**Angular**

Step 1: Install node js

Step 2: Install npm install-g @angular/cli

How to verify angular cli

1. Node –v (It’s check version of node)
2. Ng –v (It’s check version of angular cli)

Step 3: Install visual studio code

COMMAND: for creating new angular project using following command

Ng new AngularProject –skip-test

Step 4. Open visual studio code through command use ***.Code***

Step 5: Install bootstrap in angular project using following command

Npm install [bootstrap@4.0](mailto:bootstrap@4.0) –save

Step 6: for running angular use following command: ng serve

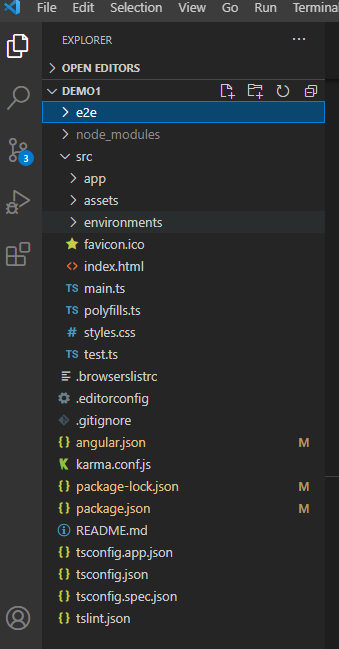
***Version in Angular***

| VERSION | STATUS | RELEASED | ACTIVE ENDS | LTS ENDS |
| --- | --- | --- | --- | --- |
| ^11.0.0 | Active | Nov 11, 2020 | May 11, 2021 | May 11, 2022 |
| ^10.0.0 | LTS | Jun 24, 2020 | Dec 24, 2020 | Dec 24, 2021 |
| ^9.0.0 | LTS | Feb 06, 2020 | Aug 06, 2020 | Aug 06, 2021 |

Note: Angular version has ***major, minor and Patch changes***

1. Feature of angular
2. Single page application
3. Speed and performance
4. Smaller application
5. Modular application
6. Cross-platform (web, mobile and desktop)
7. Use Typescript

Folder structure in Angular



**.editorConfig :-** this file use for configure rule that rule use by Team member from our team.

**Angular.json file**: - In this file contain configuration of application

Main.ts is entry point of application.

Whenever first boot our application check main.ts file.

In this file contain bootstrapmodule property that called to app.module.ts file.

Main.ts file contain following line of code

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

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

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

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

if (environment.production) {

  enableProdMode();

}

platformBrowserDynamic().bootstrapModule(AppModule)

  .catch(err => console.error(err));

App.module file contains configuration of multiple module and component files.

Please find below line of code in app.module.ts file

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

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

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

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

@NgModule({

  declarations: [

    AppComponent

  ],

  imports: [

    BrowserModule,

    AppRoutingModule

  ],

  providers: [],

  bootstrap: [AppComponent]

})

export class AppModule { }

* **Declaration:** Whatever component we created in our application that component need to declare in **app.module.ts** file
* **Imports:** Whenever any new configuration install in the application that component need to import in **app.module.ts** file
* **Providers:** In this component contain service file of reference
* **Bootstrap:** This starting point of application

Every component file contain following structure of code

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

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.css']

})

export class AppComponent {

  title = 'Demo1';

}

**Polyfills.ts**

* We return code in angular application in type script language but our browser understand javascript.in this file contain complier that complier convert typescript to JavaScript
* Every not support JavaScript library in this file contain configuration of browser

**Tslint.json**:- In this file contain rule

**tsConfig.json**:- This file use for testing purpose

**Package.json:-**

Whatever mandatory data require during development and production those versions data store in this file

**Package-lock.json :-**

Whatever we installed in application those all-configuration versions store in this file

Note: - whenever you have converted application into old to new version for that use **NPM Install**

**List of all @Decorators available in Angular**

1. @NgModule
2. @Component

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.css']

})

export class AppComponent {

  title = 'Demo1';

}

1. @Injectable

@Injectable({

  providedIn: 'root'

})

export class EmployeeService {

  constructor() { }

}

1. @Directive
2. @Pipe
3. @Input
4. @Output
5. @HostBinding
6. @HostListener
7. [@ContentChild](https://medium.com/u/d220ff7175e9?source=post_page-----71bdf4ad6976--------------------------------)
8. @ContentChildren
9. @ViewChild
10. @ViewChildren

@NgModule

@NgModule({

  declarations: [

    AppComponent

  ],

  imports: [

    BrowserModule,

    AppRoutingModule

  ],

  providers: [],

  bootstrap: [AppComponent]

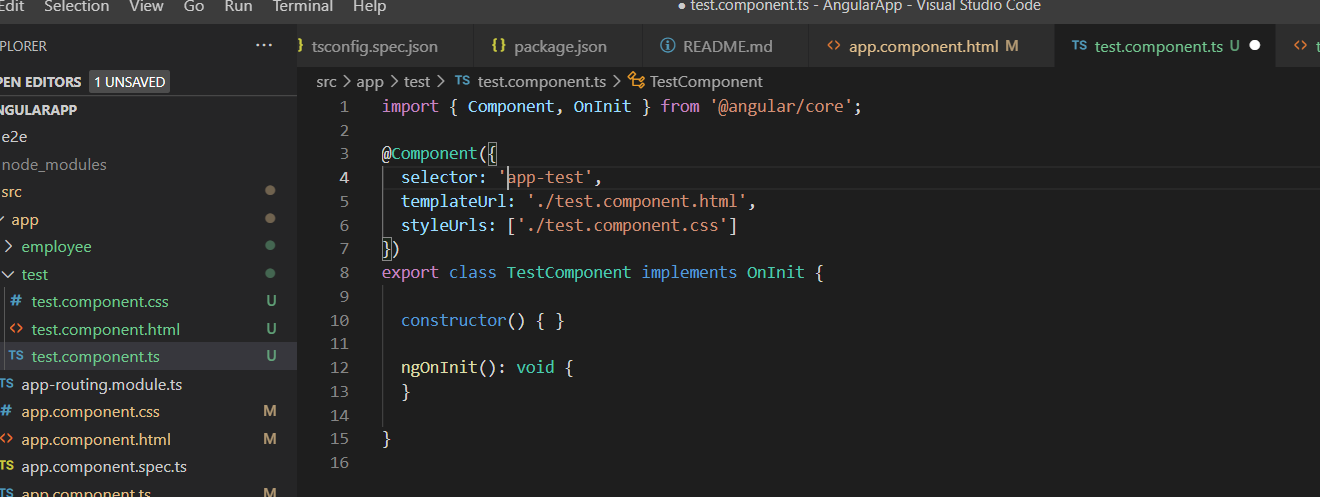
})

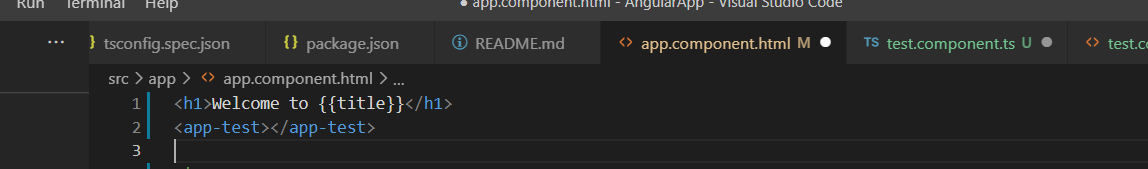
Component:-

* Angular application have collection component
* One component handle only one view.
* Component can support typescript language
* Component contains metadata like effect, style template, input-output, import etc.

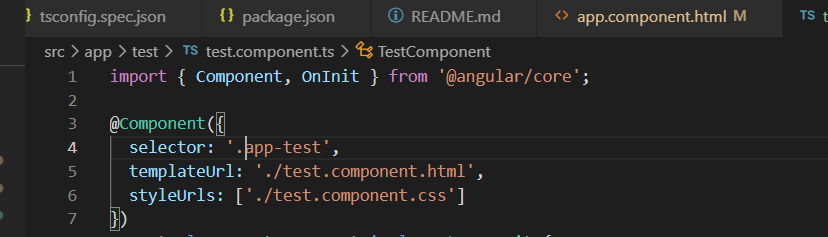
We can call the component different ways

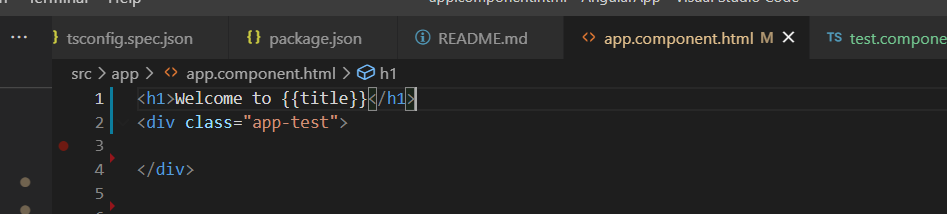
Step 1 :



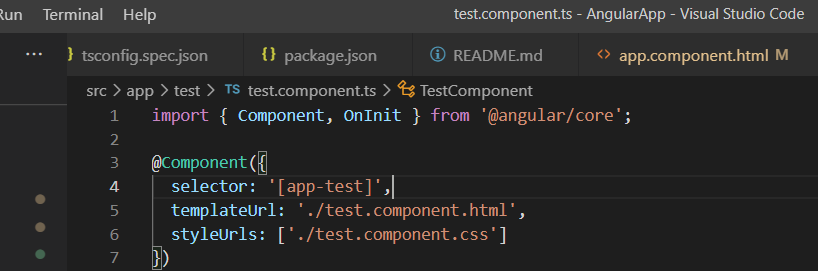


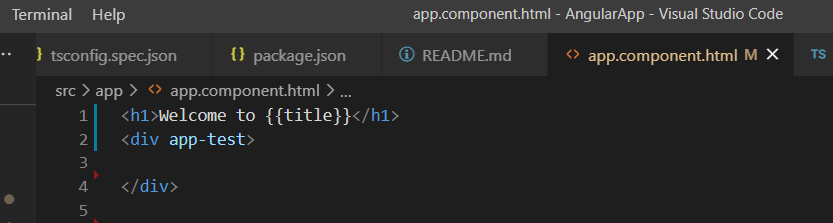
Step 2 :





Step 3:





Directive:

They are 3 type of directive

1. [Components](https://angular.io/guide/component-overview)—directives with a template. This type of directive is the most common directive type.
2. [Attribute directives](https://angular.io/guide/built-in-directives#built-in-attribute-directives)—directives that change the appearance or behavior of an element, component, or another directive.
3. [Structural directives](https://angular.io/guide/built-in-directives#built-in-structural-directives)—directives that change the DOM layout by adding and removing DOM elements.

**Built-in attribute directive**

Attribute directives listen to and modify the behavior of other HTML elements, attributes, properties, and components.

Many NgModules such as the [RouterModule](https://angular.io/guide/router) and the [FormsModule](https://angular.io/guide/forms) define their own attribute directives. The most common attribute directives are as follows:

* [NgClass](https://angular.io/guide/built-in-directives#ngClass)—adds and removes a set of CSS classes.

This is special Angular binding syntax

<div [class.extra-sparkle]="isDelightful">

ngClass is a normal Angular directive like you can build it yourself

<div [ngClass]="{'extra-sparkle': isDelightful}">

ngClass is more powerful. It allows you to bind a string of classes, an array of strings, or an object like in your example.

<td [ngClass]="val > 10 ? 'red' : 'green'">{{ val }}</td>

<input type="text" [ngClass]="control.isInvalid ? 'error' : ''" />

<td [ngClass]="getClassOf(val)">{{ val }}</td>

class MyComponent {

getClassOf(val) {

if (val >= 0 && val <= 5) {

return 'low';

} else if (val > 5 && val <= 10) {

return 'medium';

} else {

return 'high'

}

}

}

* [NgStyle](https://angular.io/guide/built-in-directives#ngstyle)—adds and removes a set of HTML styles.

[ngClass]="class" is used to set css classes to the button tag, it allows you to add it statically or based on conditions.

[ngStyle]="style" is used to set css styles to the button tag, it allows you to add it statically or based on conditions.

* [NgModel](https://angular.io/guide/built-in-directives#ngModel)—adds two-way data binding to an HTML form element.

**Built-in structural directive**

* \*ngIf
* \*ngFor
* \*ngSwtich

\*ngIf..Else

 <div \*ngIf="QuestionForm.invalid;then FormValidation else loading\_content">

</div>

 <ng-template #FormValidation>

</ng-template>

 <ng-template #loading\_content>

</ng-template>

\*ngSwitch

 <div class="card full-height" [ngSwitch]="i.StatusCode">

<div class="card-body" \*ngSwitchCase="'P'">

       </div>

       <div class="card-body" \*ngSwitchCase="'C'">

</div>

       <div class="card-body" \*ngSwitchCase="'NS'">

       </div>

<div class="card-body" \*ngSwitchDefault>

       </div>

</div>

\*ngFor

 <div class="toggle" \*ngFor="let Q of Question.lstQuestionModel;let j=index;

let F=first; let l= last; let ev=even; let od =odd;">

</div>

\*ngStyle

<ul \*ngFor="let emp of Employee">

  <li [ngStyle]="{'color':getcolor(emp.Country)}">

    {{emp.name}} - {{emp.Country}}

  </li>

</ul>

\*ngClass

<div [ngClass]="['one','two']">

    This is Blue clor

</div>

<div [ngClass]="'one two'">

    This is Blue clor

</div>

<div [ngClass]="'one'">

    This is Blue clor

</div>

Interpolation

<div>

  {{Fname + Lname}}

</div>

Property Binding

<span [innerHTML]="title">

</span>

<img [src]="">

<tr><td [colSpan]="1 + 1">Three-Four</td></tr>

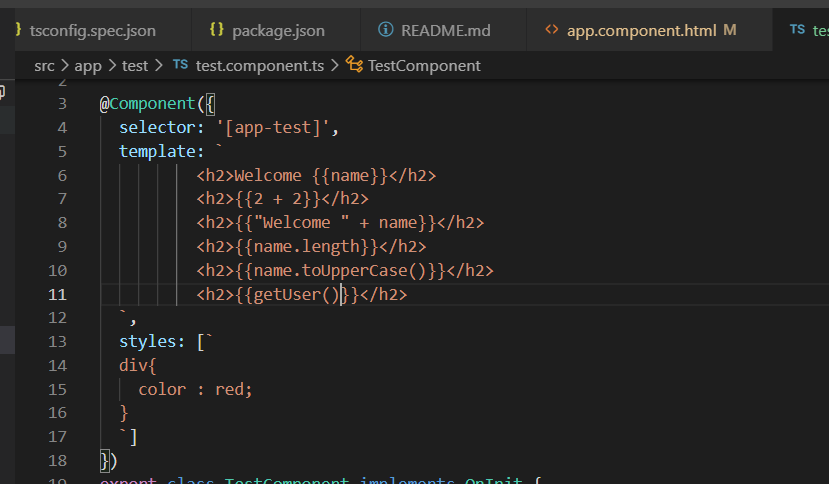
<!-- Bind button disabled [state](https://angular.io/api/animations/state) to `isUnchanged` property --> <button [disabled]="isUnchanged">Disabled Button</button>

<p [[ngClass](https://angular.io/api/common/NgClass)]="classes">[[ngClass](https://angular.io/api/common/NgClass)] binding to the classes property making this blue</p>

***Interpolation : IT will be develop using {{}} braces***

***Question : {{a=2+2}} is this logic execute by angular***

***Ans : This will not execute***



Attribute :- HTML

Property : DOM

Attribute initialize to DOM properties and then they are done. Attribute value not changes once initialize .

Property value however can be changes

## Property binding and interpolation

Often [interpolation](https://angular.io/guide/interpolation) and property binding can achieve the same results. The following binding pairs do the same thing.

## Displaying values with interpolation

Interpolation refers to embedding expressions into marked up text. By default, interpolation uses the double curly braces {{ and }} as delimiters.

<p><img src="{{itemImageUrl}}"> is the <i>interpolated</i> image.</p>

<p><img [src]="itemImageUrl"> is the <i>property bound</i> image.</p>

<p><span>{{interpolationTitle}} is the <i>interpolated</i> title.</span></p>

<p>"<span [innerHTML]="propertyTitle"></span>" is the <i>property bound</i> title.</p>

Lifecycle event sequence

Attribute Binding

<table border="1">

  <tr>

    <td [attr.colspan]="colspan">

      Student Details

    </td>

  </tr>

</table>

Http Interceptors

Often we want to enforce or apply behavior when receiving or sending HTTP requests within our application. Interceptors are a unique type of Angular Service that we can implement. Interceptors allow us to intercept incoming or outgoing HTTP requests using the **HttpClient**. By intercepting the HTTP request, we can modify or change the value of the request.

In this post, we cover three different Interceptor implementations:

* Handling HTTP Headers
* HTTP Response Formatting
* HTTP Error Handling

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

import { HttpInterceptor, HttpEvent, HttpResponse, HttpRequest, HttpHandler } from '@angular/common/http';

import { Observable } from 'rxjs';

@Injectable()

export class MyInterceptor implements HttpInterceptor {

intercept(httpRequest: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

return next.handle(httpRequest);

}

}

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

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

import { HttpClientModule, HTTP\_INTERCEPTORS } from '@angular/common/http';

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

import { MyInterceptor } from './my.interceptor';

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

@NgModule({

imports: [BrowserModule, HttpClientModule],

declarations: [AppComponent],

bootstrap: [AppComponent],

providers: [

{ provide: HTTP\_INTERCEPTORS, useClass: MyInterceptor, multi: true }

]

})

export class AppModule { }

Routing

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

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

import { UserLayoutComponent  } from './user-layout/user-layout.component';

const routes: Routes = [

  { path: '', redirectTo: '/CandidateRegister/:Testid', pathMatch: 'full' },

  { path: 'CandidateRegister/:Testid', component: UserLayoutComponent}

]

@NgModule({

  imports: [RouterModule.forRoot(routes)],

  exports: [RouterModule]

})

export class AppRoutingModule { }

export const routingComponent = [

  UserLayoutComponent

]

Lazy loading in angular: - App-routing.module.ts

## ****What is lazy loading?****

Lazy loading is a routing technique in which the JavaScript components load in the browser only when their corresponding route is activated. The main aim of lazy loading is to improve the performance of the Angular app by restricting the unnecessary loading of components.

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

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

const routes: Routes =

  [

    {

      path: 'customers', loadChildren: () => import('./customers/customers.module')

        .then(m => m.CustomersModule)

    },

    {

      path: 'orders', loadChildren: () => import('./orders/orders.module')

        .then(m => m.OrdersModule)

    },

    {

      path: '',

      redirectTo: '',

      pathMatch: 'full'

    }

  ];

@NgModule({

  imports: [RouterModule.forRoot(routes,{

    preloadingStrategy: PreloadAllModules

  })],

  exports: [RouterModule]

})

export class AppRoutingModule { }

Customer-routing.module.ts

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

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

import { CustomersComponent } from './customers.component';

const routes: Routes = [{ path: '', component: CustomersComponent }];

@NgModule({

  imports: [RouterModule.forChild(routes)],

  exports: [RouterModule]

})

export class CustomersRoutingModule { }

order-routing.module.ts

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

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

import { OrdersComponent } from './orders.component';

const routes: Routes = [{ path: '', component: OrdersComponent }];

@NgModule({

  imports: [RouterModule.forChild(routes)],

  exports: [RouterModule]

})

export class OrdersRoutingModule { }

Template Driven Validation

<form name="form" (ngSubmit)="f.form.valid && onSubmit()" #f="ngForm" novalidate>

<div class="form-group">

      <label for="firstName">First Name</label>

      <input type="text" class="form-control" [(ngModel)]="model.firstName" #firstName="ngModel" [ngClass]="{'is-invalid':f.submitted && firstName.invalid}" name="firstName" required />

      <div class="invalid-feedback">

        <div>First Name is required</div>

      </div>

    </div>

    <div class="form-group">

      <label for="lastName">Last Name</label>

      <input type="text" class="form-control" [(ngModel)]="model.lastName" #lastName="ngModel" [ngClass]="{'is-invalid':f.submitted && lastName.invalid}" name="lastName" required />

      <div class="invalid-feedback">

        <div>Last Name is required</div>

      </div>

    </div>

    <div class="form-group">

      <label for="email">Email</label>

      <input type="text" class="form-control" [(ngModel)]="model.email" #email="ngModel" [ngClass]="{'is-invalid':f.submitted && email.invalid}" name="email" required email />

      <div \*ngIf="f.submitted && email.invalid" class="invalid-feedback">

        <div \*ngIf="email.errors.required">Email Name is required</div>

        <div \*ngIf="email.errors.email">Email must be a valid email Address</div>

      </div>

    </div>

    <div class="form-group">

      <label for="password">Password</label>

      <input type="password" class="form-control" [(ngModel)]="model.password" #password="ngModel" [ngClass]="{'is-invalid':f.submitted && password.invalid}" name="password" required minlength="6" />

      <div \*ngIf="f.submitted && password.invalid" class="invalid-feedback">

        <div \*ngIf="password.errors.required">Password Name is required</div>

        <div \*ngIf="password.errors.minlength">Password must be at least 6 characters</div>

      </div>

    </div>

    <div class="form-group">

      <button class="btn btn-primary">Register</button>

    </div>

  </form>

Reactive Form Validation

App.component.html

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

    <div class="form-row">

         <div class="form-group col">

              <label>Title</label>

                    <select formControlName="title"

class="form-control"

[ngClass]="{ 'is-invalid': submitted && f.title.errors }">

                        <option value=""></option>

                        <option value="Mr">Mr</option>

                        <option value="Mrs">Mrs</option>

                        <option value="Miss">Miss</option>

                        <option value="Ms">Ms</option>

                    </select>

                    <div \*ngIf="submitted && f.title.errors" class="invalid-feedback">

                        <div \*ngIf="f.title.errors.required">Title is required</div>

                    </div>

          </div>

          <div class="form-group col-5">

               <label>First Name</label>

               <input type="text" formControlName="firstName"

class="form-control"

[ngClass]="{ 'is-invalid': submitted && f.firstName.errors }" />

                    <div \*ngIf="submitted && f.firstName.errors"

class="invalid-feedback">

                         <div \*ngIf="f.firstName.errors.required">

First Name is required

</div>

                    </div>

          </div>

          <div class="form-group col-5">

               <label>Last Name</label>

               <input type="text" formControlName="lastName"

class="form-control"

[ngClass]="{ 'is-invalid': submitted && f.lastName.errors }" />

                    <div \*ngIf="submitted && f.lastName.errors"

class="invalid-feedback">

                        <div \*ngIf="f.lastName.errors.required">

Last Name is required

</div>

                    </div>

             </div>

  </div>

            <div class="form-group">

                <label>Email</label>

                <input type="text" formControlName="email"

class="form-control"

[ngClass]="{ 'is-invalid': submitted && f.email.errors }" />

                <div \*ngIf="submitted && f.email.errors" class="invalid-feedback">

                    <div \*ngIf="f.email.errors.required">Email is required</div>

                    <div \*ngIf="f.email.errors.email">

Email must be a valid email address

</div>

                </div>

            </div>

            <div class="form-row">

                <div class="form-group col">

                    <label>Password</label>

                    <input type="password" formControlName="password" class="form-control" [ngClass]="{ 'is-invalid': submitted && f.password.errors }" />

                    <div \*ngIf="submitted && f.password.errors" class="invalid-feedback">

                        <div \*ngIf="f.password.errors.required">Password is required</div>

                        <div \*ngIf="f.password.errors.minlength">Password must be at least 6 characters</div>

                    </div>

                </div>

                <div class="form-group col">

                    <label>Confirm Password</label>

                    <input type="password" formControlName="confirmPassword" class="form-control" [ngClass]="{ 'is-invalid': submitted && f.confirmPassword.errors }" />

                    <div \*ngIf="submitted && f.confirmPassword.errors" class="invalid-feedback">

                        <div \*ngIf="f.confirmPassword.errors.required">Confirm Password is required</div>

                        <div \*ngIf="f.confirmPassword.errors.mustMatch">Passwords must match</div>

                    </div>

                </div>

            </div>

            <div class="form-group form-check">

                <input type="checkbox" formControlName="acceptTerms" id="acceptTerms" class="form-check-input" [ngClass]="{ 'is-invalid': submitted && f.acceptTerms.errors }" />

                <label for="acceptTerms" class="form-check-label">Accept Terms & Conditions</label>

                <div \*ngIf="submitted && f.acceptTerms.errors" class="invalid-feedback">Accept Ts & Cs is required</div>

            </div>

            <div class="text-center">

                <button class="btn btn-primary mr-1">Register</button>

                <button class="btn btn-secondary" type="reset" (click)="onReset()">Cancel</button>

            </div>

        </form>

App.component.ts

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

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

// import custom validator to validate that password and confirm password fields match

import { MustMatch } from './\_helpers/must-match.validator';

@Component({ selector: 'app', templateUrl: 'app.component.html' })

export class AppComponent implements OnInit {

    registerForm: FormGroup;

    submitted = false;

    constructor(private formBuilder: FormBuilder) { }

    ngOnInit() {

        this.registerForm = this.formBuilder.group({

            title: ['', Validators.required],

            firstName: ['', Validators.required],

            lastName: ['', Validators.required],

            email: ['', [Validators.required, Validators.email]],

            password: ['', [Validators.required, Validators.minLength(6)]],

            confirmPassword: ['', Validators.required],

            acceptTerms: [false, Validators.requiredTrue]

        }, {

            validator: MustMatch('password', 'confirmPassword')

        });

    }

    // convenience getter for easy access to form fields

    get f() { return this.registerForm.controls; }

    onSubmit() {

        this.submitted = true;

        // stop here if form is invalid

        if (this.registerForm.invalid) {

            return;

        }

        // display form values on success

        alert('SUCCESS!! :-)\n\n' + JSON.stringify(this.registerForm.value, null, 4));

    }

    onReset() {

        this.submitted = false;

        this.registerForm.reset();

    }

}

this.myForm.reset();

* ***Value changes and status changes***

Are property of form control, form array and form group class .value change and status change return observable return and we can subscribe them to get value.

Whatever we are changes in respective control that data print using value change

If control is empty then control status print using status change

 this.registerForm.valueChanges.subscribe((data : any) => {

        console.log("Name " + data)

      });

this.registerForm.statusChanges.subscribe((data : any) => {

        console.log(data)

      });

FormArray: - Use for generating dynamic control

<https://stackblitz.com/edit/form-array-angular?file=src%2Fapp%2Fapp.component.ts>

app.compoent.ts

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

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

@Component({

  selector: 'my-app',

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

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

})

export class AppComponent  {

  orderForm: FormGroup;

items: FormArray;

constructor(private formBuilder: FormBuilder) {}

ngOnInit() {

  this.orderForm = new FormGroup({

    items: new FormArray([])

  });

}

createItem(): FormGroup {

  return this.formBuilder.group({

    name: '',

    description: '',

    price: ''

  });

}

addItem(): void {

  this.items = this.orderForm.get('items') as FormArray;

  this.items.push(this.createItem());

}

}

App.compoent.html

<hello name="{{ name }}"></hello>

<p>

  Start editing to see some magic happen :)

</p>

<div [formGroup]="orderForm">

<div formArrayName="items"

  \*ngFor="let item of orderForm.get('items').controls; let i = index;">

  <div [formGroupName]="i">

    <input formControlName="name" placeholder="Item name">

    <input formControlName="description" placeholder="Item description">

    <input formControlName="price" placeholder="Item price">

  </div>

  Chosen name: {{ orderForm.controls.items.controls[i].controls.name.value }}

</div>

</div>

<button (click)="addItem()">Add</button>

­

  GetCorporateCandiateData(TestId): Observable<any> {

    return this.\_http.get<any>(this.rooturl + '/api/CorporateUser/GetCandiateData' + '/' + TestId).pipe(catchError(this.\_ErrorHandlerService.handleError));

  }

this.\_userSvc.GetCorporateCandiateData(this.TestId).subscribe(data => {})

**promise:**

* having one pipeline
* usually only use with async data return
* not easy to cancel

**observable:**

* are cancellable
* are re-triable by nature such as retry and retryWhen
* stream data in multiple pipelines
* having array-like operations like map, filter etc
* can be created from other sources like events
* they are functions, which could be subscribed later on

**Promise**

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

Another important thing to remember regarding promises is that a request initiated from a promise is not canceled.

 Disadvantages of Promise:

1. User could not cancel a request to the API.
2. User could not retry a failed call.
3. As our application gets bigger, promises become hard to manage.

**Observable**

An Observable is an array or a sequence of events over time. It has at least two participants, the creator (the data source) and the subscriber (subscription where data is being consumed). Compared to a promise, an observable can be canceled.

RxJS is all about unifying the ideas of promise callbacks and data flow and making them easier to work with. Observables provide operators, like map, forEach, reduce...similar to an array.

There are also powerful operators like retry(),  reply(), retryWhen(), delay().

Ng g guard auth

CanActivate

Interface that a class can implement to be a guard deciding if a route can be activated. If all guards return true, navigation continues. If any guard returns false, navigation is cancelled.

# CanActivateChild

Interface that a class can implement to be a guard deciding if a child route can be activated. If all guards return true, navigation continues. If any guard returns false, navigation is cancelled.

# CanDeactivate

Interface that a class can implement to be a guard deciding if a route can be deactivated. If all guards return true, navigation continues. If any guard returns false, navigation is cancelled.

# CanLoad

Interface that a class can implement to be a guard deciding if children can be loaded. If all guards return true, navigation continues. If any guard returns false, navigation is cancelled.

@ViewChild

1. **import** {
2. Component
3. } from '@angular/core';
4. @Component({
5. selector: 'app-secondcomponent',
6. templateUrl: './secondcomponent.component.html',
7. styleUrls: ['./secondcomponent.component.css']
8. })
9. **export** **class** SecondcomponentComponent {
10. constructor() {}
11. Test() {
12. **debugger**;
13. **return** "hello";
14. }
15. }

**SecondComponent.html**

1. <p>secondcomponent works!</p>

Now let’s use this second component in our main appcomponent.html with the help of selector “app-secondcomponent” so that our appcomponent.html looks as like below :

1. <div **class**="row">
2. <input type="text"><br/>
3. <button >Click Me</button>
4. </div>
5. <app-secondcomponent></app-secondcomponent>

Now, let’s make use of @viewchild to access the “Test()” in appcomponent.

1. **import** {
2. Component,
3. ViewChild,
4. ElementRef,
5. OnInit,
6. AfterViewInit
7. } from '@angular/core';
8. **import** {
9. SecondcomponentComponent
10. } from './secondcomponent/secondcomponent.component';
11. @Component({
12. selector: 'app-root',
13. templateUrl: './app.component.html',
14. styleUrls: ['./app.component.css']
15. })
16. **export** **class** AppComponent **implements** AfterViewInit {
17. title = 'Temprefapp';
18. @ViewChild(SecondcomponentComponent, {  **static**: **false**
19. }) vcvariable: SecondcomponentComponent
20. ngAfterViewInit() {
21. **debugger**;
22. console.log(**this**.vcvariable.Test());
23. }
24. }

After your application instantiates a component or directive by calling its constructor, Angular calls the hook methods you have implemented at the appropriate point in the lifecycle of that instance.

Angular executes hook methods in the following sequence. You can use them to perform the following kinds of operations.

|  |  |  |
| --- | --- | --- |
| **Hook method** | **Purpose** | **Timing** |
| **ngOnChanges()** | Respond when Angular sets or resets data-bound input properties. The method receives a [SimpleChanges](https://angular.io/api/core/SimpleChanges) object of current and previous property values.  Note that this happens very frequently, so any operation you perform here impacts performance significantly. See details in [Using change detection hooks](https://angular.io/guide/lifecycle-hooks#onchanges) in this document. | Called before ngOnInit() and whenever one or more data-bound input properties change.  Note that if your component has no inputs or you use it without providing any inputs, the framework will not call ngOnChanges(). |
| **ngOnInit()** | Initialize the directive or component after Angular first displays the data-bound properties and sets the directive or component's input properties. See details in [Initializing a component or directive](https://angular.io/guide/lifecycle-hooks#oninit) in this document. | Called once, after the first ngOnChanges(). |
| **ngDoCheck()** | Detect and act upon changes that Angular can't or won't detect on its own. See details and example in [Defining custom change detection](https://angular.io/guide/lifecycle-hooks#docheck) in this document. | Called immediately after ngOnChanges() on every change detection run, and immediately after ngOnInit() on the first run. |
| **ngAfterContentInit()** | Respond after Angular projects external content into the component's view, or into the view that a directive is in.  See details and example in [Responding to changes in content](https://angular.io/guide/lifecycle-hooks#aftercontent) in this document. | Called *once* after the first ngDoCheck(). |
| **ngAfterContentChecked()** | Respond after Angular checks the content projected into the directive or component.  See details and example in [Responding to projected content changes](https://angular.io/guide/lifecycle-hooks#aftercontent) in this document. | Called after ngAfterContentInit() and every subsequent ngDoCheck(). |
| **ngAfterViewInit()** | Respond after Angular initializes the component's views and child views, or the view that contains the directive.  See details and example in [Responding to view changes](https://angular.io/guide/lifecycle-hooks#afterview) in this document. | Called *once* after the first ngAfterContentChecked(). |
| **ngAfterViewChecked()** | Respond after Angular checks the component's views and child views, or the view that contains the directive. | Called after the ngAfterViewInit() and every subsequent ngAfterContentChecked(). |
| **ngOnDestroy()** | Cleanup just before Angular destroys the directive or component. Unsubscribe Observables and detach event handlers to avoid memory leaks. See details in [Cleaning up on instance destruction](https://angular.io/guide/lifecycle-hooks#ondestroy) in this document. | Called immediately before Angular destroys the directive or component. |

***ngOnChange***

App.Component.ts

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

@Component({

  selector: 'my-app',

  template: `

  <input [(ngModel)]="firstName">

  <input [(ngModel)]="lastName">

  <input [(ngModel)]="age" type="number">

  <app-doctor-card

    [firstName]="firstName"

    [lastName]="lastName"

    [age]="age"

    >

  </app-doctor-card>

  `,

  styles: [`input { display:  block; }`]

})

export class AppComponent {

  firstName = 'Tomasz'

  lastName = 'Kula'

  age = 28;

}

Doctor-card.compoent.ts

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

@Component({

  selector: 'app-doctor-card',

  template: `

  {{ firstName }} {{ lastName }} {{ age }}

  `

})

export class DoctorCardComponent implements OnChanges {

 @Input() firstName: string;

 @Input() lastName: string;

 @Input() age: number;

 fullName: string = ''

  ngOnChanges(changes: SimpleChanges) {

    if (changes.firstName || changes.lastName){

      this.fullName = `${this.firstName} ${this.lastName}`

    }

  }

}

**DoCheck**

App.compent.ts

import {

  Component,

  DoCheck,

  KeyValueDiffers,

  HostListener

} from "@angular/core";

@Component({

  selector: "my-app",

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

  styleUrls: ["./app.component.css"]

})

export class AppComponent implements DoCheck {

  name = "Angular 5";

  differ: any;

  hideTOP = true;

  constructor(private differs: KeyValueDiffers) {

    setTimeout(() => {

      this.name = "Angular 5 is New";

    }, 2000);

    this.differ = this.differs.find({}).create();

  }

  @HostListener("document:scroll")

  scroll() {

    console.log(document.body.scrollTop);

    const scrollHeight = Math.max(

      document.body.scrollTop,

      document.documentElement.scrollTop

    );

    if (scrollHeight > 20) {

      this.hideTOP = false;

    } else {

      this.hideTOP = true;

    }

  }

  ngDoCheck() {

    debugger;

    const change = this.differ.diff(this);

    if (change) {

      change.forEachChangedItem(item => {

        console.log("item changed", item);

      });

    }

  }

  gotoTop() {

    window.scrollTo(0, 0);

    this.hideTOP = true;

  }

}

**ngAfterViewInit**

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

import { TestComponentComponent } from './test-component/test-component.component';

@Component({

  selector: 'my-app',

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

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

})

export class AppComponent implements OnInit, AfterViewInit  {

  name = 'ViewChild() example';

  ngOnInit() {

    console.log('ngOnInit');

  }

  ngAfterViewInit() {

    console.log('ngAfterViewInit')

    console.log(this.testComponent.testNumber)

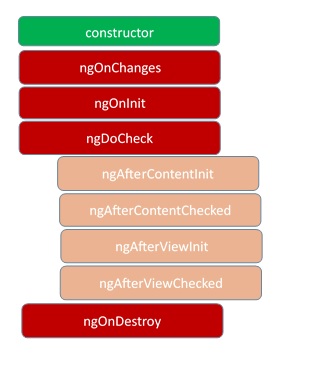
    console.log(this.testComponent.testString)

  }

  @ViewChild(TestComponentComponent) testComponent: TestComponentComponent;

  number: number;

}



|  |  |  |
| --- | --- | --- |
| State | Class if true | Class if false |
| The control has been visited | Ng-touched | Ng-untouched |
| Control’s value has changed | Ng-dirty | Ng-pristine |
| The Control value is valid | Ng-valid | Ng-invalid |

  constructor(private \_EnrollSvc: EnrollmentService,

              private fb :FormBuilder) {

    console.log("HI, this is component");

  }

registrationForm=this.fb.group({

  userName:['Avadhut'],

  Password:[''],

  ConfrimPassword:['']

})

  registrationForm = new FormGroup({

    userName: new FormControl('Avadhut'),

    Password : new FormControl(''),

    ConfrimPassword : new FormControl('')

  });

Changedetectorref in Angular 8

Change Detection means updating the DOM whenever data is changed. Angular provides two strategies for Change Detection.

abstract class [ChangeDetectorRef](https://angular.io/api/core/ChangeDetectorRef) {

[abstract **markForCheck**(): void](https://angular.io/api/core/ChangeDetectorRef#markForCheck)

[abstract **detach**(): void](https://angular.io/api/core/ChangeDetectorRef#detach)

[abstract **detectChanges**(): void](https://angular.io/api/core/ChangeDetectorRef#detectChanges)

[abstract **checkNoChanges**(): void](https://angular.io/api/core/ChangeDetectorRef#checkNoChanges)

[abstract **reattach**(): void](https://angular.io/api/core/ChangeDetectorRef#reattach)

}

detach() {

    this.changeDetectorRef.detach();

  }

  attach() {

    this.changeDetectorRef.reattach();

  }

  markForCheck() {

    this.changeDetectorRef.markForCheck();

  }

  trigger() {

    this.changeDetectorRef.detectChanges();

  }

Ahead-of-time (AOT) compilation

The Angular [ahead-of-time (AOT) compiler](https://angular.io/guide/glossary#aot) converts your Angular HTML and TypeScript code into efficient JavaScript code during the build phase before the browser downloads and runs that code.

Angular offers two ways to compile your application:

* *Just-in-Time* (JIT), which compiles your app in the browser at runtime. This was the default until Angular 8.
* *Ahead-of-Time* (AOT), which compiles your app and libraries at build time. This is the default since Angular 9.

How to create dynamic component

import { Component, ComponentFactoryResolver, ViewChild, ViewContainerRef } from '@angular/core';

import { DynamicComponent } from './dynamic/dynamic.component';

@Component({

  selector: 'my-app',

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

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

})

export class AppComponent  {

  @ViewChild('container', { read: ViewContainerRef }) container: ViewContainerRef;

  private \_counter = 1;

  constructor(private componentFactoryResolver: ComponentFactoryResolver) { }

  add(): void {

    // create the component factory

    const componentFactory = this.componentFactoryResolver.resolveComponentFactory(DynamicComponent);

    // add the component to the view

    const componentRef = this.container.createComponent(componentFactory);

    // pass some data to the component

    componentRef.instance.index = this.\_counter++;

  }

}

@ViewChild

App.compoent.html

<hello name="{{ name }}"></hello>

<div #myDiv>

</div>

App.Compoent.ts

@Component({

  selector: 'my-app',

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

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

})

export class AppComponent implements AfterViewInit {

  name = 'Angular 8 by Example: ElementRef';

  @ViewChild("myDiv") divView: ElementRef;

  ngAfterViewInit(){

    console.log(this.divView);

    this.divView.nativeElement.innerHTML = "Hello Angular 8!";

  }

}

Rxjs contain following member

FromEvent

Observable

Of

Map()

Filter()

Concat()

flatMap()

interval

https://stackoverflow.com/questions/49698640/flatmap-mergemap-switchmap-and-concatmap-in-rxjs

map(), filter(), concat(), and flatMap() these are the operators offered by the RxJS library. You can use pipes to link these operators together. Pipes let you combine multiple functions into a single function

import { filter, map } from 'rxjs/operators';

const squareOddVals = pipe(

filter((n: number) => n % 2 !== 0),

map(n => n \* n)

);

import { from, Observable } from 'rxjs';

import { interval } from 'rxjs'; // Create an Observable that will publish a value on an interval

const secondsCounter = interval(1000); // Subscribe to begin publishing values

const subscription = secondsCounter.subscribe(n => console.log(`It's been ${n + 1} seconds since subscribing!`));

FromEvent :- Angular

export class AppComponent implements AfterContentInit {

  name = 'Angular';

  @ViewChild('app') app;

ngAfterContentInit() {

const click$ = fromEvent(this.app.nativeElement, 'click');

const mouseOver$ = fromEvent(this.app.nativeElement, 'mouseover');

const mouseOut$ = fromEvent(this.app.nativeElement, 'mouseout');

click$.subscribe(val => console.log('Bölgeye tıklandı'));

mouseOver$.subscribe(val => console.log('mouse Bölgeye girdi'));

mouseOut$.subscribe(val => console.log('mouse Bölgeden çıktı'));

}

}

Pipe : Angular

1) **Pipes in Angular**: A pipe takes in data as input and transforms it to the desired output  
<https://angular.io/guide/pipes>

2) **pipe() function in RxJS**: You can use pipes to link operators together. Pipes let you combine multiple functions into a single function.

// RxJS

import { of, combineLatest } from 'rxjs';

// 1. Define multiple observables

let color = of('Black', 'Red', 'Blue');

let brand = of('Jaguar', 'Ford', 'BMW');

let price = of(100, 200, 300);

// 2. Call combineLatest operator, inject multiple observables in array

const joinStream = combineLatest(color, brand, price);

// 3. Subscibe combineLatest observable

const subscribe = joinStream.subscribe(([color, brand, price]) => {

console.log(color + ' ' + brand + ' ' + 'costed me' + ' ' + price );

})

const ob1$ = of( 1 , 2 , 3, 4 );

const ob2$ = interval( 1000 ).pipe(take( 4 ));

const result = concat(ob1$, ob2$);

result.subscribe(val => console.log(val));

 concat subscribes to each observables one at a time in a sequence and merges their result.

● it will subscribe to the first observable and emit its values until it completes, then move to the next.This will be repeated till the concat runs out of observables.

● when the last observable is completed, the concat also completes.

● at one time only one observable can emit values.

* **flatMap/mergeMap** - creates an Observable immediately for any source item, all previous Observables are kept alive
* **concatMap** - waits for the previous Observable to complete before creating the next one
* **switchMap** - for any source item, completes the previous Observable and immediately creates the next one

Question 1: Subject and Behavior subject

Question 2: View Child and content child

Question 3: host listener and host bind

Question 4: on push and change Detection

Question 5: Print number from 1 to 100 without using any variable

Animation :

Some of trigger status in animation

Rxjs Tutorial

Rxjs is a library which helps us in making async calls and implement callback mechanism

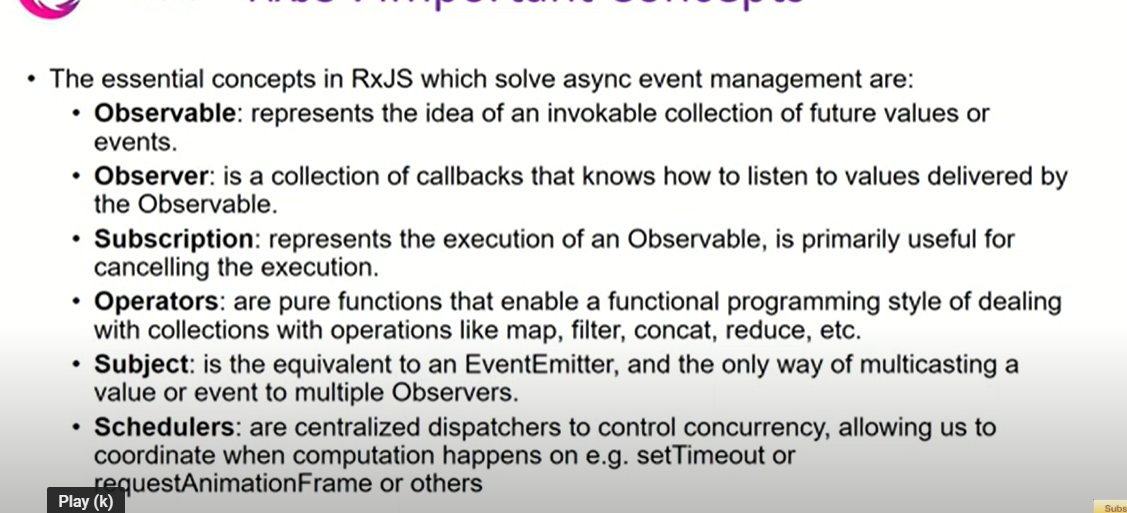
Rxjs consist of 3 main thing

* 1. observable
  2. satellite Data type – observers, schedulers, subject
  3. Operators
     1. Array methods (map, filter, reduce etc)
  4. Help us in data stream

Observable :- It’s a data stream

Continuously get the data over period of the time and which keep changing

Subscriber :- It will listen to the observable for data changes/updates



Operator is very important part of Rxjs

Operator in Rxjs is a pure function