**Angular interview questions  
Angular project creation with specific version - npx -p @angular/cli@14 ng new {project name}**

1. **Angular advantages**

* Custom reusable components : We can create our own components as per requirements and which can be reusable across the application
* Productivity and code consistency : Components and services
* Easy testing – Unit testing and we can add debugger in browser with sourcemap as true
* High compatibility : Cross platform compatible e.g chrome or firefox. Can develop small, medium, large applications. JS is in constant state of flux so angular is a framework for which company can rely on as it provides constant support and upgradation

1. **Router**

* [@angular/router](https://v17.angular.io/api/router) API
* Define routes object
* // AppRouting.module.ts  
  const routes = [

{

path : ‘home’,

component: HomeComponent,

title: ‘Home’,

canActivate: [CanActivateAuthGuard],

loadChildren: () =>

import('./menu/menu.module').then((x) => x.MenuModule), // it will load menumodule where we will specify menu routes

pathMatch: full | prefix,  
},

{  
path: ‘learn’,

component: LearnComponent,

children:[

{

path : ‘learnAngular’,

component: LearnAngularComponent

},

path : ‘learnReact’,

component: LearnReactComponent

]},

{

path: 'team/:id',

component: TeamComponent

},

// default route

{

path: '',

redirectTo: '/home’,

pathMatch: 'full'

},

// 404 page not found if any route not matches

{

path: '\*\*',

component: WildcardComponent

},

],

@NgModule({

imports: [RouterModule.forRoot(routes)],

exports: [RouterModule]

})

export class AppRoutingModule { }

// MenuRoutes.modules.ts

export const MenuRoutes: Routes = [

{

path: 'menu',

component: MenuListComponent

},

{

path: 'menu/:id',

component: MenuSingleComponent

}

];

@NgModule({

imports: [RouterModule.forChild(MenuRoutes)]

})

export class MenuModule {}

1. **Components**

* Main building blocks.
* Components provide us the way to build ui using template, logic file ts , styles files.
* Components are reusable
* @Component({

selector : ‘app-component-overview’,

templateUrl:’./component-overview.component.html’,

styleUrls : [‘./component-overview.component.scss’]  
})

1. **Compilation types**

* AOT
  + Best and recommended Compiler
  + Default compilation from Angular 9
  + Code compiles before loading into the browser
  + Source code translate to optimized JS code during the development phase
  + Reduces initial load time as it pre-compiles
  + Detects and reports errors during compilation phase so developer can resolve it and load application
  + Leading to reliable application with fewer runtime issues
  + ng build –aot
  + ng serve –aot
  + To create a production build with AOT optimizations, run the following command: ng build –prod

* JIT
  + Compiles code during run time, right into your browser
  + Detects and reports error during runtime
  + When opens web app in browser JIT starts compilation
  + JIT compiler reads and understands the code
  + Then translate to javascript code understandable by browser
  + Browser uses this translated JS code to build and display the applicatopn
  + More Chances of getting error during running the application in browser
  + You can use this during development but recommended AOT for prod

1. **What are the advantages of TypeScript over JavaScript?**
2. **Directives**

* Directives are used to add some behavior to DOM elements
* We can build custom directives too
* Attribute directive
  + Change look and feel of element
  + ngStyle , ngClass
  + Custom attribute directive

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

@Directive({

selector: '[appHighlight]'

})

export class HighlightDirective {

constructor(private eleRef: ElementRef) {

eleRef.nativeElement.style.background = 'red';

}

}

* Structural directive
  + Add or remove element
  + \*ngIf
  + Custom structural directive

import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';

@Directive({

selector: '[appNot]'

})

export class AppNotDirective {

constructor(

private templateRef: TemplateRef<any>,

private viewContainer: ViewContainerRef) { }

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

if (!condition) {

this.viewContainer.createEmbeddedView(this.templateRef);

} else {

this.viewContainer.clear(); }

}

}

1. **Services**

* Common tasks can be written is services
* It is used to share the data within application
* Reusable
* Code organization
* Dependency injection – Services are injected into components using DI

1. **Dependency injection**

* For service to be used in component first it needs to be injected. DI takes care of it
* Component doesn’t need to worry about creating the service as it will be take care by DI
* Modular so reusable
* We can use service in component, directive, pipes and other service

1. Providers
   1. Object for managing and creating instances of dependencies that can be injected into component and services
   2. Import { Component, Injectable } from ‘@angular/core’;

@Injectable()

export class MyService{

getData() {  
 return ‘some data’;

}

}

@Component({

Selector: ‘app-my-component’,

Providers: [ MyService ],

Template: ‘templateurl’

})

export class MyComponent {

constructor(private myService: MyService) {}

data = this.myService.getData();

}

1. Injectors
   1. Injectors responsible for managing and creating dependencies
   2. import { Component, Injectable, Injector } from '@angular/core';

@Injectable()

export class MyService {

getData() {

return "Data from MyService";

}

}

@Component({

selector: 'my-component',

template: '{{ data }}'

})

export class MyComponent {

constructor(private injector: Injector){}

data = this.injector.get(MyService).getData();

}

1. Normal way
   1. import { Component, Injectable, Injector } from '@angular/core';

@Injectable(

{

providedIn: 'root'

}

)

export class MyService {

getData() {

return "Data from MyService";

}

}

@Component({

selector: 'my-component',

template: '{{ data }}'

})

export class MyComponent {

constructor( private myService: MyService ){}

data = this.myService.getData();

}

1. **Angular 12 Features**

* **nullish coalescing operator**

old syntax  
{{age !== null && age !== undefined ? age : calculateAge() }}

new

**{{ age ?? calculateAge() }}**

* **TypeScript 4.2**

improvements in operator, type alias, tuple types, and abstract classes

* **Deprecating support for IE11**

As Microsoft stop support for IE11

* **Sass support for inline styles**

We can use SAAS together with inline styles in component’s metadata

e.g

@Component({

Styles: [

`  
@import ‘’style/base’

.heading{

.headigText{

Margin-bottom:40px

}

}

`

]

})

To enable this feature add “inlineStyleLanguage”: “scss” to angular.json

* **Production build by default**

Running ng build now defaults to production which saves teams some extra steps and helps to prevent accidental development builds in production!

* **Router changes**

routerLinkActiveOptions – possible to specify whether we need exact match or not for different part of url to apply css class

* **Http improvements**
* Introduced human-readable names for HTTP status code

e.g if(response.status === HttpStatusCode.Ok){}  
HttpStatusCode.Ok is 200

* httpParams has new appendAll method

const params = new HttpParams().appendAll({  
 ‘pageId’:’invest’,

‘project’:’PE’  
})

Const url = ‘https://test:8080/data’;

Const newUrl = url + ‘?’ + params.toString();

Console.log(newUrl) // https://test:8080/data?pageId=invest&project=PE

1. **Observables**

* To pass the data within components or application
* Asynchronous operations
* Observer will send the data
* Observable will listen to data passed by observer
* Observable part of RxJS library
* Subscriber take three optional parameters  
  this.myObservable.subscribe(next, error, complete);
* Next, error, complete
* Observable will emit data only if there is subscriber (observer)
* Its unicast because each subscribed observer has its own execution
* E.g  
  public observerValue = new Observable((observerVal)=>{  
   observerVal.next(Math.floor(Math.random() \* 99 ) + 1);   
  })

ngOnInit(){  
 observerValue.subscribe((val)=>{  
 console.log(val)  
})

observerValue.subscribe((val)=>{  
 console.log(val)  
})  
}

// Output - it will print random different numbers because each has its own execution  
99  
20

* It has separate observer and observable

1. **Subject**

* Emits data even if there is no subscriber
* Subject are multicast which means execution shared among multiple subscribers
* E.g   
  let mySubject = new Subject<any>();

mySubject.subscribe((val) => {  
 console.log(vale)  
})

mySubject.subscribe((val) => {  
 console.log(vale)  
})

mySubject.next(Math.floor(Math.Random() \* 99 +1))

//Output – it will print same random number because each subscriber shared the same execution  
  
99

* It works as both subscriber and observable

1. **Change detection**

* Sync the data between view and ts
* Change detection happens in below cases
  1. When event like click, submit, mouseover etc called
  2. Asynchronous functions like settimeout, setinterval etc called
  3. Api request is invoked
* Each component has its own change detection
* Even if there is click event in anyone component in application change detection will trigger for all components starting from top to bottom
* Still its doesn’t affect on performance because its unidirectional flow and after single pass change detection tree gets stable
* Default change detection strategy will check all components if event happed in any one component
* To  **optimize the no. of change detection we can use onPush strategy**
* @Component({

changeDetection : ChangeDetectionStartagy.OnPush  
})

If we add this in child component and click event happen in parent com then it will not update the view in child com

* OnPush is checkOnce , implies that change detection is skipped unless a comp is marked as dirty

1. **Pipes**

* Pipes are used for data transformation
* Types

. Pure - by default   
. Impure

* **Impure pipes**   
   a. Pipe will always transform even if data to be changed is not changed from previous
  1. Change detection will always run instead of data change or not

1. **Pure Pipes**
   1. If data has not changed since last time pipe was called, pipe won’t execute again
   2. Use to improve the performance because it will execute only when data is changed from previous run.
   3. Helps to reduce the change detection cycles
   4. We can make custom pipe as pure **with property pure: true**
   5. **Pure pipes check for object or array reference and does not do deep check.** So if any item added in array then it will not rerun the transform function as reference is not change. So it will not display updated array on ui. In this case **impure pipes**  are useful
2. Custom Pipes   
   e.g   
   import { Pipe, PipeTransform } from ‘@angular/core’;  
   @Pipe({  
    name: ‘filterByLength   
   })

export class implements PipeTransform {

transform(values: string[], minLength: number){  
 return values.filter((value) => value.length >= minLength)  
 }

}

.html file  
<ul>  
 <li \*ngFor="let value of values | filterByLength: 5">{{ value }}</li>  
 </ul>

1. Chaining the pipe

{{ myDate | date:’medium’ | upprcase }}

1. In built pipes

date   
uppercase  
async  
percent

1. Async pipe
2. Used to handle asynchronous data stream on html
3. Commonly use to subscribe to observables or promises
4. E.g  
   // component.ts  
   export class MyComponent {  
    myPromise$: Promise<string> = Promise.resolve('Hello World!');  
   }

// component.html  
<h2>Using AsyncPipe with Promise</h2>  
 <p>{{ myPromise$ | async }}</p>  
  
Output  
Using AsyncPipe with Promise  
Hello World!

1. **Data binding**

* One way and two way data binding
* String interpolation {{ propertyName }} – one way – component to view
* [(ngModel)]=”proptyName” – two way - <input type=’text’ [(ngModel)]=’name’ />
* Property binding - <app-child [dataToPass]=”dataPass”></app-child>  
  @Input() dataToPass: string = ‘dataPassed”

Event **binding** - <app-child [dataToPass]=”dataPass” (dataThatPassed)=”receivedData()”></app-child>  
@Output() dataThatPassed;  
  
clickToPassData(){  
this.dataThatPassed.emit(‘name’)  
}

1. **Find most occurred character in string**  
   function maxChar(str) {

const charMap = {};

let max = 0;

let maxChar = '';

// create character map

for (let char of str) {

if (charMap[char]) {

// increment the character's value if the character existed in the map

charMap[char]++;

} else {

// Otherwise, the value of the character will be increamented by 1

charMap[char] = 1;

}

}

// find the most commonly used character

for (let char in charMap) {

if (charMap[char] > max) {

max = charMap[char];

maxChar = char;

}

}

return maxChar;

}

1. **Angular 15 features**

* StandAlone component
* Directive composition API
* NgOptimizedImage directive– was introduced in v14 but got stable in v15

1. **Standalone Component**

* Streamline the authoring experience by reducing the need for NgModules
* Components, directives and pipes declare with standalone: true need not to be declare in NgModule (Compiler will give an error if you try to declare in NgModule)
* One standalone component can directly import any other standalone component
* When you want to use any other component, directives or pipes which are not even standalone but instead declared and exported by NgModule. In this case you can directly import NgModule directly into standalone component.

E.g

@Component({

standalone: true,

selector:’photo-gallery’,

imports: [MatButtonModule],

templateUrl:’html file path’,

styleUrls:’scss file path’

})

- Standalone components can be imported in existine NgModule

- An angular application can bootstrapped without NgModule by using standalone component as the application’s root component.

E.g

// in the main.ts file

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

import {PhotoAppComponent} from './app/photo.app.component';

bootstrapApplication(PhotoAppComponent);

* We can lazy load the Standalone component with below code

const routes = [{

path: ‘admin’,

loadComponent: ()=> import(‘component file path’).then(mod => mod.Component class name)

}]

1. **Directive Composition API**

* To create reusable and modular directives
* Works only on standalone directives
* Building multiple smaller task specific directives and then combine them as needed
* Example data table which has filtering, searching, pagination. So we can create multiple small task specific directives and then combine to form data table directive
* Uses

Modularity

Reusability

Flexibility

Readability

* So lets see if we have color , border directive and section component needs to have directives as needed
* Older way

<app-section colrDirective borderDirective></app-section>

<app-section colrDirective borderDirective></app-section>

<app-section colrDirective borderDirective></app-section>

<app-section colrDirective borderDirective></app-section>

* Now lets see if we want to remove one colorDirective then we have to remove it from all places
* Instead if we use directive composition api we will need to only remove it from hostDirectives
* E.g

@Component({

selector : ‘app-section’,

templateUrl : ‘’path,

stylesUrls:[’path’],

hostDirectives:[ColorDirective, BorderDirective] // Add/ Remove directives as required

})

* We can add one directive inside another directive
* Lets see create one more directive ColorBorderDirective
* E.g

@Directive({

Selector: [‘ColorBorder’],

hostDirectives:[ColorDirective, BorderDirective],

Standalone:true

})

1. **NgOptimizedImage**

* Helps to improve image loading performance
* Enforces best practices on using images
* Import NgOptimizedImage in appmodule.

E.g

<img [ngSrc]="mountainImg" width="474" height="306" priority />

<img [ngSrc]="roadImg" width="474" height="266" />

<img [ngSrc]="sunRays" width="474" height="355" />

<img [ngSrc]="lakeImg" width="640" height="426" />

* We will get error if we don’t provide the width and height to image which we usually don’t get when using src. So it resolves shift layout issue where text might appear in between till image is loaded
* We can provide priority attribute to the image which takes lot of time to load.
* By default image are lazy loaded. So it loads as required so it reduces application initial loading time
* Automatically generating a `srcset` attribute

1. **Server side rendering**

* Improved performance
* Delivering fully rendered HTML to client, which the browser can parse and display even before it downloads the application js
* Useful for users low bandwidth connection or mobile devices
* Improved core web vitals
* Core Web Vitals (CWV)](https://web.dev/learn-core-web-vitals/) statistics, such as reduced
* First Contentful Paint ([FCP](https://developer.chrome.com/en/docs/lighthouse/performance/first-contentful-paint/)) and
* Largest Contentful Paint ([LCP](https://web.dev/lcp/)), as well as
* Cumulative Layout Shift ([CLS](https://web.dev/cls/)).
* Better SEO search engine optimization of web applications by making it easier for search engines to crawl and index then content of the application
* To create new SSR project use ng new –ssr
* To add ssr in existing project use ng add @angular/ssr
* These commands create and update application code to enable SSR and adds extra files to the project structure.
* my-app
* |-- server.ts # application server
* └── src
* |-- app
* | └── app.config.server.ts # server application configuration
* └── main.server.ts # main server application bootstrapping

1. **Signals**

* Offers granular tracking of state changes
* Introduces in V16 and became stable in V17
* Signals can notify interested consumers when they change
* Three types
* a. Writable signals – allows to modify value directly
* b. Computed signals
* c. Effects
* Writable Signals
* -Create signal using signal() function with initial value
* e.g

import {signal} from ‘@angular/core’

const count = signal(0);

colors = signal([‘Red’, ‘Green’]);

* Signals are getter function so calling them wil get value
* console.log(‘ The count is ’,count()); // 0
* To change the value use set directly
* to update the value base on previous value use update
* count.set(3)
* Update the value
* count.update( value => value + 1 ); // 1
* colors.mutate( values => values.push(‘Blue’) ); Red, Green, Blue // V16
* Colors.update(values => […values, ‘Blue’]) // V17 above way is deprecated in v17
* Computed signals
* Values derived from other signals
* Computed signals will be recalculated if any one signal changes
* Computed signals can be used when computation needs to be done basis on the signal
* Read only, we can not set values
* import {omputed} from ‘@angular/core’

length = signal(20);

Breadth = signal(40);

area = computed(() => this.length() \* this.breadth() )

increase (){

length.set(30)

}

* // first computed area signal will calculate 800
* // once click on increase it will take updated value of signal and will recalculate the are // 1200
* Effects
* it is an operation that runs when one or more signal values changes
* Always runs at least once
* Keeps track of their dependencies dynamically
* Uses
* . to log something when signal changes
* . to update local storage when signal changes
* Do not update signal inside effects because it may trigger infinite circular updates and trigger unnecessary detection cycle
* To avoid this signal is disallowed by default. Writing signals inside computed or effects is not allowed by default
* If we are having normal variables and calculating a+b will not change the output after changing a to some other value
* but signals takes the updated value always so computed signals will recalculate if a signal value is changed
* Effects can be used when need to refresh some data based on signal value changes ( only if we are sure it will not create infinte loop ) but by default setting signal inside effect is not allowed.
* We can allowed setting signal inside effect using
* e.g

effect(() => {

this.refreshCart()

}, { allowedSignalWrites : true } );

usually write effect inside constructor

Effects destroys when components gets destroy

1. **Input Signals**

* Input signals can be used to bound values from parent component
* We can pass input signal value from parent to child component
* Its more powerful than @Input because updated values will be passed to child and recalculation happens on new values.
* With @Input we need to write extra line of code like ngonchanges to get the updated values and perform the operation
* Input signals are read only
* E.g
* Parent component
* <input type=”number” #inputNum />
* <button (click)=”calculate(inputNum)” > Calculate </button>
* <app-square [inputNum]=”num” />
* .ts

num = 0;

calculate(el){

this.num = el.value

};

* Square component

Import { input, computed } from ‘@angular/core’;

<p> Square is {{ result }} </p>

* .ts

inputNum = input(0);

result = computed(() => this.inputNum() \* this.inputNum() );

/\* without input signal

@Input() inputNum: number = 0;

result = this.inputNum \* this.inputNum; // this will not update if input property changes so need to write below ngonchanges code

ngOnChanges (changes: SimpleCahnges) :voide {

this.result = this.inputNum \* this.inputNum;

}

\*/

* Optional input - inputNum = input(0);
* Required input – inputNum = input.required<number>()
* Aliasing input signal
* inputNum = input( 0, { alias : ‘num’ } );
* Using in template
* <p> {{ inputNum() }} </p>
* Required inputs do not require initial values
* Signal inputs, when used in templates, will automatically mark OnPush components as dirty.
* Values can be easily derived whenever an input changes using computed.
* Easier and more local monitoring of inputs using effect instead of ngOnChanges or setters.

1. **Model inputs**

* Model inputs are writtable
* Two way data binding is possible with model inputs
* Propagate new value back to another component
* Allow to write the value into the property
* E.g
* 1. Parent child communication
* square can write value to squareOfNum which then propagates the value back to square property of parent component.
* main.ts => parent component
* <app-square [inputNum]="num" [(squareOfNum)]="square"/>
* <p> From child square {{ square }}
* square.ts => child component
* @Component({
* selector: 'app-square',
* standalone: true,
* template: `
* <p>{{ result()}}</p>
* <button (click)="calculate()" > Square for parent </button>
* `,
* })
* export class Square {
* inputNum = input(0);
* result = computed(() => this.inputNum() \* this.inputNum());
* squareOfNum = model(2);
* calculate() {
* this.squareOfNum.set(this.inputNum() \* this.inputNum());
* }
* }
* 2. Two way binding in same component
* <div (click)=”toggle()” ></div>
* export class CustomCheckBox{
* checked = model(false);
* Disabled = input(false);
* Toggle () {
* this.checked.set( !this.checked() );.
* }
* }
* When component writes a new value into model, component can propagate new value back to component that binding the into that input.
* Implicit change event
* When you declare a model input in a component or directive, Angular automatically creates a corresponding output for that model. The output's name is the model input's name suffixed with "Change".e.g checkedChange

1. **Differences between model() and input()**

* Model defines both input and output
* Model writable i.e changed from anywhere with set and update method
* Input can be changed from template
* Model inputs do not support input transforms while signal inputs do.

1. **Queries as Signals**

* Component or directive can define queries which can find child elements and read values from their injectors
* Commonly used to retrieve reference to component, directives or Dom elements and more
* Two categories
* View queries
* Content queries
* View Queries
* . view queries retrieve result from elements in component’s own template ( view )
* import { Component,

signal,

computed,

input,

model,

viewChild,

ElementRef,

viewChildren,

} from '@angular/core';

. @Component({

selector: 'app-square',

standalone: true,

template: `

<input type="text" #textInput>

<input type="text" #textInput>

<input type="text" #textInput2>

<button (click)="focus()">Focus input </button>

<button (click)="focus2()">Focus input 2 </button>

`,

})

export class Square {

textInp = viewChildren<ElementRef>('textInput');

textInp2 = viewChild<ElementRef>('textInput2');

focus() {

this.textInp()[0]?.nativeElement.focus();

}

focus2() {

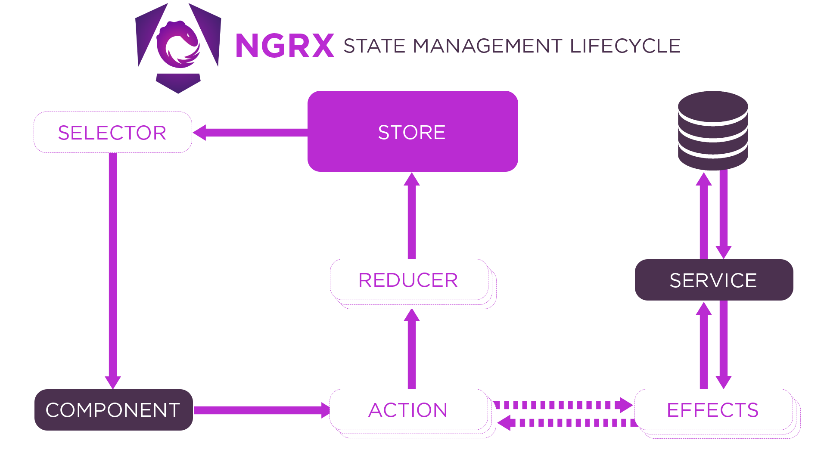
alert(this.textInp2()?.nativeElement.value);

}

}

* Content queries
* Content queries retrieve results from the elements in the component's content
* the elements nested inside the component tag in the template where it's used.
* You can query for a single result with the contentChild function.
* You can also query for multiple results with the contentChildren function.

1. **Signal and Subject difference**
2. **NgRx Store**

* Global state management for angular application
* **Actions –** describe events that are dispatched from components and services
* **Reducers -** state changes are handles by reducers that take the current state and last action to compute new state
* **Selectors –** pure functions used to select, derive and compose pieces of state
* **Store -** state is accessed with store, Observable of state and observer of actions
* 
* **E.g**app/counter.actions.ts

Import { createActions } from ‘@ngrx/store’;  
  
export const increment = createAction(‘[Counter Component ] Increment ’);  
  
app/counter.reducers.ts  
  
import { createReducer, on } from ‘ngrx/store’;

Import {increment } from ‘app/counter.actions.ts’;

Export const initialState = 0 ;  
export const counterReducer = createReducer (  
 initialState,  
 on(increment, (state) => state+1)  
);

App.module.ts

Import { StoreModule } from ‘@ngex/store’;

Import { counterReducer } from ‘./counter.reducer.ts’;

@NgModule({

Imports : [BrowserModule, StoreModule.forRoot( { count : counterReducer} )]

})

my-counter.componet.ts

import { Store } from ‘@ngrx/store’;

import { increment } from ‘../counter.actions’;

export class MyCounter {

count$ : Observable<number>;

constructor(private store: Store< { count: number} >) {  
 this.count$ = store.select(‘count’);  
}

// Method will call when increment button is clicked

increment() {

this.store.dispatch( increment() )

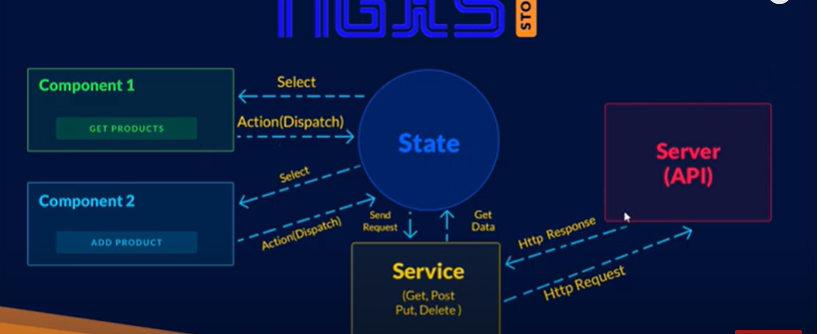
}

}

1. **NgRx Effects**

* https://youtu.be/uyW48EQge-8?si=4gGlFxzLPFj1XJdc
* Effects are used to fetch data from network call, long running tasks that produce multiple events where component don’t need to have knowledge about these interactions.
* In larger application this is very important because you have multiple sources of data , with multiple services to fetch data and those services again relying on other services to fetch data .
* Effects handle external data and interactions, allowing your services to be less stateful and only perform tasks related to external interactions.
* Effects is your service which listens to all your actions and it will react to those actions and will return again an action depends on true and false.
* **Send == Component => Action => Effect => Service => Server**
* **Receive == Server => Observable => Service => Plug in our service to effect which is observable stream => Observable => Effect ( outside of store ) => Dispatch an action => Reducer => State ( updates ) => Component**

1. **NgXs**

* ****
* Global state management pattern + library for Angular
* Reduces boilerplate with moder Typescript such as classes and decorators
* NgXs tries to make thing as **simple** and accessible as possible
* NgXs get rid of switch statement as library is responsible for knowing when functions needs to be called
* NgXs make sure to make use of **Dependency injection** through allowing services can be injected into state classes making it easier to take advantage of angular features
* 4 Pillars  
  Store  
  Actions  
  State  
  Select
* **Store :**
* Global state management that dispatches action that state listens to and provides way to select data slices out from global state
* This.store.dispatch(action)  
  this.store.snapshot() // will return entire value of store for that point in time
* store.reset(myNewStateObject) will reset the entire state to the passed argument without firing any actions or life-cycle events.
* **Actions :**
* Actions can either be thought of command which should trigger something to happen or as the resulting event of something that has already happened
* **export class FeedAnimals {**
* **static readonly type = '[Zoo] Feed Animals';**
* **}**
* Actions with Metadata
* **export class FeedZebra {**
* **static readonly type = '[Zoo] Feed Zebra';**
* **constructor(**
* **public name: string,**
* **public hayAmount: number**
* **) {}**
* **}**
* **Action handlers -** The action handler is an Observable that receives all the actions dispatched before the state takes any action on it.  
  **ofAction:** triggers when any of the below lifecycle events happen
* **ofActionDispatched:** triggers when an action has been dispatched
* **ofActionSuccessful:** triggers when an action has been completed successfully
* **ofActionCanceled:** triggers when an action has been canceled
* **ofActionErrored:** triggers when an action has caused an error to be thrown
* **ofActionCompleted:** triggers when an action has been completed whether it was successful or not (returns completion summary)
* **State :**
* States are classes along with decorators that define metadata and action mappings
* In @State decorator we define metadata about state  
  @State<string[]>({  
  name:’animals’;   
  defaults:[]  
  })  
    
  @Injectable()  
  export class AnimalState{

}

**State metadata includes**name – unique name of state  
defaults – default state of object/array for this state slice

* State listens to an action through an @Action decorator.
* E.g

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

import { State, Action, StateContext } from '@ngxs/store';

export class FeedAnimals {

static readonly type = '[Zoo] FeedAnimals';

}

export interface ZooStateModel {

feed: boolean;

}

@State<ZooStateModel>({

name: 'zoo',

defaults: {

feed: false

}

})

@Injectable()

export class ZooState {

@Action(FeedAnimals)

feedAnimals(ctx: StateContext<ZooStateModel>) {

const state = ctx.getState();

ctx.setState({

...state,

feed: !state.feed

});

}

}

* **E.g**
* App/module.ts  
  NgxsModule.forRoot([EmployeeState])
* Employee.action.ts

Import { Employee } from ‘employee modal file path’;  
  
export class GetEmployee {  
 static readonly type = ‘ [Employee] Get’];  
}

Export class AddEmployee {  
 static readonly type = ‘[Employee] Add’;  
 constructor( public payload:Employee ) { }  
}

* Employee.state.ts  
  import { State, Selector, StateContext } from ‘@ngxs/store’;

Import { Employee } from ‘modal file path’;

Import { GetEmployee } from ‘action file path’;  
  
// State modal  
export class EmployeeStateModel {   
 employees : Employee[];

employeeLoaded :boolean  
}  
  
// State  
@State<EmployeeStateModal> ({  
 name : ‘emploees’,

defaults : {  
 employees : [],

employeeLoaded : false   
 }  
})  
  
@Injectable()  
export class EmployeeState {  
  
constructor(private empService : EmployeeService)

// Selector has logic

// Get employees   
 @Selector()  
 static getEmployeesData (state : EmployeeStateModal) {  
 return state.employees;  
 }  
  
 @Selector()  
 static employeeLoaded(state : EmployeeStateModel) {  
 return state.employeeLoaded;  
 }  
  
 @Action(GetEmployee)

getEmployees( { getState, setState } : StateContext<EmployeeStateModel>)

{

return this.empService.getEmployeeList().pipe(tap(res => {  
 console.log(‘tap resp’,res);  
 const state = getState();

setState(  
 …state,

employees :res,

employeeLoaded : true   
// logic to avaoid multiple networkmcall if already data in state   
 } )

)  
 }

@Action(AddEmployee)  
addEmployee({getState, patchState} : StatContext<EmployeeStateModel>, { payload } : AddEmployee)  
{  
 return this.empService.addEmployee(payload).pipe (  
 tap ( res => {

Const state = getSate() ;  
  
 patchState ({  
 employees : [ …state.employees, res]  
 })  
 }))  
}  
  
@Action( DeleteEmployee)

deleteEmployee( {getState, setState} : StateContext , { id } : DeleteEmployee){  
 return this.empService.deleteEmployee(id).pipe(tap(res=>{  
 const state = getState() ;

const filteredEmployees = state.employees.filter( emp => emp.\_id !== id);  
  
 setState({  
 …state,  
 employees : filteredEmployees  
 })  
 }))  
}  
}

* **Employee.component.ts**

import { Store, Select } form ‘@ngxs/store’;

import { GetEmployee } from ‘action file path’;

import { EmployeeState } from ‘stata file path’;

Constructor (private store : Store ) { }  
@Select ( EmployeeState.getEmployeeList ) emploees$ : Observable<Employee[]>;

@Select ( EmployeeState.employeeLoaded ) employeeLoaded$ Observable<Boolean>;

employeeLoadedSub : Subscription

ngOnInit() {  
 this.employees$.subscribe( res = > cosole.log(‘ Data from select ’, res) ;)  
 }

getEmployees() {  
 this.employeeLoadedSub = thid.employeeLoaded.subscribe( isLoaded => {  
 if( !isLoaded ) {  
 this.store.dispatch( new GetEmployee() );  
 }  
})  
}  
  
addEmployee() {  
 this.store.dispatch( new AddEmployee() );  
}

ngOnDestroy() {  
 this.employeeLoadedSub.unsubscribe();  
 }  
  
employee.component.html   
  
<div \*ngFor=”let emp of employee$ | async ”>{{emp.\_id}}</div>

* **LifeCycle**
* ngxsOnChanges
* ngxsOnInit
* ngxsAfterBootstrap
* **Select**
* Use to select the data from state
* **@Select – select data of state – Observable where we access state slice data**
* **@Selector – Logic to select the data**

1. **Promise**
2. **RxJs**

* **Check code**

1. **Ng -container**

* We can use it as container
* Don’t add extra element in DOM
* As we cant add more than one directive on elements so it will add unnecessary elements in DOM so we can use ng-container
* <b>E.g Behavior Subject | </b>
* <ng-container \*ngIf="exclusiveBehav">
* <p \*ngFor="let ele of arrOfColors">{{ ele }}</p>
* </ng-container>