**Interview:**

<https://stackblitz.com/edit/angular-np1v8q-hhecny?file=src%2Fapp%2Fapp.component.html,src%2Fapp%2Fservice.service.ts,src%2Fapp%2Fapp.component.ts,src%2Fapp%2Fapp.module.ts,src%2Findex.html,src%2Fapp%2Fproduct-list%2Fproduct-list.component.html>

**Creating Ionic New App?**

* will need to have Node.js and Node Package Manager (NPM) installed on your computer
* You will need to have the corresponding SDKs (Software Development Kits) installed on your computer.
* Install the Ionic CLI: npm install -g @ionic/cli
* For iOS you will need [Xcode](https://apps.apple.com/us/app/xcode/id497799835?mt=12) (which means you will need to be working on a Mac), and for Android you will need [Android Studio](https://developer.android.com/studio/install).
* Create a New Ionic Angular Project: ionic start myAppName tabs --type=angular
* Add Capacitor to the Project:

npm install @capacitor/cli @capacitor/core

npx cap init myAppName com.example.myapp

* Integrate With iOS, Android

npx cap add ios -> npx cap open ios

npx cap add android ->npx cap open android

**ng Angular Commands?**

**ng new**: Creates a new Angular project.

**ng generate (or ng g)**: Generates components, directives, services, modules, and more.

ng generates component my-component

ng g directive my-directive

ng g service my-service

ng g module my-module

**ng serve**: Builds and serves the application in development mode, with live reload.

**ng build**: Builds the application for production.

**ng test**: Runs unit tests using Karma and Jasmine.

**ng e2e**: Runs end-to-end tests using Protractor.

**ng lint**: Lints the codebase using TSLint or ESLint.

**ng add**: Adds new capabilities to the project by installing and configuring libraries and packages.

**ng update**: Updates the dependencies of the project to the latest versions.

**ng config**: Sets or gets Angular configuration values.

**ng help**: Displays helpful information about Angular CLI commands.

**What is Angular coding language?**

Angular itself is not a coding language.

It is a framework for building web applications using TypeScript.

TypeScript is a superset of JavaScript that adds static typing and additional features to JavaScript.

Angular leverages TypeScript to provide a more structured and scalable approach to web development.

**Lifecycle hooks in Angular?**

1. **ngOnChanges**: Called whenever one or more data-bound input properties change. It receives a **SimpleChanges** object containing the previous and current values of the properties.
2. **ngOnInit**: Called once, after the first **ngOnChanges** (if any) and before the rendering of the component. This is where initialization logic should go.
3. **ngDoCheck**: Called during every change detection cycle, which means very frequently. Use it to implement your own change detection logic.
4. **ngAfterContentInit**: Called after Angular projects external content into the component's view (such as inserting content into **<ng-content>**).
5. **ngAfterContentChecked**: Called after Angular checks the content projected into the component.
6. **ngAfterViewInit**: Called after Angular initializes the component's views and child views.
7. **ngAfterViewChecked**: Called after Angular checks the component's views and child views.
8. **ngOnDestroy**: Called just before Angular destroys the component. Clean-up logic, such as unsubscribing from observables or detaching event handlers, should go here.

**Ionic LifeCycle Hooks?**

* **ionViewWillEnter**: Before the view is presented. Good for initializing data.
* **ionViewDidEnter**: After the view is fully presented. Ideal for starting animations or processes.
* **ionViewWillLeave**: Before the view is left. Useful for pausing or stopping tasks.
* **ionViewDidLeave**: After the view has been left. Suitable for cleanup tasks.
* **ionViewWillUnload**: Before the view is destroyed. Good for final teardown tasks.

**Why ngOnInit when we have constructor?**

### Constructor

Purpose: The constructor of a class is a special method that is called when an instance of the class is created. In Angular components, the constructor is primarily used for dependency injection.

* Usage:
  + Initialize basic properties of the component.
  + Inject services or dependencies needed by the component.
* Limitations: The constructor is called before Angular has fully initialized the component, which means that many Angular-specific features (such as inputs, outputs, and the component's view) are not yet available.

### ngOnInit

* Purpose: ngOnInit is a lifecycle hook provided by Angular. It is part of the OnInit interface and is called by Angular after the constructor and after Angular has initialized all data-bound properties of the component.
* Usage:
  + Perform complex initialization that requires access to inputs and outputs.
  + Fetch data from services or APIs.
  + Set up the initial state of the component that depends on bindings.
* Advantages:
  + By the time ngOnInit is called, Angular has finished setting up the component, so you can safely access input properties and other component features.
  + Helps in organizing initialization logic separately from the basic setup and dependency injection done in the constructor.

### Comparison and Best Practices

* Initialization Timing:
  + Constructor: Called immediately when the component is created.
  + ngOnInit: Called after Angular initializes the component’s data-bound properties.
* Usage:
  + Constructor: Best used for simple initialization and dependency injection.
  + ngOnInit: Best used for complex initialization, especially when it depends on inputs or requires interaction with services.

**Building Block of Angular?**

The architecture diagram identifies the eight main building blocks of an Angular application:

* [Modules](https://v2.angular.io/docs/ts/latest/guide/architecture.html#modules)
* [Components](https://v2.angular.io/docs/ts/latest/guide/architecture.html#components)
* [Templates](https://v2.angular.io/docs/ts/latest/guide/architecture.html#templates)
* [Metadata](https://v2.angular.io/docs/ts/latest/guide/architecture.html#metadata)
* [Data binding](https://v2.angular.io/docs/ts/latest/guide/architecture.html#data-binding)
* [Directives](https://v2.angular.io/docs/ts/latest/guide/architecture.html#directives)
* [Services](https://v2.angular.io/docs/ts/latest/guide/architecture.html#services)
* [Dependency injection](https://v2.angular.io/docs/ts/latest/guide/architecture.html#dependency-injection)

**How does an Angular application work?**

Every Angular app consists of a file named **angular.json**. This file will contain all the configurations of the app. While building the app, the builder looks at this file to find the entry point of the application. Following is an image of the angular.json file:

"build": {

"builder": "@angular-devkit/build-angular:browser",

"options": {

"outputPath": "dist/angular-starter",

"index": "src/index.html",

"main": "src/main.ts",

"polyfills": "src/polyfills.ts",

"tsConfig": "tsconfig.app.json",

"aot": false,

"assets": [

"src/favicon.ico",

"src/assets"

],

"styles": [

"[./node\_modules/@angular/material/prebuilt-themes/deeppurple-amber.css](mailto:./node_modules/@angular/material/prebuilt-themes/deeppurple-amber.css)",

"src/style.css"

]

}

}

Inside the build section, the main property of the options object defines the entry point of the application which in this case is **main.ts**.

The main.ts file creates a browser environment for the application to run, and, along with this, it also calls a function called **bootstrapModule**, which bootstraps the application. These two steps are performed in the following order inside the main.ts file:

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

…..

……

platformBrowserDynamic().bootstrapModule(AppModule)

In the above line of code, **AppModule** is getting bootstrapped.

The AppModule is declared in the app.module.ts file. This module contains declarations of all the components.

Below is an example of app.module.ts file:

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

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

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

@NgModule ({

declarations: [

AppComponent

],

imports: [

BrowserModule

],

providers: [],

entryComponents: [],

bootstrap: [AppComponent]

})

export class AppModule { }

As one can see in the above file, **AppComponent** is getting bootstrapped.

This component is defined in **app.component.ts** file. This file interacts with the webpage and serves data to it.

Below is an example of app.component.ts file:

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

@Component ({

selector: 'app-root',

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

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

})

export class AppComponent {

title = 'angular';

}

Each component is declared with three properties:

* **Selector** - used for accessing the component
* **Template/TemplateURL** - contains HTML of the component
* **StylesURL** - contains component-specific stylesheets

After this, Angular calls the **index.html** file. This file consequently calls the root component that is **app-root**. The root component is defined in **app.component.ts**. This is how the index.html file looks:

<!doctype html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>Angular</title>

<base href="/">

<meta name="viewport" content="width=device-width, initial-scale=1">

</head>

<body>

<app-root></app-root>

</body>

</html>

The HTML template of the root component is displayed inside the <app-root> tags.

This is how every angular application works.

**Bootstrapping Module?**

Bootstrapping module refers to the primary module of an Angular application responsible for starting the application and initializing its components.

Each Angular application has at least one root module, which serves as the entry point for the application.

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

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

import { AppComponent } from './app.component'; // Assuming AppComponent is the root component

@NgModule ({

imports: [BrowserModule],

declarations: [AppComponent],

**bootstrap**: [AppComponent] // **AppComponent is specified as the root component to be bootstrapped**

})

export class AppModule {}

In this example, AppModule is the bootstrapping module, which imports BrowserModule and declares AppComponent.

The **bootstrap array specifies that AppComponent** should be bootstrapped when the application starts. ￼

Overall, the bootstrapping module in Angular serves as the entry point for the application, defining the root components and initiating the application startup process.

Here's how a bootstrapping module works:

* **Root Module (App-Module)**: The bootstrapping module is typically the root module of the Angular application. It is responsible for importing other modules, such as feature modules, and configuring the application by declaring components, providers, and other application-specific configurations.
* **Bootstrap Array**: The @NgModule decorator of the bootstrapping module includes a bootstrap property, which is an array that specifies the root component or components to be bootstrapped when the application starts. These components will be instantiated and rendered in the application's root DOM element.
* **Platform Bootstrap(main.ts)**: To start an Angular application, you need to use the platformBrowserDynamic().bootstrapModule() method provided by @angular/platform-browser-dynamic. This method takes the bootstrapping module as an argument and initializes the application by bootstrapping the specified root components.

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

…

const appModule = require('./src/app/app.module');

platformBrowserDynamic()

bootstrapModule(appModule.AppModule)

(err => console.log(err));

**What is transpiling in Angular?**

Transpiling in Angular refers to the process of converting TypeScript code into JavaScript code that web browsers can execute.

Angular applications are built using TypeScript, a superset of JavaScript that adds static typing and additional features to the language.

Since browsers can only run JavaScript, the TypeScript code needs to be **transpiled** into JavaScript before it can be executed.

This is typically done using the **TypeScript compiler (tsc)** or build tools like Angular CLI

**PolyFill file?**

polyfill files play a crucial role in enabling Angular applications to run smoothly across different browsers by filling in the gaps in browser support for modern JavaScript features and APIs. They help ensure compatibility, standardization, and consistent behavior, allowing developers to focus on building robust and feature-rich Angular applications.

* **Feature Support**: Polyfills provides support for modern JavaScript features and APIs in browsers that do not natively support them. For example, older versions of Internet Explorer may lack support for features introduced in ECMAScript 6 (ES6) or later, such as Promise, Array.prototype.includes, or Object.entries. Polyfills ensure that these features are available and behave consistently across different browsers.
* **Standardization**: Polyfills help ensure standardisation across browsers by providing a consistent implementation of JavaScript features according to the ECMAScript specification. This helps developers write code using modern JavaScript syntax and APIs without worrying about browser-specific quirks or inconsistencies.
* **Compatibility**: Angular relies on various modern JavaScript features and APIs to provide its functionality. By including polyfills, Angular applications can be run on a wider range of browsers, including older ones, without sacrificing functionality or performance.
* **Integration**: Angular CLI, the official command-line interface for Angular, includes built-in support for adding polyfills to Angular projects. The CLI automatically adds necessary polyfills to the project configuration based on the targeted browsers specified in the **browserslist** configuration file. This ensures that polyfills are seamlessly integrated into Angular applications without manual intervention.

**SCSS File => SASS File => Global.scss??**

In Angular applications, SCSS (Sassy CSS) files serve as a powerful tool for styling components, modules, and the overall user interface. SCSS is a superset of CSS, meaning it extends CSS with additional features like variables, nesting, mixins, and more, making CSS code more maintainable and organised.

Inside angular, json file.

….build": {

"options": {

"styles": [

"src/styles.scss"

]

}

},

Here are several key uses of SCSS files in Angular:

* **Styling Components**: SCSS files are commonly used to style individual Angular components. Each component can have its own SCSS file, which encapsulates styles specific to that component. This approach helps in maintaining a modular and scalable codebase, as styles are scoped to individual components, reducing the risk of style conflicts and making it easier to manage and refactor styles.
* **Global Styles**: SCSS files can also be used to define global styles that apply across [the](http://directives) entire Angular application. These global styles can be included in the main styles.scss file, which serves as the entry point for all styles in the application. Global styles may include reset styles, typography definitions, layout configurations, and other common styles that need to be applied universally.
* **Variables and Mixins**: SCSS allows the use of variables and mixins, which enable developers to define reusable values and styles that can be used throughout the application. Variables can be used to define colors, font sizes, spacing, and other commonly used values, making it easy to update styles consistently across the application. Mixins allow developers to define reusable sets of CSS rules, reducing duplication and improving maintainability.
* **Nesting and Selectors**: SCSS supports nesting of CSS selectors, allowing developers to write more concise and readable styles by nesting child selectors within parent selectors. This helps in organizing styles hierarchically, making it clear which styles apply to which elements within a component. Additionally, SCSS provides powerful selector features such as parent reference (&), which allows styles to be scoped to specific states or conditions.
* **Importing Stylesheets**: SCSS supports the @import directive, which allows stylesheets to be imported into other stylesheets. This feature enables developers to modularize styles by splitting them into separate files and importing them as needed. It also facilitates the use of third-party CSS frameworks and libraries, which can be imported into Angular projects via SCSS.

**NPX-Node Package Execute**

* In the command npx cap init myAppName com.example.myapp,
* It's a package runner tool that comes with npm (Node Package Manager) versions 5.2.0 and higher.
* It is used to execute binaries from npm packages that aren't globally installed.
* In this specific command, it's being used to initialize a new Capacitor project named "myAppName" with the given app ID "com.example.myapp".

Capacitor is a cross-platform app development tool for building web apps, hybrid apps, or native apps for iOS, Android, and Electron.

**Project-specific Node.js and npm Versions**:  
 nvm stands for Node Version Manager.

It is a command-line utility that allows you to manage multiple installations of Node.js and npm (Node Package Manager) on your system.

nvm enables you to easily switch between different versions of Node.js and npm, ensuring compatibility with different projects that may require specific versions.

You can specify Node.js and npm versions on a per-project basis by creating a .nvmrc file in the root directory of your project.

This file contains the desired Node.js version, and nvm will automatically use that version when you navigate to the project directory.

**Why Capacitor?**

Capacitor simplifies the process of building cross-platform apps by providing a unified development environment and easy access to native features, allowing developers to focus on building their app's functionality rather than worrying about platform-specific complexities.

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array>

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/string>

**RxJS**

RxJS (Reactive Extensions for JavaScript) is a library for reactive programming using Observables, which enables you to work with asynchronous data streams and events. It provides a powerful set of operators for transforming, combining, and reacting to data streams in a declarative and composable manner.

RxJS operators can be broadly categorized into the following types:

**Creation Operators**:

* + Creation operators are used to create observables from various data sources such as arrays, events, promises, timers, or other observables.

Examples:

* + **of**: Creates an observable from a sequence of values.
  + **from**: Converts an array, promise, or iterable into an observable.
  + **interval**: Emits sequential numbers at a specified interval.
  + **fromEvent**: Converts DOM events into an observable sequence.

**Transformation Operators**:

* + Transformation operators are used to transform the items emitted by observables into new observables or modify the stream in some way.

Examples:

* + **map**: Transforms each item emitted by an observable into another item.
  + **filter**: Filters items emitted by an observable based on a predicate function.
  + **pluck**: Extracts a specific property value from each emitted object.
  + **mergeMap** / **switchMap** / **concatMap**: Maps each item emitted by an observable to a new observable and flattens them into a single observable.

**Filtering Operators**:

* + Filtering operators are used to selectively emit items from an observable stream based on certain conditions.

Examples:

* + **filter**: Filters items emitted by an observable based on a predicate function.
  + **take**: Emits only the first n items emitted by an observable.
  + **distinct**: Emits unique items emitted by an observable (removes duplicates).
  + **debounceTime**: Emits a value from the source observable only after a specified time has passed without emitting any other value.

**Combination Operators**:

* + Combination operators are used to combine multiple observables into a single observable or perform operations on multiple observables simultaneously.

Examples:

* + **merge:** Combines multiple observables into one that emits items from all source observables**.**
  + **concat**: Concatenates multiple observables, emitting items from each source observable in sequence.
  + **combineLatest**: Combines the latest values emitted by multiple observables into an array or a new object.

**Error Handling Operators**:

* + Error handling operators are used to handle errors emitted by observables and recover from them.

Examples:

* + **catchError**: Handles errors emitted by an observable by returning a new observable or value.
  + **retry**: Automatically resubscribes to the source observable a specified number of times if it encounters an error.

These are just some of the categories and examples of RxJS operators. RxJS provides a vast array of operators for various use cases, enabling you to express complex asynchronous behavior in a concise and declarative manner. Understanding and mastering these operators is essential for effectively working with reactive programming in Angular applications and JavaScript in general.

**Fork Join (); & combineLatest() <= RxJs Library;**

**Fork Join ();**  
 **Multiple api calls, when all Api executed then it will subscribe ()**

In Angular, forkJoin is an operator provided by the RxJS library.

It's used to combine multiple observable sequences into a single observable sequence by running them concurrently and emitting an array of the last values emitted by each source observable once all observables complete.

import {Observable, forkJoin } from 'rxjs';

….

ngOnInit(): void {

// Example API endpoints

const endpoint1 = '<https://api.example.com/endpoint1>';

const endpoint2 = '<https://api.example.com/endpoint2>';

// Calling multiple HTTP requests concurrently using forkJoin

**forkJoin([**

**this.http.get(endpoint1),**

**this.http.get(endpoint2)**

**]). subscribe**(

([response1, response2]) => {

// Handle responses here

console.log ('Response from endpoint 1:', response1);

console.log ('Response from endpoint 2:', response2);

},

error => {

// Handle errors

console.error('Error:', error);

}

);

}

**combineLatest();**

**Multiple api calls, it will emit value at least one API return the value.**

In Angular, combineLatest is another operator provided by the RxJS library.

It is used to combine multiple observables into a single observable, which emits an array of the most recent values from each source observable whenever any of the source observables emits a new value.

import {Observable, combineLatest } from 'rxjs';

combineLatest([

observable1,

observable2

]). subscribe(

([response1, response2]) => {

// Handle responses here

console.log ('Response from observable 1:', response1);

console.log ('Response from observable 2:', response2);

},

error => {

// Handle errors

console.error('Error:', error);

}

);

**Differences b/w combineLatest() and forkJoin()**

* **Behavior**:
  + **combineLatest()**: Emits an array containing the latest values from each source observable whenever any of the source observables emits a new value. It waits for all observables to emit at least one value before emitting a combined value.
  + **forkJoin()**: Waits for all source observables to complete and then emits an array containing the last value from each source observable. It doesn't emit until all source observables complete, and if any of them errors out, it will also emit an error.
* **Triggering**:
  + **combineLatest()**: Emits whenever any of the source observables emit a new value.
  + **forkJoin()**: Waits for all source observables to complete.
* **Usage**:
  + **combineLatest()**: Useful when you want to combine multiple observables and react to changes in any of them. For example, combining user input observables (e.g., form fields) to perform a calculation.
  + **forkJoin()**: Useful when you want to wait for multiple observables to complete and then do something with their final values. For example, making multiple HTTP requests in parallel and processing the combined response.
* **Error Handling**:
  + **combineLatest()**: If any source observable errors out, combineLatest() continues to emit values from the remaining observables, but the error is propagated to the subscriber.
  + **forkJoin()**: If any source observable errors out, forkJoin() will immediately emit an error and unsubscribe from the remaining observables.

In summary, combineLatest() is typically used for real-time data synchronization or combining observables that represent ongoing streams of data, while forkJoin() is used when you need to wait for multiple asynchronous tasks to complete and combine their results.

**SwitchMap();**

Suppose you have an application where users can search for items using a search input field. Each time the user types a search query, an HTTP request is made to fetch the search results. However, you want to ensure that only the latest search results are displayed, and any previous requests that haven't been completed yet are canceled.

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

import { HttpClient } from '@angular/common/http';

import {Observable, Subject} from 'rxjs';

import { switchMap, debounceTime, distinctUntilChanged } from 'rxjs/operators';

@Component ({

selector: 'app-search',

template: `

<input type="text" (input)="search($event.target.value)" placeholder="Search...">

<ul>

<li \*ngFor="let item of searchResults$ | async"> {{item}} </li>

</ul>

`

})

export class SearchComponent {

searchResults$: Observable<string []>;

private searchTerms = new Subject<string> ();

constructor (private http: HttpClient) {

this.searchResults$ = this.searchTerms.pipe(

debounceTime(300), // Wait for 300ms after each keystroke

distinctUntilChanged(), // Only emit if the search term has changed

**switchMap**(term => this.searchItems(term)) // Switch to new search observable, canceling previous requests

);

}

search (term: string): void {

this.searchTerms.next(term); // Push search term into observable stream

}

searchItems(term: string): Observable<string []> {

if (!term.trim()) {

// If empty search term, return empty array immediately

return of ([]);

}

// Otherwise, make HTTP request to search API

return this.http.get<string[]>(`https://api.example.com/search?term=${term}`);

}

}

**App initializer in angular?**

In Angular, an app initializer is a feature introduced in Angular version 9 that allows you to execute a function or perform initialization tasks before the application is bootstrapped. This provides a way to perform any necessary setup or configuration before the application starts running.

App initializers are typically used for tasks such as:

1. Initializing configuration settings.
2. Fetching data from an API or performing other asynchronous tasks.
3. Setting up localization or internationalization.
4. Preparing the application environment.

To define an app initializer in an Angular application, you use the **APP\_INITIALIZER** provider provided by Angular. This provider allows you to register one or more functions that will be executed before the application is initialized.

import { NgModule, APP\_INITIALIZER} from '@angular/core';

import { HttpClientModule } from '@angular/common/http';

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

// Function to be executed as an app initializer

export function initializeApp() {

return () => {

// Perform initialization tasks here, such as fetching data

console.log ('Initializing application...');

// For example, fetch some configuration data from an API

// return httpClient.get('/api/config').toPromise();

};

}

@NgModule ({

declarations: [

// Declare your components, directives, etc. here

],

imports: [

BrowserModule,

HttpClientModule

],

providers: [

// Register the app initializer

{provide: APP\_INITIALIZER, useFactory: initializeApp, multi: true}

],

bootstrap: [/\* AppComponent \*/]

})

export class AppModule { }

In this example:

* We define a function **initializeApp** that performs initialization tasks. This function returns another function that will be executed as the app initializer.
* In the **providers** array of the **AppModule**, we register the **APP\_INITIALIZER** provider and specify our **initializeApp** function using the **useFactory** property.
* The **multi: true** option tells Angular that there may be multiple app initializer functions.

When Angular bootstraps the application, it will execute all registered app initializer functions before proceeding with the normal initialization process. This allows you to ensure that any required setup or initialization tasks are completed before the application starts running.

**Pipes:**

**Pure Pipes:**

Pure pipes are the default type of pipes in Angular.

They are stateless and do not depend on any external factors.

Pure pipes are executed only when Angular detects a pure change to the input value (i.e., the input reference or primitive value has changed).

import {Pipe, PipeTransform } from '@angular/core';

@Pipe ({

name: 'myPurePipe',

pure: true // This is the default behavior for pipes, but it's good practice to explicitly set it

})

export class MyPurePipe implements PipeTransform {

transform (value: any, args?: any): any {

// Perform transformation here

return transformedValue;

}

}

**Impure Pipes:**

Impure pipes, on the other hand, can have side effects and are executed on every change detection cycle.

They might be triggered more frequently than necessary, potentially impacting performance.

import {Pipe, PipeTransform } from '@angular/core';

@Pipe ({

name: 'timeAgo',

pure: false

})

export class TimeAgoPipe implements PipeTransform {

transform (completedAt: Date): string {

const elapsed = Date.now() - completedAt.getTime();

const seconds = Math.floor(elapsed / 1000);

const minutes = Math.floor(seconds / 60);

const hours = Math.floor(minutes / 60);

const days = Math.floor(hours / 24);

if (days > 0) {

return `${days} days ago`;

} else if (hours > 0) {

return `${hours} hours ago`;

} else if (minutes > 0) {

return `${minutes} minutes ago`;

} else {

return 'Just now';

}

}

}

**PipeExmpl with Argument**

import {Pipe, PipeTransform } from '@angular/core';

@Pipe ({

name: 'truncate'

})

export class TruncatePipe implements PipeTransform {

transform (value: string, length: number): string {

if (! value) return '';

if (value.length <= length) return value;

return value.substring(0, length) + '...';

}

}

<p> {{ longText | truncate: 50}} </p>

**Pipes Сhain in Angular**

<p [innerHTML]="input | getLength | mark"></p>

**Sync and Async pipes in Angular**

sync pipes are used for synchronous data transformation, while async pipes are used for handling asynchronous data streams, especially when dealing with Observables and Promises.

<p> {{(**date$ | async**) | date: 'shortDate'}} </p>

Yes, you can use both the sync and async pipes together in Angular. This can be useful in scenarios where you want to transform data synchronously before applying asynchronous operations, or vice versa.

**NOTE**:  
**Async pipe will perform three tasks**

1. It will subscribe to Observable or promises and return the latest value.
2. Whenever a new value is emitted, it marks the component to be checked.
3. That means angular will need to run change Detector () for that component in next cycle.
4. No need to take care about unsubscribe ()

**Is it possible to use sync and async pipes in Impure pipes?**

No, it's not possible to directly use both the sync and async pipes together in an impure pipe in Angular.

Impure pipes are designed to be synchronous and are meant to run synchronously on every change detection cycle.

The sync pipe operates synchronously, whereas the async pipe handles asynchronous operations.

However, you can achieve a similar effect indirectly.

You can create a custom impure pipe that internally uses both sync and async pipes.

This would require you to use an observable or a promise within the pipe and then apply synchronous transformations afterward.

transform (data: Observable<any>): any {

return data.pipe(

map ((value: any) => {

// Apply synchronous transformation

value = value.toUpperCase(); // Example synchronous transformation

return value;

})

);

}

**Decorators**

In Angular, decorators are functions that modify TypeScript classes, methods, properties, or parameters. They are used to add metadata to these elements, which Angular then uses to understand and process the class or its members appropriately. Decorators play a crucial role in defining and configuring various aspects of Angular applications, such as components, services, modules, and directives.

**Types of Decorators:**

* **Class decorators** like @Component, @NgModule, @Directives, @Pipe
* **Property decorator**s like @Input and @Output
* **Method decorators** like @HostListener
* **Parameter decorators** like @Injectable

There are several types of decorators commonly used in Angular:

* **@Component**: This decorator is used to mark a class as an Angular component. It provides metadata such as the HTML template, styles, and selector for the component.
* **@Directive**: Directives are like components but are used to add behaviour to existing DOM elements or components. This decorator marks a class as an Angular directive.
* **@Pipe**: Pipes are used to transform data in templates. The @Pipe decorator marks a class as an Angular pipe, allowing it to be used in template expressions to transform data.
* **@Injectable**: This decorator marks a class as a provider of a service, allowing Angular to inject it into other components or services.
* **@NgModule**: NgModule is used to define a module in an Angular application. This decorator provides metadata such as declarations, imports, exports, and providers for the module.
* **@Input**: Used to define an input property for a component. This decorator marks a class property as an input property, allowing the parent component to bind data to it.
* **@Output**: Like @Input, @Output is used to define an output property for a component. It marks a class property as an output property, allowing the component to emit events to its parent component.
* **@HostListener**: This decorator is used to listen to events on the host element of a directive or component.
* **@HostBinding**: Used to bind a directive or component property to a property on its host element.

**Angular Directive**

Directives are a powerful feature that allows you to extend the behavior of HTML elements.

They can be used to create reusable components, add behavior to elements, manipulate the DOM, and more.

There are three types of directives in Angular:

* **Component Directives**:
  + Components are directives with a template. They are the most common type of directive used in Angular applications.
  + Components encapsulate a piece of UI along with its associated behavior and state. They can have their own templates, styles, and lifecycle hooks.
  + Components are typically used to create reusable UI components, such as buttons, forms, menus, and more.
* **Attribute Directives**:
  + Attribute directives are **used to change the behavior or appearance of an existing HTML/DOM element**.
  + They are applied as attributes to HTML elements and modify the element's behavior or appearance based on the specified logic.
  + Examples of attribute directives include **ngModel, ngClass, ngStyle,** and custom attribute directives created by developers.
* **Structural Directives**:
  + Structural directives are **used to add or remove elements from the DOM based on certain conditions**.
  + They are applied as attributes to HTML elements and change the structure of the DOM based on the provided conditions.
  + Examples of structural directives include **ngFor, ngIf, ngSwitch,** and custom structural directives created by developers.

**How do you create a Custom Angular directive?**

To create a custom Angular directive, you need to define a directive class and then apply the @Directive decorator to that class.

**Example:** Let's say you want to create a custom directive that changes the background color of an element when the user hovers over it.

* First, create a new TypeScript file for your custom directive, e.g., highlight.directive.ts. Inside this file, import the necessary modules and define the directive class:

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

@Directive ({selector: '[***appHighlight***]'})

export class HighlightDirective {

@Input('appHighlight') highlightColor: string;

constructor (private el: ElementRef) {}

@HostListener('mouseenter') onMouseEnter() {

this.highlight(this.highlightColor || 'yellow');

}

@HostListener('mouseleave') onMouseLeave() {

this.highlight(null);

}

private highlight (color: string) {

this.el.nativeElement.style.backgroundColor = color;

}

}

* Next, you need to declare the directive in the appropriate module. In this case, let's assume it's the app.module.ts file:

import { ***HighlightDirective*** } from './highlight.directive';

@NgModule ({

imports:[ BrowserModule, FormsModule ],

declarations: [ AppComponent, HelloComponent, ***HighlightDirective*** ],

bootstrap: [ AppComponent ],

providers: [

{provide: COMPILER\_OPTIONS, useValue: {}, multi: true},

{provide: CompilerFactory, useClass: JitCompilerFactory, deps: [COMPILER\_OPTIONS]},

{provide: Compiler, useFactory: createCompiler, deps: [CompilerFactory]}

]

})

export class AppModule { }

* Finally, you can use the custom directive in your component's HTML template:

*<p* **appHighlight***="lightblue">Hover over me to change the background color! </p>*

**Resolution Modifiers:**

**Used to modify the default resolution modifier**

**What happens when we use the script tag within a template?**

In Angular, using the <script> tag within a template is generally not recommended, and it may not behave as expected. This is because Angular templates are primarily intended for defining the structure and behaviour of the user interface, and including JavaScript code directly within templates can lead to several issues:

Or

When a script tag is used within a template, the browser treats it as part of the HTML content and attempts to execute it. However, Angular's template compiler does not process or execute scripts within templates. Instead, scripts should be placed in separate JavaScript files and included using the appropriate Angular mechanisms, such as component logic or Angular modules.

Instead of using the **<script>** tag within templates, it is recommended to follow Angular's best practices for implementing dynamic behaviour:

* Define component methods in the component class and bind them to HTML elements using Angular's event binding syntax ((event)="method ()").
* Utilize Angular directives and pipes for manipulating data and rendering dynamic content.
* Leverage Angular services for implementing business logic and performing side effects.
* If necessary, use Angular's platform-agnostic renderer API to interact with the DOM programmatically from within component code.

**What happen,**

**When you attempt to get an element by its ID/Class using JavaScript and**

**the specified ID/Class does not exist in the HTML document?????**

the JavaScript method will return null. This means that the variable you assigned the result will hold the value null.

It's important to handle this case in your JavaScript code to avoid errors when attempting to manipulate or access properties of the returned element. You can check if the element is null before performing any further operations on it.

**Why do we need to avoid using 'any’ Keyword in Typescript?**

* TypeScript is a statically typed superset of JavaScript that allows you to specify types for variables. Using any essentially disables TypeScript's type checking for that variable, which can lead to runtime errors that could have been caught at compile time.
* When you specify the type of a variable explicitly, it makes the code more readable and understandable, both for yourself and for other developers who may work on the codebase later.

**What is the use of Generics in TS?**

Generics in TypeScript allow you to create reusable components, functions, and classes that can work with a variety of data types while providing type safety.

**Example:**

function getFirstElement<T>(arr: T[]): T | undefined {

return arr.length > 0 ? arr[0] : undefined;

}

// Usage examples

const stringArray: string[] = ['apple', 'banana', 'orange'];

const firstString: string | undefined = getFirstElement(stringArray);

console.log(firstString); // Output: apple

const numberArray: number[] = [1, 2, 3, 4, 5];

const firstNumber: number | undefined = getFirstElement(numberArray);

console.log(firstNumber); // Output: 1

const emptyArray: any[] = [];

const firstElement: any | undefined = getFirstElement(emptyArray);

console.log(firstElement); // Output: undefined

<https://testbook.com/interview/angular-coding-interview-questions>

**How can you implement two-way data binding in Angular?**

**How can you create a service in Angular and inject it into a component?**

**How do you handle form validation in Angular?**

**How do you implement lazy loading of modules in Angular?**

**How do you handle HTTP requests in Angular?**

**How do you implement routing in an Angular application?**

**How do you manage the state in an Angular application using NgRx?**

**How do you create and use observables in Angular with RxJS?**

**Difference between**

**@ViewChild and @ViewChildren**

**@ViewChild**:

* @ViewChild decorator allows a parent component to query for the first occurrence of a child component or element matching the specified selector.
* It is used to access a single child component or element from the parent component's template.
* The selected child component or element is available as a ViewChild property in the parent component class.

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

import { ChildComponent } from './child.component';

@Component ({

selector: 'parent-component',

template: '<child-component></child-component>'

})

export class ParentComponent {

@ViewChild (ChildComponent) childComponent: ChildComponent;

}

**@ViewChildren**:

* @ViewChildren decorator allows a parent component to query for multiple occurrences of child components or elements matching the specified selector.
* It is used to access multiple child components or elements from the parent component's template.
* The selected child components or elements are available as a QueryList in the parent component class.

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

import { ChildComponent } from './child.component';

@Component ({

selector: 'parent-component',

template: '**<child-component></child-component><child-component></child-component>**'

})

export class ParentComponent {

@ViewChildren (ChildComponent) childComponents: QueryList<ChildComponent>;

}

**Design Patterns**

* **Creational Patterns**: Singleton, Factory Method, Abstract Factory, Builder, Prototype.
* **Structural Patterns**: Adapter, Decorator, Facade, Proxy, Bridge.
* **Behavioral Patterns**: Observer, Strategy, Command, Iterator, State.

**KISS (Keep It Simple, Stupid) and DRY (Don't Repeat Yourself) Principle**

The KISS (Keep It Simple, Stupid) principle and the DRY (Don't Repeat Yourself) principle are two fundamental concepts in software development aimed at improving code **quality, maintainability, and readability**.

* **KISS Principle**:
  + The KISS principle advocates for **simplicity and clarity in design and implementation**. It suggests that systems and designs should be kept as simple as possible, avoiding unnecessary complexity and over-engineering.
  + Key aspects of the KISS principle include:
    - **Simplicity**: Designs and implementations should be straightforward and easy to understand for developers.
    - **Clarity**: Code should be clear and concise, with minimal unnecessary complexity or abstraction.
    - **Avoiding Over-Engineering**: Solutions should be kept simple and focused on solving the immediate problem without adding unnecessary features or complexity.
  + Following the KISS principle helps in reducing the risk of bugs, improving maintainability, and enhancing the overall readability of the codebase.
* **DRY Principle**:
  + The DRY principle emphasizes the **importance of code reusability and avoiding duplication.** It suggests that each piece of knowledge or logic in a system should have a single, unambiguous representation within the codebase.
  + Key aspects of the DRY principle include:
    - **Code Reuse**: Duplication of code should be minimized, and reusable components, functions, or modules should be extracted and shared across the codebase.
    - **Single Source of Truth**: Each piece of knowledge or logic should be represented in a single place within the codebase to avoid inconsistencies and maintenance issues.
    - **Abstraction**: Abstract common patterns or functionalities into reusable abstractions to promote code reuse and maintainability.
  + Following the DRY principle helps in reducing redundancy, improving maintainability, and facilitating changes and updates to the codebase without having to update multiple places where the same logic or information is duplicated.

In summary,

The KISS principle advocates for **simplicity and clarity in design and implementation,**

The DRY principle emphasizes code **reusability and avoiding duplication.**

By adhering to these principles, developers can write cleaner, more maintainable, and efficient code.

**Attribute Binding**:

* **Definition**: Attribute binding is a technique used to bind a component property to an HTML attribute. It allows you to set HTML attributes dynamically based on component properties.
* **Use Case**: Attribute binding is used when you need to set standard **HTML attributes or non-DOM properties** (e.g., aria- attributes) based on component properties.
* **Syntax**: Attribute binding is denoted by interpolation (**{{}}**) or without any special syntax.

we may need to bind HTML element attributes. For example, colspan, area, height, width, placeholder, value, etc. There are a number of HTML attributes. HTML does not have any corresponding DOM (Document Object Model) properties. So, in this case, we need to bind to HTML element attributes.

1. @Component ({
2. selector: ‘app-example’,
3. templateUrl: ‘./example.component.html’,
4. styleUrls: [‘./example.component.css’]
5. })
6. **export** **class** ExampleComponent {
8. imageHeight:number=150;
9. imageWidth:number=150;
10. imagePath:string=” assets/Images/farhan.png”;
12. }

**Property Binding:**

* **Definition**: Property binding is a one-way data binding technique that allows you to bind a component property (JavaScript property) to an **HTML element property (DOM property)**.
* **Use Case**: Property binding is primarily used to bind dynamic values from the component class to DOM properties, such as element properties, events, and component properties.
* **Syntax**: Property binding is denoted by square brackets ([]) surrounding the property to bind.

<button [disabled]="isDisabled">Click me</button>

**Guards In Angular**

**What is a Guard?**

Think of a guard as a security checkpoint in an airport. It either allows or denies access based on certain conditions.

In Angular, a guard is essentially a script that runs before certain stages of route navigation.

It decides whether to proceed with the navigation or redirect it.

The primary types of guards in Angular are:

* CanActivate
* CanActivateChild
* CanDeactivate
* Resolve
* CanLoad

Guards in Angular provide a robust way to control navigation in your application. With various types of guards available, you can handle a variety of conditions: check user authentication, warn about unsaved changes, or even pre-fetch data before completing navigation. As a developer, understanding guards is key to ensuring both the security and data integrity of the Angular apps.

**CanActivate**

The CanActivate guard checks whether a route can be activated. This is useful for protecting routes that should not be accessible unless certain conditions are met, like user authentication.

**Use Case: User Authentication**

Imagine you have an admin dashboard that should only be accessible to authenticated users. A CanActivate guard can check if the user is logged in before navigating to the admin page. If the user is not authenticated, the guard redirects them to a login page.

ng generate guard auth

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

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

@Injectable ({

providedIn: 'root'

})

export class AuthGuard implements CanActivate {

constructor (private router: Router) {}

canActivate(): boolean {

if (/\* your authentication condition \*/) {

return true;

} else {

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

return false;

}

}

}

import {AuthGuard} from './auth.guard';

const routes: Routes = [

{path**: 'protected'**, component: ProtectedComponent, canActivate: [AuthGuard] },

// ...

];

**CanActivateChild**

Similar to CanActivate, but it works on child routes. Useful for feature modules with multiple child routes.

**Use Case: Feature Access Control**

Suppose your application has a settings page with multiple child routes like “Profile Settings”, “Account Settings”, and “Privacy Settings”. You can use a CanActivateChild guard to ensure that only users with the appropriate permissions can access these child routes.

const childRoutes: Routes = [

{path**: 'child'**, component: ChildComponent, canActivateChild: [AuthGuard] },

// ...

];

**CanDeactivate**

This guard works when you're navigating away from a component. It is often used to warn the user about unsaved changes.

**Use Case: Unsaved Changes**

You have a form page for editing profile information. If a user makes changes but tries to navigate away without saving, a CanDeactivate guard can prompt them to either save the changes or confirm that they want to leave without saving.

export interface CanComponentDeactivate {

canDeactivate: () => boolean;

}

**Resolve**

A Resolve guard fetches data before navigation completes. The data can be used to populate route parameters.

**Use Case: Data Preloading**

In an e-commerce application, you have a product details page. A Resolve guard can pre-fetch the product details before the route is activated, ensuring that the user doesn't see an empty or partially loaded page.

@Injectable ({

providedIn: 'root'

})

export class DataResolver implements Resolve<Data> {

resolve (route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Data {

// Fetch your data here

}

}

**Can Load**

Can Load checks if a module should be lazy-loaded or not. This is important if you want to prevent unauthorized users from downloading parts of your application.

**Use Case: Feature Gating**

Your application has a premium feature module that should only be accessible (and thus downloaded) by premium users. A CanLoad guard can prevent unauthorized users from downloading this module, saving bandwidth and enhancing security.

const routes: Routes = [

{

path: 'feature',

loadChildren: () => import('./feature/feature.module'). then(m => m.FeatureModule),

canLoad: [AuthGuard]

},

// ...

];

**How to make a component or module lazy loading in angular?**

In Angular, lazy loading a component or module involves loading it asynchronously when it's needed, typically triggered by the user navigating to a certain route. This helps improve the initial loading time of the application by only loading the necessary code upfront and loading additional code as needed.

ng generate module lazy-module --route lazy --module app.module

This command generates a module named **LazyModule** with a route named **lazy** and adds it to the **AppModule** imports.

* First, create a feature module that contains the component or components you want to lazy load. This module should be self-contained and should not be imported into the main **AppModule**.
* Set up routes for the lazy-loaded module in your routing configuration. These routes will specify the path to the module and component that should be loaded lazily.
* In the route configuration, use the **loadChildren** property to specify the path to the module file using a string. Angular will then dynamically load that module when the corresponding route is accessed.

// app-routing.module.ts

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

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

const routes: Routes = [

{path: 'lazy', **loadChildren**: () => import('./lazy-module/lazy-module.module'). then(m => m.LazyModuleModule) },

{path: '\*\*', redirectTo: '/home'} // Handle invalid routes

];

@NgModule ({

imports: [RouterModule.forRoot(routes)],

exports: [RouterModule]

})

export class AppRoutingModule { }

When the user navigates to the specified route (e.g., **/lazy**), Angular will asynchronously load the module defined in the **loadChildren** property.

**How do you implement dynamic components in Angular?**

Dynamic components can be created and inserted into the DOM at runtime using **ComponentFactoryResolver** and **ViewContainerRef**.

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

@Component({

selector: 'app-dynamic',

template: `<ng-template #container></ng-template>`

})

export class DynamicComponent {

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

constructor(private resolver: ComponentFactoryResolver) {}

loadComponent() {

import('./lazy/lazy.component').then(({ LazyComponent }) => {

const factory = this.resolver.resolveComponentFactory(LazyComponent);

this.container.createComponent(factory);

});

}

}

// Lazy component (lazy.component.ts)

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

@Component({

selector: 'app-lazy',

template: `<p>Lazy Loaded Component</p>`

})

export class LazyComponent {}

**NgZone**

In Angular, **NgZone** is a service that helps manage the execution of asynchronous tasks and ensures that UI updates occur within the Angular zone. The zone concept in Angular is related to Angular's change detection mechanism and provides a context for tracking and managing asynchronous operations.

Here's what **NgZone** does and why it's important:

**Execution Context**:

* + **NgZone** provides an execution context, known as the Angular zone, which is responsible for triggering change detection and updating the UI.
  + When code runs within the Angular zone, Angular is aware of its execution and can detect changes to the application state and update the UI accordingly.

**Asynchronous Operations**:

* + Angular applications often involve asynchronous operations such as setTimeout, setInterval, XMLHttpRequests, or callbacks from third-party libraries.
  + These asynchronous operations can occur outside of Angular's control and may not trigger change detection automatically.
  + **NgZone** helps manage these asynchronous operations by intercepting them and ensuring that change detection is triggered when necessary.

**Zone.js Integration**:

* + **NgZone** is built on top of Zone.js, a library that enables **monkey-patching** of asynchronous operations in JavaScript.
  + Zone.js intercepts asynchronous tasks and runs them within the appropriate zone context, allowing Angular to track their execution and trigger change detection as needed.

**Zone-Aware APIs**:

* + Angular provides a set of zone-aware APIs for common asynchronous operations, such as **setTimeout**, **setInterval**, **requestAnimationFrame**, and **XMLHttpRequest**.
  + These APIs are patched by Zone.js to automatically trigger change detection when they are used within Angular's zone.

**Performance Optimization**:

* + By managing asynchronous tasks and change detection, **NgZone** helps optimize the performance of Angular applications and ensures that UI updates are efficient and responsive.

In summary, **NgZone** is a critical part of Angular's change detection mechanism and provides a way to manage asynchronous tasks within the Angular zone. It ensures that UI updates occur in response to changes in application state and helps optimize the performance and responsiveness of Angular applications. Understanding **NgZone** and how it interacts with asynchronous operations is essential for building reliable and efficient Angular applications.

**How To Transfer Data In-between Parent and Child?**

**Input Binding**: Use **@Input ()** decorator to pass data from the parent component to the child component through input properties.

**ViewChild**: Use **ViewChild** decorator to get a reference to the child component in the parent component, and then access its properties or methods directly.

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

import { ChildComponent } from './child.component';

@Component ({

selector: 'app-parent',

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

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

})

export class ParentComponent implements AfterViewInit {

@ViewChild (ChildComponent) childComponent: ChildComponent;

parentData: any = 'Data from parent';

ngAfterViewInit() {

this.childComponent.dataFromParent(this.parentData);

}

}

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

@Component ({

selector: 'app-child',

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

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

})

export class ChildComponent {

dataFromParent(data: any) {

console.log(data);

}

}

**Service**: Use a service to share data between components. Inject the service into both the parent and child components, and then use it to exchange data.

**How to pass and retrieve ID in routings/navigation?**

**Component A -> Component B**

<! -- component-a.component.html -->

<a [routerLink]="['/component-b', id]">Go to Component B</a>

**Configuring the Route in AppRoutingModule**:

const routes: Routes = [

{path: '', redirectTo: '/component-a', pathMatch: 'full'},

{path: 'component-a', component: ComponentAComponent },

{path: '**component-b/:id**', component: ComponentBComponent }

];

In Component B, **retrieve the ID from the route parameters using ActivatedRoute**.

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

…..

id: string;

constructor (private route: ActivatedRoute) {}

ngOnInit(): void {

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

this.id = params['id'];

});

**How will you check the size of each module?**

**Angular CLI**:

* + The Angular CLI provides the **ng build** command, which compiles the application and generates output bundles.
  + You can use the **--stats-json** option to generate a JSON file containing build statistics, including the size of each module.
  + Run the following command to generate the statistics JSON file:

ng build --stats-json

* + After the build process is complete, you'll find a **stats.json** file in the **dist** directory of your Angular project.

How many times is each module taking?

How will you check the size of each module?

**Npm packager-Web pack analyzer-it will create a structure of the DOM**

**What is CORS (Cross Origin Resources sharing) error?**  
When we are working in xyz domain (LocalHost) and tried to get data from abc Domain (google.com) now this will treat it as Cros error.   
We can handle this from backend team by enabling for xyz domai request.  
We can also handle in Front end by cross method in Node.JS

**Dump component (PresentationComponent/View Components)**

read only data, Get the data from the parent component and show it as it is in DOM. and works like get the data from the @Input decorator and emits the dta to Parent through @Output decorators.

They do not have any logic to perform the Business logic.

these are resuable and stateless.

Example: Button/inputFiled/RadioButtons

**Smart component (Container Components)**

get the data from api, pass it to child/dump component and dump component return the value and it will display in smart component only.

Managing the application state and Business logic

they often fetch the data from services

They do not have any bother about data presentation.

**bind(), call(), and apply()**  
In JavaScript and TypeScript, bind(), call(), and apply() are methods that are used to control **this** context within functions. They are commonly used for function invocation and handling **this** keyword, which is especially useful when dealing with object-oriented programming and callback functions.

### **bind()**

The bind () method creates a new function that, when called, has its this keyword set to the provided value. It also allows you to pre-set one or more arguments for the function. The new function can be invoked later.

let obj = {

x: 42

};

function printX() {

console.log(this.x);

}

let boundPrintX = printX.bind(obj);

boundPrintX(); // 42

### **call()**

The call() method calls a function with a given this value and arguments provided individually. It allows you to invoke the function immediately with a specified this context.

let obj = {

x: 42

};

function printX() {

console.log(this.x);

}

printX.call(obj); // 42

### **apply()**

The apply() method is like call(), but it takes an array of arguments instead of individual arguments. This is useful when you want to pass an array of arguments to a function.

let obj = {

x: 42

};

function printX(prefix, suffix) {

console.log(prefix + this.x + suffix);

}

printX.apply(obj, ['Value: ', '!']); // Value: 42!

**Invocation**:

* bind(): **Returns a new function, doesn't invoke the function immediately**.
* call(): **Invokes the function immediately, taking arguments individually**.
* apply(): **Invokes the function immediately, taking arguments as an array**.

**Use Case**:

* bind(): Useful for creating a new function with a specific **this** value, often used for event handlers and callbacks.
* call(): Useful for invoking functions immediately with a specific **this** value and individual arguments.
* apply(): Useful for invoking functions immediately with a specific **this** value and arguments as an array, often used with functions that accept variable arguments, such as Math.max.

// Example Combining All Three

let obj = {

x: 42

};

function printInfo(y, z) {

console.log(this.x + y + z);

}

// bind() example

let boundPrintInfo = printInfo.bind(obj, 1);

boundPrintInfo(2); // 45 (42 + 1 + 2)

// call() example

printInfo.call(obj, 1, 2); // 45 (42 + 1 + 2)

// apply() example

printInfo.apply(obj, [1, 2]); // 45 (42 + 1 + 2)

**What is Event Bubbling?**

When elements receive events, such **events propagate to their parents and ancestors upward in the DOM tree**. This is the concept of **Event Bubbling**, and it allows parent elements to handle events that occur on their children's elements.

Event objects also have the **stopPropagation()** method which you can use to stop the bubbling of an event. This is useful in cases where you want an element to receive a click event only when it is clicked and not when any of its children's elements are clicked.

const body = document.getElementsByTagName("body") [0]

const div = document.getElementsByTagName("div") [0]

const span = document.getElementsByTagName("span") [0]

const button = document.getElementsByTagName("button") [0]

**body**.addEventListener('click', () => {

console.log ("body was clicked")

})

**div**.addEventListener('click', () => {

console.log ("div was clicked")

})

**span**.addEventListener('click', () => {

console.log ("span was clicked")

})

**button**.addEventListener('click', () => {

console.log ("button was clicked")

})

Output:

button was clicked

span was clicked

div was clicked

body was clicked

**button**.addEventListener('**click**', (event) => {

**event.stopPropagation().\** // **event will not trigger above this method block**

console.log ("button was clicked")

})

OutPut

button was clicked

**div**.addEventListener('click', () => {

**event.stopPropagation().**  // event will not trigger above this method block

console.log ("div was clicked")

})

**span**.addEventListener('click', () => {

console.log ("span was clicked")

})

**button**.addEventListener('click', () => {

console.log ("button was clicked")

})

outPut:

button was clicked

span was clicked

div was clicked

**Hoisting**

Hoisting is a JavaScript mechanism **where variable and function declarations are moved to the top of their containing scope during the compilation phase, before the code is executed**.

This means that regardless of where variables and functions are declared within their scope, they are treated as if they are declared at the top.

**Variable Hoisting**:

When variables are declared using var, they are hoisted to the top of their containing function or global scope, but their initializations remain in place.

If we type code like below:

console.log(x); // Output: undefined

var x = 10;

console.log(x); // Output: 10

Interpreted code will look like this;

var x;

console.log(x); // Output: undefined

x = 10;

console.log(x); // Output: 10

**Function Hoisting**:

* Function declarations, including named functions and function expressions declared with function, are also hoisted to the top of their containing scope. This means that they can be invoked before they are declared in the code.

//typed code

foo (); // Output: "Hello, world!"

function foo () {

console.log ("Hello, world!");

}

Interpreted code:

function foo () {

console.log ("Hello, world!");

}

foo (); // Output: "Hello, world!"

Note:

However, it's important to note that function expressions declared with const, let, or var, as well as arrow functions, are not hoisted in the same way as function declarations. **Only the variable declaration (not its assignment/initialization) is hoisted.**

bar (); // Type Error: bar is not a function

var bar = function () {

console.log ("Hello, world!");

};

**SOLID Principles:**

1. Single Responsibility Principle (S)
2. Open-Closed Principle (O)
3. Liskov Substitution Principle (L)
4. Interface Segregation Principle (I)
5. Dependency Inversion Principle (D)

### **1. Single Responsibility Principle (SRP)**

Adhering to the SRP in Angular ensures that each class or component takes on only a single responsibility. For instance, an AuthService should be responsible exclusively for authentication. This fosters a clear separation of concerns and facilitates maintenance and testing.

### **2. Open/Closed Principle (OCP)**

The OCP encourages the extensibility of classes without modifying them. An Angular LoggerService could implement various logger strategies, which can be swapped by Dependency Injection, without altering the class itself. This boosts the flexibility and reusability of code.

### **3. Liskov Substitution Principle (LSP)**

The LSP ensures that subclasses can extend the functionality of their parent classes without affecting existing functionalities. In Angular, an ExtendedService that inherits from a BaseService can provide additional functions while preserving the basic functionality.

### **4. Interface Segregation Principle (ISP)**

In Angular, the ISP requires the division of large interfaces into smaller, more specific ones. This prevents classes from being overloaded with methods they don't need. An example would be a UserService split into specific interfaces like UserAuthentication and UserDataManagement.

### **5. Dependency Inversion Principle (DIP)**

The DIP dictates that dependencies should be based on abstractions and not on concretizations. A BookComponent should depend on an abstract BookService rather than a specific implementation, reducing coupling and enhancing testability.

**ES6 (ECMAScript 2015) features**

* **let and const**:
  + let and const provide block-scoped variable declarations. let allows for reassignment, while const creates variables whose values cannot be re-assigned once initialized.
* **Arrow Functions**:
  + Arrow functions provide a more concise syntax for writing function expressions. They have implicit returns when not using braces {}, and they do not have their own this context.
* **Template Literals**:
  + Template literals allow for easier string interpolation and multi-line strings using backticks (``).
* **Destructuring Assignment**:
  + Destructuring allows for extracting values from arrays or properties from objects into distinct variables, making it easier to work with complex data structures.
* **Default Parameters**:
  + Default parameter values can be specified for function parameters, which are used when no argument or an undefined argument is passed.
* **Rest and Spread Operators**:
  + The rest (...) and spread (...) operators allow for working with variable numbers of arguments, both in function definitions and when working with arrays or objects.
* **Object Literal Enhancements**:
  + Object literal shorthand syntax allows for shorter syntax when defining object properties using variable names.
  + Computed property names allow for using expressions as property names in object literals.
* **Classes**:
  + Classes provide a more natural and simpler way to create constructor functions and prototype-based inheritance in JavaScript.
* **Modules**:
  + ES6 introduced a standardized module system using import and export statements, allowing for better code organization and encapsulation.
* **Promises**:
  + Promises to provide a cleaner syntax for handling asynchronous operations compared to callbacks, with built-in support for chaining and error handling.
* **Symbol Data Type**:
  + Symbols are unique and immutable data types introduced to JavaScript, often used as object property keys to avoid unintended property collisions.
* **Iterators and Generators**:
  + Iterators and generators provide a more powerful way to traverse and work with collections and sequences of data in JavaScript.
* **Map and Set Data Structures**:
  + ES6 introduced built-in Map and Set data structures, providing alternatives to plain objects and arrays for storing unique key-value pairs and sets of values.
* **Async/Await**:
  + Async functions and the await keyword provide a cleaner and more synchronous-looking syntax for working with asynchronous code, built on top of promises.
* **Object.assign()**:
  + Object.assign() is used to copy the values of all enumerable own properties from one or more source objects to a target object.

**Anonymous function:**

An anonymous function in JavaScript is a function that is defined without a name.

Instead of declaring a named function using the function keyword followed by an **identifier**, an anonymous function is defined using the function keyword without an identifier.

**Syntax:**

// Anonymous function syntax

function(parameters)

{// Function body

}

**Function Expression**:

* An anonymous function can be defined as part of a function expression, where it's assigned to a variable or passed as an argument to another function.

// Anonymous function assigned to a variable

const greet = function(name) {

console.log (`Hello, ${name}! `);

};

greet('John'); // Output: Hello, John!

**Immediately Invoked Function Expression (IIFE)**:

* An anonymous function can be declared and immediately invoked by wrapping it in parentheses and adding () at the end.

// Immediately Invoked Function Expression (IIFE)

(function () {

console.log ('This is an IIFE');

})();

**Arrow Function**:

* Arrow functions provide a more concise syntax for defining anonymous functions, especially for single-expression functions.

// Arrow function

const square = (x) => {

return x \* x;

};

console.log(square (5)); // Output: 25

**Callback Functions**:

* Anonymous functions are commonly used as callback functions, where they're passed as arguments to other functions to be executed later.

// Callback function example

const numbers = [1, 2, 3, 4, 5];

const squared = numbers.map(function(x) {

return x \* x;

});

console.log(squared); // Output: [1, 4, 9, 16, 25]

**Currying in JavaScript?**

At its core, currying is a functional programming technique that involves breaking down a function that takes multiple arguments into a series of functions that take one argument each. This creates a chain of functions, where each function returns to another function until the result is achieved.

function calculateVolume(length) {

return function (breadth) {

return function (height) {

return length \* breadth \* height;

}

}

}

console.log(calculateVolume(4)(5)(6));

**Infinite Currying in JavaScript**

**Nothing but recursion function;**

**Initially:**

|  |  |
| --- | --- |
| function sum(a) { |  |
|  | return function(b){ |
|  | return function(c){ |
|  | return a+b+c; |
|  | } |
|  | } |
|  | } |
|  | console.log (sum (1)(2)(3) ()); |

**Later:**

function sum(a) {

return function(b) {

if (! b) {

return a;

}

return sum(a+b);

}

}

console.log (sum (1)(2)(3)(4)(5)(6) ()); //21

**Inheritance**

**Using Component Inheritance**:

* Angular supports component inheritance, where one component can extend another component to inherit its behavior and template.
* This is achieved by creating a base component with common functionality and then creating child components that extend the base component using TypeScript's extends keyword.

**Using Composition**:

* Instead of inheritance, you can achieve code reuse through composition by creating reusable services or utility classes that can be injected into multiple components.
* Components can use these shared services or utility classes to encapsulate common functionality.

**Using Directive Inheritance**:

* Like component inheritance, you can create directive inheritance where one directive extends to another to inherit its behavior and functionality.
* Directives can be used to add behavior to DOM elements, and inheritance can be used to create reusable sets of directives.

**Encapsulation**

Angular provides several mechanisms for encapsulation:

* **Component Encapsulation**:
  + Angular components encapsulate their templates, styles, and behavior into reusable units.
  + By default, Angular components have view encapsulation enabled, which means styles defined in component-specific CSS files are scoped to that component's view only and do not leak to other parts of the application.
  + View encapsulation modes include Emulated (default), None, and ShadowDom.
* **Input and Output Properties**:
  + Components can define input properties using @Input () decorators and output properties using @Output () decorators.
  + Input properties allow parent components to pass data to child components, while output properties allow child components to emit events to parent components.
  + This facilitates communication between components while maintaining encapsulation, as the internals of the child component are shielded from direct manipulation by the parent component.
* **Services and Dependency Injection**:
  + Angular services encapsulate shared logic and data that can be reused across multiple components.
  + Services can be injected into components using Angular's dependency injection system, enabling components to access shared functionality without directly coupling their implementations.
  + This promotes separation of concerns and encapsulation by allowing components to focus on presentation logic while delegating business logic to services.
* **Private Members**:
  + TypeScript provides support for defining private members within classes using the private modifier.
  + Private members are accessible only within the class in which they are defined, providing encapsulation by hiding implementation details from external code.

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

@Component ({

selector: 'app-user',

template: `<h2> {{user.name}} </h2>`

})

export class UserComponent {

@Input () user: {name: string};

private isLoggedIn: boolean;

constructor () {

this.isLoggedIn = false;

}

login (username: string, password: string) {

// Implementation details of login functionality

this.isLoggedIn = true;

}

logout () {

// Implementation details of logout functionality

this.isLoggedIn = false;

}

}

In Above example:

* The UserComponent encapsulates the presentation logic for displaying user information.
* Input property user allows external components to pass user data to the UserComponent.
* Private member isLoggedIn encapsulates the state of the user's login status within the UserComponent, preventing direct access from external code.
* Methods login () and logout () encapsulate the behavior for logging in and logging out, respectively, shielding the implementation details from external components.

**Angular14 feature:**

**Angular provides better error catching and diagnostics for your templates, helping you identify issues faster:**

**<div>Product Name: {{ product.namee }}</div>**

Error:

* You might see an error message in the browser console, but it wouldn't necessarily tell you the exact line or the reason for the error.
* The message could be something generic like "Error evaluating expression '{{ product.namee }}' in component SomeComponent"

Current error Error:

Property 'namee' does not exist on type 'Product' at line 5 in some-component.html

**Optional Injectors** (Angular 14+):

Streamlines dependency injection by letting components be created without explicitly providing an injector.

This reduces boilerplate code.

// Example (pre-angular 14)

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

@Component ({

selector: 'app-some-component',

template: `<h1>Some Component</h1>`

})

export class SomeComponent {

constructor (**private injector: Injector**) {

const someService = this.injector.get(SomeService);

// Use someService

}

}

// Example (Angular 14+)

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

@Component ({

selector: 'app-some-component',

template: `<h1>Some Component</h1>`

})

export class SomeComponent {

constructor (**@Optional () private someService: SomeService**) {

// someService might be null here

if (this.someService) {

// Use someService

} else {

// Handle the case where SomeService is not available

}

}

}

**Benefits:**

* **Reduced boilerplate code:** You don't need to explicitly handle injector creation or retrieval in the constructor.
* **Improved code readability:** Makes it clear that a dependency is optional.
* **Flexibility:** Allows you to handle scenarios where a dependency might not be available in all parts of your application.

Higher order function

Event bubling/ Mozilla developer // done

Subject and its types

Routings/ id passing and query passing

Decorators and annotation

If Dom element not exist, if we are consuming in ts file, what will happen????

View child and view children

callback function -

ES6 features

anonymous function

closures

hoisting

css3

HTML5

inheritance

RxJS

routing data transfer

Shadow shell/ don shell

**What is shallow copy and deep copy in JavaScript?**

**Shallow Copy**:

* Shallow copy creates a new object or array and copies the references of the original object's properties or elements. If the properties or elements are objects themselves, only their references are copied, not the actual objects.

// Original object

const originalObject = {a: 1, b: {c: 2}};

// Shallow copy

const shallowCopy = {...originalObject };

// Modifying a property of the shallow copy

shallowCopy.a = 10;

// Modifying a nested property of the shallow copy

shallowCopy.b.c = 20;

console.log(originalObject); // Output: { a: 1, b: { c: 20 } }

console.log(shallowCopy); // Output: { a: 10, b: { c: 20 } }

modifying the nested property b.c in the shallow copy also affects the original object because they both share the same reference to the nested object { c: 2 }.

**Deep Copy**:

* Deep copy creates a new object or array and recursively copies all properties or elements, including nested objects or arrays. This ensures that any changes made to the copied object or array do not affect the original.

// Original object

const originalObject = { a: 1, b: { c: 2 } };

// Deep copy

const deepCopy = JSON.parse(JSON.stringify(originalObject));

// Modifying a property of the deep copy

deepCopy.a = 10;

// Modifying a nested property of the deep copy

deepCopy.b.c = 20;

console.log(originalObject); // Output: { a: 1, b: { c: 2 } }

console.log(deepCopy); // Output: { a: 10, b: { c: 20 } }

In this example, modifying the nested property b.c in the deep copy does not affect the original object because they are completely separate copies.

**NOTE:**

This JSON string is passed to the JSON.parse() method which then transforms it into a JavaScript object.

This method is useful when the object is small and has serialisable properties. But if the object is very large and contains certain non-serialisable properties then there is a risk of data loss.

Especially if an object contains methods, then JSON.stringify() will fail as methods are non-serialisable.

There are better ways to a deep clone of which one is **Lodash** which allows cloning methods as well.

**Lodash To Deep Copy**

Lodash is a JavaScript library that provides multiple utility functions and one of the most used functions of the Lodash library is the cloneDeep() method. This method helps in the deep cloning of an object and clones the non-serializable properties which were a limitation in the JSON.stringify() approach.

const lodash = require('lodash');

let employee = {

eid: "E102",

ename: "Jack",

eaddress: "New York",

salary: 50000,

details: **function** () {

**return** "Employee Name: "

+ **this**.ename + "-->Salary: "

+ **this**.salary;

}

}

let deepCopy = **lodash.cloneDeep**(employee);

**Programming**

**Display the Character Count in string and ascending order?**

var data = 'asdfawedsf';

function charCount(str) {

const result = [...str]. reduce ((acc, value) => {

acc[value] = acc[value]? acc[value] + 1: 1;

return acc;

}, {});

return result;

}

console.log(charCount(data));

const newArray = replaceId(array, 3, 100);

function replaceId(array, 3, 100) {

array.map(value => {

if (old === replace) {

return {…value, id: newId };

}

return array;

});

}

**Remove Duplicate string/Character in string?**

function removeDuplicates(str) {

var setValue= new Set(str);

return [...setValue]. join('');

}

**Array Sorting**

const arr = [12, 44, 55, 1, 3, '**a**', '**dv**'];

l

// Sort the array numerically

arr.sort((a, b) => {

return a - b;

});

console.log(arr); // Output: [ 1, 3, 12, 44, 55, 'a', 'dv' ]

The strings 'a' and 'dv' are converted to NaN (Not-a-Number) when subtracted from numbers. When sorting, NaN values are treated as greater than any number, so they appear at the end of the sorted array.

const arr = ['banana', 'apple', 'orange'];

arr.sort();

// Output: ['apple', 'banana', 'orange']

By default, **the sort () method sorts array elements as strings and arranges them in lexicographic (alphabetical) order.**

const arr = [

{name: 'John', age: 30},

{name: 'Alice', age: 25},

{name: 'Bob', age: 35}

];

arr.sort((a, b) => a.age - b.age);

// Output: [{name: 'Alice', age: 25}, {name: 'John', age: 30}, {name: 'Bob', age: 35}]

const arr = ['apple', 'banana', 'orange'];

arr.sort((a, b) => {

if (a.length === b.length) {

return a.localeCompare(b); // Sort alphabetically if lengths are equal

}

return a.length - b.length; // Sort by length otherwise

});

// Output: ['apple', 'orange', 'banana']

const inputString = "hello";

const stringWithoutDuplicates = removeDuplicates(inputString);

console.log ("String without duplicates:", stringWithoutDuplicates);

**Add only Numbers in a string?**

var temp1= "vijay1234 kumar9876";

function sumofnumbers(temp1) {

const numbers= temp1.split(''). map (Number); //all string letters will store as NaN

console.log(numbers);

const result1= numbers.reduce((sum,value) => {

if(value){

sum = sum + value;

}

return sum;

},0);

return result1;

}

const result = sumofnumbers(temp1);

console.log(result);

**Print the Value in between array?**

function inbetween(array) {

for (let i=0; i<array.length; i++) {

const start= array[i];

const end = array[i+1];

for (let j=start+1; j<end; j++){

console.log(j); // 2,3,4,6,8

}

}

}

array= [1,5,7,9,10];

inbetween(array);

**Print Longest word of the array?**

function findLongestWord(array) {

let longestWord = "";

for (let i = 0; i < array.length; i++) {

if (array[i]. length > longestWord.length) {

longestWord = array[i];

}

}

return longestWord;

}

const wordsArray = ["apple", "banana", "kiwi", "strawberry", "pineapple"];

const longestWord = findLongestWord(wordsArray);

console.log ("The longest word is:", longestWord);

**Sort an Odd numbers in an array and Even Numbers should be in their index?**

const arr = [9, 8, 7, 6, 5, 4, 3, 2, 1, 0];

const sortedOdd = arr.filter(num => num % 2 !== 0).sort((a, b) => a - b);

let oddIndex = 0;

const result = arr.map(num => {

if (num % 2 !== 0) {

return sortedOdd[oddIndex++]

} else {

return num;

}

});

console.log(sortedOdd);

console.log(result); // Output: [1, 8, 3, 6, 5, 4, 7, 2, 9, 0]

**Reverse An Array**

function reverseArray(array){

const reverseArray = [];

for (let i=array. length-1; i>=0; i--) {

reverseArray.push(array[i])

}

return reverseArray;

}

let array= [1,5,7,9,10];

console.log(reverseArray(array)); // [ 10, 9, 7, 5, 1]

or

let array = [1,5,7,9,10];

array.reverse();

console.log(array); //[ 10, 9, 7, 5, 1]

**Reverse A String:**

function reverseString(str){

return str.split('').reverse().join('');

}

**OR**

function reverseString(str) {

let reverseStr = ''”;

for(let i=str. length-1; i>=0; i--){

reverseStr += str[i];

}

return reverseStr;

}

**Second Max Number in an Unsorted Array?**

function findSecondMax(arr) {

let max = -Infinity;

let secondMax = -Infinity; //It will give least negative value when comparing for the first time

for (let i = 0; i < arr.length; i++) {

if (arr[i] > max) {

secondMax = max;

max = arr[i];

} else if (arr[i] > secondMax && arr[i] !== max) {

secondMax = arr[i];

}

}

if (secondMax === -Infinity) {

return "There is no second maximum number";

} else {

return secondMax;

}

}

// Example usage:

const arr = [3, 1, 4, 5, 9, 2, 8];

console.log("Second maximum number:", findSecondMax(arr));

**Replace ID:**

const array = [

{ id: 1, name: 'Hari' },

{ id: 2, name: 'Hero' },

**{ id: 3, name: 'Sai' },**

{ id: 4, name: 'Arpit' }

];

function replaceId(array, oldId, newId) {

return array.map(obj => {

if (obj.id === oldId) {

return { ...obj, id: newId };

}

return obj;

});

}

const newArray = replaceId(array, 3, 100);

console.log(newArray);

OR  
const array = [

{ id: 1, name: 'Hari' },

{ id: 2, name: 'Hero' },

{ id: 3, name: 'Sai' },

{ id: 4, name: 'Arpit' }

];

// Find the index of the object with id equal to 3

const index = array.findIndex(obj => obj.id === 3);

// If the object is found, update its id property to 100

if (index !== -1) {

array[index].id = 100;

}

console.log(array);

**Input** [1,2,3,4,5,”A”,”b”,”c”]

**Output**. [2,4,6,8,10,a2,b2,c2]

Logic:

let array= [1,2,3,4,5,”A”,”b”,”c”];

let newArray =[];

array.forEach((element)=>{

if(typeof element === 'number'){

newArray.push(element\*2);

}else{

newArray.push(element+'2');

}

}

);

console.log(newArray);

**for...in loop**:

* It iterates over the keys (property names) of an object rather than its values.

const obj = { a: 1, b: 2, c: 3 };

for (const key in obj) {

console.log(key, obj[key]);

}

**for...of loop**:

* The **for...of** loop was introduced in ES6 and is used to iterate over iterable objects such as arrays, strings, maps, sets, etc.
* It iterates over the values of an iterable object rather than its keys.
* It provides a cleaner syntax compared to traditional **for** loops and is especially useful when dealing with arrays.

const array = [1, 2, 3, 4, 5];

for (const value of array) {

console.log(value);

}

**First letter of each word capitalised**

function capitalizeFirstLetter(sentence) {

return sentence.split(' ').map(word => word.charAt(0).toUpperCase() + word.slice(1)).join(' ');

}

let sentence = "this is a sample sentence.";

let capitalizedSentence = capitalizeFirstLetter(sentence);

console.log(capitalizedSentence);

**Anagrams : ("listen" and "silent" are anagrams of each)**

function anagrams(s1, s2){

return s1.split('').sort().join('') === s2.split('').sort().join('') ;

}

console.log(anagrams('vijay','Kumar') ); // true

console.log(anagrams('listen','silent') ); // false

const a = {

a1:10,

a2:20

}

a.a1=100;

console.log(a); //a1:100, a2: 20

// we can be able to modify property of an object even if it is a const Bz we can modify the existing property, but we can't add new propertyasync pic

0ppp

console.log (1 +"2" +"2"); //122

console.log (1 + +"2" +"2"); //32

console.log (1 + -"2" +"2"); //-13

console.log (+"1" + "1" +"2"); //112

console.log ("A" - "B" +"2"); //NaN2

console.log ("A" - "B" + 2); // NaN

console.log (3 \*\* '3'); 27 exponential values

**Higher Order Functions:**

A higher-order function is a function that takes one or more functions as arguments or returns a function as its result.

Or

It's a function that operates on other functions by either accepting them as arguments, returning them, or both.

**Case 1: higher-order function that takes another function as an argument**

function applyOperation(x, y, **operation**) {

return **operation (**x, y);

}

function add (x, y) {

return x + y;

}

function subtract (x, y) {

return x - y;

}

console.log (applyOperation(5, 3, add)); // Output: 8

console.log (applyOperation(5, 3, subtract)); // Output: 2

**Case 2: higher-order function that returns another function**

function createGreeter(greeting) {

return function(name) {

return `${greeting}, ${name}!`;

};

}

const sayHello = createGreeter('Hello');

console.log(sayHello('John')); // Output: Hello, John!

const sayGoodbye = createGreeter('Goodbye');

console.log(sayGoodbye('Jane')); // Output: Goodbye, Jane!

**Case 3: higher-order function that takes and returns another function**

function compose (func1, func2) {

return function(value) {

return func1(func2(value)); // (square (double (3)))

};

}

function double(x) {

return x \* 2; ￼-> 3\*2=6 ->return 6

}

function square(x) {

return x \* x; -> 6\*6 -> return 36

}

const doubleThenSquare = compose (square, double);

console.log(doubleThenSquare(3)); // Output: 36 (square (double (3)))

**There are various built in HOFs,**

**some of the most common ones are map (), filter () and reduce ()**

* Use **map** when you want to transform an array
* Use **filter** to select a subset of data from an array, and
* Use **reduces** when you want to return a single value as a result.

**MAP ():**

**Add 10 to every element in a array**

const arr = [1, 2, 3, 4, 5];

const output = **arr.map**((num) => num += 10)

console.log(arr); // [1, 2, 3, 4, 5]

console.log(output); // [11, 12, 13, 14, 15]

**Suppose we only want their first and last name**

const users = [

{firstName: 'John', lastName: 'Doe', age: 25},

{firstName: 'Jane', lastName: 'Doe', age: 30},

{firstName: 'Jack', lastName: 'Doe', age: 35},

{firstName: 'Jill', lastName: 'Doe', age: 40},

{firstName: 'Joe', lastName: 'Doe', age: 45},

]

const result = **users.map**((user) => user.firstName + ' ' + user.lastName)

console.log(result); // ['John Doe', 'Jane Doe', 'Jack Doe', 'Jill Doe', 'Joe Doe']

**FILTER ()**

**Odd numbers from an array of numbers.**

const arr = [1, 2, 3, 4, 5];

const output = arr.filter((num) => num % 2) // filter out odd numbers

console.log(arr); // [1, 2, 3, 4, 5]

console.log(output); // [1, 3, 5]

**Need objects age >30**

const users = [

{firstName: 'John', lastName: 'Doe', age: 25},

{firstName: 'Jane', lastName: 'Doe', age: 30},

{firstName: 'Jack', lastName: 'Doe', age: 35},

{firstName: 'Jill', lastName: 'Doe', age: 40},

{firstName: 'Joe', lastName: 'Doe', age: 45},

]

const output = users.filter(users => users.age > 30)

console.log(output); // [{firstName: 'Jack', lastName: 'Doe', age: 35}, {firstName: 'Jill', lastName: 'Doe', age: 40}, {firstName: 'Joe', lastName: 'Doe', age: 45}]

**REDUCE ()**

You should is used it when you want to perform some operation on the elements of an array and return a single value as a result.

The "single value" refers to the accumulated result of repeatedly applying a function to the elements of a sequence.

**sum up all the elements in an array**

const numbers = [1, 2, 3, 4, 5];

const sum = numbers.reduce((total, currentValue) => {

return total + currentValue;

}, 0)

console.log(sum); // 15

**find the maximum value in an array**

let numbers = [5, 20, 100, 60, 1];

const maxValue = numbers.reduce((max, curr) => {

if(curr > max) max = curr;

return max;

});

console.log(maxValue); // 100

**merge different objects in a single object**

const obj1 = { a: 1, b: 2 };

const obj2 = { c: 3, d: 4 };

const obj3 = { e: 5, f: 6 };

const mergedObj = [obj1, obj2, obj3].reduce((acc, curr) => {

return { ...acc, ...curr };

}, {});

console.log(mergedObj); // { a: 1, b: 2, c: 3, d: 4, e: 5, f: 6 }

**grouping products in a shopping cart according to their brand name.**

const shoppingCart = [

{name: 'Apple', price: 1.99, quantity: 3},

{name: 'Apple', price: 1.99, quantity: 3},

{name: 'Xiomi', price: 2.99, quantity: 2},

{name: 'Samsung', price: 3.99, quantity: 1},

{name: 'Tesla', price: 3.99, quantity: 1},

{name: 'Tesla', price: 4.99, quantity: 4},

{name: 'Nokia', price: 4.99, quantity: 4},

]

const products = shoppingCart.reduce((productGroup, product) => {

const name = product.name;

if(productGroup[name] == null) {

productGroup[name] = [];

}

productGroup[name]. push(product);

return productGroup;

}, {});

console.log(products);

//OUTPUT

{

Apple: [

{name: 'Apple', price: 1.99, quantity: 3},

{name: 'Apple', price: 1.99, quantity: 3}

],

Xiomi: [ {name: 'Xiomi', price: 2.99, quantity: 2 } ],

Samsung: [ {name: 'Samsung', price: 3.99, quantity: 1 } ],

Tesla: [

{name: 'Tesla', price: 3.99, quantity: 1 },

{ name: 'Tesla', price: 4.99, quantity: 4 }

],

Nokia: [ {name: 'Nokia', price: 4.99, quantity: 4 } ]

}

**Guess the OutPut:**

let array = [1,2,5];

array [6] = 10;

const mapped = array.map(i => i);

console.log(array); //[ 1, 2, 5, <3 empty items>, 10]

console.log (array [4]) //undefined

console.log(array.length); // 7

console.log(mapped); // [ 1, 2, 5, <3 empty items>, 10]

Let x=10;

Let y=20;

Let z=30;

x=y;

x=y=z;

x= 30, y = 30; z=30;

function counter () {

let count =0;

return () => count++;

}

let c = counter ();

console.log(c()); //0

console.log(c()); //1

console.log(c()); //2

k

greeting.length = 10;

console.log(greeting.length); // 6

console.log(greeting);// hellow

function checkValue(value) {

const val = value?? "NAN"; **// when we use?? In between it will print whatever we pass otherwise it will print NAN**

console.log (`Value: ${val}`);

}

checkValue(24); // 24

checkValue(0); //0

checkValue(‘); //

let x = new Number("1");

let y = Number("1");

console.log(x); //[Number: 1]

console.log(y); // 1

console.log(typeof x); //object

console.log(typeof y); // number

function manipulateArray(arr){

arr.push(5); // original array is modified

arr = [1,2]; // reference is modified not original one

return arr; // will return original array

}

let list = [1,2,3,4];

manipulateArray(list);

console.log(list); //[ 1, 2, 3, 4, 5 ]

list = manipulateArray(list);

console.log(list); // [ 1, 2 ]

let x= 10;

{

console.log(x); // compile time error// For let and const, Before variable initialised, variables are in **Temporarily Dead Zone** )

let x = 20;

console.log(x);

}

let arrayA=[1,5,7,9,10];

let arrayB= arrayA;

arrayB[0]=100;

console.log(arrayA); // [ 100, 5, 7, 9, 10 ]

console.log(arrayB); // [ 100, 5, 7, 9, 10 ]

!!!!

when you assign an array to another variable using the **assignment operator (=)**, you are **not** creating a new copy of the array.

Instead, both variables reference the same array object in memory.

Therefore, any changes made to the array through one variable will affect the other variable as well.

let arrayA=[1,5,7,9,10];

let y = arrayA.indexOf(5); //1 //will get an Index of that variable

let y = arrayA.indexOf(7); //2

let y = arrayA.indexOf(10); //4

let array = [2, 3, 4];

console.log(array.unshift(1));

// Adds 1 to the beginning of the array and this function will return the length of array.

let array = [2, 3, 4];

array.unshift(1); // Adds 1 to the beginning of the array

console.log(array); // Output: [1, 2, 3, 4]

let array=[11,22,33,44,55];

array.push(array.unshift(100));

console.log(array);

**array.unshift(100)** ;

let array=[11,22,33,44,55];

array.push(array.unshift(100));

console.log(array);

**array.push**(array.unshift(100)); -> It will add the value 6 to end of the array because of the above line

Input

Let array = [1,2,3,4,5,”a”,”b”,”c”]

Output//

[2,4,6,8,10,a2,b2,c2]

let array = [1, 2, 3, 4, 5, "A", "b", "c"];

let transformedArray = array.map(item => {

if (typeof item === 'number') {

// Double the number if the item is a number

return item \* 2;

} else if (typeof item === 'string') {

// Concatenate '2' if the item is a string

return item + '2';

}

});

console.log(transformedArray);

OutPut:

console.log(a); // Output: undefined

console.log(b); // Output: ReferenceError: b is not defined

var a = b = 5;

why its saying b is undefine in JS?

So, var a; is hoisted to the top, but var b is not declared yet. Hence, b is considered an implicit global variable due to assignment without declaration (b = 5;).

Implicit global variables are generally discouraged because they can lead to unexpected behavior and bugs.

To fix this issue and ensure b is properly declared, you should explicitly declare it:

const foo = () => {

console.log(this.name); //undefine -> for empty function it will refer to window object not the sending object

}

foo.call({name:'vijay'});

const foo1 = function() {

console.log(this.name); // It will print vijay

}

foo1.call({name:'vijay'});

1. Shadow Dom:

Shadow Dom:

Shadow DOM is the group of DOM implementation to be hidden inside a single element. Encapsulate styles to the element.

**Emulated Shadow Dom:**

1. **ViewEncapsulation.None**
2. -> I will not create any encapsulation means it will uses the css form all forms(from CSS file or external links or shadow DOMs)
3. -> <https://angular.io/guide/view-encapsulation>
4. **ViewEncapsulation.Emulated**
5. -> It uses the style only from StyleUrl and inline-css and files css but not inherits css from emulated ShadowDOM.
6. -> <https://angular.io/guide/view-encapsulation>
7. **ViewEncapsulation.ShadowDom**
8. -> not a default
9. -> It uses default browser native shadow DOM. It creates the shadowRoot and used by inner components.
10. -> <https://angular.io/guide/view-encapsulation>

**Observables:**

1. Observables are mainly used to handled the asynchronous of data, Passing data between components and event handling.
2. To Handle AJAX request and response in HTTP we uses the Observables.
3. There are three method to handle Observers
4. i. next() -> Required field. handles next delivery value, called zero or more times.
5. ii. error() -> Optional field. It will handle the error from Observer.
6. iii. complete() -> Optional filed. Handles the execution complete from Observer.
7. Creating Observables using from and of method.
8. example: myObservable = of(1,2,3) or from(iterable)

**Subscribing:**

1. An Observable instance starts publishing when someone subscribe it.
2. const myObservable = of(1, 2, 3);

// Create observer object

const myObserver = {

next: (x: number) => console.log('Observer got a next value: ' + x),

error: (err: Error) => console.error('Observer got an error: ' + err),

complete: () => console.log('Observer got a complete notification'),

};

// Execute with the observer object

myObservable.subscribe(myObserver);

// Execute with the observer object

myObservable.subscribe(

result => console.log('Observer got a next value: ' + x),

err => console.error('Observer got an error: ' + err),

() => console.log('Observer got a complete notification')

);

**Multicasting:**

A multicast operator shares the single subscription created with other subscribers. The params that multicast takes in is a subject or a factory method that returns a ConnectableObservable that has to connect() method. To subscribe connect()

import { Observable,Subject } from 'rxjs';

import { take, multicast, mapTo } from 'rxjs/operators';

var observable = new Observable(function subscribe(subscriber) {

try {

subscriber.next(Math.random());

} catch (e) {

subscriber.error(e);

}

});

const multi\_op = observable.pipe(multicast(() => new Subject()));

const subscribe\_one = multi\_op.subscribe(

x => console.log("Value from Sub1 = "+x)

);

const subscribe\_two = multi\_op.subscribe(

x => console.log("Value from Sub2 = "+x)

);

multi\_op.connect();

Below are the methods used inside the pipe.

**Pipe:**

**Exponential backoff:**

Exponential backoff is **a technique used you retry an API call after failure**, it will takes the number of tries and delay

import { timer } from 'rxjs';

import { ajax } from 'rxjs/ajax';

import { retry } from 'rxjs/operators';

export function backoff(**maxTries**: number, **initialDelay**: number) {

return retry({

count: maxTries,

delay: (error, retryCount) => timer(initialDelay \* retryCount \*\* 2),

});

}

ajax('/api/endpoint')

.pipe(backoff(3, 250))

.subscribe(function handleData(data) { /\* ... \*/ });

**Router:**

<router-outlet> </router-outlet>

1. Router-outlet in angular acts as a placeholder that let the user to load the components dynamically based on the current route state.
2. The router outlet selector enables as soon as you enable routing in your application i.e., you need to import ‘Router Module’ in ‘App Module’.
3. So, there are mainly two most important events offered by router-outlet.
4. Activate — it emits when some component initiated.
5. Deactivate — it emits when some component destroyed.
6. Example:
7. <router-outlet (activate)=“onActivate($event)”></router-outlet>
8. <router-outlet (Deactivate)=“onDeactivate($event)”></router-outlet>

Angular

Angular is a platform for building mobile and desktop web applications.

Join the community of millions of developers who build compelling user interfaces with Angular.

**LocalStorage()**  
 localStorage is a web storage API provided by modern web browsers that allows web applications to store data locally within the user's browser. It provides a simple key-value pair storage mechanism and is typically used to store small amounts of data persistently.

Key features of localStorage include:

* **Simple API**: localStorage provides a straightforward API for storing and retrieving data using key-value pairs.
* **Persistent Storage**: Data stored in localStorage persists even after the browser window is closed or the user navigates away from the page. It remains available until explicitly cleared by the user or the application.
* **Client-Side Storage**: localStorage operates entirely on the client-side within the user's browser. This means that data stored in localStorage is not transmitted to the server and remains local to the user's device.
* **Limited Storage Capacity**: Each origin (combination of protocol, hostname, and port) has its own isolated storage space in localStorage. However, localStorage has limited storage capacity (usually several megabytes per origin), and attempts to exceed this limit may result in a "**QuotaExceededError**".
* **Data Persistence**: Data stored in localStorage is stored as strings. Complex data types such as objects or arrays need to be serialised into JSON strings before being stored and deserialised back into their original format when retrieved.

**Usage example:**

javascript

**Copy code**

// Storing data in localStorage

localStorage.setItem('username', 'John');

// Retrieving data from localStorage

const username = localStorage.getItem('username');

console.log(username); // Output: John

// Removing data from localStorage

localStorage.removeItem('username');

// Clearing all data from localStorage

localStorage.clear();

localStorage is commonly used in web applications for various purposes such as storing user preferences, authentication tokens, cache data, and more. However, it's important to be mindful of security considerations when using localStorage, as data stored in localStorage is accessible to any script on the same origin and can potentially be vulnerable to attacks such as cross-site scripting (XSS).

**SessionStorage()**

In Angular, as in any web application, you can use the sessionStorage API to store data temporarily on the client-side, accessible only during the lifetime of the page session. sessionStorage works similarly to localStorage, but the stored data is cleared when the browser session ends, typically when the user closes the browser window or tab.

// Save data to sessionStorage

saveData(key: string, value: any): void {

sessionStorage.setItem(key, JSON.stringify(value));

}

// Retrieve data from sessionStorage

getData(key: string): any {

const item = sessionStorage.getItem(key);

return item ? JSON.parse(item) : null;

}

// Remove data from sessionStorage

removeData(key: string): void {

sessionStorage.removeItem(key);

}

// Clear all data from sessionStorage

clearStorage(): void {

sessionStorage.clear();

}

In this example, we create a service called StorageService that provides methods for saving, retrieving, removing, and clearing data from sessionStorage. We use JSON.stringify() to serialize data before storing it in sessionStorage, and JSON.parse() to deserialize it when retrieving it.

**Difference between Local and session storage ()**

Session Storage and Local Storage are both mechanisms provided by web browsers to allow web applications to store data locally on the client-side. However, they differ in terms of lifespan, scope, and usage. Here are the main differences between Session Storage and Local Storage:

* **Lifespan**:
  + **Session Storage: Data stored in session storage persists only for the browser session.** It is cleared when the user closes the browser window or tab.
  + **Local Storage**: Data stored in local storage persists even after the browser is closed and reopened. It remains available until explicitly cleared by the user or the application.
* **Scope**:
  + Both session storage and local storage have a per-origin scope, meaning that data is isolated and accessible only within the same origin (combination of protocol, hostname, and port).
  + Data stored in session storage is accessible across multiple tabs or windows within the same browser session.
  + Data stored in local storage is also accessible across multiple tabs or windows, but it persists across browser sessions.
* **Storage Capacity**:
  + **Session Storage**: Typically, session storage has a smaller storage capacity compared to local storage. It is usually limited to a few megabytes per origin.
  + **Local Storage**: Local storage typically has a larger storage capacity compared to session storage. It can store several megabytes of data per origin, depending on the browser.
* **Usage**:
  + Session storage is commonly used for storing temporary data that needs to be available only for the duration of the browser session. It is suitable for storing session-specific preferences, temporary user data, or data that needs to be cleared when the user closes the browser.
  + Local storage is commonly used for storing persistent data that needs to be available across browser sessions. It is suitable for storing user preferences, authentication tokens, cache data, or any data that needs to be retained even after the browser is closed and reopened.

In summary, session storage is used for storing temporary data that persists only for the duration of the browser session, while local storage is used for storing persistent data that persists even after the browser is closed and reopened. Both session storage and local storage are valuable tools for web developers, and the choice between them depends on the specific requirements of the application.

**ng-template and ng-container**

**ng-template:**

* **Definition**:
  + ng-template is a directive in Angular that defines a **template block within the component's HTML without rendering its content immediately.**
* **Purpose**:
  + It acts as a placeholder for content that may need to be **conditionally rendered or dynamically manipulated**.
* **Usage**:
  + It is commonly used with structural directives like \*ngIf and \*ngFor, or with Angular components that conditionally render content.
  + It can also be used with Angular's **Template Reference Variables (#)** to reference the template and manipulate it programmatically.
* **Example**:  
   <ng-template #myTemplate> <p>This is a template content</p> </ng-template>
  + In this example, myTemplate serves as a placeholder for the paragraph content, which can be conditionally rendered or programmatically manipulated.

**ng-container:**

* **Definition**:
  + ng-container is a directive in Angular that serves as a logical container for grouping elements in the template without adding an additional element to the DOM.
* **Purpose**:
  + It allows you to apply structural directives (\*ngIf, \*ngFor) to multiple elements without introducing additional DOM elements.
* **Usage**:
  + It is commonly used with structural directives to conditionally render multiple elements or with \*ngFor to iterate over a collection of items.
  + It's particularly useful when you need to conditionally render or loop over elements without creating unnecessary HTML elements in the DOM.
* **Example**:  
   <ng-container \*ngFor="let item of items"> <div>{{ item }}</div> </ng-container>   
  + In this example, ng-container serves as a logical container for multiple div elements that are rendered conditionally based on the \*ngFor loop.

<**ng-container** \*ngIf="condition; else **templateRef**">

Content when condition is true

</ng-container>

<**ng-template** #**templateRef**>

Content when condition is false

</ng-template>

**Angular is a framework & React is a library.**

**What is the difference between a framework and a library?**

* **A Library** is a collection of pre-written code that can be reused to perform specific tasks, such as creating user interfaces or handling network requests. A library typically provides a set of functions or classes that can be called by the developer's code to accomplish a particular task. Libraries are designed to be flexible and can be used in a variety of applications.
* **A Framework** is a more comprehensive set of tools and conouventions that provides a structure for building an entire application. A framework typically includes a set of libraries and tools, as well as a set of rules and best practices that guide the development process. Frameworks provide a more prescriptive approach to application development, and developers must often follow the framework's guidelines closely to ensure that their code integrates correctly with the rest of the application.

**Here's an explanation of each of these files commonly found in Angular projects:**

1. \*\*angular.json:\*\*  
 - This file is the Angular CLI configuration file. It contains settings and configurations for your Angular project, such as build options, asset paths, and project targets.

2. \*\*package.json:\*\*  
 - This file is a manifest file for Node.js projects. It includes metadata about the project, such as dependencies (libraries and packages required for the project), scripts for building and running the project, and other project-specific configurations.

3. \*\*README.md:\*\*  
 - This file typically contains documentation and instructions for developers on how to use and contribute to the project. It's commonly written in Markdown format and is displayed on the project's repository homepage on platfortemplatems like GitHub.

4. \*\*tsconfig.json:\*\*  
 - This file is the TypeScript compiler configuration file. It specifies compiler options and settings for compiling TypeScript code into JavaScript. It can include options such as target ECMAScript version, module resolution strategy, and source map generation settings.

5. \*\*editorconfig:\*\*  
 - This file contains text editor configurations that help maintain consistent coding styles across different editors and IDEs. It defines settings like indentation style, tab size, line endings, and more.

6. \*\*gitignore:\*\*  
 - This file specifies which files and directories should be ignored by Git, the version control system. It's used to prevent certain files (such as build artifacts, dependencies, and sensitive information) from being tracked by Git and committed to the repository.

7. \*\*tsconfig.app.json:\*\*  
 - This file is a TypeScript configuration specific to the Angular application. It extends the main tsconfig.json file and may contain additional settings specific to the application's TypeScript code.

8. \*\*tsconfig.spec.json:\*\*  
 - This file is a TypeScript configuration specific to the Angular application's tests. It extends the main tsconfig.json file and may contain additional settings specific to running tests written in TypeScript.

9. \*\*main.ts:\*\*  
 - This file is the main entry point for the Angular application. It's a TypeScript file where the application is bootstrapped and initialized. It typically imports the root module of the application and uses the Angular platform to bootstrap the application.

10. \*\*favicon.ico:\*\*  
 - This file is the favicon (short for "favorites icon") for the web application. It's a small icon that is displayed in the browser's address bar, tabs, and bookmarks to represent the website.

11. \*\*index.html:\*\*  
 - This file is the main HTML file for the Angular application. It's the entry point for the application in the browser and typically contains the HTML structure, metadata, and placeholders for loading JavaScript and CSS files generated by the Angular build process.

These files are essential for setting up, configuring, and building Angular projects, and they play crucial roles in managing dependencies, configurations, documentation, and more.

**1. How can you decide to use Angular, React, or Vue based on the project size?**

**For small projects** where the application is simple and doesn't require a lot of complex functionality, [React](https://anywhere.epam.com/en/blog/senior-react-js-developer-interview-questions) or Vue would be good choices. Both frameworks are lightweight and easy to learn, making them great for small projects.

**For medium-sized projects** where the application requires more complex functionality, [Angular](https://anywhere.epam.com/en/blog/angular-interview-questions) or React might be a better choice. Angular is a complete framework with many built-in features, making it great for larger, more complex projects. React, on the other hand, is highly modular and can be used to build complex applications by integrating with other libraries and tools.

**For large-scale enterprise projects**, where the application is highly complex and requires a lot of features, Angular is often the preferred choice. Angular's extensive built-in functionality and robust architecture make it ideal for large-scale applications. Additionally, Angular provides strong support for [TypeScript](https://anywhere.epam.com/en/blog/typescript-vs-javascript-a-chief-software-engineer-s-perspective), a powerful programming language that can help ensure code quality and maintainability in large projects.

It's important to note that the choice of [JavaScript framework](https://anywhere.epam.com/en/blog/top-5-javascript-frameworks) ultimately depends on various factors, including project requirements, team skill set, and personal preferences. Evaluating each framework's strengths and weaknesses is essential to determine which is the best fit for your project.

**2. What are the main differences between SPA and SSR?**

SPA (Single Page Application) and SSR (Server-Side Rendering) are two different approaches to building web applications. Here are some key differences between them:

* **Architecture:** SPA is a client-side application where the entire web application is loaded in the browser and all the interactions happen on the client side. Conversely, SSR is a server-side application where the server generates the HTML for the web page and sends it to the browser.
* **Performance:** SPA provides a better user experience, as it loads the page once and then dynamically updates the content as the user interacts with the app. This reduces the amount of data transferred between the server and the client, resulting in faster page loads. SSR, on the other hand, provides faster initial page load times, as the server renders the HTML on the initial request.
* **SEO** (Search Engine Optimisation): SPA has poorer SEO compared to SSR, as most search engines struggle to index client-side rendered pages. SSR, on the other hand, provides better SEO, as the server generates the HTML for the web page, which can be easily crawled by search engines.
* **Development:** SPA is generally easier to develop, as the entire application logic is on the client side. SSR requires more server-side development and configuration.
* **Scalability:** SPA can be easily scaled horizontally by adding more servers to handle the increased load. SSR requires more powerful servers to handle the rendering of the HTML on the server side.
* **Security:** SPA can be more vulnerable to cross-site scripting (XSS) attacks, as most of the application logic is executed on the client side. SSR provides better security, as most of the application logic is executed on the server side.
* **Offline functionality:** SPA can provide better offline functionality, as most of the application logic and data is cached on the client side. SSR, however, needs a network connection to render the HTML on the server side.

Overall, SPA and SSR are both valid approaches to building web applications, and the choice between them depends on the specific requirements of the application.

**3. Describe the life cycle of any task in front end development?**

The life cycle of a task in front end development can vary depending on the project management methodology being used, but it generally follows the following stages:

* **Requirement gathering:** The task starts with gathering the requirements from the client or stakeholders. This involves understanding the features and functionalities that are required to be implemented in the front end.
* **Planning:** After gathering the requirements, the next stage is planning. This involves breaking down the task into smaller sub-tasks, estimating the time required for each of them, and deciding the order of execution.
* **Design:** In this stage, the design of the front end is created. This includes creating wireframes, mockups, and prototypes. Designers may use tools like Adobe Photoshop or Sketch for designing the front end.
* **Development:** Once the design is ready, developers start working on the actual development of the front end. This involves writing code using HTML, CSS, and JavaScript. Developers may use frameworks like React or Angular to speed up the development process.
* **Testing:** After the development is complete, the front end is tested to ensure that it meets the requirements and functions as expected. This involves testing for usability, compatibility, and performance.
* **Deployment:** Once the front end is tested and approved, it is deployed to a server or a hosting platform. This involves setting up the necessary infrastructure, configuring the server, and uploading the front end files.
* **Maintenance:** After deployment, the front end needs to be maintained to ensure that it continues to function properly. This involves fixing bugs, adding new features, and making necessary updates.

Throughout the task's life cycle, communication between the development team, the client, and other stakeholders is essential to ensure the project is completed successfully.

**4 Explain the sense of using several .env files. What types of environments do you know?**

Using multiple .env files in a software project is a common practice to manage different environment variables required by the application.

Environment variables are dynamic values that can be set outside of the application's code and configuration files, allowing developers to customize the behavior of the application for different environments, such as development, testing, staging, and production.

By using multiple .env files, developers can define environment-specific variables, making it easier and more practical to manage and maintain the application across different environments. For example, a development environment might use a different database or API key than the production environment.

The most common types of environments include:

* **Development:** This environment is used by developers to write, test, and debug code. It typically uses a local database and other resources to simulate the production environment.
* **Testing/Staging: This environment is used to test the application’s behavior in a similar environment to the production environment.** It is often used to test new features, updates, and bug fixes before deploying them to the production environment.
* **Production:** This is the environment where the application runs and is accessible to end users. It typically uses the same resources and configurations as the testing/staging environment, but with additional security and monitoring measures in place to ensure reliability and availability.

**Testing:**

Think of Jasmine as the recipe (specifying the ingredients and steps) for making a cake. Karma is the oven (providing the environment to execute the recipe) and the chef who follows it, bakes the cake, and tells you if it turned out well (test results).

* What is Karma, and what is its role in Angular testing?
* What is Jasmine, and why is it commonly used for testing Angular applications?
* How do you write a basic Jasmine test case for an Angular component?
* How do you configure Karma for an Angular project?
* What is the purpose of the karma.conf.js file in an Angular project?
* How do you mock dependencies in Jasmine tests for Angular services?
* How do you run tests for specific Angular components or files using Karma?
* What are some common Jasmine matchers used in Angular testing?
* How do you handle asynchronous code in Jasmine tests for Angular?
* Can you explain the role of code coverage reports generated by Karma in Angular testing?

Together, describe and it functions provide a clear and organized way to structure and write unit tests. describe helps group related test cases, while it defines the individual tests with specific assertions.

**Karma Configuration (**karma.conf.js**):**

"jasmine-core": "~3.8.0",

"jasmine-spec-reporter": "~7.0.0",

"karma": "~6.3.4",

"karma-chrome-launcher": "~3.1.0",

"karma-coverage": "~2.0.3",

"karma-jasmine": "~4.0.1",

"karma-jasmine-html-reporter": "^1.7.0",

"karma-junit-reporter": "^2.0.1",

"karma-sauce-launcher": "^4.3.6",

module.exports = function(config) {

config.set({

basePath: '', // Root directory of your project

frameworks: ['jasmine', '@angular-devkit/build-angular'], // Testing frameworks

plugins: [

require('karma-jasmine'),

require('karma-chrome-launcher'),

require('karma-jasmine-html-reporter'), // Reporter for test results

require('@angular-devkit/build-angular/plugins/karma'), // Plugin for Angular compilation

],

reporters: ['progress', 'kjhtml'], // Reporters to display test results

port: 9876, // Port used by Karma server

colors: true, // Display colorful output

logLevel: config.LOG\_INFO, // Log level (info, debug, etc.)

autoWatch: true, // Watch for file changes and re-run tests

browsers: ['Chrome'], // Browsers to run tests in

singleRun: false, // Keep Karma server running after tests

restartOnFileChange: true // Restart Karma server on file changes

});

};

**. Jasmine Test SPEC File (**your-component.spec.ts**):**

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

import { YourComponent } from './your.component'; // Path to your component

describe('YourComponent', () => {

beforeEach(async () => {

await TestBed.configureTestingModule({

declarations: [YourComponent],

}).compileComponents();

});

it('should create the component', () => {

const fixture = **TestBed.createComponent**(YourComponent);

const component = **fixture.componentInstance**;

expect(component).toBeTruthy();

});

// Add more test cases here to test different functionalities of your component

});

* The karma.conf.js file configures Karma, specifying frameworks, plugins, reporters, browsers, and other settings.
* The Jasmine test file (your-component.spec.ts) defines test cases for your Angular component (YourComponent).
  + It imports necessary modules like TestBed for testing Angular components.
  + It defines a describe block for the component and uses beforeEach to set up the testing environment.
  + The blocks define individual test cases. Here, a basic test verifies if the component is created successfully.

**H1B - SLK2024024214**

#91/1, C/o Sureka Rani,

13th Cross, K block, Rajajinagar 1st Block

Beside Old Roman Technology Bengaluru - 560010

#2,4th cross,

Bheereshwara Nilay, Sri Sathya Sai layout

Beside Prestige White Meadows

Bengaluru 560066

Given name: Vijay Kumar

Sure name : Arehalli Varadaraju

Gender : Male

Date : 04-14-94

Country Birth: India

Citizen: Indian

Address:

#2,4th cross,

Bheereshwara Nilay, Sri Sathya Sai layout

Beside Prestige White Meadows

Bengaluru 560066

#34, Arehalli, Bilagumba Post, Kutagal hobli, Ramanagara Tq & Dist, Karnataka-562159

Passport#: U5266407

Passport Issue date: 24-02-2020

Passport Exp date: 23-02-2030

Passport issue place: Bengaluru

State of birth: Karnataka

Marital status: Married

Designation : Senior Software Engineer

Exp : 7 Years

Exp in SLK : 4 Years

Contact Details:

Home : 8105242463

Mob: 7019216026

work:

extn:

Email office: [vijay.a@slkgroup.com](mailto:vijay.a@slkgroup.com)

Personal : [vijaykumar.varadaraju@gmail.com](mailto:vijaykumar.varadaraju@gmail.com)

Educations:

Bachelors Of Engineering -> EC -> 2016

Work exp:

Yellow and Red Services Put Ltd -> Full Stack Developer -> Feb 8th 2017 -> Feb 20th 2020

SLK Software Pvt Ltd -> Senior Software Engineer -> Feb 26th 2020 -> till date

I have successfully developed the Orange Money Panel on my own. I am dedicated to my work and ensure that I extend my working hours to provide overlap time for onshore team members, as I am the sole contributor from the offshore team (Scrum and Coke). I have a proactive approach and always strive to utilize my capacity to the fullest. If I encounter any issues with the stories, I take it upon myself to debug them and provide solutions to the clients.

In order to ensure minimal impact on production, I put in extra effort to address any UAT defects or observations that arise from my work. For example, when a defect related to the snapshot dynamics tile was discovered in production, I resolved it within a day and pushed a patch to production single-handedly.

Since completing the unit test training, I have noticed a significant improvement in my ability to write effective Unit Test cases. I believe in continuous learning and therefore invest my time in acquiring new skills, such as learning the basics of Spring Boot for better understanding of backend work. Additionally, I have dedicated time to insurance domain training conducted by Dr. Anupam Suri, which has greatly enhanced my understanding of the product and allows me to actively contribute to discussions in grooming sessions with Voya.

To ensure a high-quality final product, I always provide a demo to the QA team once I have completed a development task. This helps to minimise defects in my work.

**UI Developer**

**Symantic Tags:**

Semantic HTML refers to the use of HTML tags to convey the meaning and structure of the content they enclose, making the code more understandable for both humans and machines.

Semantic HTML tags describe the purpose or role of the content they contain, rather than just specifying how the content should be displayed. Some common examples of semantic HTML tags include <header>, <nav>, <main>, <section>, <article>, <aside>, <footer>, <p>, <h1> to <h6>, <ul>, <ol>, <li>, <table>, <form>, <input>, <button>, <label>, <select>, <textarea>, and so on.

Using semantic HTML tags not only improves the accessibility and SEO of your web pages but also makes the structure of your HTML clearer and easier to understand for developers who may be working with your code in the future.

Here are a few examples of how semantic HTML tags can be used:

* <header>: Used to define introductory content or navigation links at the top of a page.
* <nav>: Represents a section containing navigation links.
* <main>: Contains the main content of the document.
* <section>: Represents a thematic grouping of content, typically with a heading.
* <article>: Represents a self-contained piece of content that could be distributed and reused independently.
* <footer>: Represents a footer for its nearest sectioning content or sectioning root element.

**Non Symantic Tags:**

Non-semantic HTML tags are those tags that do not convey any specific meaning or purpose about the content they contain. These tags are typically used for formatting, layout, or styling purposes, rather than for describing the structure or meaning of the content. Non-semantic tags are often considered less descriptive and less helpful for accessibility and SEO compared to semantic tags.

Examples of non-semantic HTML tags include:

* <div>: This tag is a generic container used to group elements together for styling purposes. It doesn't provide any semantic meaning to the content it contains.
* <span>: Similar to <div>, <span> is a generic inline container used for styling inline elements. It doesn't provide any semantic meaning to its content.
* <font>: Historically used for setting font styles, sizes, and colors, <font> is now deprecated in HTML5 in favor of CSS for styling text.
* <b>, <i>, <u>, <strike>: These tags were commonly used for text formatting (bold, italic, underline, strike-through), but they are considered non-semantic because they don't convey any meaning about the text's significance or structure.
* <center>, <align>: These tags were used for text alignment and layout purposes, but they are now deprecated in HTML5. CSS should be used for layout and alignment instead.

Non-semantic tags are still widely used in HTML for layout and styling purposes, especially when building complex user interfaces. However, the trend in modern web development is to use semantic HTML elements whenever possible to improve accessibility, search engine optimization, and code maintainability. CSS is typically used for styling and layout, while HTML is used for structuring and describing the content.

**New Features of HTML5**

* **Semantic Elements**: HTML5 introduced new semantic elements such as <header>, <footer>, <nav>, <article>, <section>, <aside>, and <main>. These elements provide better structure and meaning to web documents, improving accessibility and SEO.
* **Audio and Video**: HTML5 includes native support for embedding audio and video content using the <audio> and <video> elements. This eliminates the need for third-party plugins like Flash for multimedia playback.
* **Canvas**: The <canvas> element allows for dynamic rendering of graphics, animations, and interactive visualizations using JavaScript. It provides a drawing API for rendering shapes, images, and text on the fly.
* **SVG**: HTML5 includes native support for **Scalable Vector Graphics (SVG)**, allowing for the creation and manipulation of vector graphics directly within web pages.
* **Geolocation**: HTML5 provides a Geolocation API that allows web applications to access the user's geographical location through the browser. This enables location-aware functionality in web applications, such as mapping and local search.
* **LocalStorage and SessionStorage**: HTML5 introduced the localStorage and sessionStorage APIs for storing data locally in the browser. These APIs provide a way to store key-value pairs persistently (localStorage) or for the duration of a browser session (sessionStorage).
* **Web Storage**: HTML5 introduced the localStorage and sessionStorage APIs for storing data locally in the browser. These APIs provide a way to store key-value pairs persistently (localStorage) or for the duration of a browser session (sessionStorage).
* **Web Workers**: HTML5 introduced Web Workers, which allow for multi-threaded JavaScript execution in web applications. Web Workers enable background processing without blocking the main UI thread, improving responsiveness and performance.
* **WebSockets**: HTML5 includes native support for WebSockets, which enables full-duplex communication between a web browser and a server over a single TCP connection. WebSockets facilitate real-time, low-latency communication in web applications.
* **Form Enhancements**: HTML5 introduced several enhancements to web forms, including new input types (date, email, url, number, range, etc.), attributes (placeholder, required, autocomplete, etc.), and form validation features.

These are just some of the key features introduced in HTML5. HTML5 also includes various other enhancements, improvements, and APIs aimed at making web development easier, more powerful, and more feature-rich.

**New Features of CSS3**

CSS3 introduced several new features and enhancements over its predecessor, CSS2.1, providing web developers with more powerful tools for styling and layout. Some of the key new features of CSS3 include:

* **Selectors**:
  + **Attribute Selectors**: CSS3 introduced additional attribute selectors such as ^, $, and \* for matching elements based on attribute values.
  + **Nth-child Selectors**: Select elements based on their position in a parent element using :nth-child() and related pseudo-classes.
* **Box Model**:
  + **Box-sizing Property**: Allows developers to specify whether an element's padding and border should be included in its width and height calculations.
  + **Border-radius Property**: Enables the creation of rounded corners on elements.
* **Colors and Gradients**:
  + **RGBA and HSLA Colors**: CSS3 introduced RGBA (Red, Green, Blue, Alpha) and HSLA (Hue, Saturation, Lightness, Alpha) color models, allowing for transparency and alpha-channel control.
  + **Linear and Radial Gradients**: Allows for the creation of smooth color transitions using linear and radial gradients.
* **Text Effects**:
  + **Text-shadow Property**: Adds shadows to text elements.
  + **@font-face Rule**: Allows web developers to specify custom fonts for text elements, enabling the use of non-web-safe fonts.
* **Transforms and Transitions**:
  + **Transform Property**: Enables the transformation of elements using functions like rotate(), scale(), translate(), and skew().
  + **Transitions**: Allows for the smooth transition of property values over a specified duration.
* **Animations**:
  + **Keyframes**: CSS3 introduced keyframe animations, allowing developers to define intermediate steps in an animation sequence.
  + **Animation Property**: Provides a shorthand for defining animation properties such as duration, timing function, and delay.
* **Flexbox Layout**:
  + **Flexbox Layout Module**: Introduces a flexible box layout model, making it easier to create complex layouts that adapt to different screen sizes and orientations.
* **Grid Layout**:
  + **Grid Layout Module**: Enables the creation of grid-based layouts with rows and columns, offering precise control over the placement and alignment of elements.
* **Media Queries**:
  + **Responsive Design**: CSS3 introduced media queries, allowing developers to apply styles based on the characteristics of the device, such as screen size, resolution, and orientation. This enables the creation of responsive web designs that adapt to different devices and viewport sizes.
* **Multiple Column Layout**:
  + **Multi-column Layout Module**: Allows for the creation of multi-column layouts within a single element, similar to newspaper columns.

These are just some of the key features introduced in CSS3. CSS3 brought significant improvements and new capabilities to web styling, empowering developers to create more dynamic, visually appealing, and responsive web experiences.

**CSS**

1. **Grid Layout**
2. The CSS Grid Layout Module offers a grid-based layout system, with rows and columns, making it easier to design web pages without having to use floats and positioning.  
    **display: grid;**  
    **grid-template-columns: auto auto auto auto auto;**
3. **What is the Box model in CSS? Which CSS properties are a part of it?**
4. A rectangle box is wrapped around every HTML element. The box model is used to determine the height and width of the rectangular box. The CSS box model is a container that contains multiple properties including borders, margin, padding, and the content itself. It is used to create the design and layout of web pages.
   * **Padding:** Area surrounding the content (Space between the border and content).
   * **Border:** Area surrounding the padding.
   * **Margin:** Area surrounding the border.
   * **Content:** Actual Content of the box where the text or image is placed.
5. **What are Class loaders in Bootstrap?**
6. Class loaders also perform the process of converting a named class into its equivalent binary form.
7. **What is Bootstrap Grid System?**
8. grid system that appropriately scales up to 12 columns as the device or viewport size increases.
9. <div class = "container">
10. <div class = "row">
11. <div class = "col-sm-4”></div>
12. <div class = "col-md-2”></div>
13. </div>
14. <div class = "row">...</div>
15. </div>

1. **How do you make images responsive?**
2. The class applies max-width: 100%; and height: auto;
3. **What do you understand by the universal sector?**
4. A universal selector is a selector that matches any element type’s.
5. \*{
6. color:blue;
7. font-size-10px;
8. }

**Define z-index**.

1. Z-index is used to specify the stack order of elements that overlap with each other. Its default value is zero and can take both negative and positive values. A higher z-index value is stacked above the lower index element. It takes the following values- auto, number, initial, and inherit.   
   **.class {**  
     **z-index: 1;**  
    **}**
2. **Tell us about the property used for image scroll controlling?**
3. body {
4. background-image: url(‘url\_of\_image’);
5. background-repeat: no-repeat;
6. background-attachment: fixed;
7. }

**Pseudo-Elements:**

Pseudo-elements are used to style certain parts of an element's content. They allow you to style parts of an element that are not part of the actual HTML structure. Pseudo-elements are denoted by double colons (::) and include common examples like :: before, ::after, ::first-line, and ::first-letter.

p::first-line {

font-weight: bold;

}

p::before {

content: "Before text: ";

}

**Pseudo-Classes:**

Pseudo-classes are used to define the special state of an element. They allow you to style an element based on its state or relationship with other elements, such as hover, focus, active, first-child, etc. Pseudo-classes are also denoted by a single colon (:).

a:hover {d

color: red;

}

input:focus {

border-color: blue;

}

li:first-child {

font-weight: bold;

}

1. **What are the limitations of CSS?**  
    **->** We cannot request a webpage through CSS.
2. -> Some selectors can lead to cross-browser issues due to their less browser-friendly behavior.
3. -> The parent selector tag is not available, thus you can’t select the parent selector tag.
4. -> CSS cannot always assure compatibility with every browser; you need to be cautious while choosing the style selector.

**HTML**

1. **What is the advantage of collapsing white space?**
2. the browser collapses multiple spaces into a single space, you can indent lines of text without worrying about multiple spaces. This enables you to organize the HTML code into a much more readable format.  
      
   **Advantages:** It can help reduce the size of HTML documents, making them load faster. This is because white space, such as extra spaces or line breaks, takes up unnecessary bytes in the document.
3. **What is the ‘class’ attribute in HTML?**
4. -> The ‘class’ attribute specifies the class name for an HTML element. Multiple HTML elements can have the same class value.
5. -> The ‘id’ attribute specifies the unique Id for an HTML element. Multiple HTML elements cannot have the same id value.
6. **Basic Structure of HTML.**

1. **What is Grouping?**
2. More than one selector shares the same declaration.
3. example: h1, h2, h3 {background: red;}
4. **How to create a new HTML element using JavaScript?**
5. <script>
6. document.createElement(“myElement”)
7. </script>
8. <myElement>hello Intellipaat!</myElement>
9. **Local Storage Session Storage**
10. The localStorage object stores data indefinitely. SessionStorage object stores them for only one session.
11. If browser window is closed, the data will not be erased. The data will get erased after the browser window is closed.
12. The data that is stored in the localStorage can be The data that is stored in the sessionStorage is accessible
13. only in the current browser window.
14. accessed by multiple browser windows.
15. **How can you create nested web pages in HTML?**
16. <iframe> tag
17. **!IMPORTANT CSS**
18. CSS style rules “cascade” in the sense that they follow an order of precedence. !important will help to override the existing CSS procedures.
19. **What are the feature of HTML5**
20. geolocation = navigator.geolocation  
     **Audio**  
     **Video**  
     **Canvas**  
     **SVG**  
     **Picture**  
     **manifest cache**
21. **What is the use of a novalidate attribute for the form tag in HTML5?**  
     The novalidate attribute for the <form> tag in HTML5 is used to disable native browser form validation. When this attribute is present, the browser's built-in form validation, which checks input fields based on their required, pattern, min, max, and other attributes, is bypassed.

**What is a manifest file in HTML5?**

<!DOCTYPE html>

<html lang="en" manifest="**manifest.appcache**">

In HTML5, a manifest file refers to a file named **manifest.appcache** that is used in the context of web applications to provide information about resources that should be cached locally by the browser. This mechanism is known as the Application Cache or AppCache.

**The manifest file allows web developers to specify a list of files that should be cached by the browser, enabling offline access to web applications**. It typically includes HTML, CSS, JavaScript files, images, and other assets necessary for the web application to function offline.

CACHE MANIFEST

CACHE:

index.html

styles.css

script.js

logo.png

NETWORK:

\*

FALLBACK:

/ offline.html

**HR:**

**Why are you interested in Wipro?**

I am interested in working at wipro.com because **it is a leading global information technology, consulting and business process services company**. I believe that I can contribute to the company's success and help it grow further. wipro.com has a strong focus on innovation and technology, which I find very appealing.

"What are your greatest strengths?"

"One of my greatest strengths is my ability to adapt to new situations quickly. I enjoy challenges and can remain composed under pressure. Additionally, I'm highly organized and detail-oriented, which helps me manage multiple tasks efficiently."

Can you tell me about a time when you demonstrated leadership skills?"

In my previous role, I was tasked with leading a cross-functional team to implement a new project management system. I effectively delegated tasks, motivated team members, and ensured open communication throughout the process. As a result, we successfully launched the system ahead of schedule and within budget."

What do you consider your strongest technical skills?"

I have strong technical skills in [specific technology or tool], which I've honed through hands-on experience and continuous learning. I'm proficient in [specific programming languages, frameworks, or software], and I'm often sought out by colleagues for assistance and advice in these areas."

**Git:**

# Add the upstream remote

git remote add upstream https://github.com/example/upstream-repo.git

# Fetch the branches and their respective commits from the upstream repository

git fetch upstream

# Create a new branch and check it out

git checkout -b feature/new-upstream-changes

# Merge the changes from the 'main' branch of the upstream repository into your new branch

git merge upstream/main

**What is the Status Code of HTTP response??**

### **1xx: Informational**

* **100 Continue**: The initial part of a request has been received and has not yet been rejected by the server.
* **101 Switching Protocols**: The server understands and is willing to comply with the client's request to switch protocols.

### **2xx: Success**

* **200 OK**: The request has succeeded. The meaning of success depends on the HTTP method (GET, POST, etc.).
* **201 Created**: The request has been fulfilled and has resulted in one or more new resources being created.
* **202 Accepted**: The request has been accepted for processing, but the processing has not been completed.
* **204 No Content**: The server successfully processed the request, but is not returning any content.

### **3xx: Redirection**

* **301 Moved Permanently**: The resource has been moved to a new URL permanently.
* **302 Found**: The resource has been temporarily moved to a different URL.
* **304 Not Modified**: The resource has not been modified since the version specified by the request headers.

### **4xx: Client Errors**

* **400 Bad Request**: The server could not understand the request due to invalid syntax.
* **401 Unauthorized**: The request requires user authentication.
* **403 Forbidden**: The server understood the request, but refuses to authorize it.
* **404 Not Found**: The server cannot find the requested resource.
* **405 Method Not Allowed**: The request method is known by the server but is not supported by the target resource.

### **5xx: Server Errors**

* **500 Internal Server Error**: The server encountered an unexpected condition that prevented it from fulfilling the request.
* **501 Not Implemented**: The server does not support the functionality required to fulfill the request.
* **502 Bad Gateway**: The server, while acting as a gateway or proxy, received an invalid response from the upstream server.
* **503 Service Unavailable**: The server is currently unable to handle the request due to a temporary overload or scheduled maintenance.
* **504 Gateway Timeout**: The server, while acting as a gateway or proxy, did not receive a timely response from the upstream server.

**Which grade i will be belongs to for my position?**

**Salary will be slightly negociable?**

**Which Location i will be working in banglore?**  
**In hand Salary?**

**Performance Bonus divide is like Quaterly or Half Year or How it is?**  
**Expected Project Team if you aware about?**

**When i can expect Offer Letter!? Because i need to inform for my another Employer which i have an offer which is about to come in next week. Suppose if i reject the offer for the same then they will go for another replacement of mine. Otherwise it will be difficult for them. I know this is though choice but i need to decline ,**