Angular question and ofc learning

Use salesforce knowledge also -> while your interview that also good

Learn through note – no video – micro frontend

1. Package and package – lock
2. Entry components
3. Sonar questions , ts lint , vs code extension , tslint issue sonar , %20 no issue in route with template string – bugs , critical bugs , complexity ,
4. High and low level idea – how things working , input op
5. Git commands , bitbucket , Jenkins , jira
6. Microfrontend arch , vendor common , vendor and site – purpose of different module and repo
7. Cross dependency , circular dependancy
8. Routing – redirectTo , pathmatch ,
9. Wrapper component

Steps ->

1. Basic from office work -> SSP , testing , rxjs , arch related , basic question searching , Api call
2. Plural all angular – speed up
3. All last questions and ngrx
4. Other repo check and compare other design pattern – Kiran – what all design pattern u use and check their codebase
5. LLD and HLD , code base , design pattern

Basics of Angular

webpack-bundle-analyzer

Declaration , imports , exports , entry components , export of comp if you want using selector

Multiple use – Lib , site ,

Entry components -

Specifies a list of components that should be compiled when this module is defined. For each component listed here, Angular will create a ComponentFactory and store it in the ComponentFactoryResolver.

---Update Angular 9 or Angular 8 with Ivy explicitly enabled---

Entry Components With Ivy are not required anymore and now are deprecated

---for Angular 9 and 8 with Ivy disabled---

In the case of a dynamically loaded component and in order for a ComponentFactory to be generated, the component must also be added to the module’s entryComponents:

-> package.json and package-lock.json difference – above angular 5 , npm I both created

To avoid differences in installed dependencies on different environments and to generate the same results on every environment we should use the package-lock. json file to install dependencies. ... json file and you will able to generate the same results as you developed with that particular package

* Why is package-lock.json created?
* What is the purpose or use of package-lock.json?
* Why should we commit package-lock.json with our project source code?

Ambiguity within different environment ,  it will install the exact latest version of that package in your project and save the dependency in package.json with a carat (**^**) sign. Like, if the current version of a package is 5.2.3 then the installed version will be 5.2.3 and the saved dependency will be ^5.2.3. Carat (^) means it will support any higher version with major version 5 like 5.3.1 and so on. Here, package-lock.json is created for locking the dependency with the installed version.

**What is the purpose or use of package-lock.json?**

To avoid differences in installed dependencies on different environments and to generate the same results on every environment we should usethe **package-lock.json** file to install dependencies.

Ideally, this file should be on your source control with the package.json file so when you or any other user will clone the project and run the command “npm i”, it will install the exact same version saved in package-lock.json file and you will able to generate the same results as you developed with that particular package.

## Why should we commit package-lock.json with our project source code?

During deployment, when you again run “npm i” with the same package.json file without the package-lock.json, the installed package might have a higher version now from what you had intended.

Now, what if you wanted to have that particular version for your dependency during deployment which you used at the time of development. This is the need of creating a package-lock.json file and keeping it with the source code. This file is created with the details of the specific version installed in your project.

Keep locking your dependencies!!

Entry components

Entry components are deprecated,

This is for dynamically added components that are added using ViewContainerRef.createComponent(). Adding them to entryComponents tells the offline template compiler to compile them and create factories for them

The components registered in route configurations are added automatically to entryComponents as well because router-outlet also uses ViewContainerRef.createComponent() to add routed components to the DOM.

If you don't list a dynamically added component to entryComponents you'll get an error message a bout a missing factory because Angular won't have created one.

A component loaded declaratively via its selector is not an entry component.

Most application components are loaded declaratively. Angular uses the component's selector to locate the element in the template. It then creates the HTML representation of the component and inserts it into the DOM at the selected element. These aren't entry components.

A few components are only loaded dynamically and are never referenced in a component template.

The bootstrapped root AppComponent is an entry component. True, its selector matches an element tag in index.html. But index.html isn't a component template and the AppComponent selector doesn't match an element in any component template.

Angular loads AppComponent dynamically because it's either listed by type in @NgModule.bootstrap or boostrapped imperatively with the module's ngDoBootstrap method.

Components in route definitions are also entry components. A route definition refers to a component by its type. The router ignores a routed component's selector (if it even has one) and loads the component dynamically into a RouterOutlet.

The compiler can't discover these entry components by looking for them in other component templates. You must tell it about them by adding them to the entryComponents list.

Angular automatically adds the following types of components to the module's entryComponents:

The component in the @NgModule.bootstrap list.

Components referenced in router configuration.

You don't have to mention these components explicitly, although doing so is harmless.

<https://medium.com/ngconf/bye-bye-entrycomponents-a4cd933e8eaf>

what is angular ivy

Ivy is **a complete rewrite of Angular's rendering engine**. In fact, it is the fourth rewrite of the engine and the third since Angular 2. But unlike rewrites two and three, which you might not have even noticed, Ivy promises huge improvements to your application.

Before Ivy, Angular would create Ngfactories for all the components declared in the template and as per the NgModule configuration. During the runtime it would enable tree shaking for the components not used. This is why the dynamic components with no Ngfactories could not be rendered and would throw an error like:

Adding the component to the entryComponents array would then make the factories for these dynamic components available at runtime.

**Bootstrapping the component – app component – entry using index.html**

Another way of bootstrapping your component is using the ngDoBootstrap() method wherein we can imperatively define which component to be bootstrapped on the launch of the app in the browser. This is more imperative way to write since you create an element for the component selector and check for change detection.

Specifying components in route definitionsThis is the other way Angular specifies a component as an entry component. If we look at the routing definitions, we always specify the routable component class in the definitions and this is when the CLI registers all the routable components as entryComponents.

As we discussed above, entryComponents are mostly specified in two ways: bootstrapping them or defining them in a router definition. But since these happen under the hood, we hardly notice it. However, when working with dynamic components, or web components in Angular, we explicity define the components as entry Components inside the entryComponents array.

**Role of entryComponents in smaller bundle sizes?**

Alright, think for a minute. When we declare multiple components inside the declarations array of our modules, does that mean all these components will be included into the final bundle?

This is where **entryComponents** have a role to play. So first of all, the answer to the above question is NO. All declared components aren’t necessarily present in the final produced bundle. They’d be present in the produced bundle only if they are specified as entryComponents

This basically means that all the routable components will be present in the bundle for sure and also the bootstrap component obviously. This would also include the bundles that are declared inside the templates of other components. However, the tree shaking process will get rid of all the unused components with no reference without having to include them inside the package.

EntryComponents are mostly explicitly defined when dealing with dynamic components, as I said before. This is because there needs to be a reference for the compiler to understand that, THOUGH there is no reference for a particular component in the template or router for now, there is a possibility for it to be rendered dynamically when required. The ComponentFactoryResolver takes care of creating this dynamic component for us but we specify this inside the entryComponents array inside NgModule.

# **Coming to the point why entryComponents are no longer required.**

Now having an idea about why we need entryComponents, let’s discuss a scenario wherein we have created a dynamic component and have added it to the entryComponents array.  
This basically means that since we explicitly declared it as an entryComponent, the tree shaker would not prune this component thinking that it doesn’t have a reference in the template. Also, specifying it as an entryComponent would create a component factory for this dynamic component.

First, the entryComponent for a particular dynamic component could be added automatically whenever a dynamic component was created to be used. So this would save the developer from specifying it every time to make sure the compiler knows the component. One more issue with using entryComponent was about referencing the entryComponents declared inside a lazily loaded module. So if a lazy loaded module contains a modal component as an entry component, you’d face an error like No component factory found for this component. This was because the root injector couldn’t be referenced to create a component factory for the entryComponent. One solution, though not very promising, was creating a component resolver factory yourself for a particular entry component inside the lazy loaded module to be able to execute it.

# **However,**

With Angular 9 coming in and Ivy as the new rendering engine, all the components would be considered as entering components and do not necessarily need to be specified inside the entryComponents array.

# **Why?**

With Ivy’s principle of locality, importing dynamic components will always work regardless of presence of entryComponents or ANALYSE\_FOR\_ENTRY\_COMPONENTS.

A demo here shows how entryComponents are no longer required with Angular 9

import { NgModule, DoBootstrap, ApplicationRef } from '@angular/core';

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

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

@NgModule({

  imports: [

    BrowserModule

  ],

  declarations: [

    AppComponent

  ],

})

export class AppModule implements DoBootstrap {

  ngDoBootstrap(appRef: ApplicationRef) {

    appRef.bootstrap(AppComponent); // Or some other component

  }

}

# **All you need to know about Ivy, The new Angular engine!**

## Smaller bundles, faster compilations, Better debugging, dynamic loading of module and components and advanced concepts like Higher order components. Angular Ivy — The complete guide for the 3rd generation of the Angular renderer.

<https://medium.com/angular-in-depth/all-you-need-to-know-about-ivy-the-new-angular-engine-9cde471f42cf>

Now let’s opt-in Ivy by editing the tsconfig.app.json and add a section of angularComplierOptionand set the enableIvyto true. for new Angular CLI projects, you can just use the --enableIvy flag when running your ng newscript.

# Difference Among Angular 8, 7, 6, 5, 4, 3, 2 — Breakdown, New Features, and Changes

<https://medium.com/@lifenshades/difference-among-angular-8-7-6-5-4-3-2-breakdown-new-features-and-changes-811fb5f8e6f0>

Ang 1 – 2010 – Ang js

Ang 2- 2016 – ts , ecma , complete rewrite

2017 – Ang4 , no ang 3 as @angular/core , router and compiler , router was already in version 3

Ngif , else , ts improved , Aot compiler , performance ,

– 2017 Ang 5 ,

Build optimizer : It helps to removed unnecessary code from your application

Angular Universal State Transfer API and DOM Support — By using this feature, we can now share the state of the application between the server side and client side very easily.

Compiler Improvements: This is one of the very nice features of Angular 5, which improved the support of incremental compilation of an application

Preserve White space: To remove unnecessary new lines, tabs and white spaces we can add below code(decrease bundle size)

// in component decorator you can now add:  
“preserveWhitespaces: false”  
// or in tsconfig.json:  
“angularCompilerOptions”: { “preserveWhitespaces”: false}`

Increased the standardization across all browsers: For internationalization we were depending on `i18n` , but in ng 5 provides a new date, number, and currency pipes which increases the internationalization across all the browsers and eliminates the need of i18n polyfills.

* exportAs: In Angular 5, multiple names support for both directives and components
* HttpClient: until Angualar 4.3 [@angular/HTTP](http://twitter.com/angular/HTTP) was been used which is now depreciated and in Angular 5 a new module called HttpClientModule is introduced which comes under [@angular/common](http://twitter.com/angular/common)/http package.
* Improved in faster Compiler support:  
  A huge improvement made in an Angular compiler to make the development build faster. We can now take advantage of by running the below command in our development terminal window to make the build faster.  
  ng serve/s — aot
* 2018 – Ang6 No major breaking changes
* Dependency on RxJS 6
* Synchronizes major version number of the:  
  — Angular framework  
  — Angular CLI  
  — Angular Material + CDK
* Remove support for *<template>* tag and “*<ng-template>*” should be used.
* Registering provider: To register new service/provider, we import Service into module and then inject in provider array. e.g:

// app.module.ts **import** {MyService} **from** './my-service';  
...  
providers: [...MyService]  
...

* But after this upgrade you will be able to add providedIn property in injectable decorator. e.g:
* // MyService.ts@Injectable({ providedIn: 'root'})  
  export class MyService{}
* CLI Changes: Two new commands have been introduced  
  — ng update <package>  
  \* Analyse package.json and recommend updates to your application  
  \* 3rd parties can provide update scripts using schematics  
  \* automatically update code for breaking changes  
  \* staying update and low maintenance  
  — ng add  
  \* add new capablities to your applicaiton  
  \* e.g ng add [@angular/material](http://twitter.com/angular/material) : behind the scene it add bit of necessary code and changes project where needed to add it the thing we just told it to add.

2018 – Ang7

* Released on October 2018
* This is a major release and expanding to the entire platform including-  
  — Core framework,  
  — Angular Material,  
  — CLI
* CLI Prompts: Added a new interface — DoBootstrap interface
* Angular 7 added a new compiler — Compatibility Compiler (ngcc)
* Introduce a new Pipe called — KeyValuePipe
* Added a new router features — warn if navigation triggered outside Angular zone

2019 – Ang 8

* Allow passing state to routerLink Directives in the Router
* Allow passing state to NavigationExtras in the Router
* Added support for SASS
* Resolve generated Sass/Less files to .css inputs

# **Angular and SOLID principles**

**Testable , maintainable , understandable , flexible**

American software engineer and instructor Robert C. Martin (Uncle Bob)

If you don’t architect your codebase in a way that can allow changes, later on, you will pay an expensive price!

[**https://indepth.dev/posts/1414/angular-and-solid-principles**](https://indepth.dev/posts/1414/angular-and-solid-principles)

The primary benefits of a SOLID architecture are that you will write code:

* that is testable
* that is easy to understand
* that can be adjusted and extended quickly without producing bugs
* that separates the policy (rules) from the details (implementation)
* that allows for implementations to be swapped out
* where business logic is where it is expected to be
* where classes do what they are intended to do

S: Single Responsibility Principle - A*class or function should only have one reason to change*

O: Open-Closed Principle - *A software artifact should be open for extension but closed for modification*

L: Liskov-Substitution Principle - Introduced by Barbara Liskov in the 1980s. This principle defines that objects of a superclass shall be replaceable with objects of its subclasses without breaking the application. That requires the objects of the subclasses to behave in the same way as the objects of the superclass.

I: Interface Segregation Principle - Prevent classes from relying on things that they don’t need

D: Dependency Inversion Principle - Abstractions should not depend on details. Details should depend on abstractions

**HW – check for design pattern where we created helper service , where we will do all service method call and subscription and later in component , just call helper service methods. So idea is component is very clean and loose file.**

**Naïve Approach**

export class ListComponent implements OnInit {

public showNbOfArticlesDropdown$: Observable<boolean>;

public mainList$: Observable<Authors[] | Articles[]>;

public listLevel$: Observable<ListLevel>;

public pagination$: Observable<Pagination>;

public topic$: Observable<string>;

public nbArticles$: Observable<number>;

constructor(

private apiService: ApiService,

private storeService: StoreService

) {

this.listLevel$ = this.storeService.select('listLevel');

this.pagination$ = this.storeService.select('pagination');

this.topic$ = this.storeService.select('topic');

this.nbArticles$ = this.storeService.select('nbOfArticles');

}

ngOnInit(): void {

this.mainList$ = this.getMainList();

this.showNbOfArticlesDropdown$ = this.showNbOfArticles();

}

private getMainList(): Observable<Authors[] | Articles[]> {

return combineLatest(

this.listLevel$,

this.pagination$,

this.topic$,

).pipe(

switchMap(([listLevel, pagination, topic]) => {

const { first, last } = pagination;

switch (listLevel) {

case ListLevel.AUTHORS:

return this.nbArticles$.pipe(

switchMap(nbArticles => {

return this.apiService.fecthAuthors({first, last, topic, nbArticles})

})

);

case ListLevel.ARTICLES:

return this.apiService.fecthArticles({ first, last, topic });

default:

break;

}

})

)

}

private getArticlesByAuthor(authorId: number): Observable<Articles[]> {

return this.apiService.fetchArticlesByAuthor(authorId);

}

private showNbOfArticles(): Observable<boolean> {

return this.listLevel$.pipe(

map(listLevel => {

switch (listLevel) {

case ListLevel.AUTHORS:

return true;

case ListLevel.ARTICLES:

return false;

default:

break;

}

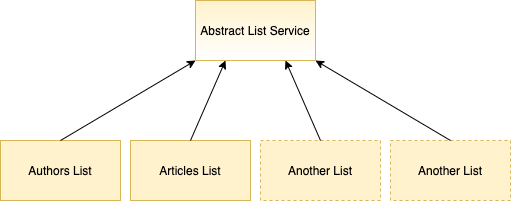
})

)

}

}

Using solid principle



**Conclusion**

Every software engineer should know and understand SOLID principles. It is a concept that helps to get a strong architecture foundation for any application.

Writing clean and SOLID code does increase the number of files introduced in your project because of abstractions. But, it enables any large team of developers to scale quickly, without producing bugs. Particularly in large enterprise applications where product requirements evolve every time.

# **How OOP is mistreated in Angular**

**Oop is a** programming  **paradigm.**

If we want to share functionality of class A in class B without inheritance, Angular has a remedy for that: dependency injection. We should inject class B in class A and use it through a provided reference.

[**https://indepth.dev/posts/1473/how-oop-is-mistreated-in-angular**](https://indepth.dev/posts/1473/how-oop-is-mistreated-in-angular)

**Saw api call also here**

### **In Conclusion**

In this article, we have explored some common malpractices with OOP in Angular codebases. While there are lots of other programming paradigms (some also used by Angular projects like function programming, reactive programming and so on), OOP has a central place in the Angular ecosystem, so using it correctly is paramount if we want to achieve flexibility and maintainability.

# **How to refactor an Angular codebase**

[**https://indepth.dev/posts/1425/how-to-refactor-an-angular-codebase**](https://indepth.dev/posts/1425/how-to-refactor-an-angular-codebase)

Important things to consider:

* I advise you upgrade version one by one, say, if you want to go from 7 to 10, go from 7 to 8, then from 8 to 9, then from 9 to ten. This way you might dodge some problems

ng lint , ng lint --fix

# **Angular 9: What’s Angular and Angular 9 New Features**

In nutshell these are the new features of Angular 9:

* Smaller bundle sizes and augmented performance
* Faster testing
* Better debugging
* Improved CSS class and style binding
* Improved type checking
* Improved build errors
* Improved build times, enabling AOT on by default
* Improved Internationalization
* The Ivy compiler: The default use of the Ivy compiler is the most important feature of Angular 9, Ivy is what actually designed to solve the major problems of Angular i.e the performance and large file size
* Selector-less bindings support for Angular Ivy
* Support for TypeScript Diagnostics Format
* Support for more scopes in providedIn
* A New Type-Safe TestBed.inject() Method Instead of TestBed.get()
* Improvements to differential loading
* AOT compilation everywhere – rendering fast , already downloaded code not required to read , new changes only
* Bundle sizes
* Globalisation
* Additional provider scopes
* Improved developer experience
* New debugging API in development mode
* Strict mode
* Improved component and directive class inheritance
* Latest TypeScript versions
* Improved server-side rendering with Angular Universal
* Improved styling experience
* Stabel Bazel release as opt-in option
* Angular Components
* Testing

# **5 TypeScript Design Patterns You Should Know**

Routing

Budget increased – install everything

Circular dependency

Budget increase

Lazy loading and module

Tsconfig.json

Angular.json

Also it only checks "application" type projects in the angular.json as budgets doesn't do anything for libraries

Class A, B C

You are using selector , that module should be imported and exported

Don’t include one feature module to another with route

Deprecated

Rxjs

Ngrx

Unit testing

LLD / Design patterns

Node

Office ->

@injectable

The @[Injectable](https://angular.io/api/core/Injectable)() decorator specifies that Angular can use this class in the DI system. The metadata, providedIn: 'root', means that the HeroService is visible throughout the application.

For clarity and maintainability, it is recommended that you define components and services in separate files.

If you do combine a component and service in the same file, it is important to define the service first, and then the component. If you define the component before the service, Angular returns a run-time null reference error.

If any life cycle hook even constructor required then use it else not required

.pipe(catchError(this.errorCallback(this.errorCallbackKey)));

 fetchMyVendorDetails(vendorId: string): Observable<IInventoryResponse<IInventoryVendorNames>> {

    return this.httpClient

      .get<IInventoryResponse<IInventoryVendorNames>>(

        `/api/inventory/v2/inventories/vendors/${vendorId}`

      )

      .pipe(catchError(this.errorCallback(this.errorCallbackKey)));

  }

fetchMyVendorAssessmentDetails(

    paginationParams: IPaginationGridParams,

    vendorId: string,

    templateTypes?: string

  ): Observable<IPageableOf<IInventoryVendorAssessment>> {

    let params = new HttpParams()

      .set('page', paginationParams.page.toString())

      .set('size', paginationParams.size.toString())

      .set('sort', paginationParams.sort ? `${paginationParams.sort}` : '');

    if (templateTypes) {

      params = params.append('types', templateTypes);

    }

    return this.httpClient

      .post<IPageableOf<IInventoryVendorAssessment>>(

        `/api/assessment-v2/v2/assessments/inventory/${vendorId}/assessments/lists`,

        [],

        {

          params,

        }

      )

      .pipe(catchError(this.errorCallback(this.errorCallbackKey)));

  }

A pure pipe is only called when **Angular detects a change** in the value or the parameters passed to a pipe.An impure pipe is called for every change detection cycle no matter whether the value or parameter(s) changes

Async is an example of **an impure pipe**. It is always checking for new input data. Pure will be true if not specified. The pure property tells Angular whether or not the value should be recomputed when its input changes.

Pure and impure function

@HostBinding('class') readonly hostClass =

    'min-height-1 slds-col height-100-percent slds-grid slds-grid--vertical';

Directives are **classes that add additional behavior to elements in your Angular applications**

@Output() selectedRId = new EventEmitter<string>();

selectedRowId(rowId: string) {

    this.selectedRId.emit(rowId);

  }

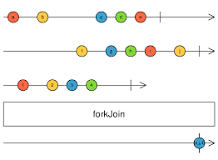
(selectedRId)="selectedRId($event)"

 selectedRId(rowId: string) {

    this.rowId.emit(rowId);

  }

What is forkJoin RxJS?

[[](https://www.google.com/search?q=What+is+forkJoin+RxJS?&sxsrf=AOaemvJ48VC9CfhZ-UeD5BbQjsB034oOcw:1638939864520&tbm=isch&source=iu&ictx=1&fir=BeiikYYn7AbMWM%252CJEE0ldzuRx2KJM%252C_&vet=1&usg=AI4_-kS5JH4vOEuryGGdG1JE2SYabWJeQg&sa=X&ved=2ahUKEwih7LrjttP0AhU1S2wGHanRBuYQ9QF6BAgGEAE#imgrc=BeiikYYn7AbMWM)](https://www.google.com/search?q=What+is+forkJoin+RxJS?&sxsrf=AOaemvJ48VC9CfhZ-UeD5BbQjsB034oOcw:1638939864520&tbm=isch&source=iu&ictx=1&fir=BeiikYYn7AbMWM%252CJEE0ldzuRx2KJM%252C_&vet=1&usg=AI4_-kS5JH4vOEuryGGdG1JE2SYabWJeQg&sa=X&ved=2ahUKEwih7LrjttP0AhU1S2wGHanRBuYQ9QF6BAgGEAE" \l "imgrc=BeiikYYn7AbMWM)

forkJoin is **an operator that takes any number of input observables which can be passed either as an array or a dictionary of input observables**. If no input observables are provided (e.g. an empty array is passed), then the resulting stream will complete immediately.

What is SwitchMap RxJS?

The switchMap operator maps **each value to an observable**, then it flattens all of the inner observables. It basically projects each source value to an observable which is then merged in the output observable, emitting values only from the most recently projected observable.

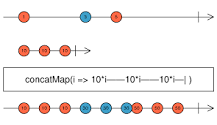
What is mergeMap in RxJS?

RxJS mergeMap() operator is **a transformation operator that applies a project function on each source value of an Observable**, which is later merged in the output Observable. ... This operator is best to use when you want to flatten an inner observable and manually control the number of inner subscriptions.

What is difference between mergeMap and switchMap?

So here's the simple difference — **switchMap cancels previous HTTP requests that are still in progress**, while mergeMap lets all of them finish. In my case, I needed all requests to go through, as this is a metrics service that's supposed to log all actions that the user performs on the web page, so I used mergeMap .

What is concatMap in RxJS?

[[](https://www.google.com/search?q=What+is+concatMap+in+RxJS?&sxsrf=AOaemvIagWs9ozzoDVe5tmwM5o72cB8VMA:1638940332605&tbm=isch&source=iu&ictx=1&fir=ipd4_opNObZAkM%252C9u74yuI1ftKTwM%252C_&vet=1&usg=AI4_-kS8OyyO5zr-dz_vlw57kpv6r4sfUw&sa=X&ved=2ahUKEwiIr9TCuNP0AhUdTmwGHSo1CrwQ9QF6BAg1EAE#imgrc=ipd4_opNObZAkM)](https://www.google.com/search?q=What+is+concatMap+in+RxJS?&sxsrf=AOaemvIagWs9ozzoDVe5tmwM5o72cB8VMA:1638940332605&tbm=isch&source=iu&ictx=1&fir=ipd4_opNObZAkM%252C9u74yuI1ftKTwM%252C_&vet=1&usg=AI4_-kS8OyyO5zr-dz_vlw57kpv6r4sfUw&sa=X&ved=2ahUKEwiIr9TCuNP0AhUdTmwGHSo1CrwQ9QF6BAg1EAE" \l "imgrc=ipd4_opNObZAkM)

function **stable**. **Projects each source value to an Observable which is merged in the output Observable**, in a serialized fashion waiting for each one to complete before merging the next.

What is the difference between switchMap concatMap and mergeMap in RxJS?

Use **mergeMap** if you simply want to flatten the data into one Observable, use switchMap if you need to flatten the data into one Observable but only need the latest value and use concatMap if you need to flatten the data into one Observable and the order is important to you.

getBusinessOwnerVendorsAndSchemaDetails() {

    this.isLoading = true;

    const filterCriteria = {

      AND: [

        {

          columnId: 'owner',

          attributeKey: 'owner',

          dataType: 50,

          operator: 'EQ',

          values: [this.currentUser.Id],

        },

      ],

    };

    forkJoin([

      this.myVendorsSspApiService.fetchVendorSchemaDetails(),

      this.myVendorsSspApiService.fetchBusinessOwnersVendorNames(

        this.paginationParam,

        filterCriteria

      ),

    ])

      .pipe(

        filter(res => Boolean(res)),

        map((response: [IVendorSchemaDetails[], IPageableData<IInventoryVendorNames>]) => {

          this.vendorsList = response[1];

          this.schemaDetails = response[0];

          this.inventoryVendorList = response[1].data;

          return this.inventoryVendorList

            .filter(data => data.primaryContact !== undefined)

            .map(data => data.primaryContact[0].id);

        }),

        switchMap((unknownUsers: string[]) => {

          return this.userStore.fetchUsersById(unknownUsers).pipe(

            catchError(() => {

              return of(EMPTY);

            })

          );

        }),

        takeUntil(this.destroy$)

      )

      .subscribe({

        next: (response: IBasicUserIdMap) => {

          this.isLoading = false;

          this.schemaDetails.forEach(item => {

            if (Object.values(AttributeKey).includes(item.fieldName as AttributeKey)) {

              this.columnSelect[item.fieldName] = item;

            }

          });

          this.columns = ResultantTableColumns.filter(data =>

            Object.keys(this.columnSelect).includes(data.cellValueKey)

          );

        },

        error: () => {

          this.isLoading = false;

        },

      });

  }

 \*ngIf="!isLoadingAssessment; else contentLoadings"

 <ng-template #contentLoadings>

            <ot-loading otPageLoading></ot-loading>

          </ng-template>

        </section>

Ng-container , ng-template

this.router.navigate([`${route.path}`]);

this.router.navigate([`/ssp/vendors/details/vendor/${rowId}`], {

      queryParams: { tabId: 'details' },

    });

  }

this.route.queryParams.subscribe(params => {

      this.currentTab = params['tabId'];

    });

fetchWorkFlowBuilderStage(workflowId: string) {

    this.myVendorsSspApiService

      .getWorkflowVendorStage(workflowId)

      .pipe(takeUntil(this.destroy$))

      .subscribe({

        next: (res: IWorkflowBuilderStage[]) => {

          this.isLoadingDetails = false;

          this.workflowBuilderStage = res;

          this.workflowBuilderStage = this.workflowBuilderStage.sort((a, b) =>

            a.sequence > b.sequence ? 1 : -1

          );

          this.workflowBuilderStage.forEach(data => {

            const workflowStage: string = (this.selectedRow.workflowStage as IWorkflowStage).stage

              .value;

            if (workflowStage === data.name) {

              this.pathItems.push({ label: data.name, isActive: true });

            } else {

              this.pathItems.push({ label: data.name, isActive: false });

            }

          });

          const activeIndex: number = this.pathItems.findIndex(

            activeIdx => activeIdx.isActive === true

          );

          for (let i = 0; i < activeIndex; i++) {

            this.pathItems[i].isActive = true;

          }

        },

        error: () => {

          this.isLoadingDetails = false;

        },

      });

  }

this.vendorId = this.route.snapshot.paramMap.get('id');

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

const routes: Routes = [

  {

    path: 'vendor',

    canActivate: [MasterGuard],

    data: {

      guards: [environment.AuthGuardServices, OmniGuard, NgxPermissionsGuard],

      guardsRelation: 'AND',

      permissions: {

        only: 'VendorRiskManagement',

      },

    },

    component: WrapperComponent,

    children: [

{

        path: 'attribute-manager',

        children: [

          {

            path: 'home',

            canActivate: [NgxPermissionsGuard],

            data: {

              permissions: {

                only: ['VRMAttributeManager', 'VendorAttributes'],

                redirectTo: '404',

              },

            },

            loadChildren: () =>

              import(

                './attribute-manager/vendor-attribute-manager-home/vendor-attribute-manager-home.module'

              ).then(m => m.VendorAttributeManagerHomeModule),

          },

          {

            path: 'list',

            loadChildren: () =>

              import(

                './attribute-manager/engagement-attribute-manager-wrapper/engagement-attribute-manager-wrapper.module'

              ).then(m => m.EngagementAttributeManagerWrapperModule),

          },

          {

            path: 'detail',

            loadChildren: () =>

              import(

                './attribute-manager/contract-attribute-manager-wrapper/contract-attribute-manager-wrapper.module'

              ).then(m => m.ContractAttributeManagerWrapperModule),

          },

          {

            path: '',

            redirectTo: '/vendor/attribute-manager/home',

            pathMatch: 'full',

          },

        ],

      },

Feature module route

const routes: Routes = [

  {

    path: '',

    pathMatch: 'full',

    component: ContractAttributeManagerWrapperComponent,

  },

];

 RouterModule.forChild(routes)

Unit testing

// Angular

import { ComponentFixture, TestBed } from '@angular/core/testing';

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

// OT Library

import { CoreTestingModule } from '../core/core-testing.module';

import { MyVendorsSspListsComponent } from '@onetrust/vendor';

// component

import { MyVendorsListWrapperComponent } from './my-vendors-list-wrapper.component';

import { MockComponents } from 'ng-mocks';

describe('MyVendorsListWrappereComponent', () => {

  let component: MyVendorsListWrapperComponent;

  let fixture: ComponentFixture<MyVendorsListWrapperComponent>;

  let routerSpy = { navigate: jasmine.createSpy('navigate') };

  beforeEach(() => {

    TestBed.configureTestingModule({

      imports: [CoreTestingModule],

      declarations: [MyVendorsListWrapperComponent, MockComponents(MyVendorsSspListsComponent)],

      providers: [{ provide: Router, useValue: routerSpy }],

    }).compileComponents();

  });

  beforeEach(() => {

    fixture = TestBed.createComponent(MyVendorsListWrapperComponent);

    component = fixture.componentInstance;

    fixture.detectChanges();

  });

  it('should create', () => {

    expect(component).toBeTruthy();

  });

  it(`should navigate to vendor detail page`, () => {

    const vendorId = 'fa147582-7274-445a-a6bd-6f934c513feb';

    component.selectedRowId(vendorId);

    expect(routerSpy.navigate).toHaveBeenCalledWith([`/ssp/vendors/details/vendor/${vendorId}`], {

      queryParams: { tabId: 'details' },

    });

  });

});

Shallow and deep copy

**var** employee = {

    eid: "E102",

    ename: "Jack",

    eaddress: "New York",

    salary: 50000

}

console.log("Employee=> ", employee);

**var** newEmployee = employee;    // Shallow copy

console.log("New Employee=> ", newEmployee);

console.log("---------After modification----------");

newEmployee.ename = "Beck";

console.log("Employee=> ", employee);

console.log("New Employee=> ", newEmployee);

// Name of the employee as well as

// newEmployee is changed.

Deep copy

**var** employee = {

    eid: "E102",

    ename: "Jack",

    eaddress: "New York",

    salary: 50000

}

console.log("=========Deep Copy========");

**var** newEmployee = JSON.parse(JSON.stringify(employee));

console.log("Employee=> ", employee);

console.log("New Employee=> ", newEmployee);

console.log("---------After modification---------");

newEmployee.ename = "Beck";

newEmployee.salary = 70000;

console.log("Employee=> ", employee);

console.log("New Employee=> ", newEmployee);

Graphical user interface, text, application

Description automatically generated

# **Designing Angular architecture - Container-Presentation pattern**

<https://indepth.dev/posts/1478/designing-angular-architecture-container-presentation-pattern>

### **Crucial Design Points**

I want to highlight a few crucial points from my point of view. When planning architecture for a new feature, I'm always trying to make decisions on those points, preferably before starting to code. These key points are:

* What components, services, and other classes will be used (do I need to create new ones? are there any classes that I can reuse?)
* What are the responsibilities of those classes? Are they limited enough?
* What is the data flow between those classes, especially the components?
* What are the dependencies in the components?
* Is my plan open for extension in the future? (this is very similar to "O" in the SOLID principle)

### **Why Container-Presentation?**

The pattern that I want to introduce to you is called the Container-Presentation, and it clearly answers all those points above. Why have I chosen that particular pattern? I think this is an instrumental design that can be applied on an almost everyday basis - I believe many of you already intuitively use it in your feature design which is excellent. I want to describe it clearly and show step by step how it can be applied and extended. Although the use case for that pattern is very common, developers tend to omit important parts of that pattern which I will highlight and try to convince you to include in your implementation.

Before digging into the pattern architecture, I would like to highlight the key reasons why this pattern could be useful for you:

* it provides flexible and open for extension split of the components
* separation of the components makes it easy to apply efficient change strategy to each type of the component
* component’s dependencies are limited and well organized, which makes them easy to manage, preferably some of the components don’t have any dependencies at all!
* most of the components are highly generic, and the pattern itself compels you to create them so that they can be easily reused in future work
* logic and UI are separated, which means it’s pretty safe to change the UI without causing any issues with the implemented logic for the feature

To sum up that very shortly:

* Container: takes care of the data, distributes that, handles service calls and most of the logic
* Presentation: present data, sometimes receives events from the users and passes them to Container component

## Summary

I hope you enjoyed that little story about the Container-Presentation pattern and learned new stuff. I’m sure you were familiar with most of those techniques, but bringing them together to make a solid architecture is the key.

What were the main ingredients in this guide:

* Container Component- responsible for retrieving data and distributing them to Presentation Components
* Presentation Component - responsible for rendering data and optionally informing Container about events (for instance, user actions)

What should be used for communication between these two types of components?

* Input / Output for most cases
* Service for rather complex scenarios

The other concepts I used:

* vm - View Model convention for wrapping Input properties into a single object. It should be typed to increase safety.
* Change Detection OnPush - the most efficient strategy for most cases, usually can be used without issues in Presentation Components
* vm Class - if the model grows, or it needs to keep some logic to either remove code duplication or extract the logic from other places, it is recommended to create a Class with appropriate properties, getters, functions, etc.
* cloning techniques - it's usually important to clone the objects that we work with, not work on the same data instances. You need to think if your solution requires shallow or deep cloning and use the appropriate algorithm. Spread operator can be useful for shallow cloning.

Lastly, I want to remind you of the key idea that goes through this article and is essential. Introducing such patterns in your architecture will make the solution more solid, concise which will help you with:

* creating a code without duplication
* providing clear responsibility for each class
* being ready for future requirement changes - which means it will be easy to extend your architecture without the need to modify existing code

# **How pure and impure pipes work in Angular Ivy**

When pipe is pure, transform() method is invoked only when its input arguments change. Pipes are pure by default.  
If the pipe has internal state (that is, the result depends on the state other than its arguments), set pure to false. In this case, the pipe is invoked on each change detection cycle, even if the arguments have not changed.

Another interesting feature of the pure pipes is that Angular creates only one instance of a pure pipe regardless of how many times a pipe is used in a template:

# **RxJS in Angular: Part I**

### **Using RxJS to reduce the state of our components**

Here is how we do some stuff *wrong:*

1. When we have a new task at hand, we think about how the state will change
2. We store *new*pieces of state in our component (new properties, nested objects and etc)
3. We devise new methods that encapsulate ways how our new state changes
4. Write convoluted logic inside our template.

### **More reactive thinking**

Now we are going to devise a simple three-step plan of thinking. Trying to solve the same problem, but now using Reactive Forms and RxJS, we will do the following:

1. Understand what part of the state affects the UI and make it an Observable stream;
2. Use RxJS operators to perform calculations and derive the final state to be used in UI
3. Use the async pipe to put the result of our computations in the template

### **In conclusion**

RxJS is a powerful tool — no wonder such a huge enterprise framework like Angular is built around it; It has lots of concepts and tricks which may be used to make our code in Angular better, more readable, easier to reason about, more understandable

# **Learn Angular Component Design Patterns — Creating a Drawer Component**

Telling about reusable component

## Lazy load for performance — v6

Every time you create a component that is intended for reuse (every component must be intended for reuse, by the way), you should ask these questions to your application

**“Is it imperative to render this component at the initial load of the page? Is the information in it vital to be rendered in the initial load?”**

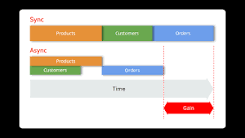
If your answer is "no" or "maybe", you should consider lazy loading the component

## Wrapping it up

Let’s wrap our marathon refactoring here for now. Throughout this article we have learned how we could use efficient tools Angular provides to create high quality components. We learned that techniques such as multiple content projection, lazy loading, exportAs etc can be really handy when creating highly efficient, reusable components. Mastering these concepts/techniques can really make your development more fun and productive.

Synchronous JavaScript: As the name suggests synchronous **means to be in a sequence**, i.e. every statement of the code gets executed one by one. So, basically a statement has to wait for the earlier statement to get executed

What is asynchronous and synchronous in JavaScript?

[[](https://www.google.com/search?q=What+is+asynchronous+and+synchronous+in+JavaScript?&sxsrf=AOaemvL6sRvG-ZOLj-_w_0FGYtrl0UYypQ:1639149276639&tbm=isch&source=iu&ictx=1&fir=15DYPhOAI3M-XM%252CS5f29DyJ089TJM%252C_&vet=1&usg=AI4_-kQZ8zOUVM1VoBxLYg400Jb7lS-tMw&sa=X&ved=2ahUKEwj6zPfywtn0AhWqwYsBHf22DooQ9QF6BAgEEAE#imgrc=15DYPhOAI3M-XM)](https://www.google.com/search?q=What+is+asynchronous+and+synchronous+in+JavaScript?&sxsrf=AOaemvL6sRvG-ZOLj-_w_0FGYtrl0UYypQ:1639149276639&tbm=isch&source=iu&ictx=1&fir=15DYPhOAI3M-XM%252CS5f29DyJ089TJM%252C_&vet=1&usg=AI4_-kQZ8zOUVM1VoBxLYg400Jb7lS-tMw&sa=X&ved=2ahUKEwj6zPfywtn0AhWqwYsBHf22DooQ9QF6BAgEEAE" \l "imgrc=15DYPhOAI3M-XM)

In synchronous operations tasks are performed one at a time and only when one is completed, the following is unblocked. In other words, you need to wait for a task to finish to move to the next one. In asynchronous operations, on the other hand, **you can move to another task before the previous one finishes**

**View Encapsulation**

View encapsulation defines whether the template and styles defined within the component can affect the whole application or vice versa. Angular provides three encapsulation strategies:

* Emulated (default) - styles from main HTML propagate to the component. Styles defined in this component's @Component decorator are scoped to this component only.
* ShadowDom - styles from main HTML do not propagate to the component. Styles defined in this

component's @Component decorator are scoped to this component only.

* None - styles from the component propagate back to the main HTML and therefore are visible to all components on the page. Be careful with apps that have None and Native components in the application. All components with None encapsulation will have their styles duplicated in all components with Native encapsulation.

change detection in angular

In the Default strategy, whenever any data to @Input() decorated properties are changed, Angular runs the change detector to update the view. In the onPush strategy, Angular runs change detector only when a new reference is **passed** to the @Input() decorated properties

**The part of the Angular framework that does this is called the “change detector.”** Every component has a change detector that reads the binding on the template and makes sure that the data model and view are in sync with each other. Whenever, for any reason (actually there are three reasons which we cover later in the article), data model changes, it is the change detector that projects the updated data to the view, so that the view and the data model are in sync with each other.

To optimize the number of checks, Angular provides two change detection strategies:

1. Default strategy
2. onPush strategy

In the **Default strategy**, whenever any data to @Input() decorated properties are changed, Angular runs the change detector to update the view. In the **onPush** strategy, Angular runs change detector only when a **new reference** is passed to the @Input() decorated properties.

The onPush change detection strategy instructs Angular to run change detector on the component and its subtree only when a new reference is passed to the @Input decorated properties.

What's the difference between async and defer Javascript?

Async vs Defer  
  
With async, **the file gets downloaded asynchronously and then executed as soon as it's downloaded**. With defer, the file gets downloaded asynchronously, but executed only when the document parsing is completed. With defer, scripts will execute in the same order as they are called

## The localStorage Object

The localStorage object stores the data with no expiration date. The data will not be deleted when the browser is closed, and will be available the next day, week, or year.

## The sessionStorage Object

The sessionStorage object is equal to the localStorage object, **except** that it stores the data for only one session. The data is deleted when the user closes the specific browser tab.

async pipe in angular

The async pipe **subscribes to an Observable or Promise and returns the latest value it has emitted**. When a new value is emitted, the async pipe marks the component to be checked for changes. When the component gets destroyed, the async pipe unsubscribes automatically to avoid potential memory leaks.

# **Singleton services**

A singleton service is a service for which only one instance exists in an application.

## Providing a singleton service

There are two ways to make a service a singleton in Angular:

* Set the providedIn property of the @[Injectable](https://angular.io/api/core/Injectable)() to "root".
* Include the service in the AppModule or in a module that is only imported by the AppModule
* **View Encapsulation**
* View encapsulation defines whether the template and styles defined within the component can affect the whole application or vice versa. Angular provides three encapsulation strategies:
*  Emulated (default) - styles from main HTML propagate to the component. Styles defined in this component's @Component decorator are scoped to this component only.
*  ShadowDom - styles from main HTML do not propagate to the component. Styles defined in this
* component's @Component decorator are scoped to this component only.
*  None - styles from the component propagate back to the main HTML and therefore are visible to all components on the page. Be careful with apps that have None and Native components in the application. All components with None encapsulation will have their styles duplicated in all components with Native encapsulation.

A BehaviorSubject holds one value. When it is subscribed it emits the value immediately. A Subject doesn't hold a value.

# Introduction to NgRx - Angular Reactive Extensions

### Key concepts

Let's start with introducing a few key concepts and building blocks of [NgRx](https://ngrx.io/). NgRx is a state management tool inspired by Redux, so the building blocks might be familiar to Redux users. To build and operate on our state, we will need the following basic building blocks:

* **Store** is where state of the application will be stored. It's both an Observable of a state and an Observer of action.
* **Actions** describe all the possible and unique events that can occur within the application. They can occur ie. by user interaction, communication with the server or can be a result of other actions.
* **Reducers** are what binds the actions and the state. All the state changes must occur as result of an action. Those changes are handled by pure functions called reducers. They take the current state and the latest action and compute new state value based on that.
* **Selectors** - to retrieve part of the state that we're interested in, we use pure functions which extract the portion of the state that a given component might be interested in.

import {createAction} from '@ngrx/store';

export const likePhoto = createAction('[Photo List] Like Photo');

sh ng add @ngrx/store@latest

import {createReducer, on} from '@ngrx/store';

import {dislikePhoto, likePhoto} from './photo.actions';

export type PhotoState = number;

const initialState: PhotoState = 0;

export const photoReducer = createReducer(

initialState,

on(likePhoto, state => state + 1),

on(dislikePhoto, state => state - 1)

);

imports: [

/\* ... other modules \*/

StoreModule.forRoot({

photo: photoReducer

},

{}

)

],

import {createSelector} from '@ngrx/store';

import {PhotoState} from './photo.reducer';

const selectPhotoFeature = (state: { photo: PhotoState }) => state.photo;

export const selectPhoto = createSelector(selectPhotoFeature, likes => ({

title: 'Introduction to NgRx',

url: 'https://ngrx.io/assets/images/ngrx-badge.png',

likes

})

);

import {Store} from '@ngrx/store';

interface AppState {

photo: PhotoState;

}

@Component({/\* ... \*/})

export class AppComponent {

constructor(private store: Store<AppState>) {

}

}

import {select, Store} from '@ngrx/store';

import {selectPhoto} from './store/photo.selectors';

import {dislikePhoto, likePhoto} from './store/photo.actions';

@Component({/\* ... \*/})

export class AppComponent {

photo$ = this.store.pipe(select(selectPhoto));

constructor(private store: Store<AppState>) {

}

onLike(): void {

this.store.dispatch(likePhoto());

}

onDislike(): void {

this.store.dispatch(dislikePhoto());

}

}

@Component({

selector: 'app-root',

template: `

<div class="photos">

<app-photo class="photo" [photo]="photo$ | async" (like)="onLike()" (dislike)="onDislike()"></app-photo>

</div>

`,

})

Routing

pathMatch: 'full' means, that the whole URL path needs to match and is consumed by the route matching algorithm.

pathMatch: 'prefix' means, the first route where the path matches the start of the URL is chosen, but then the route matching algorithm is continuing searching for matching child routes where the rest of the URL matches.

 [routerLink]="['/products']"

         routerLinkActive="active"

         [routerLinkActiveOptions]="{ exact : true}">Product List</a>

 imports: [

    RouterModule.forRoot([

      { path: 'welcome', component: WelcomeComponent },

      {

        path: 'products',

        loadChildren: () =>

          import('./products/product.module').then(m => m.ProductModule)

      },

      { path: '', redirectTo: 'welcome', pathMatch: 'full' },

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

    ])

  ],

  exports: [RouterModule]

  <option \*ngFor="let category of categories$ | async"

                    [value]="category.id">{{ category.name }}</option>

          </select>

 suppliers$ = this.http.get<Supplier[]>(this.suppliersUrl)

    .pipe(

      tap(data => console.log('suppliers', JSON.stringify(data))),

      shareReplay(1),

      catchError(this.handleError)

    );

Sharereplay , tap , mergeMap , concatmap , switchMap , withLatestFrom , combineLatest , forkJoin , takeUntil , filter , catcheError

suppliersWithConcatMap$ = of(1, 5, 8)

    .pipe(

      tap(id => console.log('concatMap source Observable', id)),

      concatMap(id => this.http.get<Supplier>(`${this.suppliersUrl}/${id}`))

    );

  suppliersWithMergeMap$ = of(1, 5, 8)

    .pipe(

      tap(id => console.log('mergeMap source Observable', id)),

      mergeMap(id => this.http.get<Supplier>(`${this.suppliersUrl}/${id}`))

    );

  suppliersWithSwitchMap$ = of(1, 5, 8)

    .pipe(

      tap(id => console.log('switchMap source Observable', id)),

      switchMap(id => this.http.get<Supplier>(`${this.suppliersUrl}/${id}`))

    );

private handleError(err: any) {

    // in a real world app, we may send the server to some remote logging infrastructure

    // instead of just logging it to the console

    let errorMessage: string;

    if (err.error instanceof ErrorEvent) {

      // A client-side or network error occurred. Handle it accordingly.

      errorMessage = `An error occurred: ${err.error.message}`;

    } else {

      // The backend returned an unsuccessful response code.

      // The response body may contain clues as to what went wrong,

      errorMessage = `Backend returned code ${err.status}: ${err.body.error}`;

    }

    console.error(err);

    return throwError(errorMessage);

  }

private productsUrl = 'api/products';

productsWithCategory$ = combineLatest([

    this.products$,

    this.productCategoryService.productCategories$

  ]).pipe(

    map(([products, categories]) =>

      products.map(product => ({

        ...product,

        price: product.price \* 1.5,

        category: categories.find(c => product.categoryId === c.id).name,

        searchKey: [product.productName]

      }) as Product)

    ),

    shareReplay(1)

  );

 private productSelectedSubject = new BehaviorSubject<number>(0);

  productSelectedAction$ = this.productSelectedSubject.asObservable();

private productInsertedSubject = new Subject<Product>();

  productInsertedAction$ = this.productInsertedSubject.asObservable();

addProduct(newProduct?: Product) {

    newProduct = newProduct || this.fakeProduct();

    this.productInsertedSubject.next(newProduct);

  }

 onSelected(categoryId: string): void {

    this.categorySelectedSubject.next(+categoryId); -> convert to integer

  }

What is PreloadingStrategy?

Angular Preloading Strategy is **yet another way to speed up the load time of the Angular Apps**. We build Modular apps using the Angular Modules. The Angular loads all the modules, when the user requests for the first time. ... The Angular allows us further optimize our app using a technique called PreLoading.

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

import { Route, PreloadingStrategy } from '@angular/router';

import { Observable, of } from 'rxjs';

@Injectable({

  providedIn: 'root'

})

export class SelectiveStrategy implements PreloadingStrategy {

  preload(route: Route, load: Function): Observable<any> {

    if (route.data && route.data['preload']) {

      return load();

    }

    return of(null);

  }

}

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

 <input class="form-control"

                   id="userNameId"

                   type="text"

                   placeholder="User Name (required)"

                   required

                   ngModel

                   name="userName"

                   #userNameVar="ngModel"

                   [ngClass]="{'is-invalid': (userNameVar.touched || userNameVar.dirty) && !userNameVar.valid }" />

            <span class="invalid-feedback">

              <span \*ngIf="userNameVar.errors?.required">

                User name is required.

              </span>

            </span>

# **Angular Route Guards for Authentication**

In your angular web application you might need to restrict access to certain pages for some users, or in other words you might be looking for a method to authenticate in-app navigation from client-side. This is where Angular Route Guards will make your life easy. **Angular Route Guard is an interface which can be implemented to decide if a route can be activated.**

There are 5 types of guards in Angular namely **CanActivate, CanActivateChild, CanDeactivate, Resolve**and**CanLoad.**

<https://levelup.gitconnected.com/angular-route-guards-for-authentication-d77fb01f04ae>

JSON WEB TOKENS

|  |
| --- |
| import { Injectable } from '@angular/core'; |
|  |  |
|  | @Injectable({ |
|  | providedIn: 'root' |
|  | }) |
|  | export class AuthService { |
|  |  |
|  | constructor() { } |
|  |  |
|  | isLoggedIn() { |
|  | const token = localStorage.getItem('token'); // get token from local storage |
|  | const payload = atob(token.split('.')[1]); // decode payload of token |
|  | const parsedPayload = JSON.parse(payload); // convert payload into an Object |
|  |  |
|  | return parsedPayload.exp > Date.now() / 1000; // check if token is expired |
|  |  |
|  | } |
|  |  |
|  | } |

|  |
| --- |
| import { Injectable } from '@angular/core'; |
|  | import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, Router } from '@angular/router'; |
|  | import { Observable } from 'rxjs'; |
|  | import { AuthService } from './auth.service'; |
|  |  |
|  | @Injectable({ |
|  | providedIn: 'root' |
|  | }) |
|  | export class AuthGuard implements CanActivate { |
|  |  |
|  | constructor(private authService: AuthService, private router: Router) {} |
|  |  |
|  | canActivate( |
|  | next: ActivatedRouteSnapshot, |
|  | state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean { |
|  |  |
|  | if (!this.authService.isLoggedIn()) { |
|  | this.router.navigate(['/login']); // go to login if not authenticated |
|  | return false; |
|  | } |
|  | return true; |
|  | } |
|  | } |

|  |
| --- |
| import { NgModule } from '@angular/core'; |
|  | import { Routes, RouterModule } from '@angular/router'; |
|  | import { HomeComponent } from './home/home.component'; |
|  | import { LoginComponent } from './login/login.component'; |
|  | import { AuthGuard } from './auth.guard'; |
|  |  |
|  | const routes: Routes = [ |
|  | { path: '', redirectTo: '/home', pathMatch: 'full' }, |
|  | { path: 'login', component: LoginComponent }, |
|  | { path: 'home', component: HomeComponent, |
|  | canActivate: [AuthGuard], // visit home only if authenticated |
|  | }, |
|  | ]; |
|  |  |
|  | @NgModule({ |
|  | imports: [RouterModule.forRoot(routes)], |
|  | exports: [RouterModule] |
|  | }) |
|  | export class AppRoutingModule { } |

JwtHelperService

npm install --save @auth0/angular-jwt

// src/app/auth/auth.service.tsimport { Injectable } from '@angular/core';  
import { JwtHelperService } from '@auth0/angular-jwt';@Injectable()  
export class AuthService { constructor(public jwtHelper: JwtHelperService) {} // ...  
 public isAuthenticated(): boolean { const token = localStorage.getItem('token'); // Check whether the token is expired and return  
 // true or false  
 return !this.jwtHelper.isTokenExpired(token);  
 }}

Router-outlet in Angular works as **a placeholder which is used to load the different components dynamically based on the activated component or current route state**. Navigation can be done using router-outlet directive and the activated component will take place inside the router-outlet to load its content.

@NgModule({

  imports: [

    RouterModule.forRoot([

      { path: 'welcome', component: WelcomeComponent },

      {

        path: 'products',

        canActivate: [AuthGuard],

        data: { preload: false },

        loadChildren: () =>

          import('./products/product.module').then(m => m.ProductModule)

      },

      { path: '', redirectTo: 'welcome', pathMatch: 'full' },

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

    ], { preloadingStrategy: SelectiveStrategy })   // , { enableTracing: true, preloadingStrategy: SelectiveStrategy }

  ],

  exports: [RouterModule]

})

export class AppRoutingModule { }

The canload guard **determines whether a particular lazy loaded child route can be loaded**.

What is CanLoad guard in Angular?

The CanLoad Guard **prevents the loading of the Lazy Loaded Module**

Resolve guard is used in the scenario when we **want to ensure whether there is data available or not before navigating to any route**. If there is no data then it has no meaning to navigate there. It means we have to resolve data before navigating to that route.

export class AuthGuard implements CanActivate, CanLoad

import { [Directive](https://angular.io/api/core/Directive), [ElementRef](https://angular.io/api/core/ElementRef) } from '@angular/core'; @[Directive](https://angular.io/api/core/Directive)({ selector: '[appHighlight]' }) export class HighlightDirective { constructor(el: [ElementRef](https://angular.io/api/core/ElementRef)) { el.nativeElement.style.backgroundColor = 'yellow'; } }

Debouncing and throttling techniques are used to limit the number of times a function can execute. Generally, how many times or when a function will be executed is decided by the developer. But in some cases, developers give this ability to the users. Now, it is up to the user to decide when and how many times to execute that function.

Graphical user interface, application

Description automatically generated

In JIT compilation, the application compiles inside the browser during runtime.  
Whereas in the AOT compilation, the application compiles during the build time.  
  
The advantages of using AOT compilation are:

 Since the application compiles before running inside the browser, the browser loads the executable code and renders the application immediately, which leads to **faster rendering**.

 In AOT compilation, the compiler sends the external HTML and CSS files along with the application, eliminating separate AJAX requests for those source files, which leads to **fewer ajax requests**.

 Developers can detect and handle errors during the building phase, which helps in **minimizing errors**.

 The AOT compiler adds HTML and templates into the JS files before they run inside the browser. Due to this, there are no extra HTML files to be read, which provide **better security** to the application.

### 10. How are observables different from promises?

The first difference is that an Observable is **lazy** whereas a Promise is **eager**.

|  |  |
| --- | --- |
| Promise | Observable |
| Emits a single value | Emits multiple values over a period of time |
| Not Lazy | Lazy. An observable is not called until we subscribe to the observable |
| Cannot be cancelled | Can be cancelled by using the unsubscribe() method |
|  | Observable provides operators like map, forEach, filter, reduce, retry, retryWhen etc. |

Consider the following Observable:

const observable = rxjs.Observable.create(observer => {

console.log('Text inside an observable');

observer.next('Hello world!');

observer.complete();

});

console.log('Before subscribing an Observable');

observable.subscribe((message)=> console.log(message));

Now let’s consider a Promise:

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

console.log('Text inside promise');

resolve('Hello world!');

});

console.log('Before calling then method on Promise');

greetingPoster.then(message => console.log(message));

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.

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 { }

In JavaScript, only objects and arrays are mutable, not primitive values. ... A mutable object is an object whose state can be modified after it is created. Immutables are **the objects whose state cannot be changed once the object is created**. Strings and Numbers are Immutable.

|  |
| --- |
| <h2>Choose Brand Colors:</h2> |
|  | <color-sample |
|  | [color]="primary" |
|  | #primaryColorSample |
|  | (click)="open()"> |
|  | </color-sample> |
|  | ... the rest of the AppComponent template ... |

What is forkJoin in angular?

'forkJoin' is the easiest way, when you need to wait for multiple HTTP requests to be resolved. 'forkJoin' waits for each HTTP request to **complete and group's all the observables returned by each HTTP call into a single observable array** and finally return that observable array.

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.

It means that the component with the async pipe will be marked for check every time a new value is emitted. And Angular will check the component next time it **runs change detection even** if the value hasn't changed.

forkJoin - When all observables complete, emit the last emitted **value** from each. combineLatest - When any observable emits a value, emit the latest value from each.