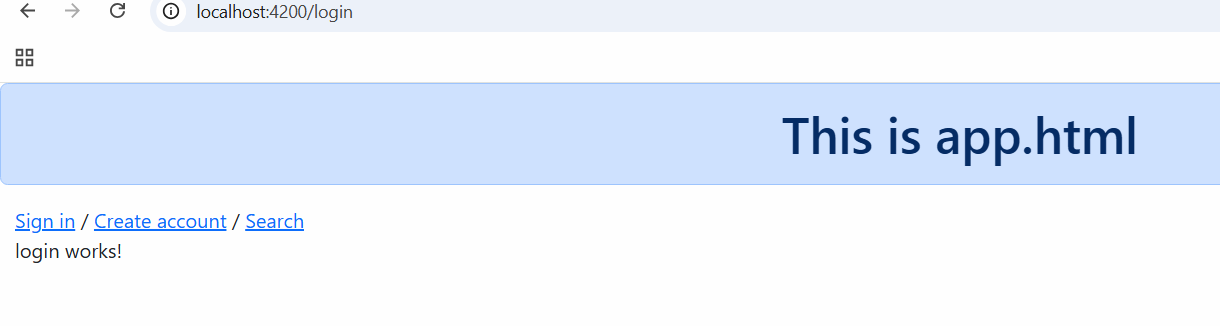


Use <router-outlet> and routerLink in the component to add in-app navigation in the root component

app.html



Output:

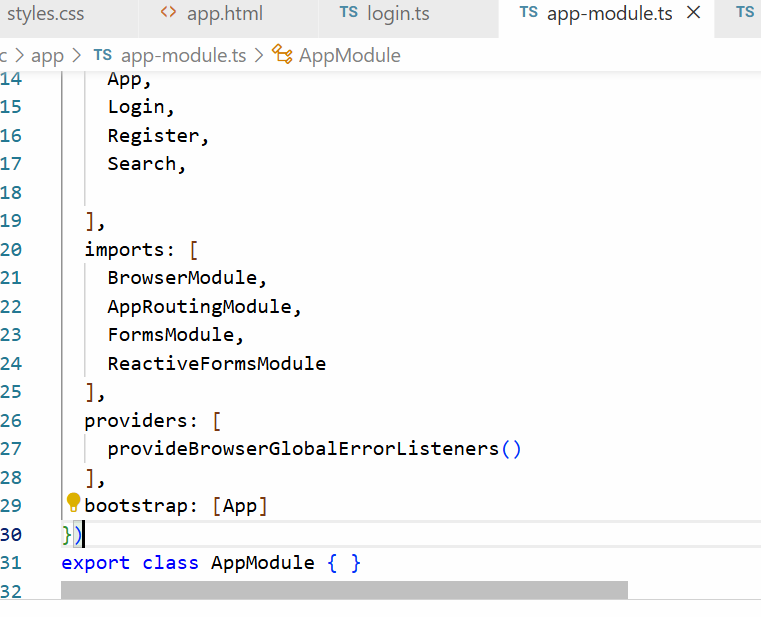


Activity: use Model driven form that creates Login Form, Registration Form and Search Form in the components Login, Register & Search, Apply bootstrap css to them to make them look good.

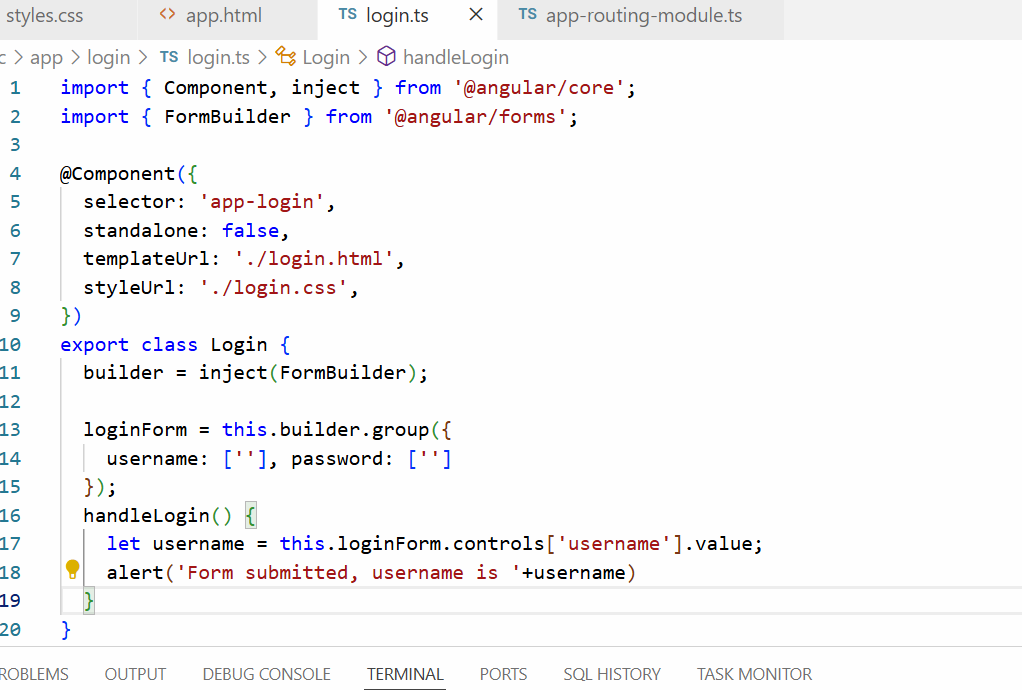
Note: When you submit any form just alert some message

Solution:

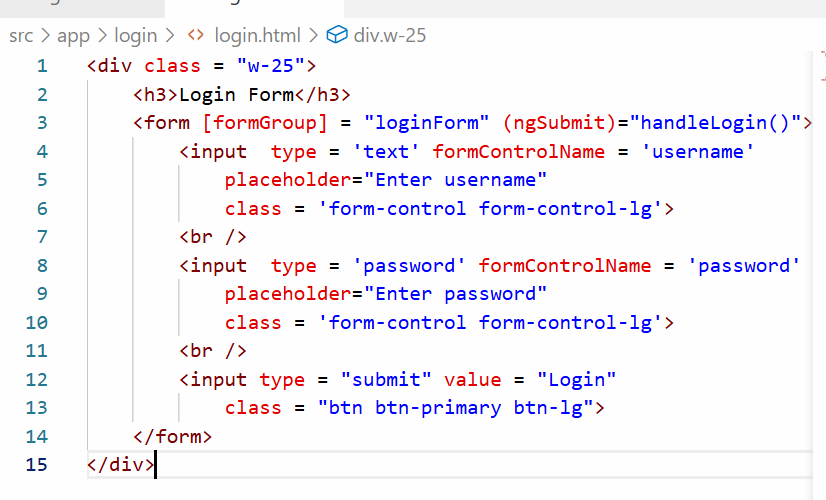
Add FormsModule & ReactiveFormsModule in app-module.ts



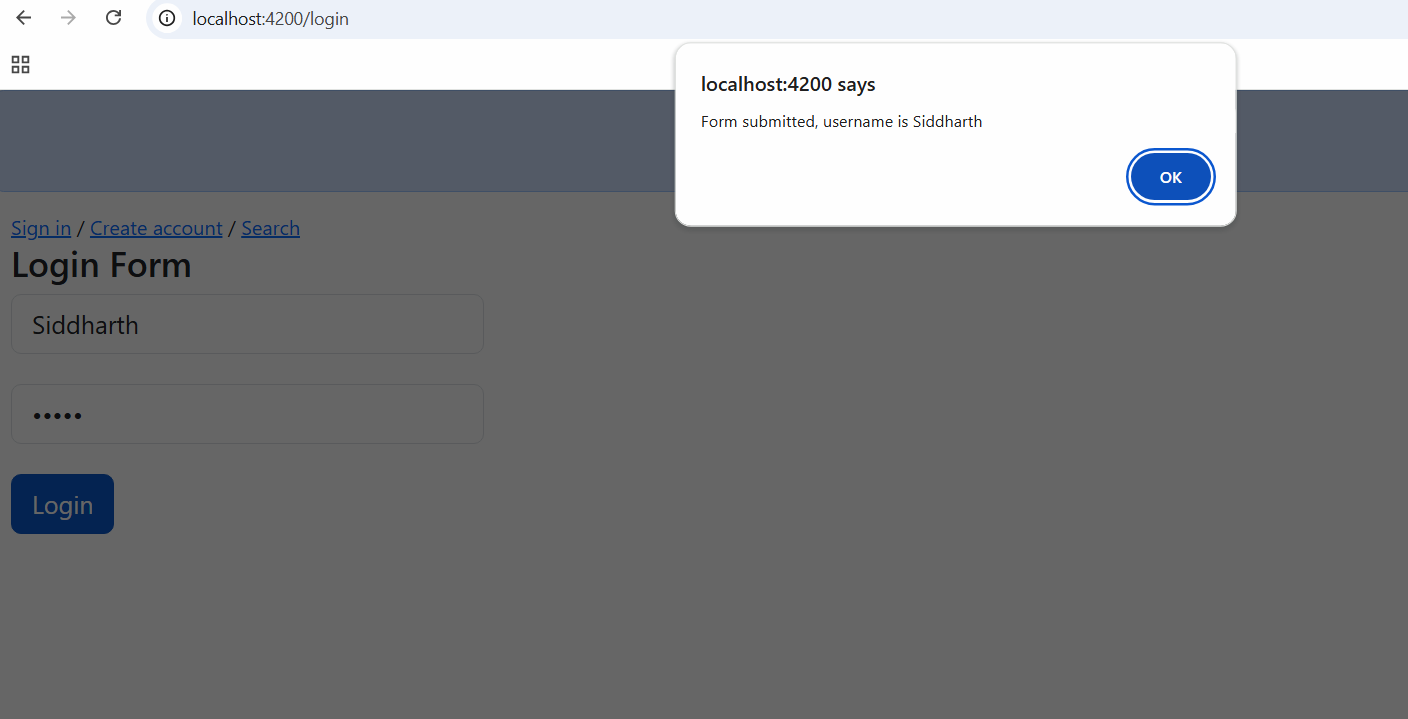
login.ts will have form group and form control and a function to print the username



login.html

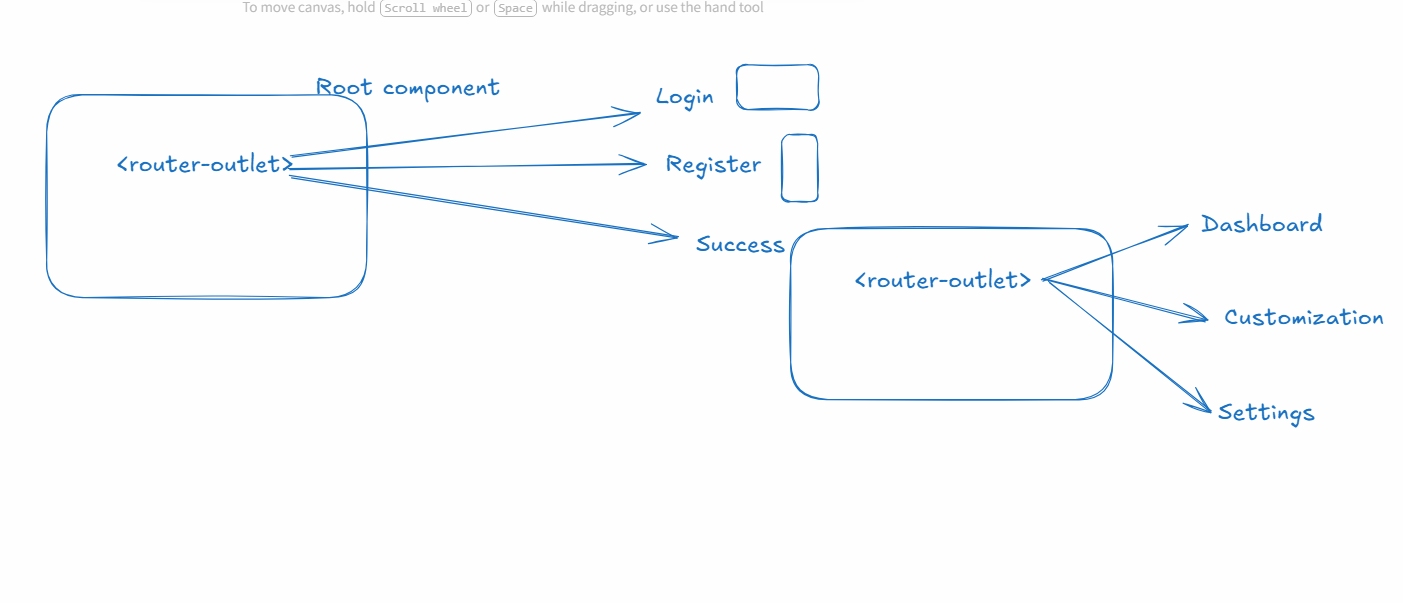


Output:



Child routing or nested routing

Child routes allow you to nest routes inside another route



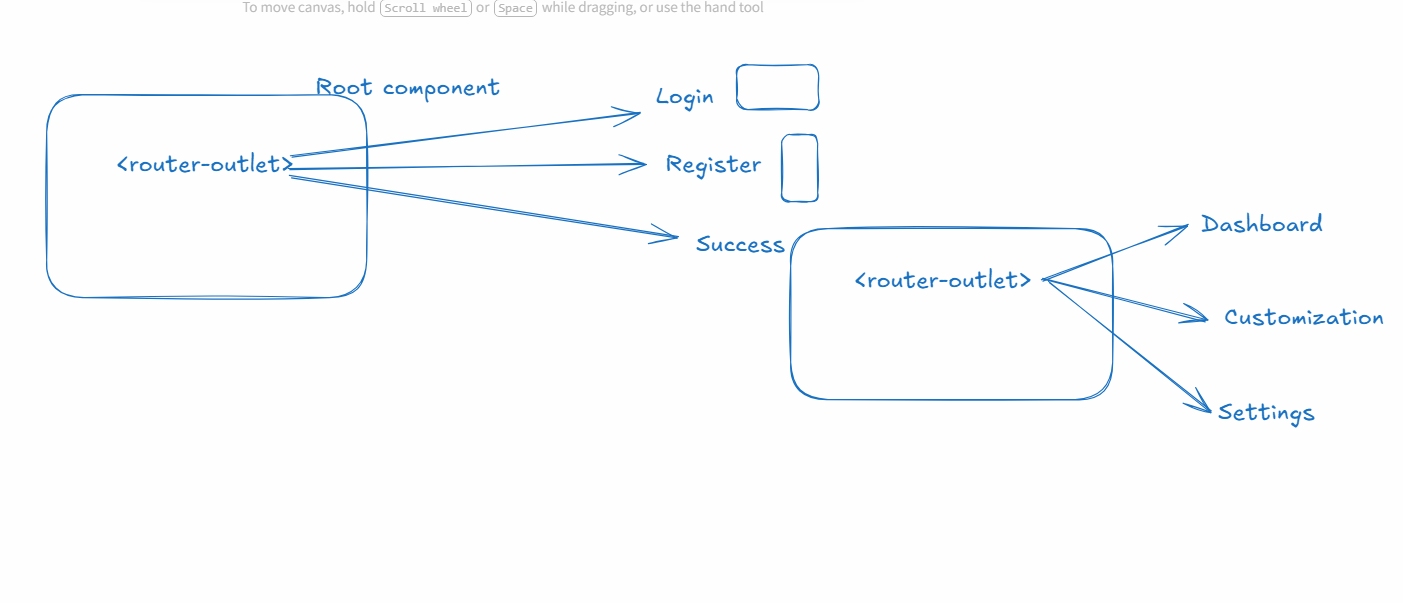
When it comes to navigating from one component to another component angular provides two types of navigation

1. routerLink: <a routerLink = “url”>LinkName</a>
2. Router object: It has a navigate([“url”]); method - this is used inside the code when the application wants to decide the navigation based on some condition

You just need to tell angular to inject the Router object using the following code

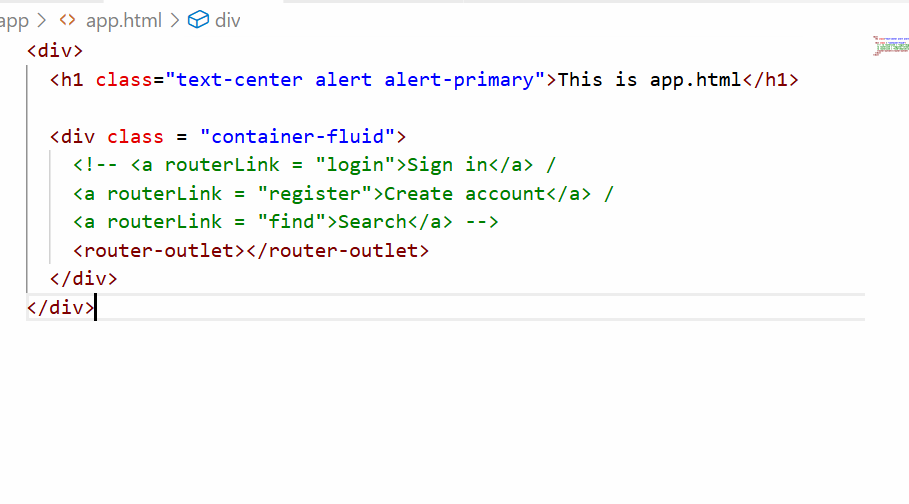
router = inject(Router);

To create the routes for the following diagram



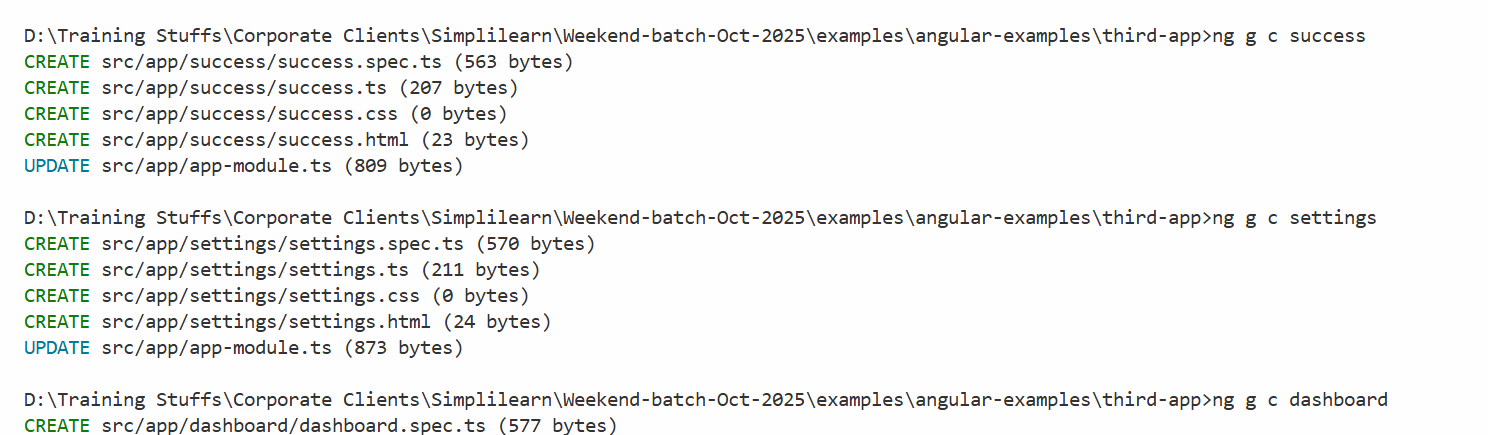
We need to use <router-outlet> that can navigate to 3 different components in Root component & success will have <router-outlet> that can navigate to other components, but those components are part of <router-outlet> inside success, not inside Root component

app.html



Components required

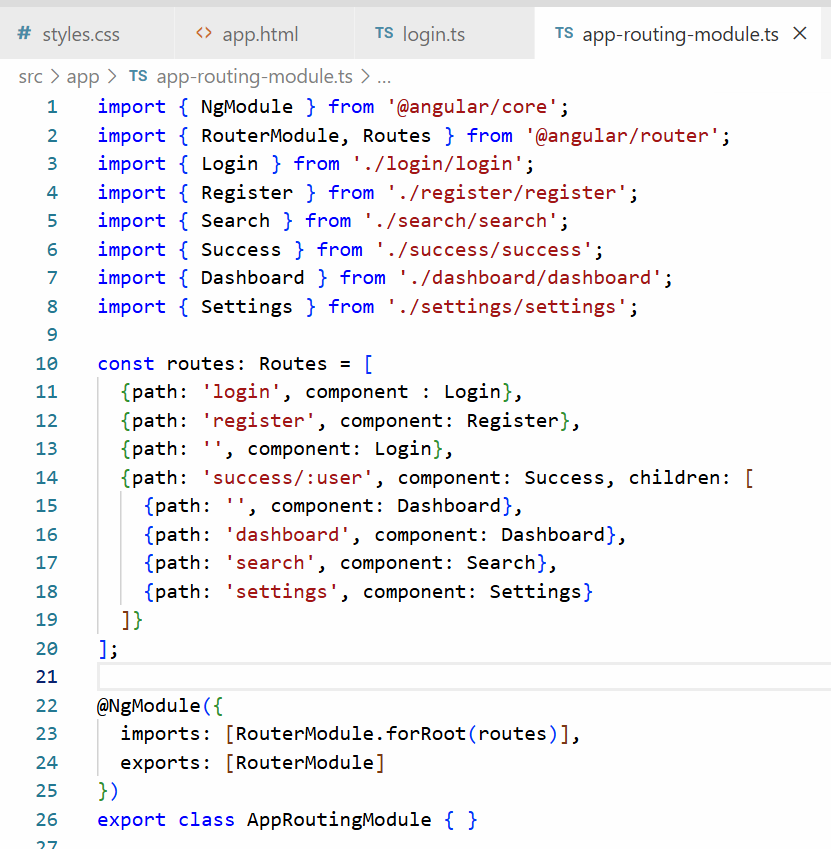
1. login
2. register
3. success
4. search
5. dashboard
6. settings



Child routing is configured with children property

{path : ‘success/:user’, component : Success, children : [ {…}, {…} ] }

We need to configure the routes once again to have child routes in the app-routing-module.ts



What success/:user does is it is a dynamic path, where user is a path parameter and can accept various values, it matches like success/Kishor, success/Atharv, success/Siddharth and so on, the user is parameter that holds value like Kishor, Atharv, Siddharth when the success components loads, so that you can read those parameters

What login component does

router = inject(Router)

username = usernameValue

if(valid) router.navigate([“success”, username]); success/username

else router.navigate([“/login”])

What success component does

active = inject(ActivatedRoute)

ActivatedRoute can read route parameter like success/:user, that :user value it can read using some built-in functions

active.params.subscribe(obj) -> returns the value associated with :user

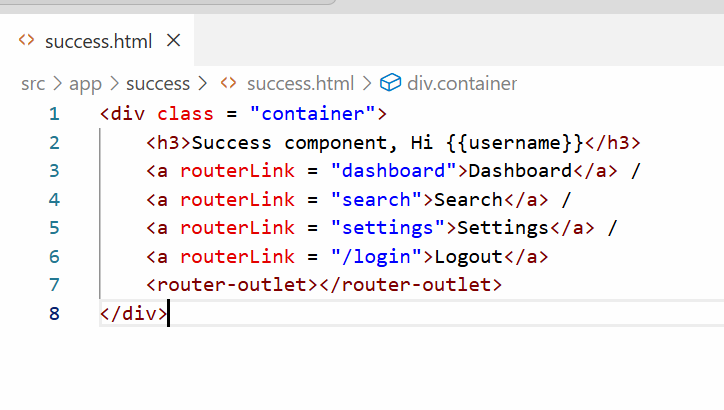
login.ts



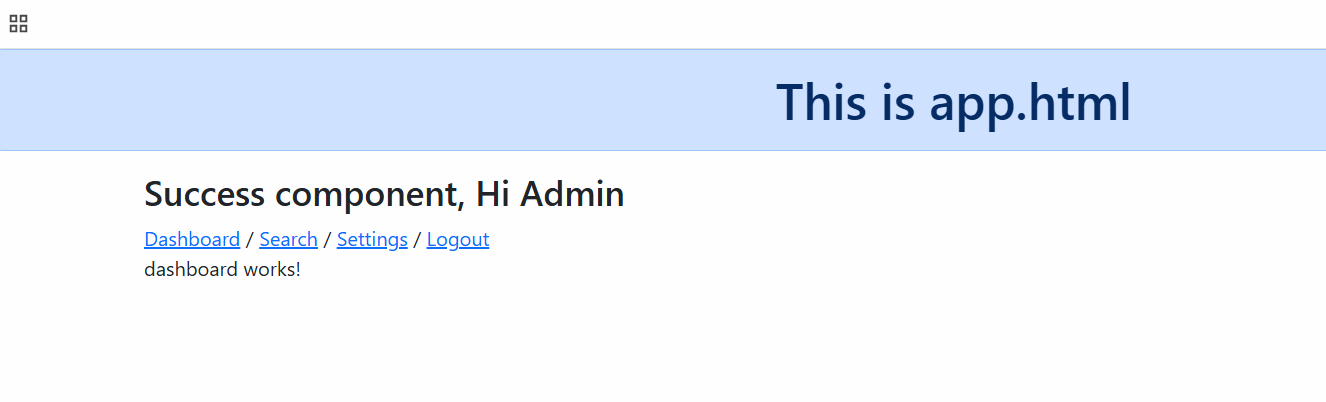
success.ts



success.html



Output:



**Dec 13**

Model driven forms

Template driven forms

Services Class – business logic that is called by component classes

Backend Services – HTTPClient to make HTTP calls

HTTPClient – Observable, subscribe

Angular used for single page applications,. UI or mobile, web, native.

Injectiable decorator that can be injected into various components

Dependency injection = the component class becomes dependent on the service class and this is called dependency injection. Angular Framework creates the object and supplies the object where requried.

Backend services = programs that would interact with DBs and other apis.

Observable, - container with the response from backend service. Note: Observable will not send request unless some one subscribes.

Session 17

13-12-2025

Validations on model driven form or reactive forms

The form control in the form builder can use Validators.compose([]) function which accepts array of validators

  userForm = this.builder.group({

    firstname:['', Validators.compose([Validators.required])], lastname:[''], phone : ['']

  });

Now angular made it a little simpler to apply validators just by adding nested array inside the form control value

userForm = this.builder.group({

    firstname:['', [Validators.required]], lastname:[''], phone : ['']

  });

mdf.ts

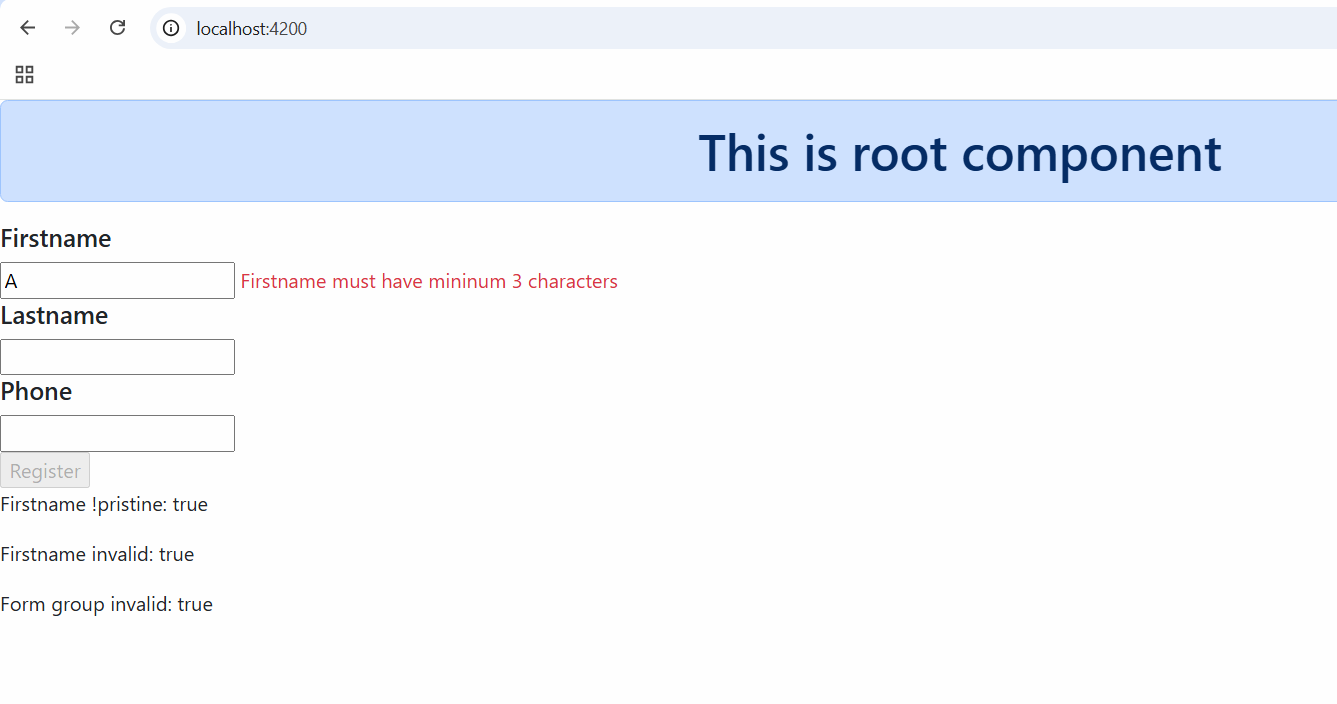


The pattern is a regular expression where ^ means start with a string and [0-9] means any digits between 0 to 9 and $ end of the string

mdf.html



Output:



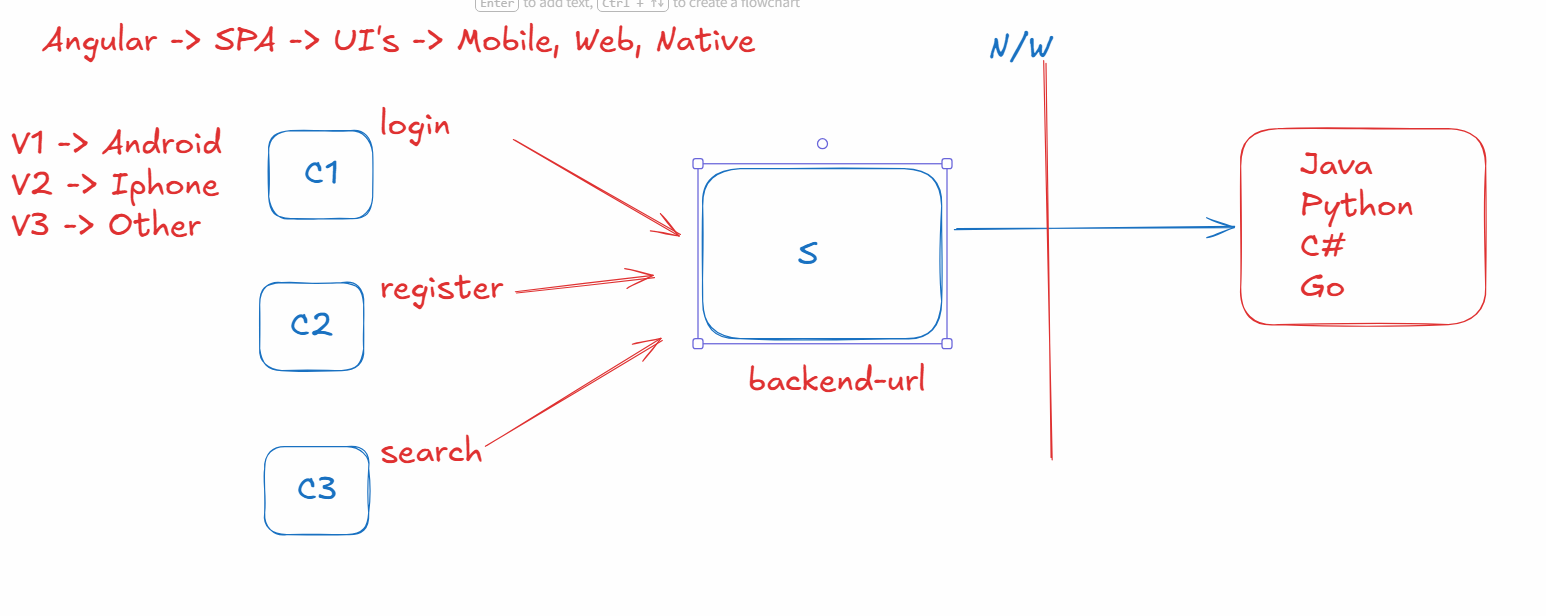
Angular Services

These are reusable logics that can be called by multiple components, you can use these to separate UI logics with business logics

Component class: It can have UI logics and the data the UI wants to maintain

Service class: It can have business logics that can be called by Component classes

Business logics are the logics that handle the data like invoking the backend services and performing some complex computation



In angular you can create services using ng g s service-name, this creates a class with @Injectable() decorator

Why @Injectable() decorator: this creates an object of the class automatically and any component who wants to use this object can just use using the inject() function to get the object, they don’t have to create the object

ex:   
builder = inject(FormBuilder): FormBuilder is an object which already available, instead of creating it we used the object

A service class looks like this

@Injectable()  
class ProfileService { … some logics … }

A component class can tell angular to supply the dependency of ProfileService which is called as Dependency Injection

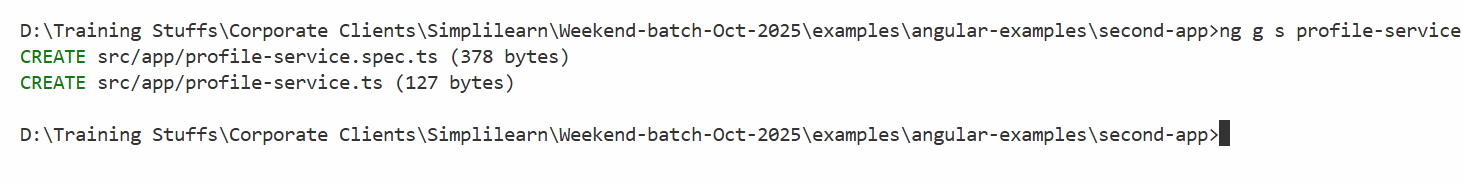
@Component({})  
class ProfileComponent {  
 service = inject(ProfileService)  
}

Dependency Injection: It is a process where we don’t create the object instead the framework creates the object & supplies the object

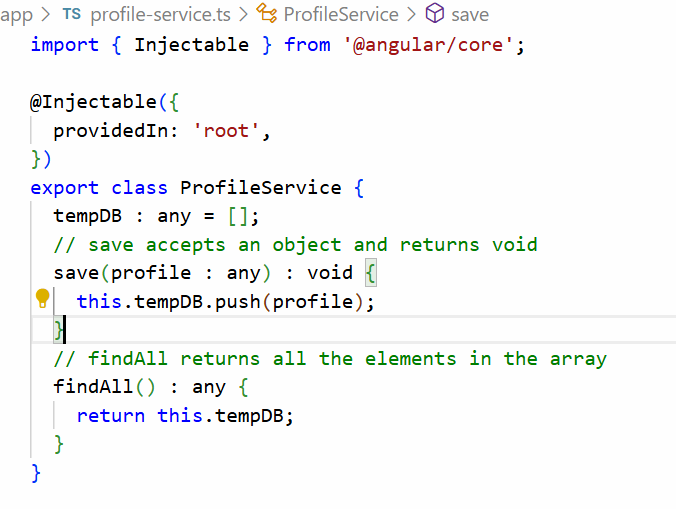
What inject() function does

It tells angular to supply the object

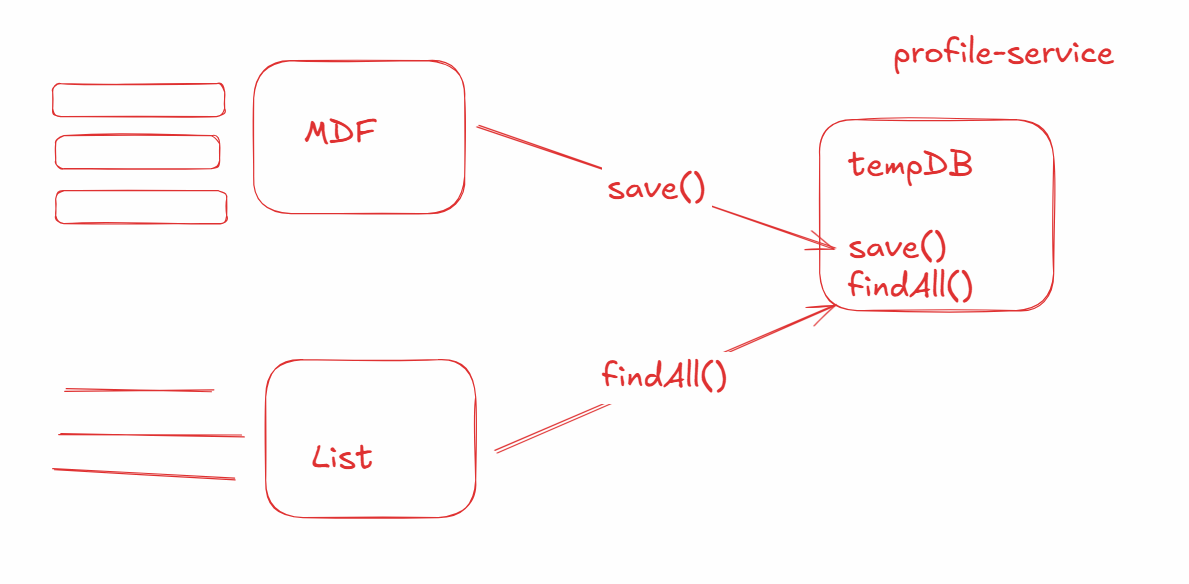
Creating a service with the name profile-service



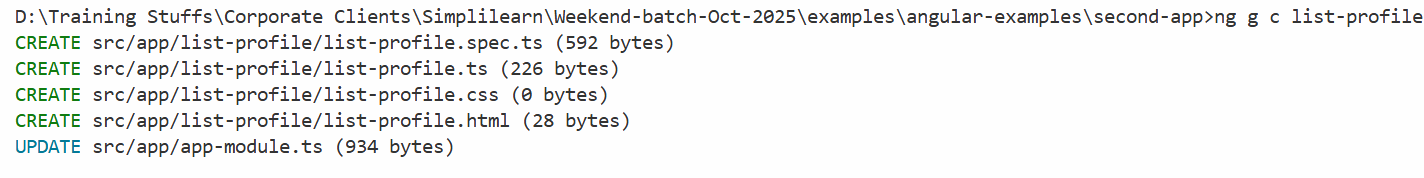
profile-service.ts



Let us have 2 components one invokes save() and the other invokes findAll()



We will create list-profile component



We will add mdf and list-profile in the app component

app.html



Invoke store in mdf and findAll in list-profiles

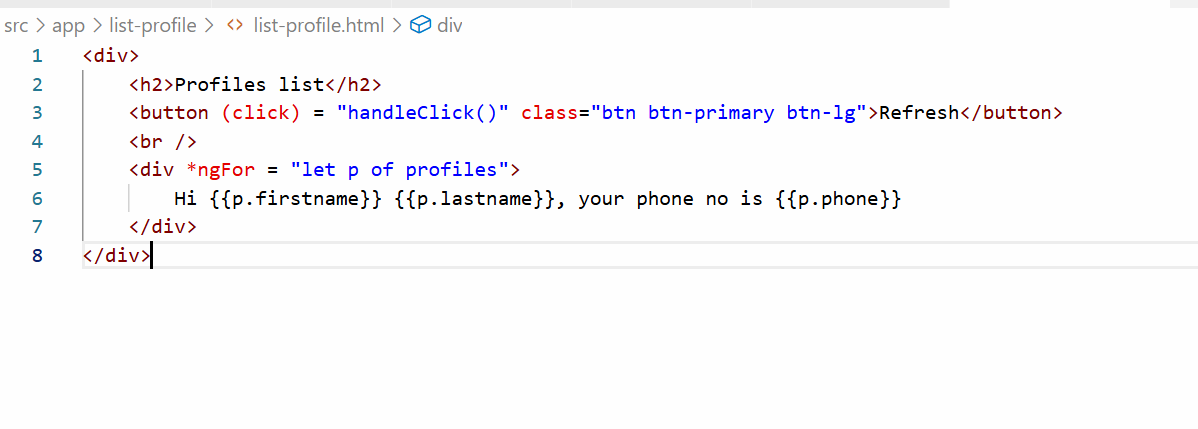
mdf.ts



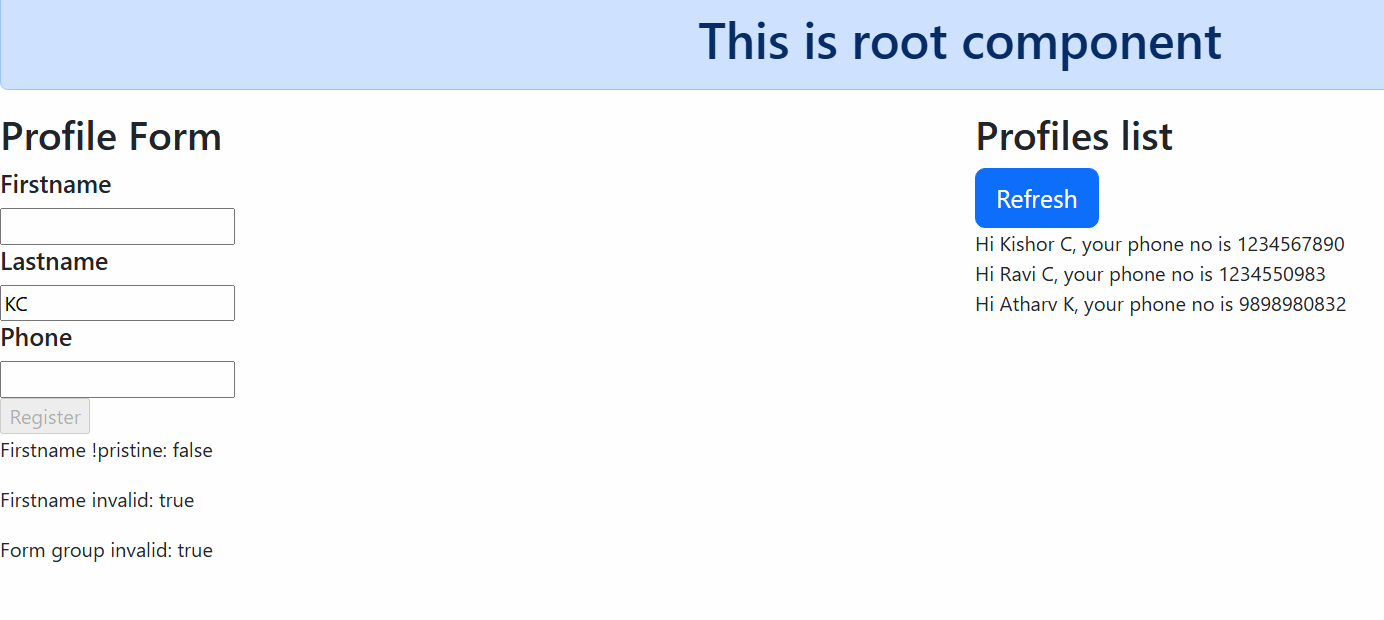
list-profiles.ts



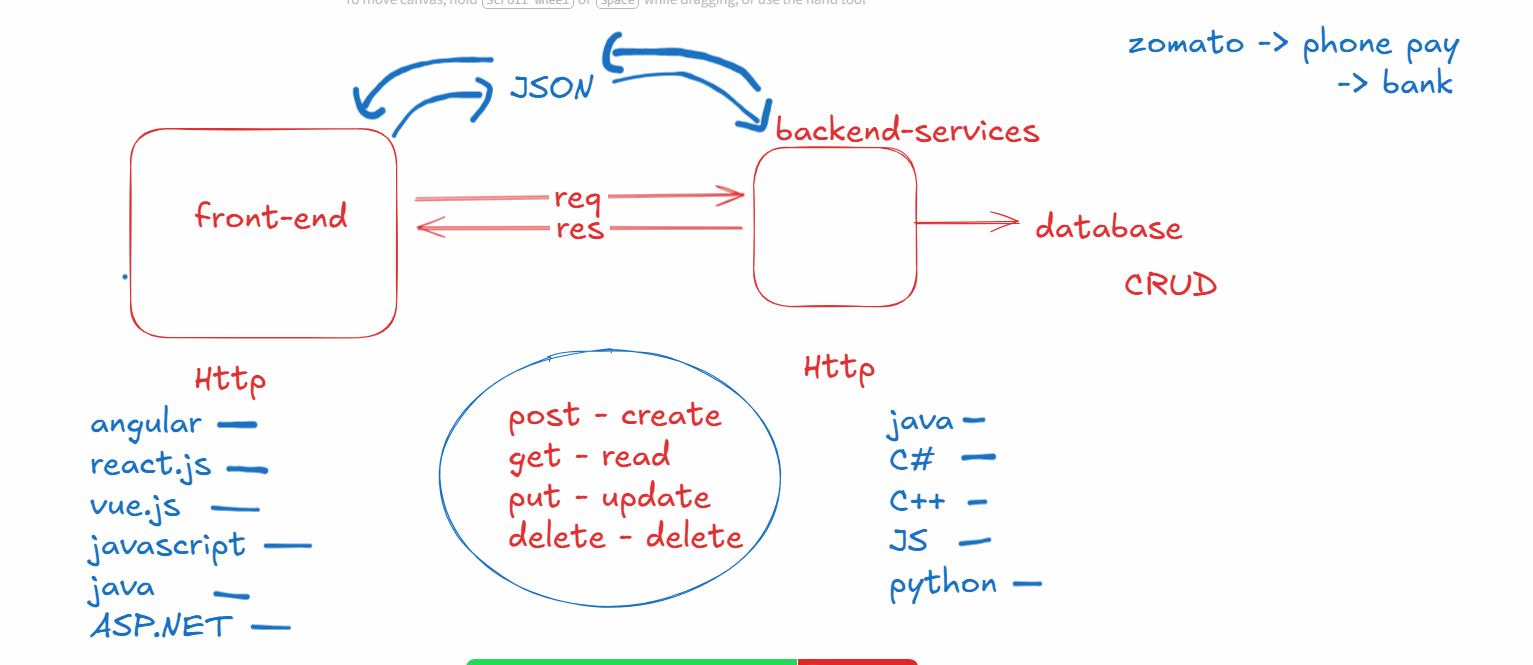
list-profile.html



Output:



Backend services



Backend services are the applications who accept the data from the front-end and share the data to the front-end, the common protocol both must use is HTTP and common structure the data must use is JSON

HttpClient: It is an object provided by angular to make HTTP calls and it provides methods like get(), post(), put() and delete()

All the HTTP methods of HttpClient returns an Observable

Observable: It is a container that will have response from the backend service, these will produce some response whenever the components subscribe to the observable

Note: Observable will not send request unless someone subscribes

When components subscribe to the observable they can receive success or error response

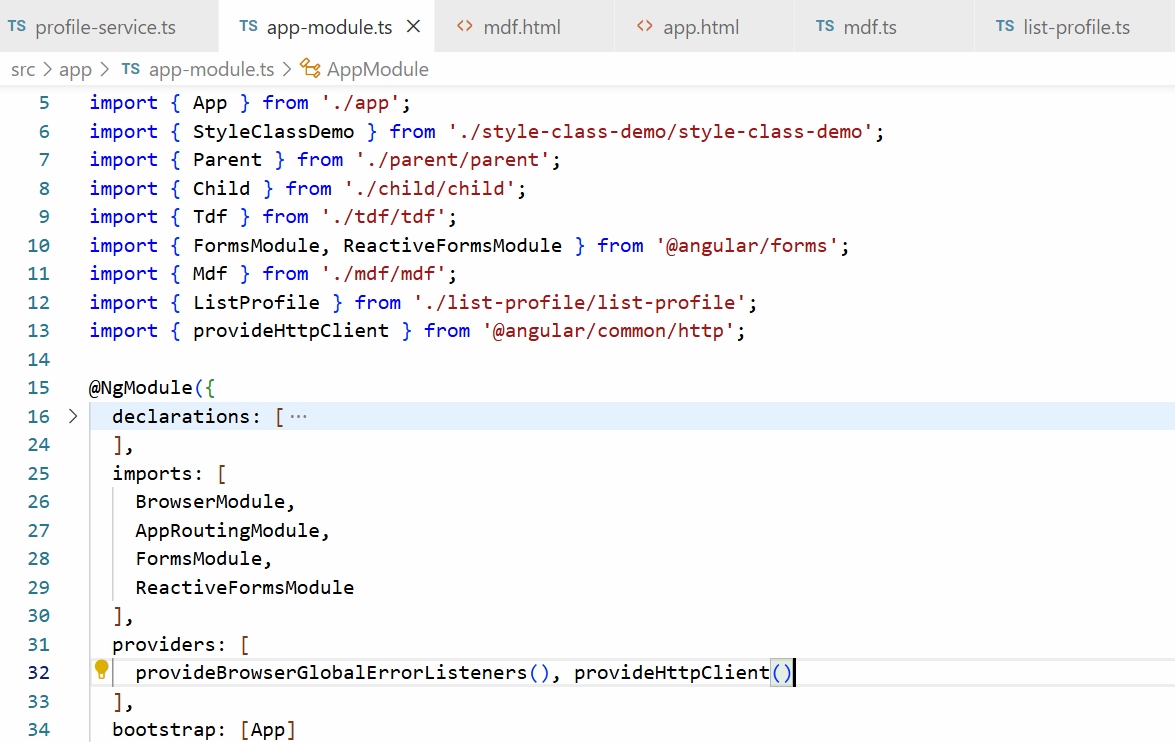
HttpClient object is created automatically by angular using a provider called provideHttpClient(), by default angular doesn’t create HttpClient object, hence you must use provideHttpClient() in the NgModule.providers property

Once angular creates HttpClient object you can use inject(HttpClient) to use the HttpClient object

Steps to access JSON Placeholder

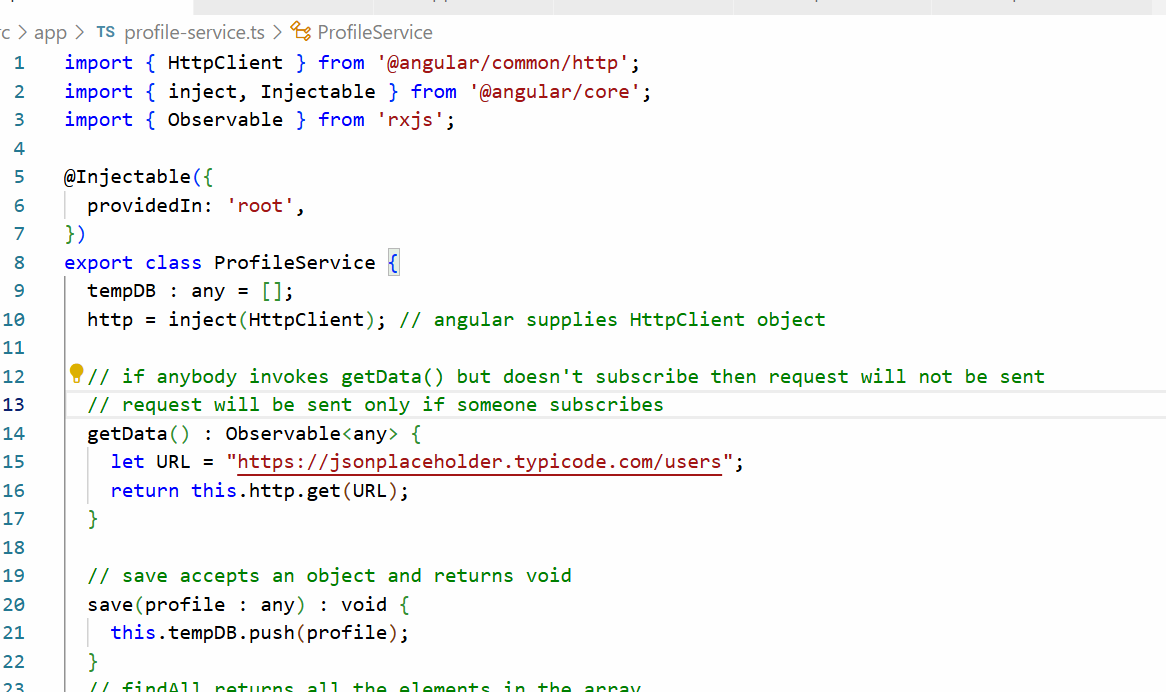
1. Mention provideHttpClient() in the @NgModule.providers
2. In the Angular service use inject(HttpClient)
3. Make HTTP requests in the Angular service that returns Observable
4. Components can subscribe to the Observables to receive the data, observable object has a subscribe() method that initiates the request to the backend

Mentioning the provideHttpClient() in the @NgModule.providers



In the Angular Service use HttpClient object & Make HTTP calls

profile-service.ts



Subscribing to the Observable

subscribe(obj): subscribe sends a request to the resource and accepts an object with 3 properties

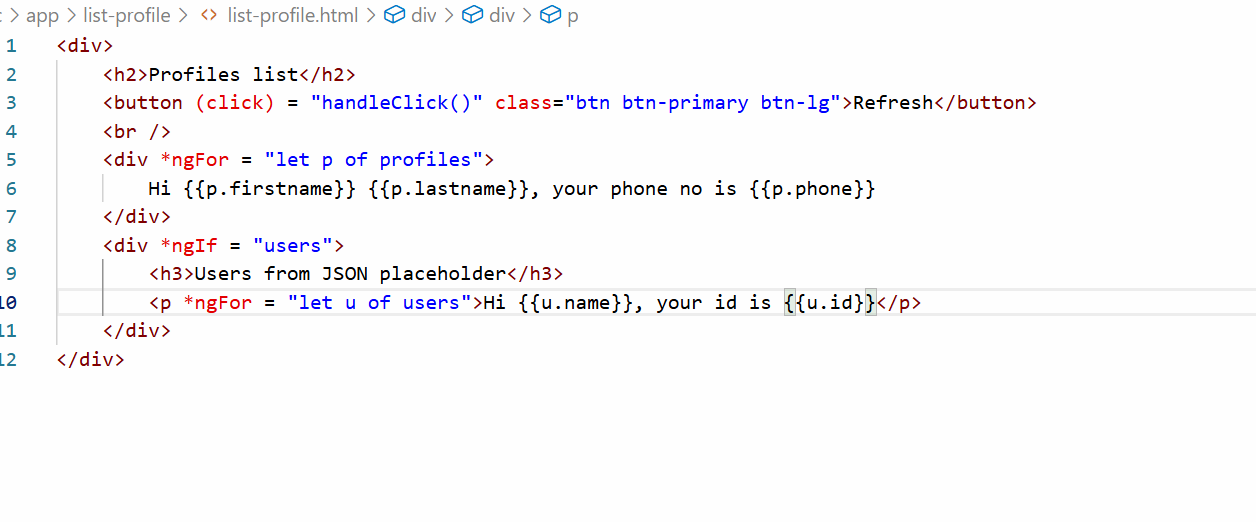
obj : { next : (v) => {}, error : (v) => {}, complete: () => {})

list-profiles.ts

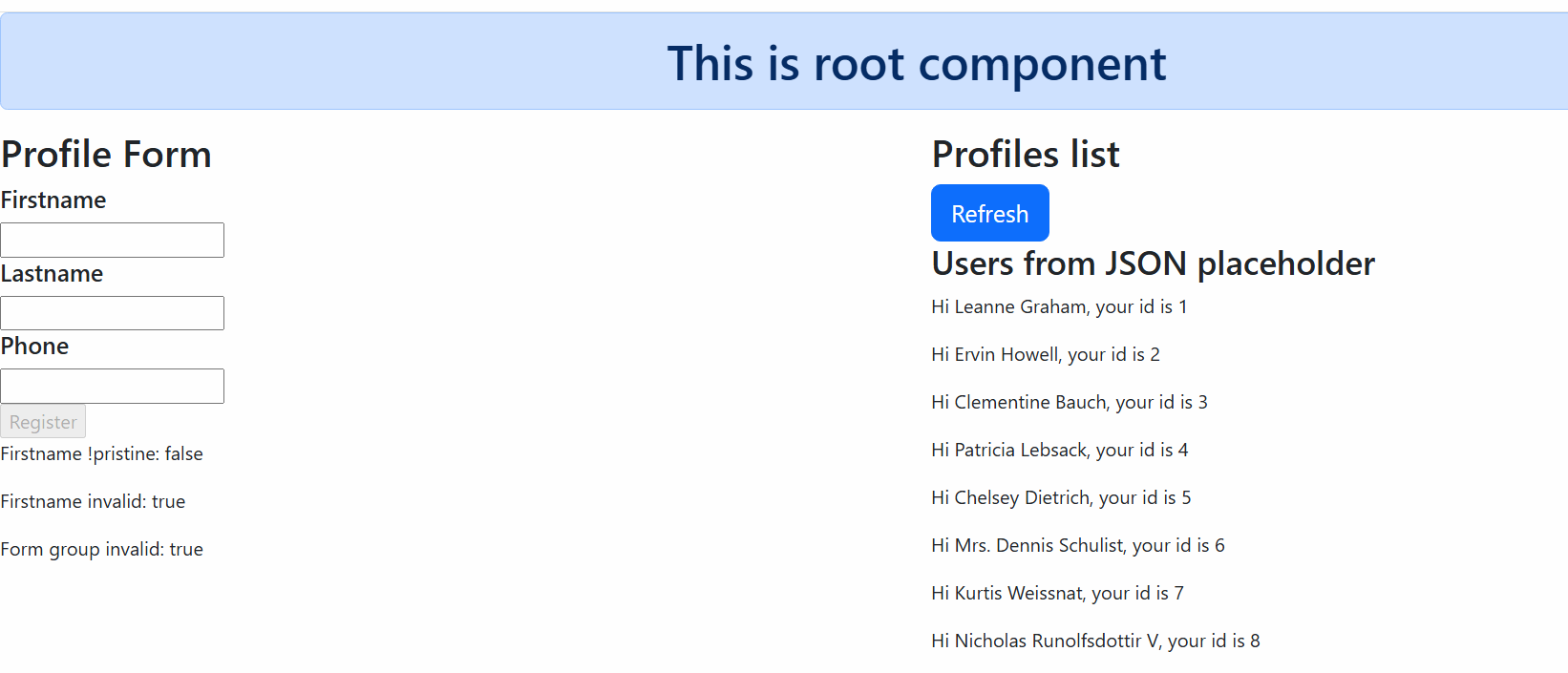


List all the users

list-profile.html



Output:



We have tried only get(URL) method from HttpClient but there are other methods in HttpClient like post(), put() and delete() they take more than one parameters but they all return Observable

* post(URL, data)
* put(URL, data)
* delete(URL, data)

Since get(URL) doesn’t accept request body it takes only one parameter

**Dec 07 2025 – Angular forms and sharing data between child and parent.**

Session 16

07-12-2025

Angular Forms

They are just similar to HTML forms that helps you to enter user input, however they come with angular data binding features, angular forms can be created in 2 ways

1. Template Driven Forms (TDF) - used for simple forms
2. Model Driven Forms (MDF) or Reactive Forms - used complex forms

Template Driven Forms

It is a way to build forms where logic and validations are handled within the HTML template, it uses angular directives (ngForm) to track the forms state / data.

You must import a module called FormsModule to use the template driven forms, this must be imported in the @NgModule.imports property which is present in the app.module.ts

With the help of FormsModule you can use ngForm, ngModel.

Thing to do while using TDF

1. Add FormsModule of @angular/forms in the @NgModule.imports
2. Use two way databinding to bind the form data to the component property
3. Use ngForm to track forms data so that it can collect all the form data in a single javascript object which will help you to manage with single object

<form #user = “ngForm” (ngSubmit) = “handleForm(user.value)”>

<input [(ngModel)] = “firstname” name = “fn”> -> binds to firstname  
<input [(ngModel)] = “lastname” name = “ln”> -> binds to lastname  
<input [(ngModel)] = “phone” name = “ph”> -> binds to phone

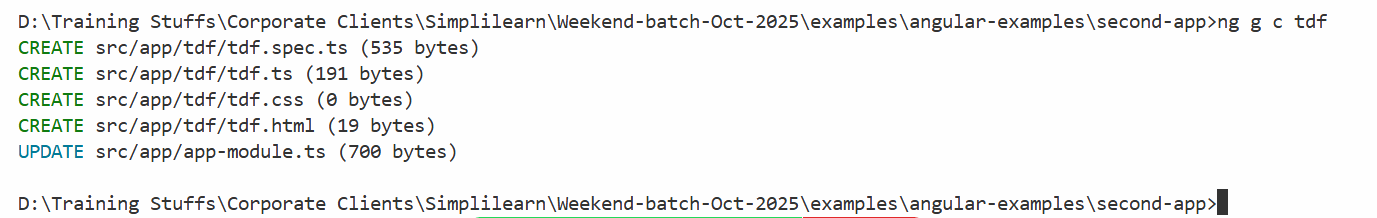
</form>

user = { fn : value, ln: value, ph : value}

In the component class you will three properties

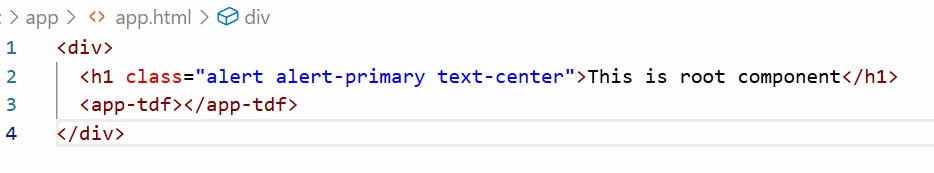
@Component({})  
class UserComponent {   
 firstname, lastname, phone;  
 // manually you will create an object, if you don’t use ngForm  
 userObj = {fn : firstname, ln : lastname, ph : phone }  
}

Create a component named tdf

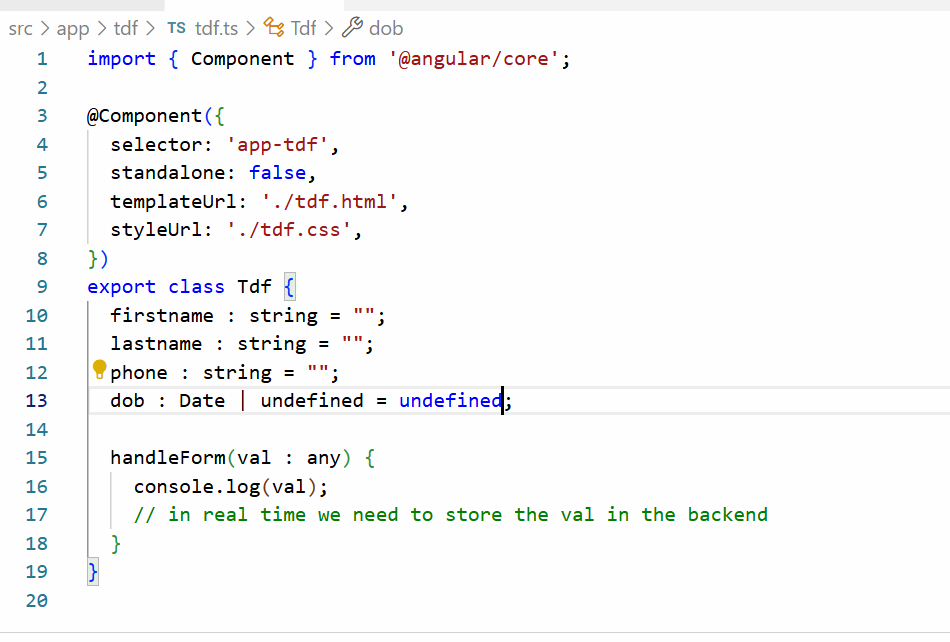


Add the component to the root component

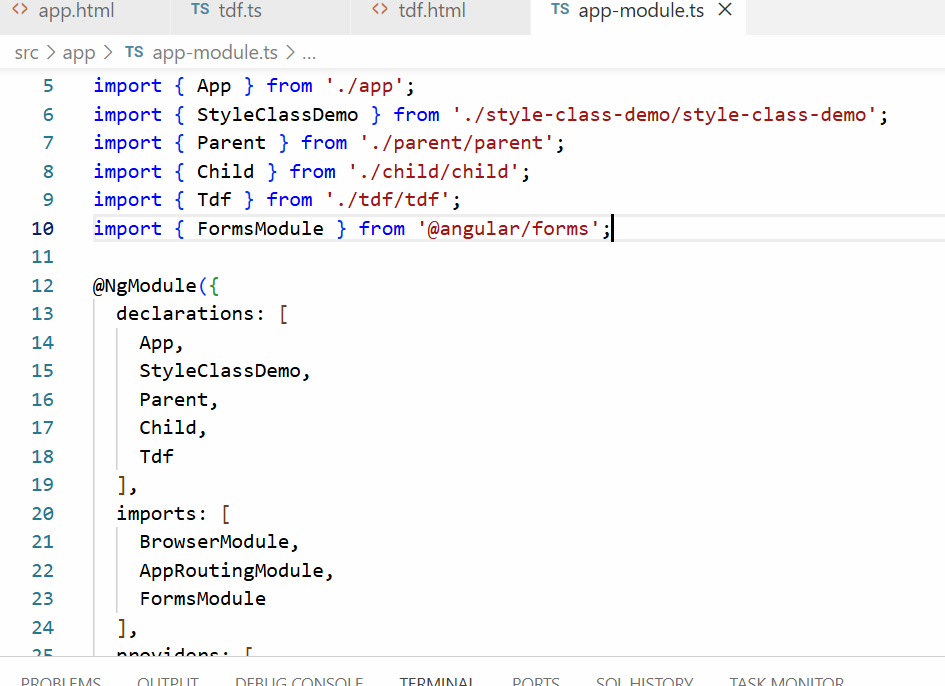
app.html



tdf.ts



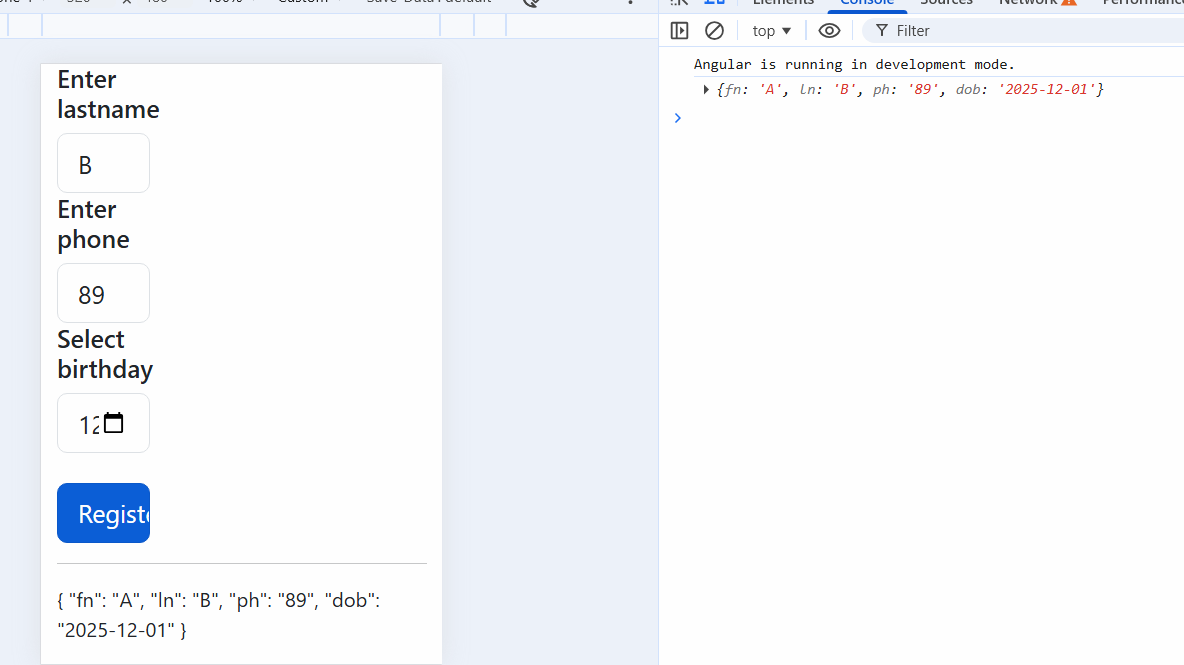
Add FormsModule of @angular/forms in the app-module.ts



tdf.html



Output:



Template Driven Form Validation

In TDF validators like required, maxLength, minLength are specified in the HTML template itself, but you must use ngModel directive instance to validate the input element, because the NgModel directive instance provides inbuilt properties like valid, invalid, value, pristine, touched, dirty which you can get from the official angular document

The input element must use ngModel with a template reference so that you can use that name in other places ex: to display error messages

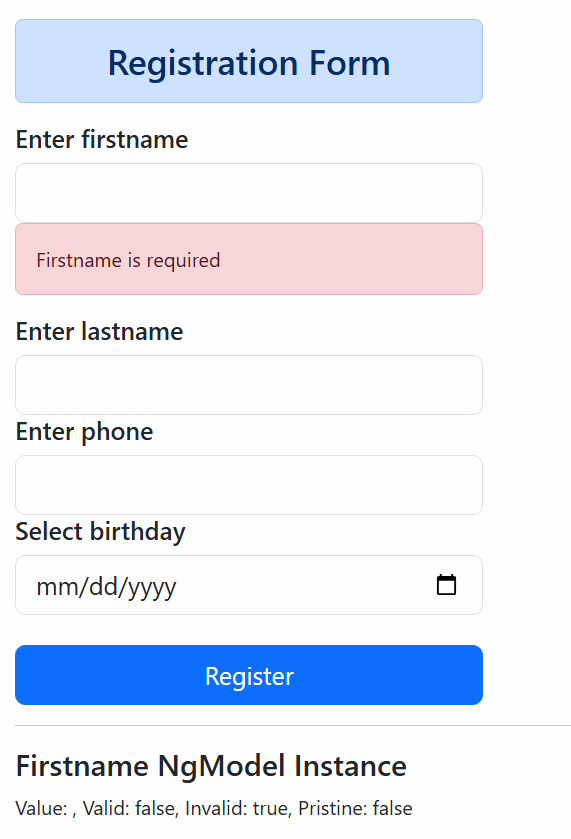
<input type = “text” [(ngModel)] = “firstname” name = “fn” #first=”ngModel” required maxLength = “15” minLength = “5”>

first.value  
first.valid  
first.invalid  
first.pristine

tdf.html



Output:

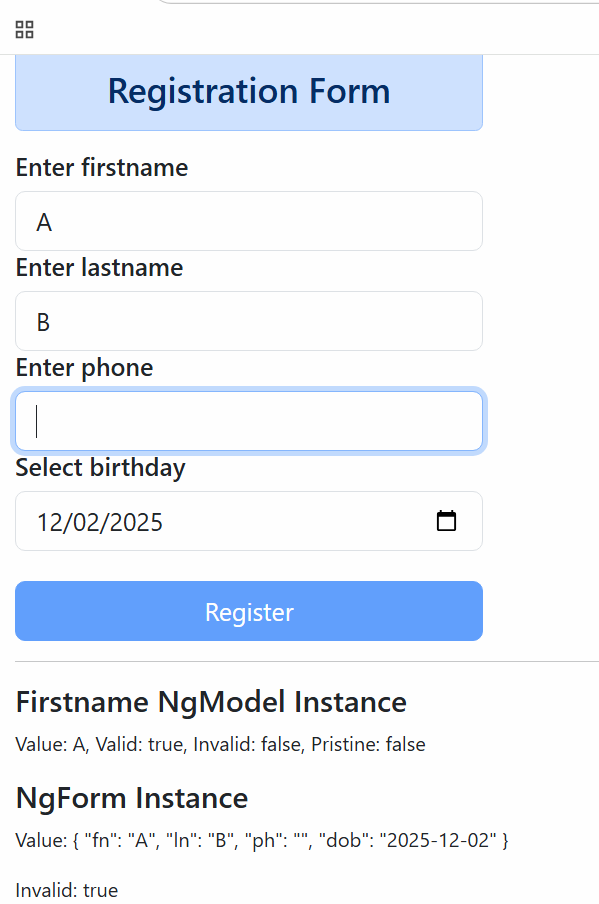


Try out displaying the validation errors for lastname, phone and dob, then disable the submit button when any one of the form input is invalid and enable the submit button only if all the inputs are valid.

How to disable the submit button



Output:



Model driven forms or Reactive forms

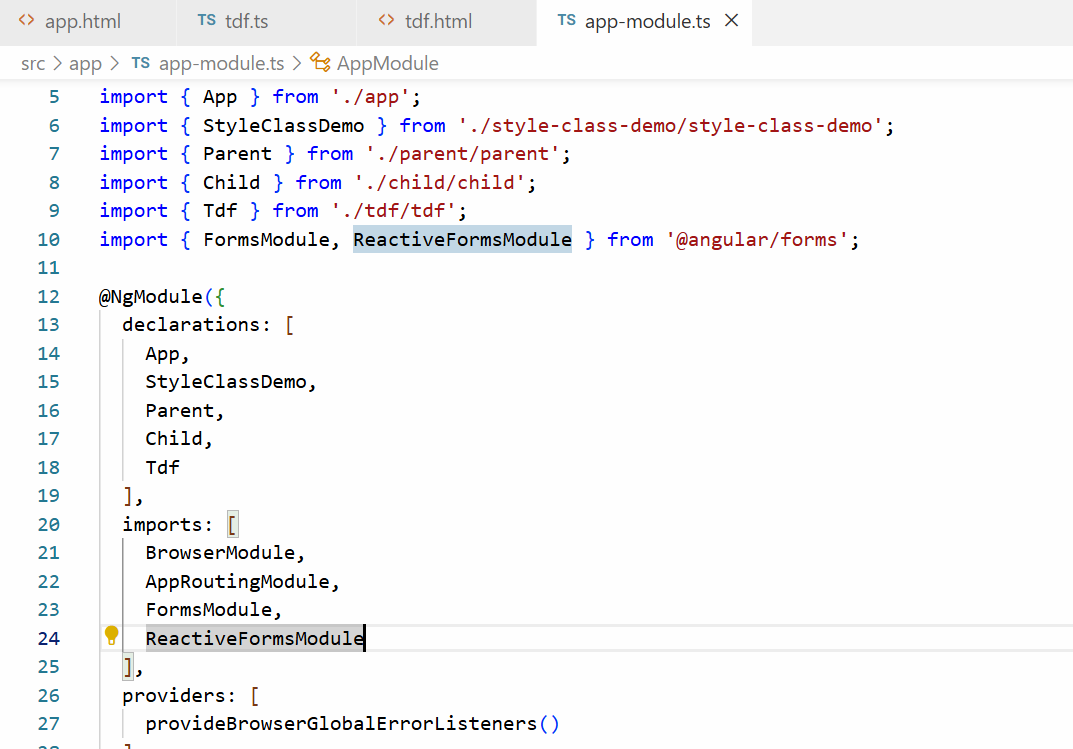
This helps to manages the forms programmatically i.e., you can manage in the typescript code instead of HTML code, you will have more control over the form when you use model driven forms hence it is used for complex forms, model driven form can handle form data, handle validators in the code itself.

To use Model Driven Form you need to use an Angular service called FormBuilder, this form build object is not available by default hence you must import in the NgModule using ReactiveFormsModule

i.e.,

NgModule.imports must use ReactiveFormsModule

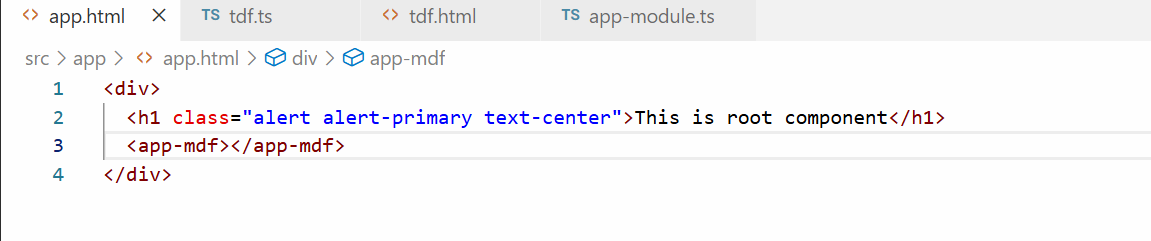
app-module.ts



Generate the component



Add this component to the root component



In Model Driven Form you create form controls in the code and HTML form in the HTML template, it makes HTML template look cleaner, because you can use Validators in the code itself, this you can achieve using one of the service called FormBuilder which you can use with the below code

builder = inject(FormBuilder);

Then you can group the form controls using group function that will take javascript properties

userForm = builider.group({  
 firstname : [],   
 lastname : [],  
 phone : [],  
 dob : []  
});

You can also add Validators to the form controls using below code

userForm.group({  
 firstname: [‘’, Validators.required], lastname : [‘’, Validators.requried]  
});

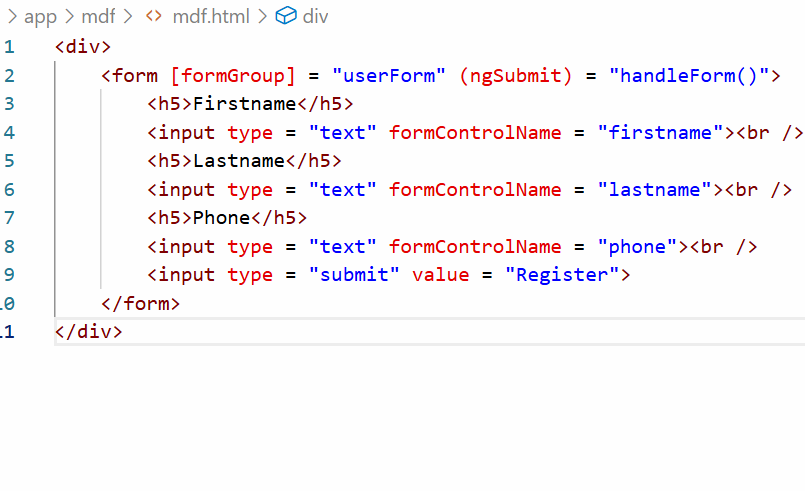
Now in the HTML template you can access the from group as below

<form [formGroup] = “userForm”>  
 <input type = “text” formControlName = “firstname”>  
 <input type = “text” formControlName = “lastname”>  
</form>

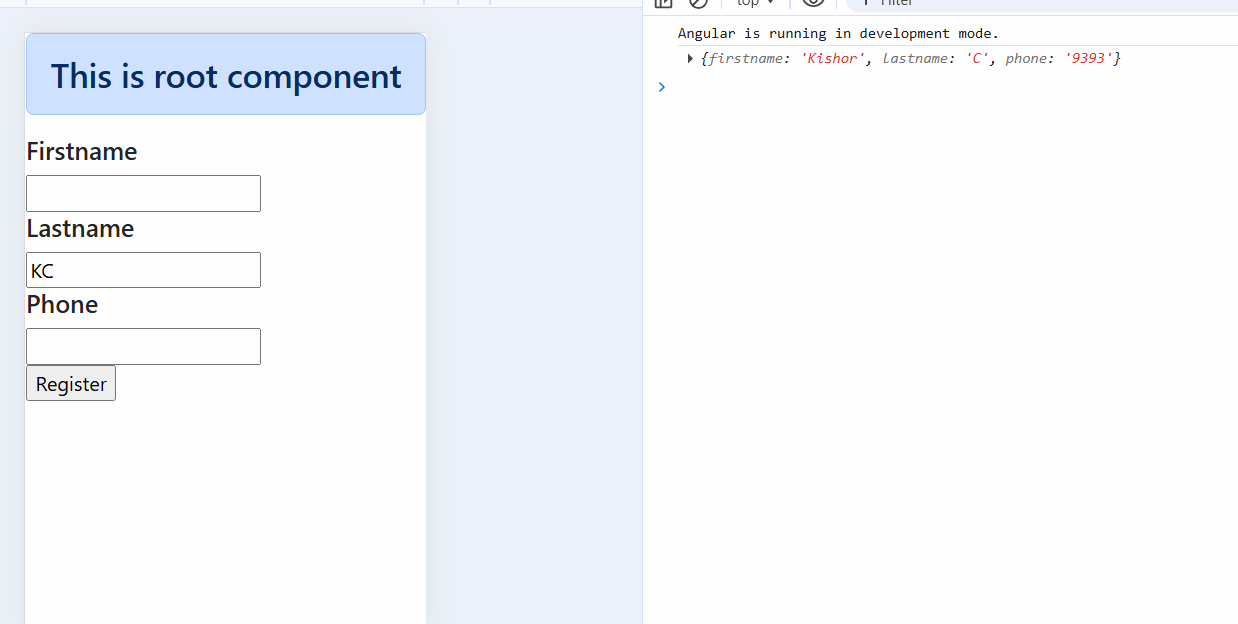
mdf.ts



mdf.html



Output:



Session 15

06-Dec-2025

Let us create a new project with a name second-app

Command: ng new second-app --no-standalone

Class & Style binding

These are property style binding which helps to add CSS classes or remove CSS classes and add style attribute to the HTML template.

NgStyle: It is used to add style to the HTML element, here you create a variable that will have the style properties and in the HTML you will use that variable using [ngStyle] binding

Example

TS code

myStyle = {“color” : “blue”; “border”:”2px solid black”, “padding”:”5px”}

HTML code

<div [ngStyle] = “myStyle”>Some content</div>

The above code adds style attribute to the div

i.e, <div style = “color : blue; border: 2px solid black; padding: 50px”>content<div>

Class Binding: This adds or removes the class, it uses [ngClass] and a CSS class name in the HTML template

Example

CSS file

.blueColor { color : blue }

TS file

status = true;

HTML

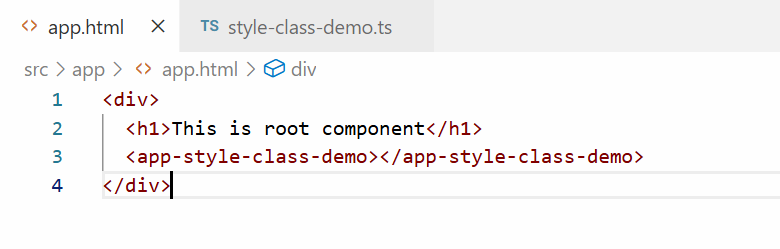
<div [ngClass] = {‘blueColor’ : status }>Some content</div>

If status is true then <div class = “blueColor”>Some content</div>  
If status if false then <div class>Some content</div>

Create a component



app.html

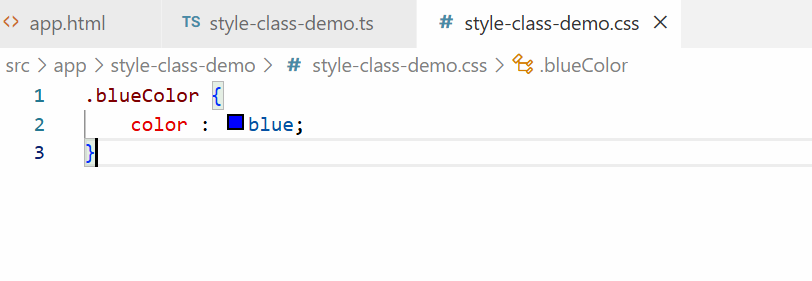


style-class-demo.ts



We are changing the values of the color in myStyles using 3 different functions like changeStylesToRed(), changeStylesToGreen() and changeStylesToDefault(), then we have status and toggle() that will add/remove CSS class

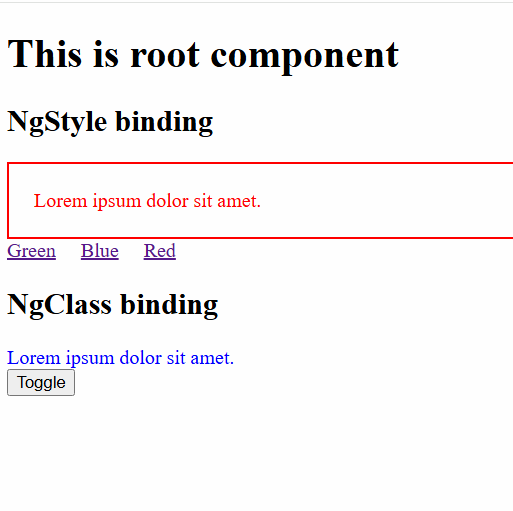
style-class-demo.css



style-class-demo.html



Output:



Adding bootstrap css to the angular project

You need to install bootstrap library in the project and add the location of the library in styles.css

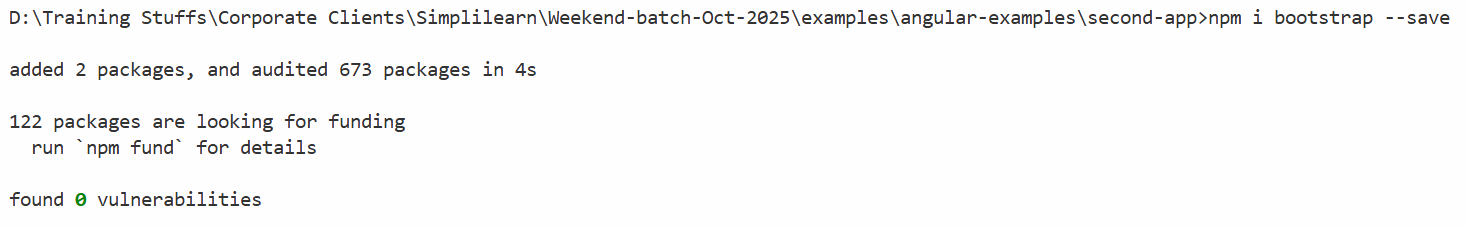
styles.css: It is a global style sheet whose style can be used in any components you create

Note: Don’t use the CDN link of bootstrap in your project, because CDN link is not reliable

How to download bootstrap

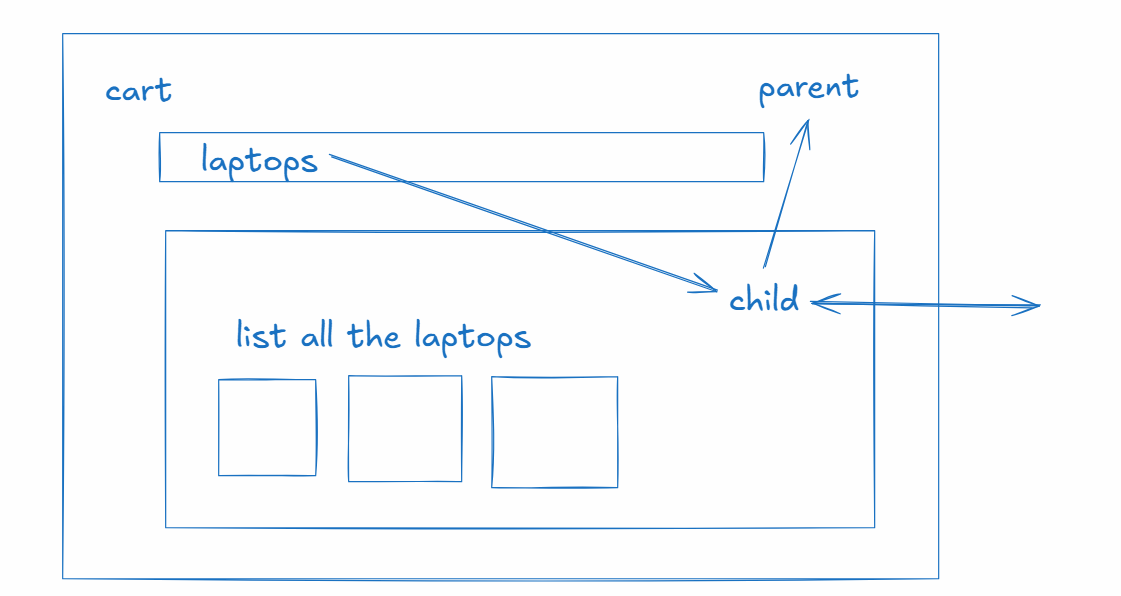
npm install bootstrap --save or npm i bootstrap --save

This downloads bootstrap in node\_modules/ folder



Component to Component interaction/communication

A component can share data to another component, especially parent to child or child to parent interaction



There are two decorators you can use to share the data between parent and child

1. @Input()
2. @Output()

@Input() It helps parent to share the data to the child component

@Output() It helps child to share the data to the parent component

Parent to child communication

parent.ts

item = “laptops”; // assume item value is dynamic

child.ts

@Input()  
product = “”; // this receives the data from the parent component

child.html

<p> {{ product }} </p>

parent.html

<child-component [product] = “item”></child-component>

Child to parent communication

child.ts

@Output()  
message = new EventEmitter<string>();  
@Output()  
product = new EventEmitter<any>();

// child component can share the data using emit function present in the EventEmitter

message.emit(“some message”) // $event  
product.emit({id: 100, name : “HP” });

parent.ts

data1 = “”;

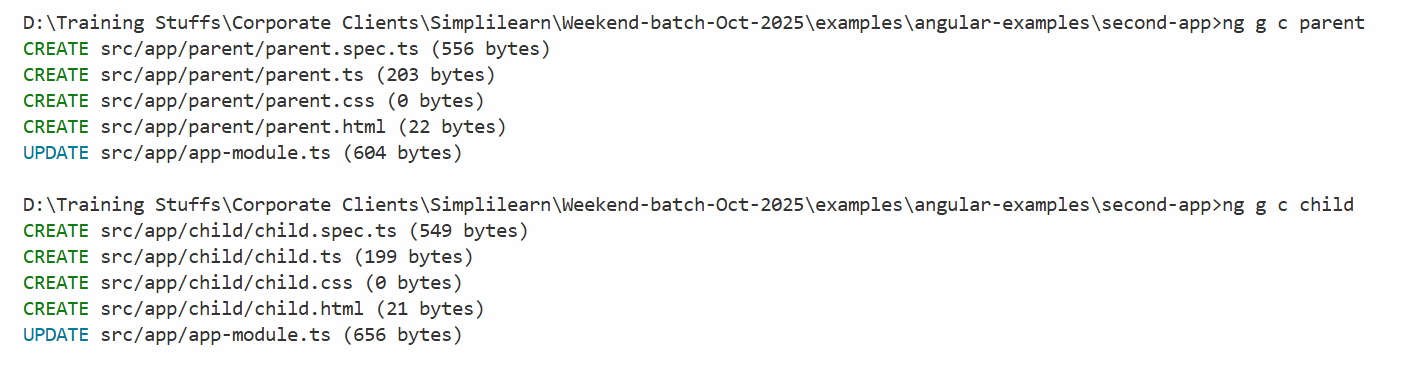
parent.html

<child-component (message) = “data1=$event” ></child-component>

Create 2 components

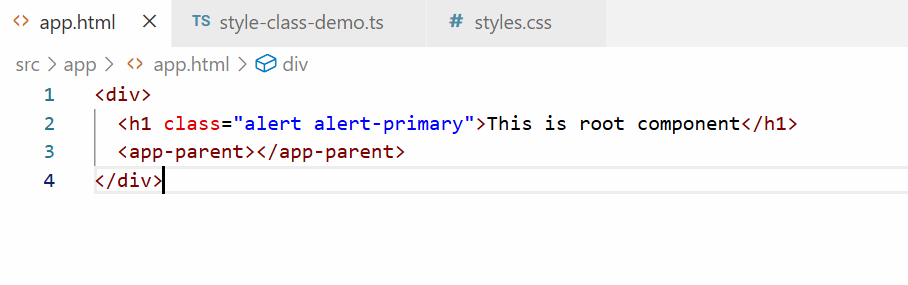
parent

child



We need to add child component to parent component & parent component to root component.

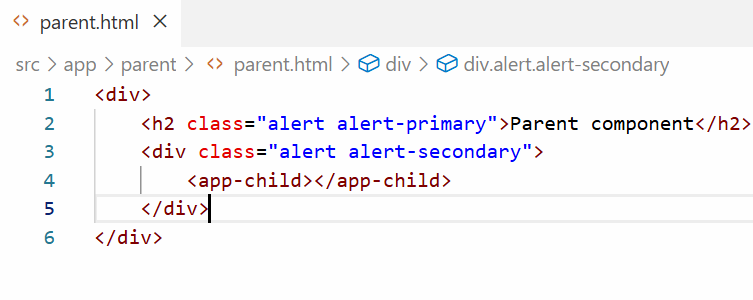
app.html



We have added parent component to the root component

Now we will add child component to parent component

parent.html



Output:



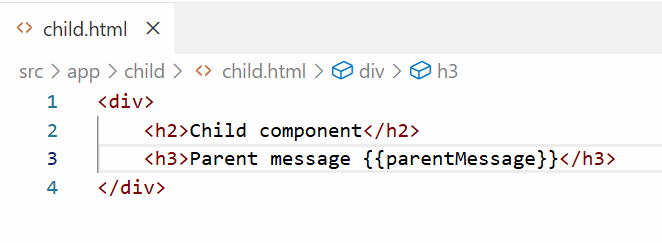
Now we will share a simple text message to the child component from the parent component, for that we need a text box in the parent and its value must be stored in the variable that will be share to child component.

Let us first create a property in child component that would have a default message

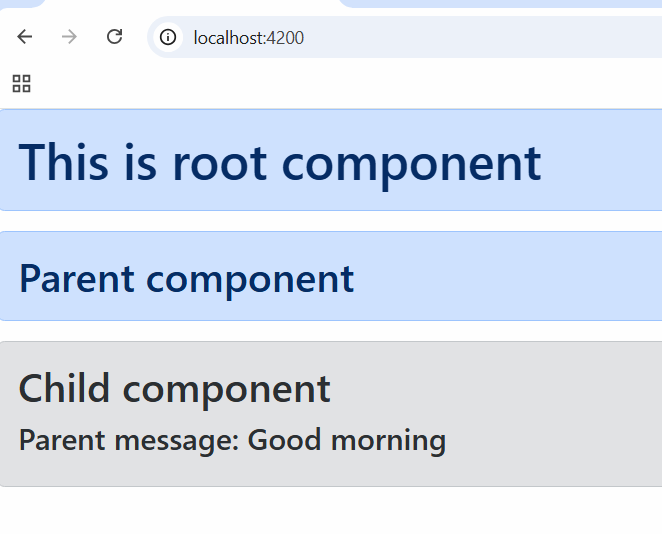
child.ts



child.html



Current we will see default output



parent.ts



parent.html



Output:

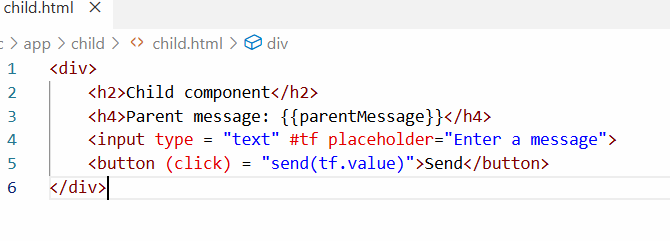


Sending the data from child to parent

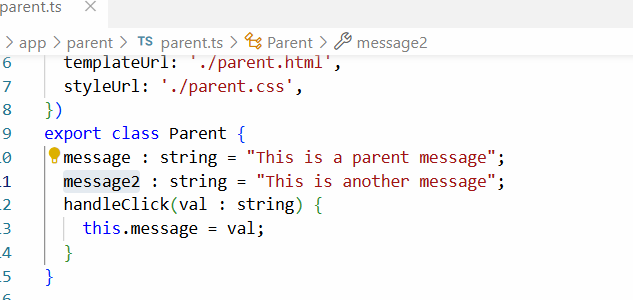
child.ts



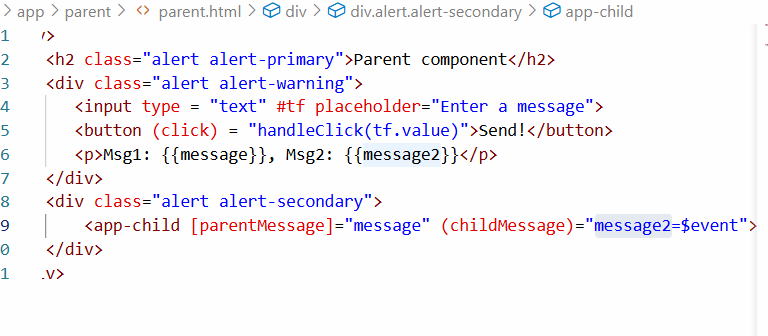
child.html



parent.ts



parent.html



Output:



30-11-2025

Directives

They allow you to work with DOM elements, there are following directives

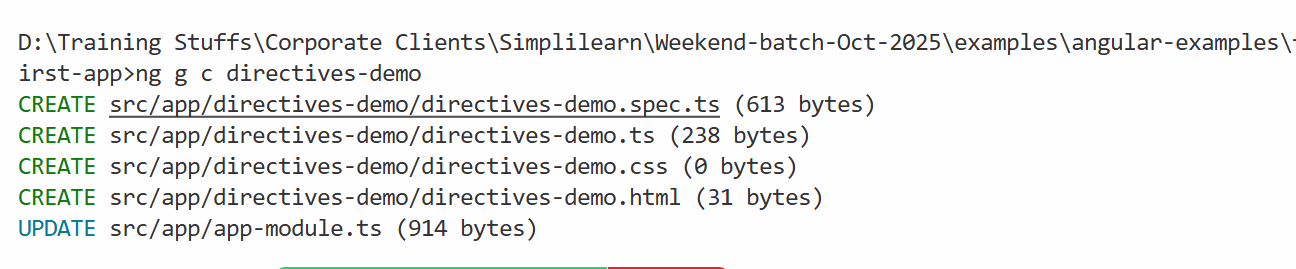
There are 3 types of directives

1. Component directive
2. Structural directive
3. Attribute directive

Component directive: a class with @Component is a component directive, most of the UI is built with component directive

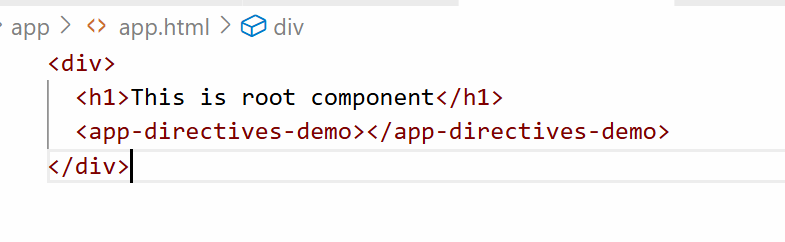
Structural directive: These change DOM structure like add/remove elements, there are 3 types

1. \*ngFor: It helps to create DOM elements at runtime while iterating the arrays
2. \*ngIf: It helps to add/remove DOM elements based on some condition
3. \*ngSwitch: It helps to add/remove DOM elements based on multiple conditions



Add the component to the root component

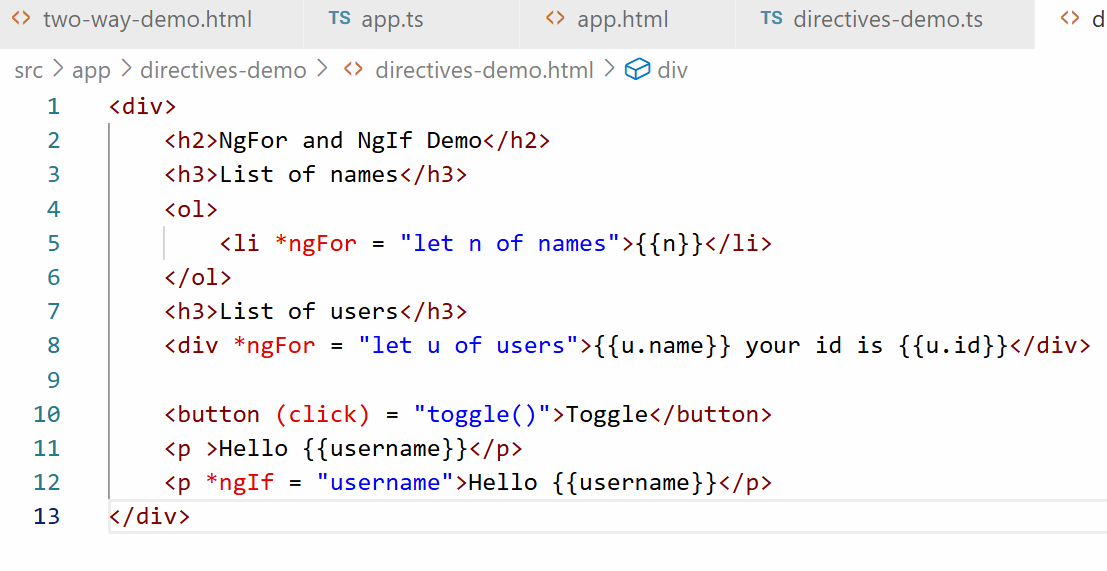
app.html



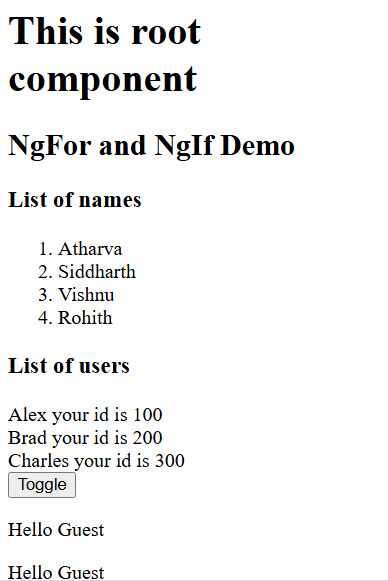
directives-demo.ts



directive-demo.html



Output:

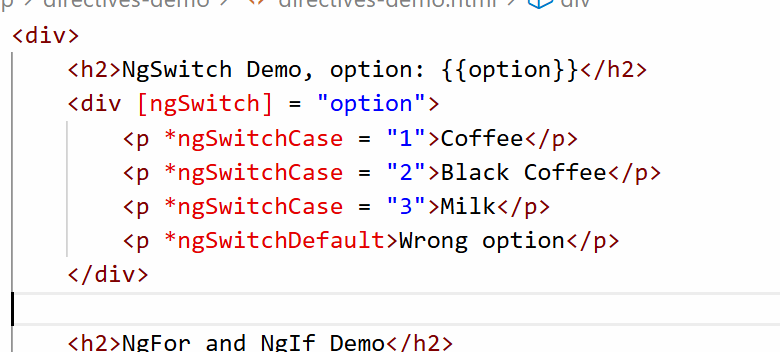


\*ngSwitch: it helps you to show only one HTML block depending on the value of an expression

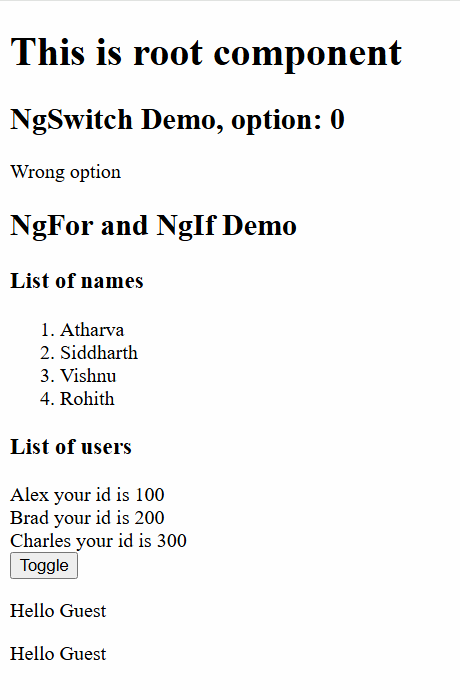
You can use \*ngSwitchCase to match the value and if none of the value matches you can use \*ngSwitchDefault which is executed when the none of the condition match.



directive-demo.html



Output:

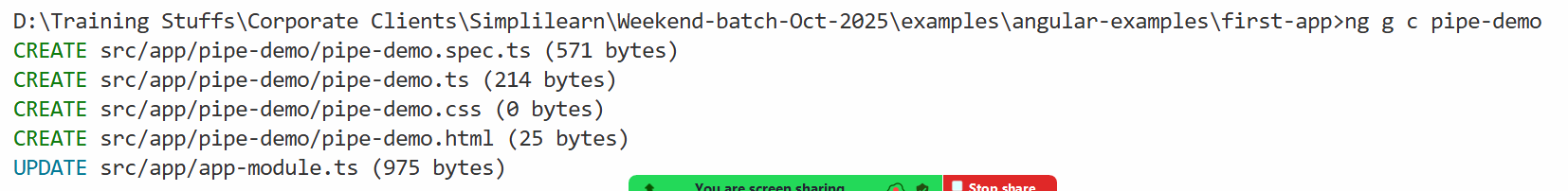


Angular Pipes

It is used to transform the output like showing the names in uppercase, lowercase, or amount in different currencies, date in different formats

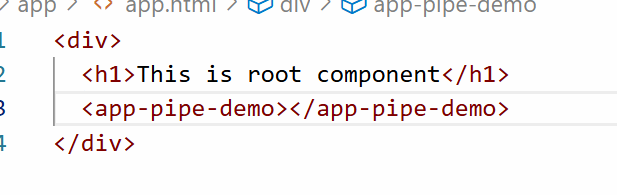
actual data is 2025-11-30 but you want to show as 11-Nov-2025 or 11-10-2025

actual data is 10000, but you will show $10,000



Add this component to the root component]

app.html



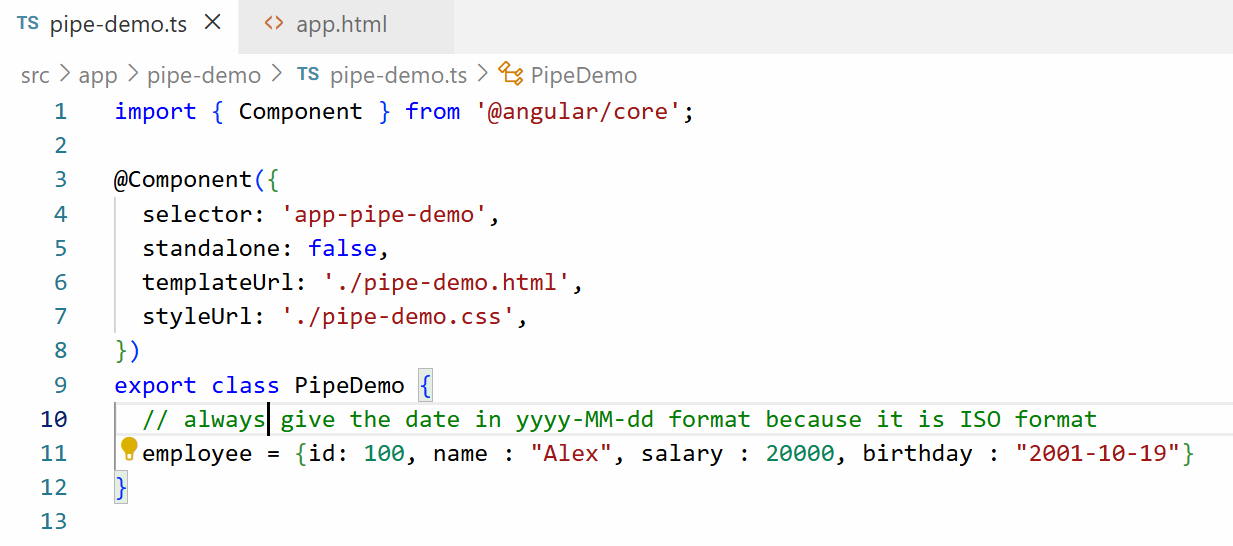
Angular has built-in pipes

* currency
* date
* uppercase
* lowercase
* json

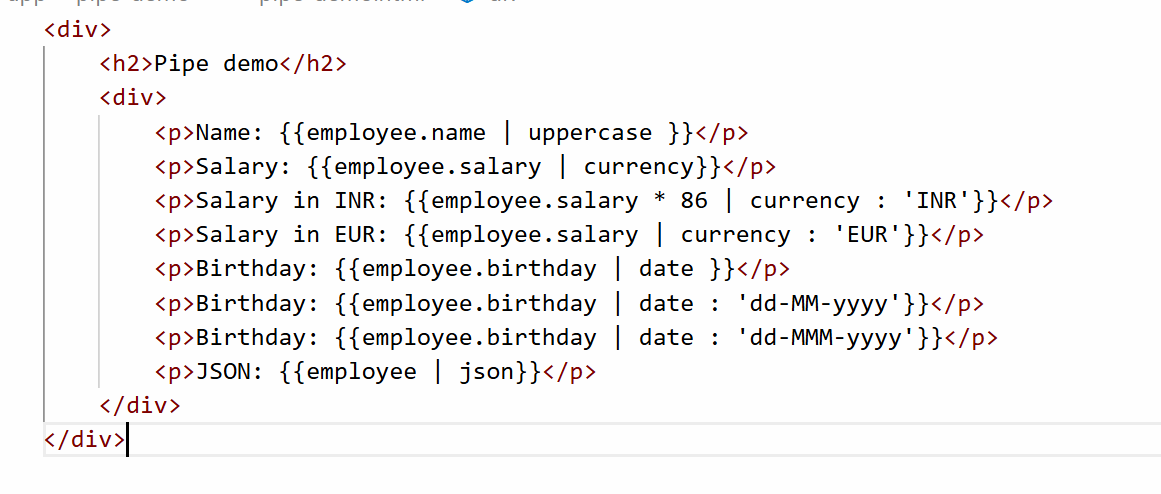
{{ salary | currency }}

{{ name | uppercase }}

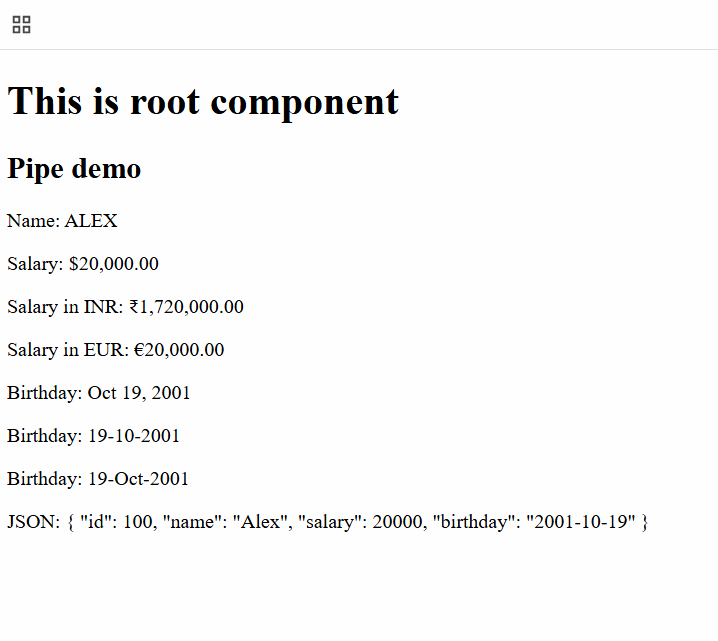
{{ birthday | date }}



pipe-demo.html



Output:



Activity

Create 3 to 4 employees in an array that will have id, name, salary, birthday, address, but address is a nested object with state, city and pin, display these employee data in a table format.

expected output

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| id | name | salary | birthday | state | city | pin |
| 100 | Alex | $20,000 | 10-Nov-2000 | KA | BLR | 50393 |
| 200 | Brad | $40,000 | 11-Oct-1999 | MH | PUN | 60293 |

Custom pipes

You can create your own pipes that would transform the data based on your requirement, to create a custom pipe you must use a following command

ng generate pipe pipe-name [or] ng g p pipe-name

This creates a class with @Pipe({}) decorator and a function called transform inside it, this function is called whenever the pipe is used in the template

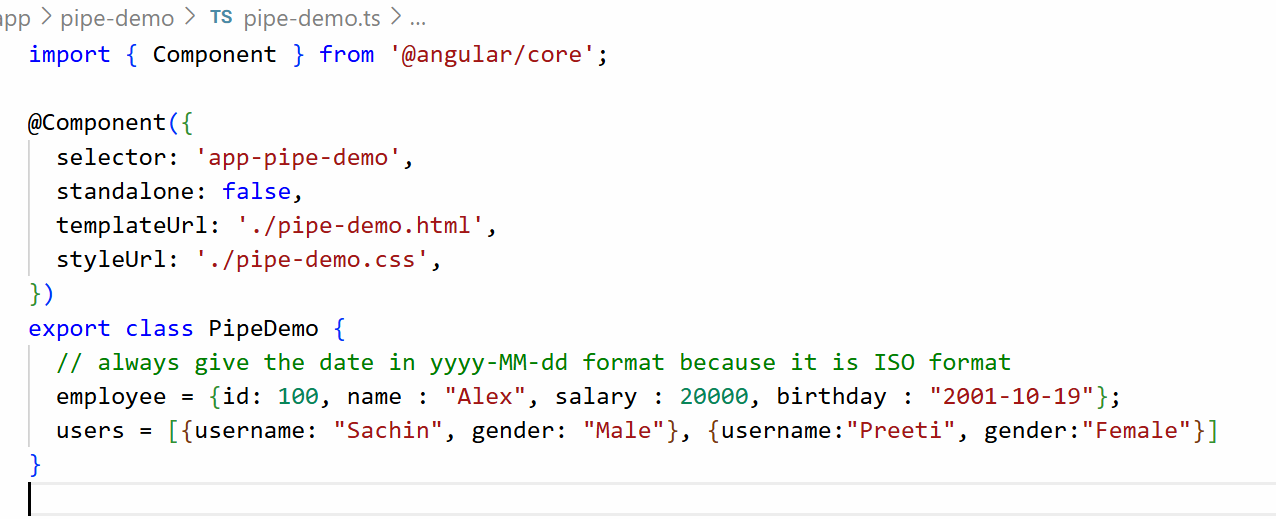
user = { username : “Sachin”, gender: “Male“ }

Output: Mr. Sachin

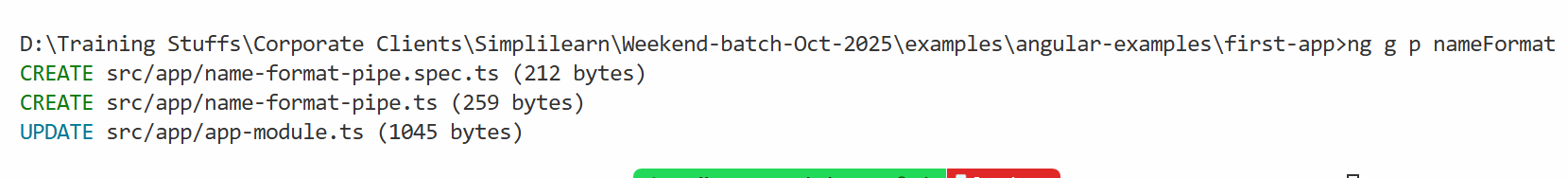
user = { username : “Preeti”, gender : “Female” }

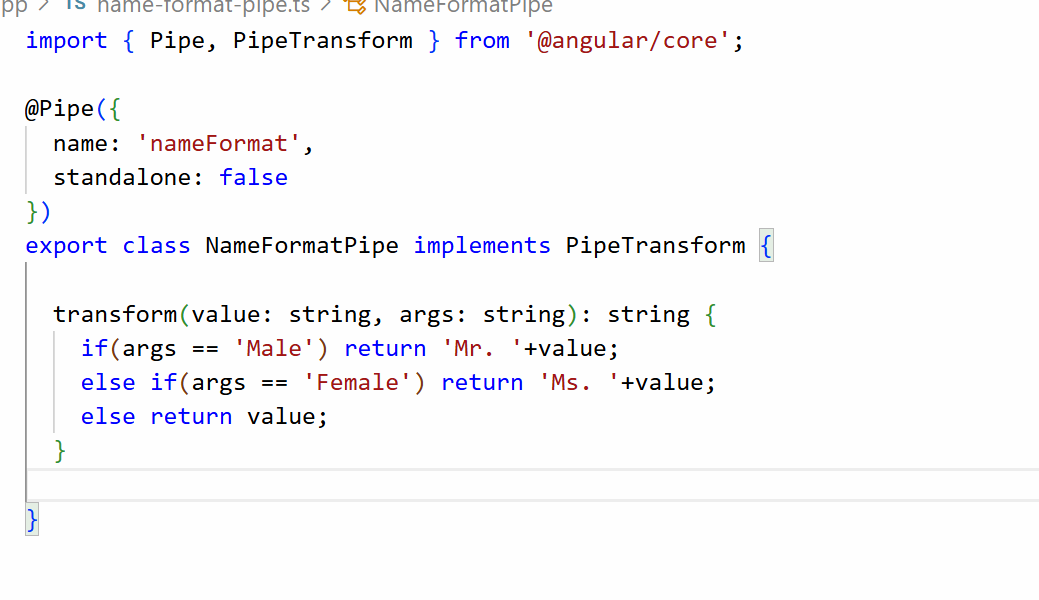
Output: Ms. Preeti

pipe-demo.ts

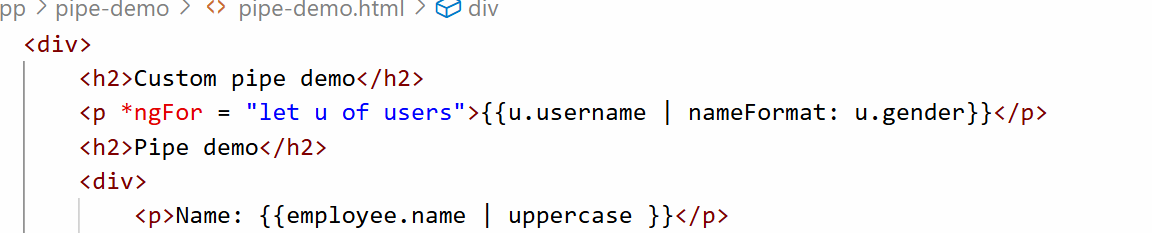


Create a pipe name-format





HTML



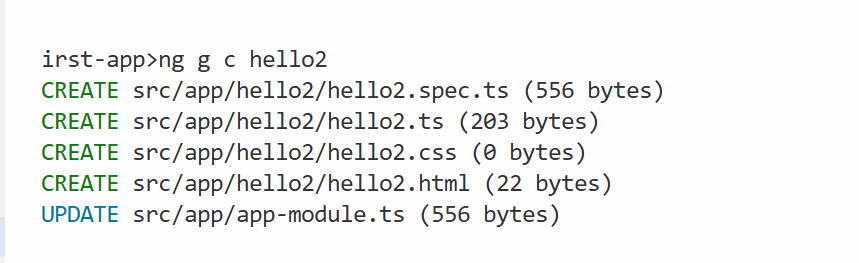
Output:

29-11-2025

Creating components using command

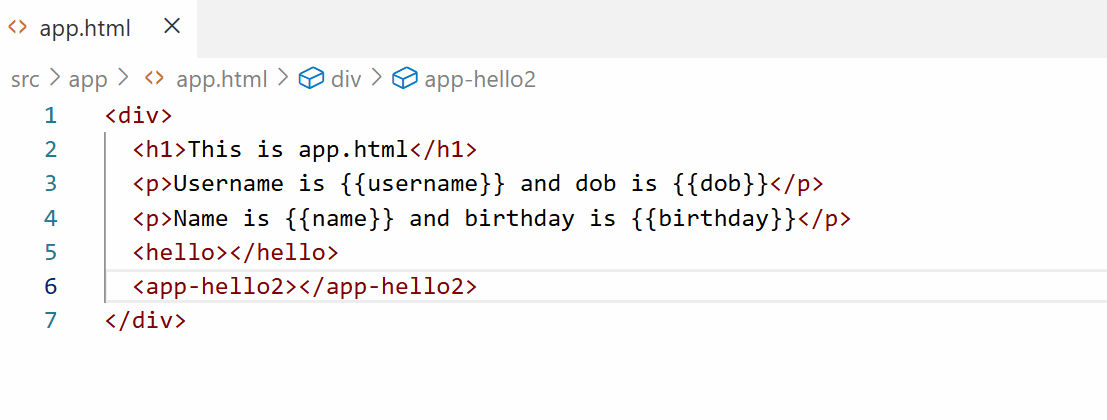
ng generate component component-name [or] ng g c component-name

Note: All the angular related commands must be entered in the parent directory of the project, i.e., project folder.

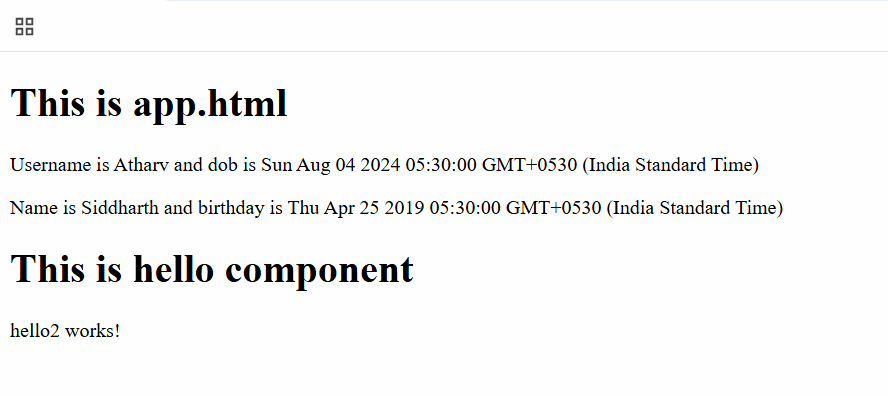


Regardless of what components you create, we must understand that only the root component is part of the index.html, so all the components we created must be part of root component or its child component.

app.html



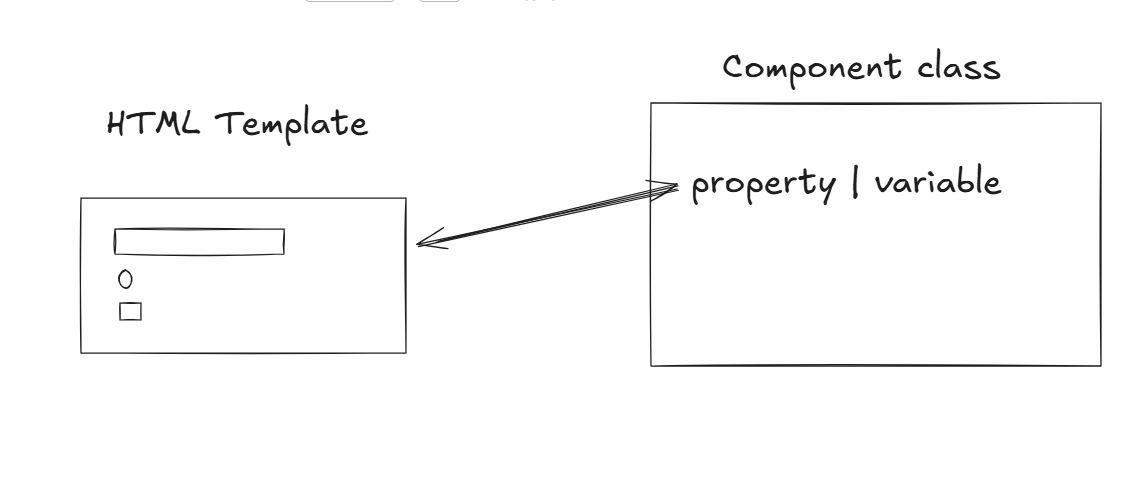
Now you can see the hello2 component result in the web page



Activity:

1. Create a component with the name employee, inside the employee component create a Javascript object called emp that will have properties like id, name, salary and dob, in it’s HTML display those id, name, salary and dob and add the employee component in the root component

Angular data-binding



Data-binding is a way where angular helps HTML template and Component class to share the data, in Angular there are mainly 4 types of Data-binding

1. Interpolation {{ }} : Here component class shares the data to the HTML template, this is mainly used to display the data
2. Event binding ( ) : Here HTML template shares the data to the Component class, here you can send form data or update the component property through events

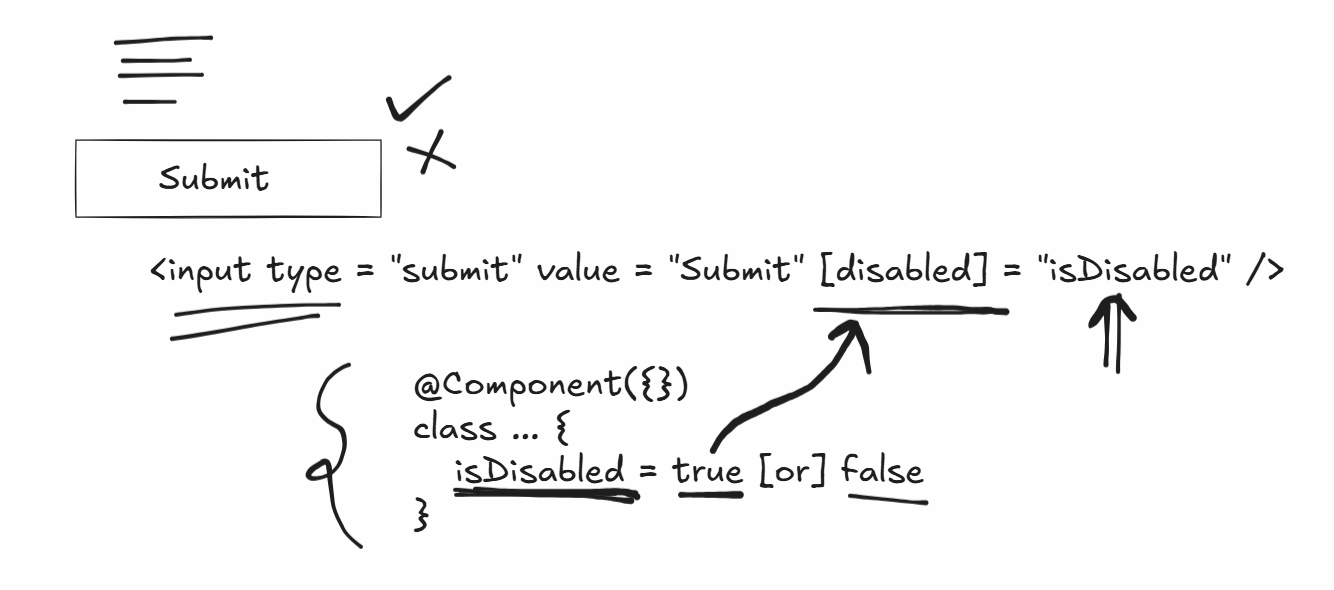
ex: <button (eventName) = “functionName()”>Button</button>

The eventName can be ngSubmit, click, input, change, blur and so on

1. Property binding [ ] : Here component class shares the data to the HTML template, this is mainly used to modify the HTML DOM properties

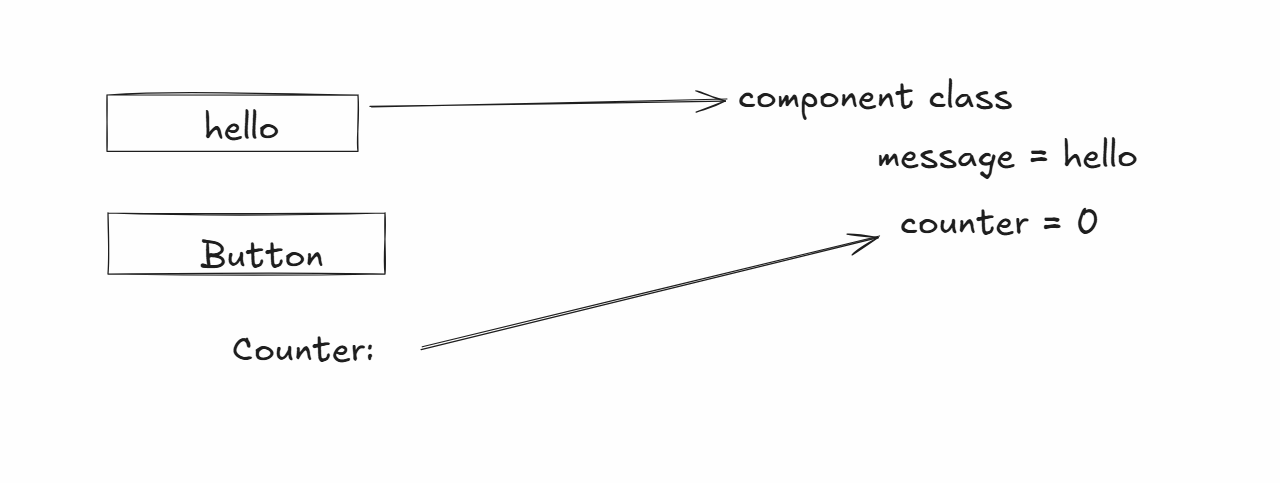
<button [disabled] = “propertyName”>Button</button>

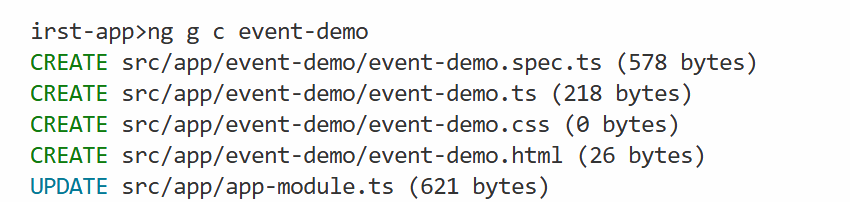
Here propertyName can have true | false value based on that the disabled property will be enabled | disabled



1. Two-way data-binding [(ngModel)] : This shares the data in both the ways i.e, from component class to html template and from html template to component class, this makes the data in sync

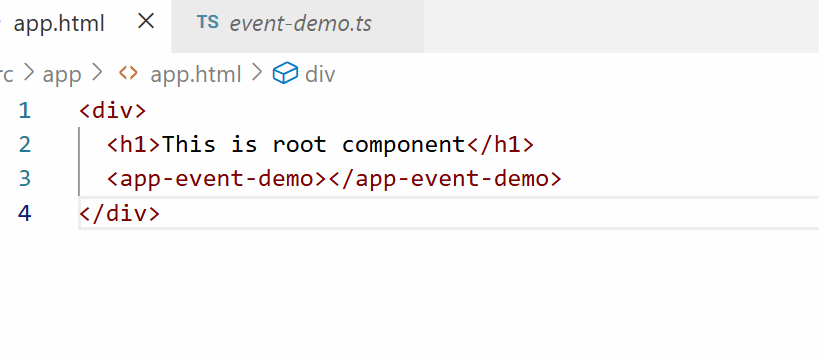
Event Binding Demo



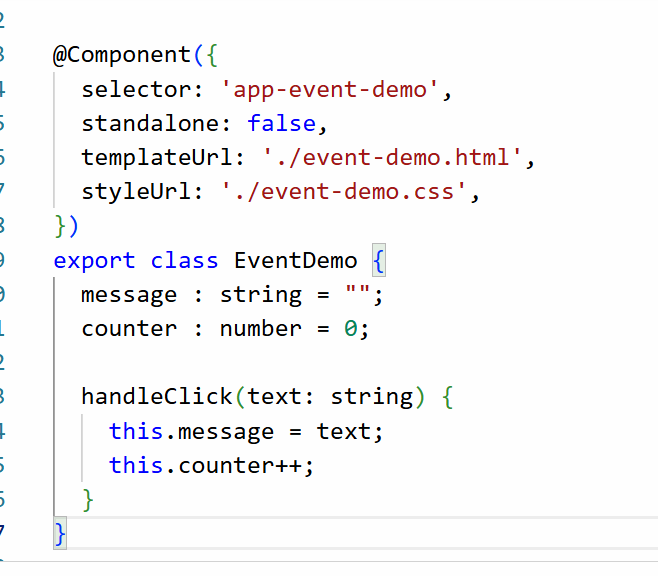


Add this component to the root component

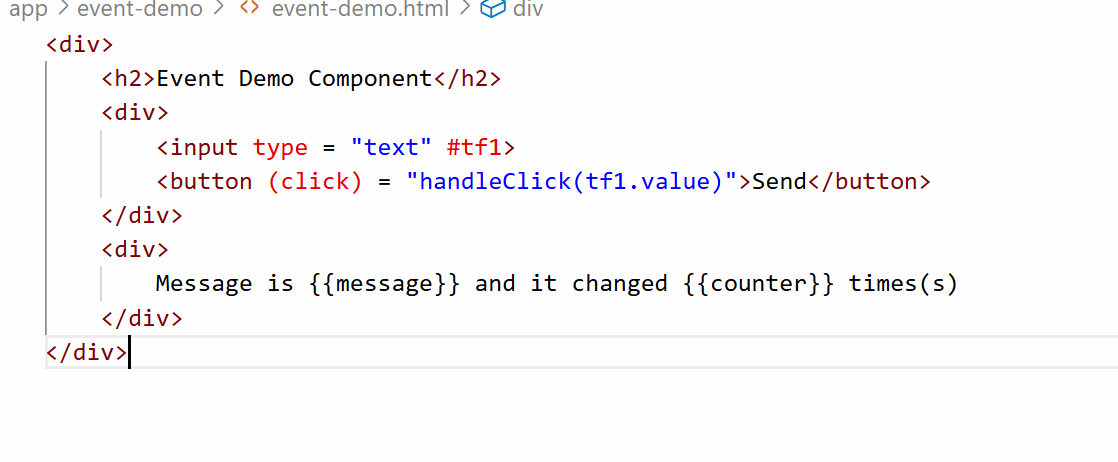
app.html



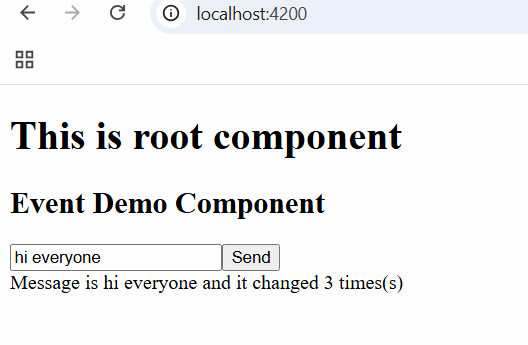
event-demo.ts



event-demo.html

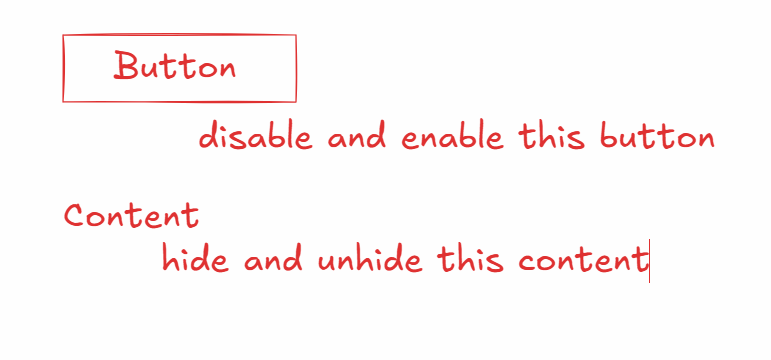


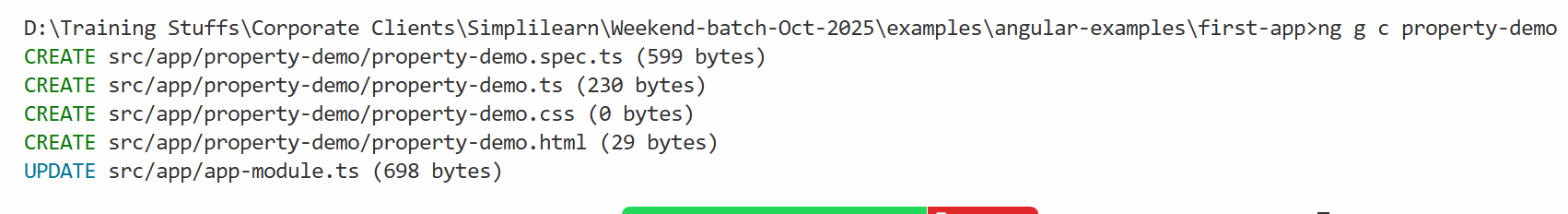
Output:



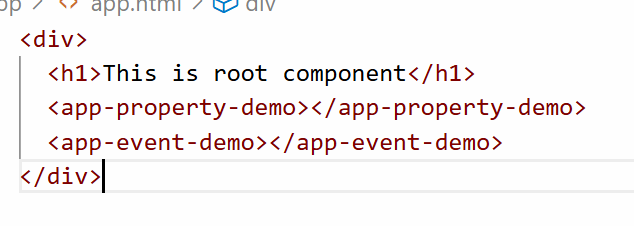
Property Binding Demo

It helps to bind the HTML DOM objects like disabled, hidden, this shares the data from component class to html template.





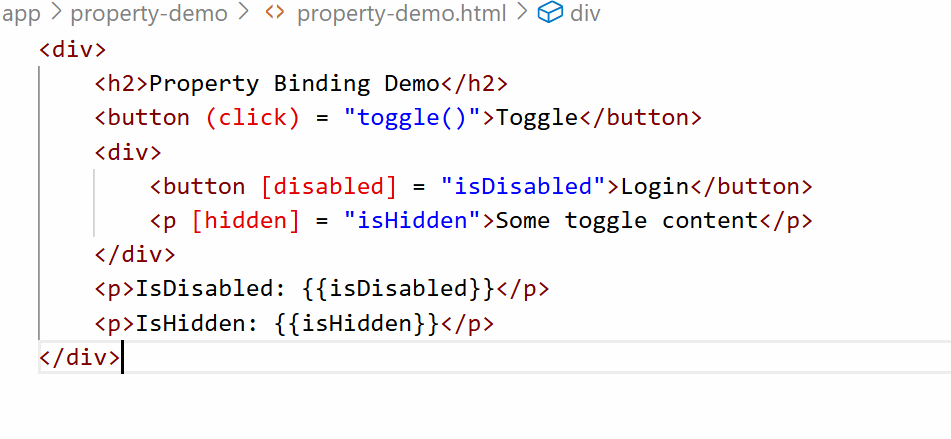
Add the component to the root component



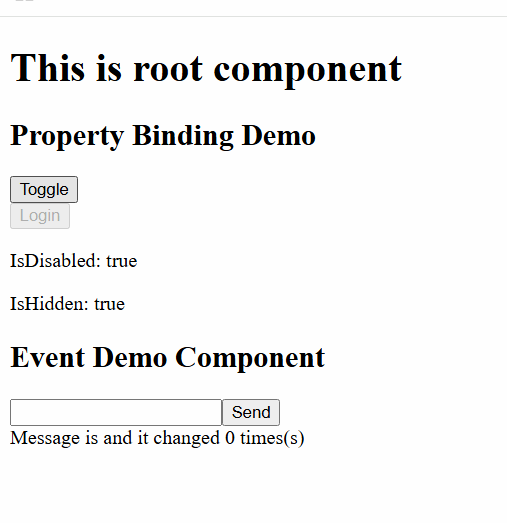
property-demo.ts



property-demo.html



Output:



Two way data binding

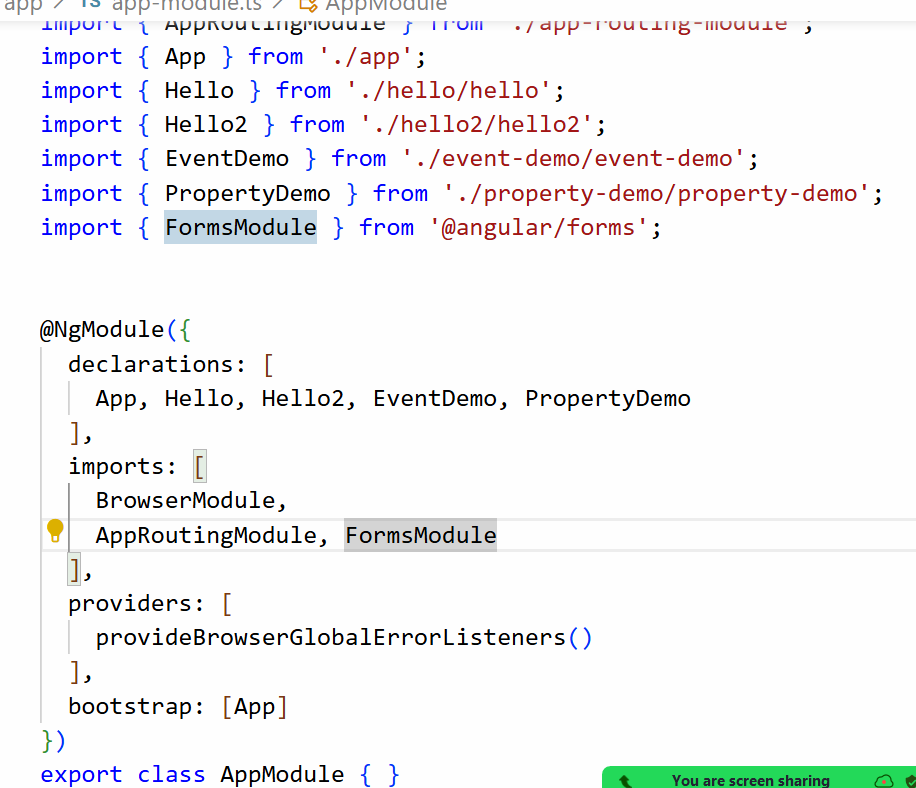
Here angular helps component class and html template to share the data in both the directions so that the data will be in sync, it is a combination of event and property binding.

By default Two way databinding doesn’t work because it uses [(ngModel)] directive, this directive is part of a built-in module called FormsModule, angular recognizes [(ngModel)] only if you add FormsModule in the application.

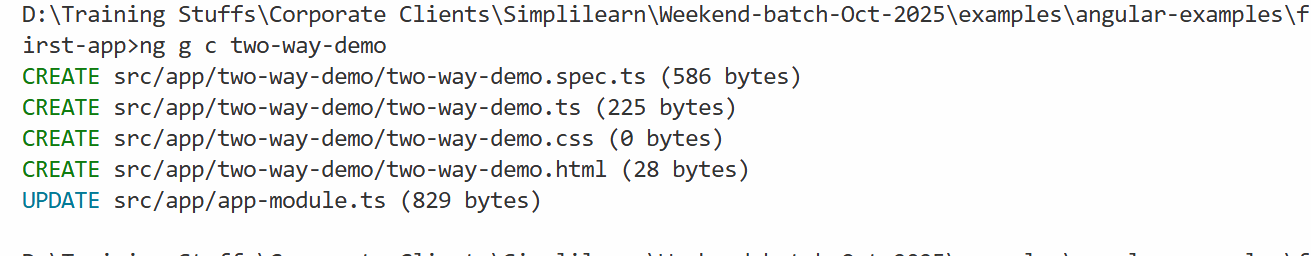
How to add FormsModule in the application

You just need to add FormsModule in the @NgModule.imports of your AppModule class which is present inside app.module.ts

app.module.ts

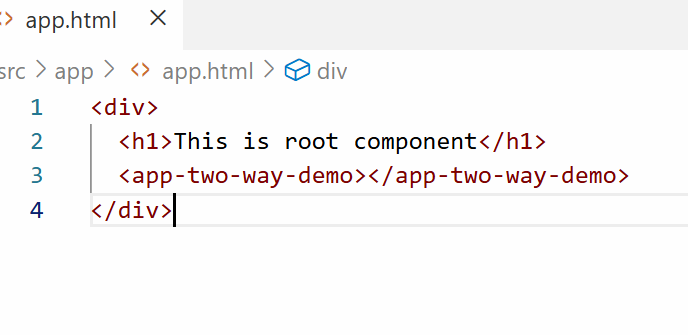


Generate two-way-demo



Add the two-demo component to the root component

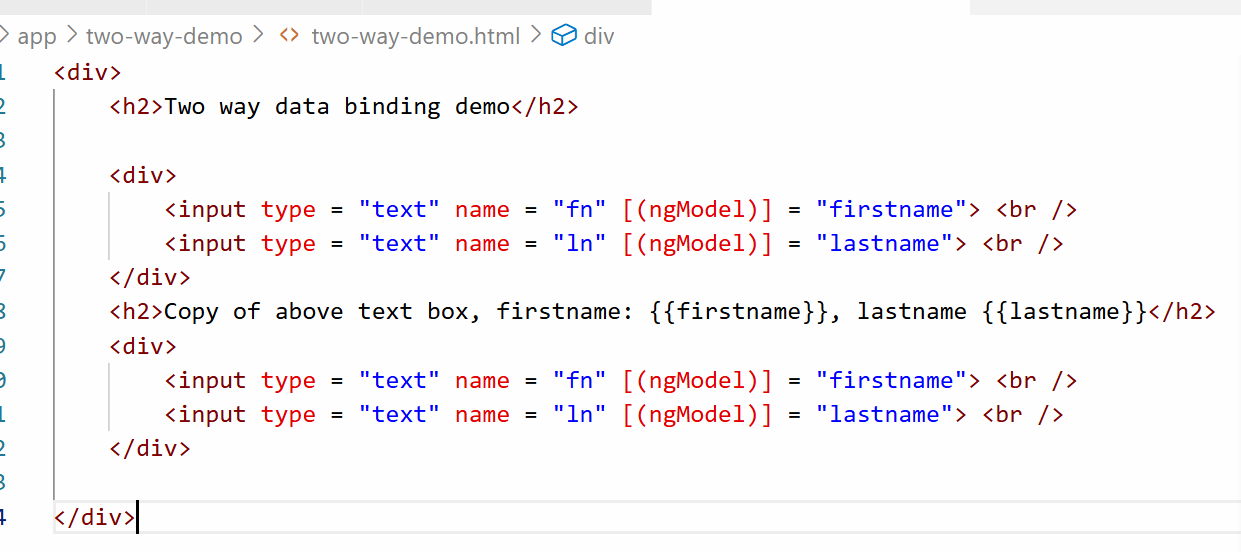
app.html



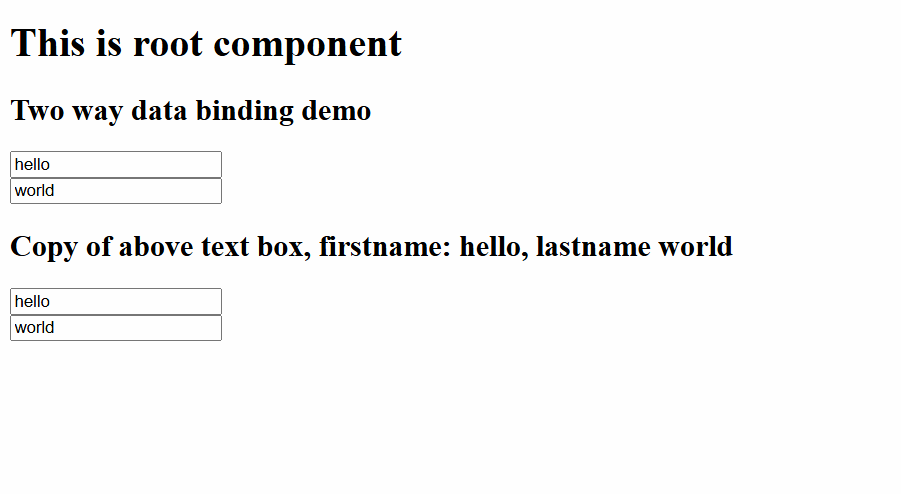
two-way-demo.ts



two-way-demo.html



Output:

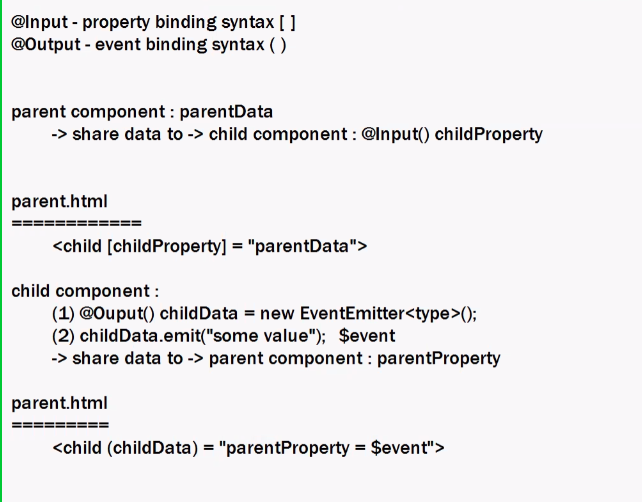


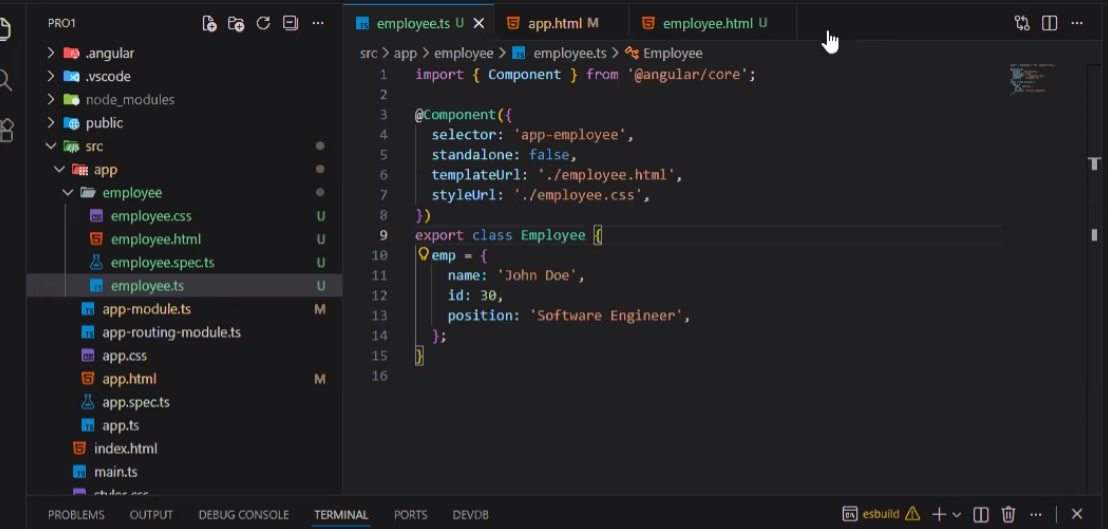
Where exactly two way databinding is useful

1. Live preview
2. When server wants to update the form when user types some value in another input box
3. In shopping cart if you change the quantity automatically the prices should update

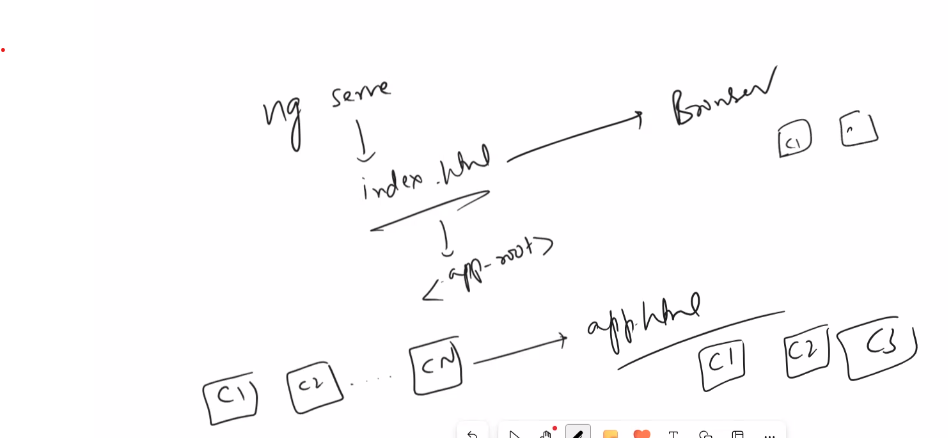
Template driven forms = way to build forms where logic and validations are handled within html templates.

From child to parent = @output and then send data through emit function.

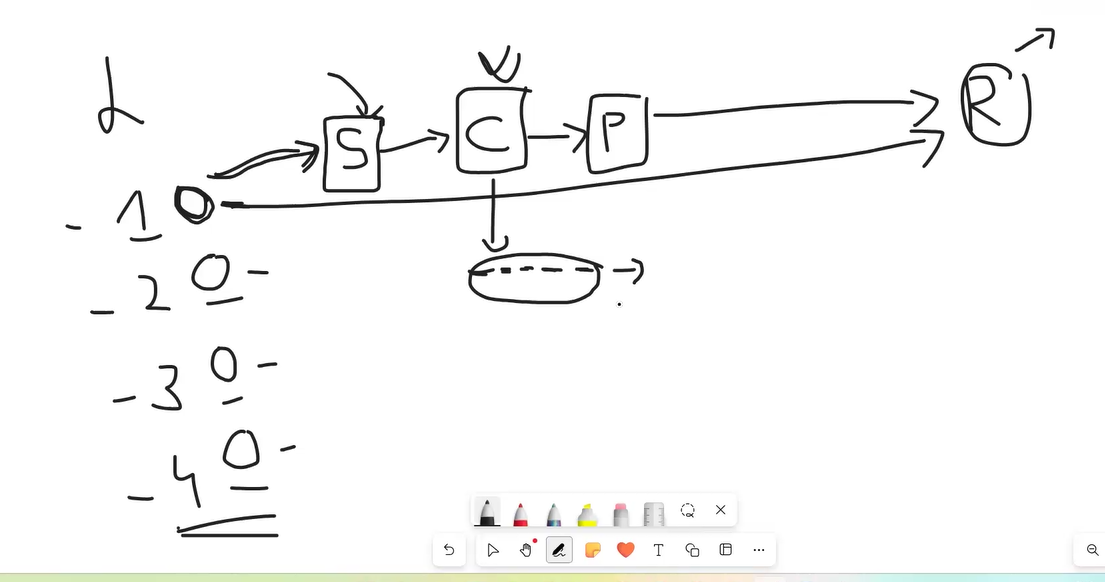




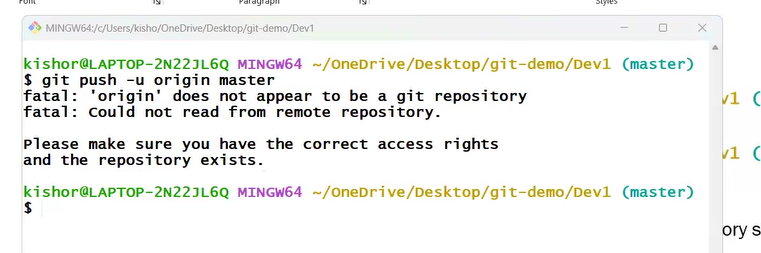




GIT STAGE COMMIT AND PUSH







**…or create a new repository on the command line**

echo "# 20251123-gitbasics" >> README.md

git init

git add README.md

git commit -m "first commit"

git branch -M main

git remote add origin https://github.com/jayashankartx/20251123-gitbasics.git

git push -u origin main

**…or push an existing repository from the command line**

git remote add origin https://github.com/jayashankartx/20251123-gitbasics.git

git branch -M main

git push -u origin main

git push -u origin master

info: please complete authentication in your browser...

Enumerating objects: 93, done.

Counting objects: 100% (93/93), done.

Delta compression using up to 14 threads

Compressing objects: 100% (88/88), done.

Writing objects: 100% (93/93), 163.86 KiB | 4.82 MiB/s, done.

Total 93 (delta 19), reused 0 (delta 0), pack-reused 0 (from 0)

remote: Resolving deltas: 100% (19/19), done.

To https://github.com/jayashankartx/20251123-jfsd.git

\* [new branch] master -> master

branch 'master' set up to track 'origin/master'.

Git Branch

git pull



Can we merge again?