**Answers for Questions**

1. Angular is a JavaScript framework which allows you to create reactive single-page-applications. It allows us to display dynamic data using typescript file
2. We can navigate around to multiple modules/functionalities in a single page html file. URL changes but it is same page(one single html). It gives the user very reactive experience. Very fast. Java scripts changes DOM, DOM changes which changes html in browser.
3. Started with Angular JS (Angular 1) has fundamental flaws. Angular 2 (re-write). Angular 4 …10,11,12…
4. It is used to bundle up the angular cli.5) It has npm(node package manager) which gives dependencies
5. Node js has npm(node package manager) which gives dependencies
6. Ng new project-name
7. Ng serve
8. In package.json
9. Dependencies for which angular uses
10. For end-to-end testing
11. Index.html
12. We use selector(app-root) adding inside index.html
13. Using {{string}}
14. It is a directive given by angular. It gives us bidirectional data binding. In input text if we give ngModel=”name” , it will update name in ts file and update html based on ts file
15. FormsModule from angular/forms
16. It is a super set of java script. Types, classes, Interfaces. It doesn’t run in browser. It compiled to javascript through cli. Java script is run in browser.
17. Bootstrap is used for better styling. Cmd to install: npm install --save bootstrap@3 . configure it to angular.json in styles block. To check if bootstrap is applied, inspect-sources-style.css-u can see bootstrap in comments.
18. Server initiates index.html where in the body we have app-root which is a selector of app.component. If u view the page-source, u can see scripts imported. They r imported by cli as bundles when we use serve cmd. These script imports have our code. First code executed is main.ts code.
19. Angular gets started then first code file it reads is main.ts file, in this file we mentioned AppModule which refers to App component. So App component rendered.
20. Components are key features. App component is root component. These allows us to separate complex application into diff parts and reusable.
21. Ng generate component server (or) ng g c server . ts file, html,css..
22. @Angular/core. Angular/core gves us core functionalities
23. Selector is unique-name we give to a component. Diff ways to write selector-check in servers component. and templateURL is html file path
24. Module is a bundle of functionalities. It says which features my Angular application uses. Angular bundles modules and have them as packages. We declare components in NgModule which comes from @angular/core.
25. Just creating a component, Angular will not load it. So we need to add the component in NgModule declarations
26. No webpack automatically takes the extensions while packing the bundle of code
27. It lets us to add other modules to our Application
28. If selector name is app-server then <app-server></app-server>
29. Using template instead of templateUrl u can directly write html code, Using ‘ single line html code ’. To use multiple lines of html code use ` code here `
30. Using styleUrls to give style.css file path. To use styling directly inside ts file use styles instead of styleUrls. styles: [` code here `]
31. Communication between typescript code and html template
32. One way and two-way data binding
33. String interpolation is a one way data binding technique to output the data from type script file to html code (use of {{serverID}} in html ) , u can also use method which returns string {{ getServerStatus() }
34. <button class="btn btn-primary" disabled>Add Server</button>
35. <button class="btn btn-primary" [disabled]="!allowNewServer">Add Server</button>
36. String interpolation: <p>{{allowNewServer}}</p>

Property binding : <p [innerHTML]="allowNewServer"></p>

If you want to change attribute vale then use property binding

1. For any events triggered by user we want something to happen

<button class="btn btn-primary" (click)="onCreateServer()">Add Server</button>

In ts file

onCreateServer() {

    this.serverCreationStatus = 'Server was created!';

  }

one-way data binding (this only updates the serverName value)

<input

type="text"

class="form-control"

(input)="onUpdateServerName($event)">

<p>{{serverName}}</p>

two-way data binding(this updates and displays the serverName value at same time)

<input

type="text"

class="form-control"

[(ngModel)]="serverName">

1. Directives are instructions in the DOM. Components(Directive with template) are instructions in the DOM.
2. Because it is a structural directive. It changes the structure of DOM

<p \*ngIf="serverCreated">Server was created, server name is {{serverName}}</p>

Ng-template is used to mark div in html

<p \*ngIf="serverCreated; else noServer">Server was created, server name is {{serverName}}</p>

<ng-template #noServer>

    <p #noServer>No Server was created</p>

</ng-template>

1. ngStyle is used to add styling values dynamically through ts file

<p [ngStyle]="{backgroundColor: getColor()}">Server with Id {{ serverId }} is {{ getServerStatus() }}</p>

1. Below online class gets added based on serverStatus condition

<p [ngClass]="{online: serverStatus === 'online'}">Server with Id {{ serverId }} is {{ getServerStatus() }}</p>

1. <app-server \*ngFor="let server of servers"></app-server>

<div \*ngFor="let logItem  of log; let i = index"

[ngStyle]="{backgroundColor: i>=4 ? 'blue' : 'transparent'}"

[ngClass]="{'whiteText': i>=4}"

>{{logItem}}</div>

1. ng g c recipes --skip-tests
2. ng g c recipes/recipe-list --skip-tests
3. It has 12 columns. col-md is for medium screen responsiveness. Md-5 is 5 columns and md-7 is seven columns width

<div class="row">

    <div class="col-md-5">

        <app-recipe-list></app-recipe-list>

    </div>

    <div class="col-md-7">

        <app-recipe-item></app-recipe-item>

    </div>

  </div>

1. <hr>
2. Model is a structure/ Object we use throughout the application. Blueprint of an object

export class Recipe {

    public name : string;

    public description: string;

    public imagePath : string;

    constructor(name: string, desc: string, imagePath: string){

        this.name = name;

        this.description = desc;

        this.imagePath = imagePath;

    }

}

1. <a

        href="#"

        class="list-group-item clearfix"

        \*ngFor="let recipe of recipes">

        <div class="pull-left">

          <h4 class="list-group-item-heading">{{ recipe.name }}</h4>

          <p class="list-group-item-text">{{ recipe.description }}</p>

        </div>

1. <a

        href="#"

        class="list-group-item clearfix"

        \*ngFor="let recipe of recipes">

        <span class="pull-right">

          <img

            [src]="recipe.imagePath"

            alt="{{ recipe.name }}"

            class="img-responsive"

            style="max-height: 50px;">

        </span>

      </a>

1. Yes we can use interface and constructor is not required
2. Check in console first (read which file and line it points to)
3. Debugging is given by developer tools. rightClick-> sources-> ctrl+p type file name -> debug point.In webpack you an directly access ts files. When we run the app it pauses at the debug point. When we hover on variable
4. Yes we can also trigger event using directives
5. Input is used to set property from parent component to child component

In parent component

<app-server-element

        [srvElement]="serverElement"

      [name]="serverElement.name">

</app-server-element>

In server component (child component) (two types of accessing)

  @Input('srvElement') element: {type: string, name: string, content: string};

  @Input() name: string;

1. output directive is used to binding object from child component to parent component

In child component

@Output() serverCreated = new EventEmitter<{serverName: string, serverContent: string}>();

  @Output('bpCreated') blueprintCreated = new EventEmitter<{serverName: string, serverContent: string}>();

In parent component

<app-cockpit

    (serverCreated)="onServerAdded($event)"

    (bpCreated)="onBlueprintAdded($event)"

  ></app-cockpit>

onServerAdded(serverData: {serverName: string, serverContent: string}) {

    this.serverElements.push({

      type: 'server',

      name: serverData.serverName,

      content: serverData.serverContent

    });

  }

1. View Encapsulation is that Angular encapsulate the css for that particular component (diff attributes being added by default to give styles for that particular component)
2. Shadow DOM is that Angular gives separate shadow DOM for each components
3. If we disable view encapsulation, the css we have given for a component will be applied across all the components globally

@Component({

  selector: 'app-server-element',

  templateUrl: './server-element.component.html',

  styleUrls: ['./server-element.component.css'],

  encapsulation: ViewEncapsulation.Emulated // None, Native None will disable it completely. Native will keep shadow DOM

1. Local reference is used to reference for that whole html element with all its properties. We use them only in html file template

<input

    type="text"

      class="form-control"

      #serverNameInput>

    <label>Server Content</label>

<button

      class="btn btn-primary"

      (click)="onAddServer(serverNameInput)">Add Server</button>

In ts file

onAddServer(nameInput: HTMLInputElement) {

    this.serverCreated.emit({

      serverName: nameInput.value,

      serverContent: this.serverContentInput.nativeElement.value

    });

  }

1. If we need to access the Element directly from ts file instead of passing through method.

 @ViewChild('serverContentInput', { static: false }) serverContentInput: ElementRef;

serverContent: this.serverContentInput.nativeElement.value

1. Lets say we have app.component as parent and it has two child components cockpit and servers. If we need to change data in servers component based on cockpit component

<div class="panel-body">

    <ng-content></ng-content>

  </div>

1. First constructor gets called then ngOnit gets called

ngOnChanges – called after a bound input property changes (this will take parameter

ngOnChanges(changes: SimpleChanges) {

    console.log('ngOnChanges called!');

    console.log(changes);

  }

)

ngOnInit – called once the component is initialized(doesn’t gives access to html element values. You can access only afterviewInit)

ngDoCheck – Called during every detection run (whenever angular checks for any changes. Lot of triggers will run)

ngAfterContentinit – called after content (ng-content) has been projected into view (It gets called only once when the content is initialized and displayed)

ngAfterContentChecked - called every time the projected content has been checked (It gets called after each change detection cycle. Means after every doCheck it gets called)

ngAfterViewInit – called after the components view (and child views) has been initialized (It happens after content checked) (gives access to html element values)

ngAfterViewChecked – called every time the view (and child views) have been checked

ngOnDestroy – called once the component gets destroyed

1. When using @child we cannot access html element content which came through ng-content(from another component). We can access this type of content using

@ContentChild('contentParagraph', {static: true}) paragraph: ElementRef;

1. Create Directive

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

@Directive({

  selector: '[appBasicHighlight]'

})

export class BasicHighlightDirective implements OnInit {

  constructor(private elementRef: ElementRef) {

  }

  ngOnInit() {

    this.elementRef.nativeElement.style.backgroundColor = 'green';

  }

}

Import it in app module declarations

In HTML

<p appBasicHighlight>Style me with basic directive!</p>

Below we are changing the color dynamically on Hover

Below is the custom directive

import {Directive,Renderer2,OnInit,ElementRef,HostListener,HostBinding,Input} from '@angular/core';

@Directive({

  selector: '[appBetterHighlight]'

})

export class BetterHighlightDirective implements OnInit {

  @Input() defaultColor: string = 'transparent';

  @Input('appBetterHighlight') highlightColor: string = 'blue';

  @HostBinding('style.backgroundColor') backgroundColor: string; // instead of using renderer we can use HostBinding

  constructor(private elRef: ElementRef, private renderer: Renderer2) { }

  ngOnInit() {

    this.backgroundColor = this.defaultColor;

    // this.renderer.setStyle(this.elRef.nativeElement, 'background-color', 'blue');

  }

  @HostListener('mouseenter') mouseover(eventData: Event) {

    // this.renderer.setStyle(this.elRef.nativeElement, 'background-color', 'blue');

    this.backgroundColor = this.highlightColor;

  }

  @HostListener('mouseleave') mouseleave(eventData: Event) {

    // this.renderer.setStyle(this.elRef.nativeElement, 'background-color', 'transparent');

    this.backgroundColor = this.defaultColor;

  }

}

In HTML

<p [appBetterHighlight]="'red'" defaultColor="yellow">Style me with a better directive!</p>

1. Structural directive with a \*. Angular transform them from \*ngIf to <ng-template [ngIf]=”!onlyodd”>
2. import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';

@Directive({

  selector: '[appUnless]'

})

export class UnlessDirective {

  @Input() set appUnless(condition: boolean) {

    if (!condition) {

      this.vcRef.createEmbeddedView(this.templateRef);

    } else {

      this.vcRef.clear();

    }

  }

  constructor(private templateRef: TemplateRef<any>, private vcRef: ViewContainerRef) { }

}

In HTML

<div \*appUnless="onlyOdd">

1. ngSwitch is used when we have lot of ngIf cases

<div [ngSwitch]="value">

        <p \*ngSwitchCase="5">Value is 5</p>

        <p \*ngSwitchCase="10">Value is 10</p>

        <p \*ngSwitchCase="100">Value is 100</p>

        <p \*ngSwitchDefault>Value is Default</p>

      </div>

1. If we have similar/same data/methods in multiple components

export class LoggingService {

  logStatusChange(status: string) {

    console.log('A server status changed, new status: ' + status);

  }

}

1. Angular construct a component so if we give constructor it will create accordingly

@Component({

  selector: 'app-account',

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

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

   providers: [LoggingService]

})

constructor(private loggingService: LoggingService) {}

  onSetTo(status: string) {

    this.loggingService.logStatusChange(status);

  }

Instead of giving providers in @Component we can mention all the providers in app.module

1. Angular dependency injector is Hierarchical injector. If we want our service data to be available/updated then we need to mention services in ‘app.module’.

If we provide them in app.component they will be available for all components in app.component

**If we provide them in any particular component Angular will create a new instance of service**

1. **@Injectable is a metadata which says angular that this service can inject another service to it**

@Injectable()

export class AccountsService {

.

.

constructor(private loggingService: LoggingService) {}

@Injectable(

{ providedIn: 'root' }

)

Above says that provide our service at root level of the application

1. In service

statusUpdated = new EventEmitter<string>();

In first component (passing updated data to it)

this.accountsService.statusUpdated.emit(status);

In second component (whenever there is a change below code will be triggered)

this.accountsService.statusUpdated.subscribe(

      (status: string) => alert('New Status: ' + status)

    );

1. We cant add all the components in single page if those are completely looks different. So we need to load other components in different page
2. In app.module.ts. Can also create separate module and add that inside app.module.ts since routing takes good amount of space

const appRoutes: Routes = [

  { path: '', component: HomeComponent },

  { path: 'users', component: UsersComponent, children: [

    { path: ':id/:name', component: UserComponent }

  ] },

  {

    path: 'servers',

    // canActivate: [AuthGuard],

    canActivateChild: [AuthGuard],

    component: ServersComponent,

    children: [

    { path: ':id', component: ServerComponent, resolve: {server: ServerResolver} },

    { path: ':id/edit', component: EditServerComponent, canDeactivate: [CanDeactivateGuard] }

  ] },

  // { path: 'not-found', component: PageNotFoundComponent },

  { path: 'not-found', component: ErrorPageComponent, data: {message: 'Page not found!'} },

  { path: '\*\*', redirectTo: '/not-found' }

];

@NgModule({

  imports: [

    // RouterModule.forRoot(appRoutes, {useHash: true})

    RouterModule.forRoot(appRoutes)

  ],

  exports: [RouterModule]

})

1. From above routes when user goes to that route page is redirected but page gets reloaded which will remove previous states. When we give RouterLink in html it wont reload. It just goes to component. routerLink catches the click and prevents the default request, it pauses and check the perfect route to load

<li role="presentation"

            routerLinkActive="active"

            [routerLinkActiveOptions]="{exact: true}">

          <a routerLink="/">Home</a>

        </li>

        <li role="presentation"

            routerLinkActive="active">

          <a routerLink="servers">Servers</a>

        </li>

1. If the current url is http://localhost:8080/servers

routerLink = “servers” is relative path – it appends to the current url try to go to (<http://localhost:8080/servers/servers>)

routerLink = “/servers” - it appends to the base url/root domain try to go to (<http://localhost:8080/servers>)

routerLink = “../servers”– it goes one step back and appends to the url try to go to (<http://localhost:8080/servers>)

1. This will make the tab active

<li role="presentation"

            routerLinkActive="active">

          <a routerLink="/servers">Servers</a>

        </li>

1. Above will make active if the servers are their in url match

routerLinkActiveOptions says only mark this if we have full path on the router link

<li role="presentation"

            routerLinkActive="active"

            [routerLinkActiveOptions]="{exact: true}">

          <a routerLink="/">Home</a>

        </li>

1. onLoadServer(id: number) {

    // complex calculation

    this.router.navigate(['/servers'])

  }

1. constructor(private serversService: ServersService,

              private router: Router,

              private route: ActivatedRoute)

onReload() {

    this.router.navigate(['servers'], {relativeTo: this.route});

  }

1. In routing module

{ path: 'users', component: UsersComponent, children: [

    { path: ':id/:name', component: UserComponent }

  ] }

1. this.route.params

      .subscribe(

        (params: Params) => {

          this.id = params[‘id’];

        }

      );

1. snapshot will not change the value if the params has been changed if the component is not reloaded and user is in same component. Angular cleans up the route subscription when user leaves the component. But its better to implement ngOnDestroy this.subscription.unsubscribe();

paramsSubscription: Subscription;

ngOnDestroy() {

    this.paramsSubscription.unsubscribe();

  }

1. [routerLink]="['/servers', 5, ‘edit’]" --- http://localhost:8080/servers/5/edit

[routerLink]="['/servers', 5, ‘edit’]"

[queryParams]="{allowEdit: '1'}" --- <http://localhost:8080/servers/5/edit?allowEdit=1>

[routerLink]="['/servers', 5, ‘edit’]"

[queryParams]="{allowEdit: '1'}"

[fragmanet]=”loading” --- <http://localhost:8080/servers/5/edit?allowEdit=1>#loading

Or in ts we can add like below

onLoadServer(id: number) {

    // complex calculation

    this.router.navigate(['/servers', id, 'edit'], {queryParams: {allowEdit: '1'}, fragment: 'loading'});

  }

this.route.queryParams

      .subscribe(

        (queryParams: Params) => {

          this.allowEdit = queryParams['allowEdit'] === '1' ? true : false;

        }

      );

<a

        [routerLink]="['/servers', server.id]"

        [queryParams]="{allowEdit: server.id === 3 ? '1' : '0'}"

1. onEdit() {

    this.router.navigate(['edit'], {relativeTo: this.route, queryParamsHandling: 'preserve'});

  }

1. Below \*\* will redirect to errorComponent if any invalid route given

{ path: 'not-found', component: ErrorPageComponent, data: {message: 'Page not found!'} },

  { path: '\*\*', redirectTo: '/not-found' }

1. When we want to make user to access component when he is loggedin/authenticated… we use guards. To protect routes
2. canActivate and canActivateChild given by Angular. Implementing these will make you implement them mandatory

@Injectable()

export class AuthGuard implements CanActivate, CanActivateChild {

  constructor(private authService: AuthService, private router: Router) {}

  canActivate(route: ActivatedRouteSnapshot,

              state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {

    return this.authService.isAuthenticated()

      .then(

        (authenticated: boolean) => {

          if (authenticated) {

            return true;

          } else {

            this.router.navigate(['/']);

          }

        }

      );

  }

  canActivateChild(route: ActivatedRouteSnapshot,

                   state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {

    return this.canActivate(route, state);

  }

}

1. promise returns a Boolean based on condition given inside that promise. Based on that Boolean value we will be executing the statements

 isAuthenticated() {

    const promise = new Promise(

      (resolve, reject) => {

        setTimeout(() => {

          resolve(this.loggedIn);

        }, 800);

      }

    );

    return promise;

  }

Use of promise

this.authService.isAuthenticated()

      .then(

        (authenticated: boolean) => {

          if (authenticated) {

            return true;

          } else {

            Return “failed”;

          }

        }

      );

1. {path: 'servers', canActivate: [AuthGuard], component: ServersComponent}
2. {

    path: 'servers',

    // canActivate: [AuthGuard],

    canActivateChild: [AuthGuard],

    component: ServersComponent,

    children: [

    { path: ':id', component: ServerComponent, resolve: {server: ServerResolver} },

    { path: ':id/edit', component: EditServerComponent, canDeactivate: [CanDeactivateGuard] }

  ] }

export interface CanComponentDeactivate {

  canDeactivate: () => Observable<boolean> | Promise<boolean> | boolean;

}

export class CanDeactivateGuard implements CanDeactivate<CanComponentDeactivate> {

  canDeactivate(component: CanComponentDeactivate,

                currentRoute: ActivatedRouteSnapshot,

                currentState: RouterStateSnapshot,

                nextState?: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {

    return component.canDeactivate();

  }

}

In routing

{ path: ':id/edit', component: EditServerComponent, canDeactivate: [CanDeactivateGuard] }

{ path: 'not-found', component: ErrorPageComponent, data: {message: 'Page not found!'} }

 this.route.data.subscribe(

      (data: Data) => {

        this.errorMessage = data['message'];

      }

    );

Observables are various data sources like events, Http Requests, triggered in code. In these we can handle data, handle error, handle completion. They are asynchronous tasks because we don’t know when they will happen and when will they complete.

1. Install rxjs . npm install --save rxjs@6

Npm install --save rxjs-compat

1. Params is Observable

interval(1000).subscribe(count => {

      console.log(count);

    });

Above code runs when that component got loaded.

1. Even if we goes to different component Observable still runs in the background. So we have to destroy the it to not to make memory increase unnecessarily. Also If we load the component again a new Observable gets started while the previous one is already running so we should destroy the observable.

this.firstObsSubscription = interval(1000).subscribe(count => {

      console.log(count);

    });

ngOnDestroy(): void {

this.firstObsSubscription.unsubscribe();}

1. // this.firstObsSubscription = interval(1000).subscribe(count => {

    //   console.log(count);

    // });

    const customIntervalObservable = Observable.create(observer => {

      let count = 0;

      setInterval(() => {

        observer.next(count);

        if (count === 2) {

          observer.complete();

        }

        if (count > 3) {

          observer.error(new Error('Count is greater 3!'));

        }

        count++;

      }, 1000);

    });

    this.firstObsSubscription = customIntervalObservable.subscribe(data => {

      console.log(data);

    }, error => {

      console.log(error);

      alert(error.message);

    }, () => {

      console.log('Completed!');

    });

  }

1. In above code if the count is 3 it errors out and ends the Observable. But when count count is 2 it is completed

In above code you can see error =>{code when it got errored out}

In above code you can see completed ()=>{code when completed}

1. Sometimes we don’t need raw data. We can filter out the content coming from subscription. Using these Operators we can filter the data
2. Subscribe will return you complete raw data.

Map operator is used to transform the data before using it.

Filter operator is used to filter data before using it. There are lot of operators

 this.firstObsSubscription = customIntervalObservable.pipe(filter(data => {

      return data > 0;

    }), map((data: number) => {

      return 'Round: ' + (data + 1);

    })).subscribe(data => {

      console.log(data);

    }, error => {

      console.log(error);

      alert(error.message);

    }, () => {

      console.log('Completed!');

    });

1. Subject is similar to Event emitter but when we use .next() (in eventEmitter we use .emit()) it will trigger all the subscribe methods for that Subject in all the components.

Subject can’t be used in @Output. We can only use Angular provided EventEmitter for that

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

import { Subject } from 'rxjs';

@Injectable({providedIn: 'root'})

export class UserService {

  activatedEmitter = new Subject<boolean>();

}

If we want to update activatedEmitter value in all the components

onActivate() {

    this.userService.activatedEmitter.next(true);}

Below will be called when we call above code .next()

this.activatedSub = this.userService.activatedEmitter.subscribe(didActivate => {

      this.userActivated = didActivate;

    });

1. Since we are creating a single page application we will need to submit form through Angular
2. Template-Driven : Angular infers the form Object from the DOM

Reactive : Form is created programmatically and redered in DOM

<form (ngSubmit)="onSubmit()" #f="ngForm">

        <div id="user-data" ngModelGroup="userData" #userData="ngModelGroup">

          <div class="form-group">

            <label for="username">Username</label>

            <input type="text" id="username" class="form-control"

              ngModel name="username" required>

          </div>

          <button class="btn btn-default" type="button" (click)="suggestUserName()">Suggest an Username</button>

          <div class="form-group">

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

            <input type="email" id="email" class="form-control" ngModel name="email" required email #email="ngModel">

            <span class="help-block" \*ngIf="!email.valid && email.touched">Please enter a valid email!</span>

          </div>

        </div>

        <p \*ngIf="!userData.valid && userData.touched">User Data is invalid!</p>

        <div class="form-group">

          <label for="secret">Secret Questions</label>

          <select id="secret" class="form-control" [ngModel]="defaultQuestion" name="secret">

            <option value="pet">Your first Pet?</option>

            <option value="teacher">Your first teacher?</option>

          </select>

        </div>

        <div class="form-group">

          <textarea name="questionAnswer" rows="3" class="form-control" [(ngModel)]="answer"></textarea>

        </div>

        <p>Your reply: {{ answer }}</p>

        <div class="radio" \*ngFor="let gender of genders">

          <label>

            <input type="radio" name="gender" ngModel [value]="gender" required>

            {{ gender }}

          </label>

        </div>

        <button class="btn btn-primary" type="submit" [disabled]="!f.valid">Submit</button>

      </form>

Ts file:

onSubmit() {

    this.submitted = true;

    this.user.username = this.signupForm.value.userData.username;

    this.user.email = this.signupForm.value.userData.email;

    this.user.secretQuestion = this.signupForm.value.secret;

    this.user.answer = this.signupForm.value.questionAnswer;

    this.user.gender = this.signupForm.value.gender;

    this.signupForm.reset();

  }

* ngModel should be given in all the fields or inputs because it exposes the data by making it a control and it can be recognized by ts code

**Template-driven approach**

* On click of submit in the html:

ngSubmit lets you to submit.

#f="ngForm"

 @ViewChild('f', { static: false }) signupForm: NgForm;

Above code will allow you to access values – check in main code above. **We get all values based on name attribute**

* Validation

Required will make it mandatory to enter the field

For type=”email” email directive will make user to enter valid email

[disabled]=”!f.valid” for button – if the form is valid then only button will be enabled

You can add css for invalid. Angular adds inbuilt classes like ng-valid if the field is valid, ng-invalid, ng-touched, ng-dirty(if the data changed).

Input .ng-invalid.ng-touched {

Border:1px solid red; // this will display red border when user touched the field and invalid data

}

To display error message

 <input type="email" id="email" class="form-control" ngModel name="email" required email

              #email="ngModel">

            <span class="help-block" \*ngIf="!email.valid && email.touched">Please enter a valid email!</span>

* Default values should be displayed – one way binding
* <select id="secret"  class="form-control"
* [ngModel]="defaultQuestion"  name="secret">
* <option value="pet">Your first Pet?</option>
* <option value="teacher">Your first teacher?</option>
* </select>

In ts file

defaultQuestion = 'teacher';

* Using two way binding to display updated value
* <div class="form-group">
* <textarea name="questionAnswer" rows="3" class="form-control"
* [(ngModel)]="answer"></textarea>
* </div>
* <p>Your reply: {{ answer }}</p>
* Making two input fields as a group. Below username and email inputs form-groups are in ngModelGroup.
* <div id="user-data" ngModelGroup="userData" #userData="ngModelGroup">
* <div class="form-group">
* <label for="username">Username</label>
* <input type="text" id="username" class="form-control"
* ngModel name="username" required>
* </div>
* <button class="btn btn-default" type="button" (click)="suggestUserName()">Suggest an Username</button>
* <div class="form-group">
* <label for="email">Mail</label>
* <input type="email" id="email" class="form-control" ngModel name="email" required email #email="ngModel">
* <span class="help-block" \*ngIf="!email.valid && email.touched">Please enter a valid email!</span>
* </div>
* </div>

Displaying error message based on complete group validation

<p \*ngIf="!userData.valid && userData.touched">User Data is invalid!</p>

* Radio button implementation
* <div class="radio" \*ngFor="let gender of genders">
* <label>
* <input type="radio" name="gender" ngModel [value]="gender" required>
* {{ gender }}
* </label></div>
* Setting and patching form values
* this.signupForm.setValue({
* userData: {
* username: suggestedName,
* email: ''
* },
* secret: 'pet',
* questionAnswer: '',
* gender: 'male'
* });

**Better approach below: if we use patch value it updates the given value and it remains other values as it is**

this.signupForm.form.patchValue({

      userData: {

        username: suggestedName

      }

    });

* Resetting forms. Empty all the inputs and reset the state for properties, attributes…. Its like the page got loaded again

this.signupForm.reset();

**Reactive approach**

* Add reactiveFormsModule to app.module.ts

Html code:

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

        <div formGroupName="userData">

          <div class="form-group">

            <label for="username">Username</label>

            <input type="text" id="username" formControlName="username" class="form-control">

            <span \*ngIf="!signupForm.get('userData.username').valid && signupForm.get('userData.username').touched"

              class="help-block">

              <span \*ngIf="signupForm.get('userData.username').errors['nameIsForbidden']">This name is invalid!</span>

              <span \*ngIf="signupForm.get('userData.username').errors['required']">This field is required!</span>

            </span>

          </div>

          <div class="form-group">

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

            <input type="text" id="email" formControlName="email" class="form-control">

            <span

              \*ngIf="!signupForm.get('userData.email').valid && signupForm.get('userData.email').touched"

              class="help-block">Please enter a valid email!</span>

          </div>

        </div>

        <div class="radio" \*ngFor="let gender of genders">

          <label>

            <input

              type="radio"

              formControlName="gender"

              [value]="gender">{{ gender }}

          </label>

        </div>

        <div formArrayName="hobbies">

          <h4>Your Hobbies</h4>

          <button

            class="btn btn-default"

            type="button"

            (click)="onAddHobby()">Add Hobby</button>

          <div

            class="form-group"

            \*ngFor="let hobbyControl of hobbyControls; let i = index">

            <input type="text" class="form-control" [formControlName]="i">

          </div>

        </div>

        <span

          \*ngIf="!signupForm.valid && signupForm.touched"

          class="help-block">Please enter valid data!</span>

        <button class="btn btn-primary" type="submit">Submit</button>

      </form>

* Ts file code initializing
* genders = ['male', 'female'];
* signupForm: FormGroup;
* forbiddenUsernames = ['Chris', 'Anna'];
* Creating formGroup
* this.signupForm = new FormGroup({
* userData: new FormGroup({
* username: new FormControl(null),
* email: new FormControl(null),
* }),
* gender: new FormControl('male'),
* hobbies: new FormArray([]),
* });
* To sync formGroup with ts code :
* <form [formGroup]="signupForm" (ngSubmit)="onSubmit()">

For form controls instead of name should give formControlName

<input type="text" id="username" formControlName="username" class="form-control">

* Submitting the form

We use same approach

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

In ts file to fetch the form

signupForm: FormGroup;

onSubmit() {

    console.log(this.signupForm);

    this.signupForm.reset();

  }

* Adding validation in ts code
* this.signupForm = new FormGroup({
* userData: new FormGroup({
* username: new FormControl(null, [Validators.required]),
* email: new FormControl(null, [Validators.required, Validators.email])
* }),
* gender: new FormControl('male'),
* hobbies: new FormArray([]),
* });
* Adding new control

<div formArrayName="hobbies">

          <h4>Your Hobbies</h4>

          <button

            class="btn btn-default"

            type="button"

            (click)="onAddHobby()">Add Hobby</button>

          <div

            class="form-group"

            \*ngFor="let hobbyControl of hobbyControls; let i = index">

            <input type="text" class="form-control" [formControlName]="i">

          </div>

        </div>

Ts code

get hobbyControls() {

    return (this.signupForm.get('hobbies') as FormArray).controls;

  }

onAddHobby() {

    const control = new FormControl(null, Validators.required);

    (<FormArray>this.signupForm.get('hobbies')).push(control);

  }

* If we don’t want to allow some Usernames or emails. Below username is normal validation but email is async validator

this.signupForm = new FormGroup({

      userData: new FormGroup({

        username: new FormControl(null, [ Validators.required, this.forbiddenNames.bind(this),

        ]),

        email: new FormControl(null, [Validators.required, Validators.email], this.forbiddenEmails),

      }),

      gender: new FormControl('male'),

      hobbies: new FormArray([]),

    });

forbiddenNames(control: FormControl): { [s: string]: boolean } {

    if (this.forbiddenUsernames.indexOf(control.value) !== -1) {

      return { nameIsForbidden: true };

    }

    return null;

  }

  forbiddenEmails(control: FormControl): Promise<any> | Observable<any> {

    const promise = new Promise<any>((resolve, reject) => {

      setTimeout(() => {

        if (control.value === 'test@test.com') {

          resolve({ emailIsForbidden: true });

        } else {

          resolve(null);

        }

      }, 1500);

    });

    return promise;

  }

* How to trigger a method when there is a status/data change in the form
* this.signupForm.statusChanges.subscribe((status) => console.log(status));
* setting/patching values
* this.signupForm.setValue({userData: {
* username: 'Max',
* email: 'max@test.com',
* },
* gender: 'male',
* hobbies: [],
* });
* this.signupForm.patchValue({
* userData: {
* username: 'Anna',
* },
* });

1. -----above
2. Allows to transforms some output inside template
3. {{server.name | uppercase}}
4. Tranforms date time with milliseconds to more readable date