Angular Important Question

Run Angular On Specific port
ng s –port 3005
For debugging sources ctrl+p
Folder structure of angular
Main.ts
Entry Point: This file acts as the entry point for the Angular application. It is responsible for bootstrapping the application.
Bootstrap Module: In main.ts, the root module (usually AppModule) is bootstrapped using the platformBrowserDynamic()
.bootstrapModule() method. This tells Angular to start the application with the specified module.

Style.css
Style.css used for the global styling of the application

Editor Config File
.editorconfig:-To enforce consistent coding styles for everyone that works in a codebase, you can add an .editorConfig file to your solution or project.EditorConfig file settings adhere to a file format specification maintained by EditorConfig.org.

.gitignore file
The purpose of gitignore files is to ensure that certain files not tracked by Git remain untracked. To stop tracking a file that is currently tracked, use git rm --cached to remove the file from the index.The filename can then be added to the .gitignore file to stop the file from being reintroduced in later commits.

Angular.json
A file named angular.json at the root level of an Angular workspace provides workspace-wide and project-specific configuration defaults for build and development tools provided by the Angular CLI. Path values given in the configuration are relative to the root workspace folder.

Index.html
This file holds the final version of the HTML code that the user's browser will fetch from the web server. Default index.html file generated by the Angular builder. Notice the few additions that have been made to the source HTML file.

Package.Json
The package.json file is a fundamental element in Node.js and JavaScript projects. It serves as the blueprint for your project, detailing everything from the project's metadata to its dependencies. This file is created when you initialize a new Node.js project using the npm init

command. It includes various fields like name, version, scripts, dependencies, and devDependencies, each serving a specific purpose in the project lifecycle.

Package-Lock.Json

The package-lock.json file is an automatically generated file in Node.js projects, created when you first run npm install. Its primary role is to lock down the exact versions of every package and its dependencies that are installed in your project. This ensures that every time you or someone else installs the project dependencies using npm install, the same versions of the dependencies are installed. This consistency is crucial for preventing the infamous "it works on my machine" problem, where code works in one environment but not in another due to different package versions.

Difference between Package.Json and Package-Lock.Json

package.json and package-lock.json are integral components of Node.js and JavaScript projects, each serving a unique and complementary role. While package.json acts as the project's manifest, detailing dependencies and scripts, package-lock.json ensures consistent and reliable installation of these dependencies across different environments. Understanding the distinction and interplay between these files is essential for any developer looking to maintain stable and consistent Node.js projects. Their collaborative use ensures that a project runs smoothly, irrespective of the environment, by locking down specific versions of packages and their dependencies, thus fostering a more predictable and controlled development process.

Angular Commands

Create Application :- ng init
Run Application :- ng serve -o
Create Component :- ng g c
ComponetInside Fol:- ng g c folder/component

What is SPA

A Single Page Application (SPA) is a type of web application that operates within a single web page, dynamically updating content as users interact with it, without requiring a full page reload. In contrast to traditional multi-page applications, where each user action results in loading a new page from the server, SPAs load the necessary HTML, CSS, and JavaScript once and rely on AJAX or Fetch API calls to fetch and display new content.

Key features of SPAs:
- Improved user experience: Since only parts of the page update, SPAs provide a smooth and responsive experience, similar to a desktop app.

- Reduced server load: By minimizing full-page reloads, SPAs reduce the load on the server.
- Examples: Gmail, Google Maps, Facebook, and Twitter are well-known SPAs.

Frameworks like React, Angular, and Vue are often used to build SPAs.

Is it possible to create spa without using any framework or how to create spa using vanilla JavaScript

Yes, it's possible to create a Single Page Application (SPA) without using any framework by leveraging **Vanilla JavaScript** (plain JavaScript) along with HTML and CSS. Although frameworks like React or Angular provide helpful tools and abstractions, you can manually build the functionality needed for an SPA. Here's how you can do it:

Key Concepts for Building a SPA with Vanilla JavaScript

1. **HTML5 History API**: Used to change the URL in the browser without reloading the page (using pushState, replaceState, and popstate).
2. **AJAX (or Fetch API)**: Used to load data dynamically from the server without refreshing the page.
3. **Event Listeners**: For capturing user interactions like clicks, and updating the content accordingly.
4. **DOM Manipulation**: Dynamically updating the page's content based on user actions or data retrieved from the server.

How angular or any frameworks Creates Single Page Application

Client-Side Routing: Frameworks like Angular handle routing without refreshing the page, allowing smooth transitions between views.
Dynamic Rendering: Components are dynamically rendered based on the route, with no need for full-page reloads.
AJAX/Fetch for Data: Data is fetched asynchronously from the server, allowing the app to update content without refreshing.
History API: Frameworks use the HTML5 History API to manage URLs and browser history, providing a seamless user experience.
Component-Based Structure: The architecture of these frameworks organizes the application into reusable components, improving maintainability and scalability.

What is difference between Javascript and Typescript

JavaScript (JS)

Definition: JavaScript is a dynamic, loosely-typed programming language primarily used to build interactive websites. It was originally designed for client-side (browser) scripting but is now also widely used for server-side programming (e.g., with Node.js).

Type System: JavaScript is dynamically typed, meaning you don't need to define variable types. The types are determined at runtime.

Syntax: JavaScript code is more flexible but can lead to runtime errors due to its lack of strict type checking.

Usage: JavaScript runs in web browsers and powers the behavior of web pages. It's the standard language for building interactive, dynamic web content.

```
let greeting = 'Hello'; // No need to declare types
console.log(greeting);  // Outputs: Hello
```

TypeScript (TS)

Definition: TypeScript is a superset of JavaScript that adds static typing and other features. It was developed by Microsoft to help developers catch errors at compile time and write more maintainable and scalable code. TypeScript code is compiled (or transpiled) into plain JavaScript to run in browsers or on platforms like Node.js.

Type System: TypeScript is statically typed, meaning you must declare the types of variables, parameters, and return values. This ensures type checking during development, reducing the likelihood of bugs.

Syntax: TypeScript extends JavaScript by adding features like type annotations, interfaces, generics, and more. It also supports all modern JavaScript features.

Usage: TypeScript is often used in larger applications where type safety, code readability, and maintainability are critical. It's popular in frameworks like Angular and libraries like React (with TypeScript support).

```
let greeting: string = 'Hello'; // Type is explicitly defined
console.log(greeting);  // Outputs: Hello
```

Key Differences:

Typing:

JavaScript: Dynamic typing (types are checked at runtime).
TypeScript: Static typing (types are checked at compile-time).
Development Errors:

JavaScript: Errors can only be caught during runtime.
TypeScript: TypeScript helps catch errors early during development/compilation.
Tooling and IDE Support:

TypeScript: has better tooling support (e.g., autocompletion, error checking in IDEs) due to its type annotations.
JavaScript: can be more error-prone in large projects due to its lack of type safety.
Compilation:

JavaScript: Directly runs in the browser or Node.js without compilation.
TypeScript: Needs to be compiled to JavaScript before it can run in a browser or Node.js.

Latest Stable version of Angular

Angular, a robust framework maintained by Google, is renowned for enabling developers to craft dynamic and high-performing web applications. Following its tradition of biannual updates to introduce innovative features and enhancements, the release of Angular 18 marks another significant milestone. This version brings forth a range of impactful changes and improvements compared to its predecessor, Angular 17. Understanding these updates is crucial for developers seeking to leverage the latest capabilities in their projects.

Angular 18:-

In Angular 18, performance improvements are a key focus. The change detection mechanism has been refined to reduce processing time for data-bound property changes, making applications more responsive. The rendering engine has also been optimized to re-render only components that have changed, rather than entire sections, boosting performance in dynamic applications. Additionally, the Ivy compiler has been enhanced to support more efficient incremental builds, speeding up the development process by compiling only necessary parts of the application.

Angular 18 introduces new features alongside performance enhancements. Standalone components allow developers to create components without requiring an NgModule, simplifying project structure and reducing boilerplate. A new dynamic component loading API enables easier runtime component addition or replacement, offering better control over memory usage and behavior. Additionally, improved support for typed forms brings strict typing to reactive forms, helping to catch errors during compilation and enhancing form reliability and maintainability.
medium arcticle angular 18

Angular has undergone significant changes since its initial release as Angular 2 (after AngularJS), with each version introducing new features, performance improvements, and

breaking changes. Here's an overview of the major versions of Angular and the key differences between them:

AngularJS (Version 1.x)

Release Date: 2010
Key Features:
Based on JavaScript.
Utilized two-way data binding, dependency injection, and directives.
Templating done with HTML and controllers.
Use of $scope for binding data.
Single-threaded.
Lacked a module system.
Limitations:
Performance issues with large applications.
No clear structure for large-scale applications.
Difficult to debug due to $scope issues.

Angular 2

Release Date: September 2016

Key Changes:

Complete rewrite of AngularJS.
Introduced TypeScript as the primary language.
Component-based architecture instead of controllers.
One-way data binding (unidirectional).
Use of modules (@NgModule) to group components, directives, and services.
Dependency injection is more advanced.
Improved templating with ngIf, ngFor, etc.
Key Differences from AngularJS:

Component-based rather than controller-based.
Strongly typed language (TypeScript).
Modular architecture with @NgModule.

Angular 4

Release Date: March 2017

Key Changes:

Skipped Angular 3 to align versioning across core packages.
Improved router with smaller and faster builds.

Animation package extracted from @angular/core to @angular/animations.
Improved Ahead-of-Time (AOT) compilation for faster rendering.
Smaller size with reduced code generation for templates.
Key Differences from Angular 2:

Smaller and more optimized code.
Faster router performance.
Separate animation module (@angular/animations).

Angular 5

Release Date: November 2017

Key Changes:

Build optimizer included, leading to smaller bundles and faster loading times.
Improved Angular Universal support (server-side rendering).
Improved HTTP Client Module (HttpClient).
Improved support for RxJS operators with pipe() function.
Key Differences from Angular 4:

Build optimizer for better performance.
Improved server-side rendering.
Enhanced HTTPClient API with improved methods.

Angular 6

Release Date: May 2018

Key Changes:

Angular Elements introduced, enabling Angular components to be used in other frameworks.
CLI Workspaces support multiple Angular projects within the same workspace.
RxJS 6 adoption with backward compatibility.
ng update and ng add commands to easily upgrade Angular versions and add new dependencies.
Tree-shakable providers for better performance.
Key Differences from Angular 5:

Angular Elements support for cross-framework compatibility.
CLI improvements with ng update and ng add.

Angular 7

Release Date: October 2018

Key Changes:

Virtual scrolling and drag-and-drop introduced.
CLI prompts introduced for user inputs during project generation.
Improved performance with bundle size reduction for smaller apps.
Improved support for Angular Material with Component Dev Kit (CDK).
Key Differences from Angular 6:

Virtual scrolling and drag-and-drop features.
CLI prompts for better developer experience.
Smaller bundle sizes.

Angular 8

Release Date: May 2019

Key Changes:

Ivy renderer introduced (opt-in), offering faster compilation and smaller bundle sizes.
Differential loading to support modern and legacy browsers, improving load times.
Web worker support for better performance in background threads.
Dynamic imports for lazy loading modules.
Key Differences from Angular 7:

Ivy renderer (initially opt-in) for faster rendering.
Differential loading for modern and legacy browsers.

Angular 9

Release Date: February 2020

Key Changes:

Ivy renderer became the default, leading to significant performance improvements.
Smaller bundle sizes.
Improved debugging with better error messages and stack traces.
Improved internationalization (i18n) with more efficient code extraction.
Key Differences from Angular 8:

Ivy as the default rendering engine.
Significant reduction in bundle size and improved debugging tools.

Angular 10

Release Date: June 2020

Key Changes:

Updated TypeScript to version 3.9.
New warnings and strict typing options for enhanced reliability.
New Date Range Picker in Angular Material.
Updated ngcc (Angular Compatibility Compiler) for improved third-party library compatibility.
Optional strict settings for more rigorous checks in projects.
Key Differences from Angular 9:

TypeScript 3.9 support.
Optional stricter settings for better type checking.

Angular 11

Release Date: November 2020

Key Changes:

Automatic inlining of fonts for faster loading.
Improvements in Hot Module Replacement (HMR).
Better logging and stricter ESLint configuration.
Faster compilation with webpack 5 support.
Key Differences from Angular 10:

Font inlining to enhance performance.
HMR support improvements for real-time updates during development.

Angular 12

Release Date: May 2021

Key Changes:

Ivy is now mandatory for all projects.
Deprecation of View Engine.
Angular CLI improvements like persistent caching for faster builds.
Webpack 5 module federation support.
Key Differences from Angular 11:

Full adoption of Ivy and deprecation of View Engine.
CLI caching for better build performance.

Angular 13

Release Date: November 2021

Key Changes:

Removal of View Engine completely.
Enhanced API support for dynamic component creation.
TypeScript 4.4 support.
Simplified package format for faster builds and smaller bundle sizes.
Key Differences from Angular 12:

Complete removal of View Engine.
TypeScript 4.4 adoption.

Angular 14

Release Date: June 2022

Key Changes:

Standalone components introduced (experimental feature), allowing components to exist without NgModule.
Improved template diagnostics with better error detection.
Strictly typed reactive forms.
Optional Angular DevTools integration for performance profiling.
Key Differences from Angular 13:

Standalone components feature.
Improved template diagnostics and form typing.

Angular 15

Release Date: November 2022

Key Changes:

Stable support for standalone components.
Enhanced support for TypeScript 4.7+.
Improved compatibility with modern web standards.
Advanced tree-shaking for even smaller bundles.
Key Differences from Angular 14:

Stable standalone components.
Better support for modern web standards.

Angular 16

Release Date: May 2023

Key Changes:

Zone.js-less mode (experimental) for better performance.
Signal support for reactivity, bringing a reactive programming style to Angular.
Improved hydration for better server-side rendering.
Key Differences from Angular 15:

Zone.js-less mode.
Introduction of signals for reactive programming.

Angular 17

Release Date: November 2023
Key Changes:
Full support for Zone.js-less mode.
Improved reactivity model.
Major performance optimizations in template and change detection.
Dynamic imports for even more efficient lazy loading.

Angular 18

Release Date: Expected 2024
Key Changes:
Enhanced performance with optimized change detection.
Standalone components refined for better flexibility.
Improved rendering engine for more efficient component updates.
Dynamic component loading API.
Typed forms with strict compilation checks.

What is Life Cycle Hooked In Angular

Lifecycle hooks in Angular are special methods that provide developers with the ability to tap into key moments in a component or directive's lifecycle. They allow developers to run custom logic during the creation, update, and destruction of a component. These hooks help manage resources, update the DOM, and interact with services at specific points in a component’s life.

1. ngOnChanges()
Purpose: Called whenever an input property (a property decorated with @Input()) of the component is changed.

Timing: Triggered right before ngOnInit() and whenever any input property changes during the component’s lifecycle.

Use Case: When you want to respond to changes in input properties.

```
ngOnChanges(changes: SimpleChanges): void {
  console.log('Input property changed:', changes);
}
```

2. ngOnInit()
Purpose: Called once, after the component's data-bound properties have been initialized.

Timing: Invoked after the first ngOnChanges() but before the component is displayed.

Use Case: Initialize the component or set up data fetching.

```
ngOnInit(): void {
  console.log('Component initialized');
}
```

3. ngDoCheck()
Purpose: Called during every change detection run, whether the change was caused by input properties or something else.

Timing: Triggered frequently during change detection.

Use Case: Implement custom change detection logic or optimize checks.

```
ngDoCheck(): void {
  console.log('Custom change detection');
}
```

4. ngAfterContentInit()
Purpose: Called after Angular has fully initialized all the content projected into the component (content inserted into the component using <ng-content>).

Timing: Runs once after the first change detection that includes content projection.

Use Case: Perform actions once the projected content is ready.

```
ngAfterContentInit(): void {
  console.log('Content projected into the component is initialized');
}
```

5. ngAfterContentChecked()
Purpose: Called after every check of the content projected into the component.

Timing: Runs after every change detection that includes projected content.

Use Case: Perform actions when content has been checked (similar to ngDoCheck but specific to projected content).

```
ngAfterContentChecked(): void {
  console.log('Content projection checked');
}
```

6. ngAfterViewInit()
Purpose: Called after the component's view and its child views have been fully initialized.

Timing: Triggered once after the view initialization.

Use Case: Perform DOM manipulations or access child components after the view is fully initialized.

```
ngAfterViewInit(): void {
  console.log('Component view initialized');
}
```

7. ngAfterViewChecked()
Purpose: Called after every check of the component's view and its child views.
Timing: Runs after each change detection affecting the component's view.

Use Case: Perform any actions or adjustments after the view has been checked (similar to ngDoCheck, but for view-related checks).

```
ngAfterViewChecked(): void {
  console.log('Component view checked');
}
```

8. ngOnDestroy()
Purpose: Called right before the component is destroyed.

Timing: Triggered when Angular is about to remove the component from the DOM.

Use Case: Clean up resources like unsubscribing from observables, detaching event listeners, or stopping intervals and timeouts.

```
ngOnDestroy(): void {
  console.log('Component destroyed');
}
```

Synchronous and Asynchronous in Angular

A synchronous operation is one where tasks are executed one after the other, in a sequential manner. Each operation waits for the previous one to complete before starting. If one task takes time (e.g., a heavy computation or a blocking process), the entire process is halted until that task is completed.

Example in Angular: If you have a sequence of functions that execute one after another without waiting for external processes (like a network request), they are synchronous.

1. Synchronous Operations

```
function syncTask() {
  console.log('First Task');
  console.log('Second Task');
}

syncTask();
```

2. Asynchronous Operations

An asynchronous operation is one where tasks are executed independently of each other. It allows other operations to proceed while waiting for an operation (such as a network request or a timeout) to complete. Asynchronous operations are critical for non-blocking behaviors in Angular, especially when dealing with I/O-bound tasks like API requests or timers.

In Angular, asynchronous operations are usually handled via:

Promises: Objects that represent a future result of an operation.
Observables (from RxJS): Streams of data that can be asynchronously processed over time.
Key Examples of Asynchronous Operations in Angular
1. Promises

A Promise in Angular represents a single future value. It can either resolve (success) or reject (failure) after the asynchronous operation completes.

```
let asyncTask = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Task Complete');
  }, 2000);
});

asyncTask.then(result => console.log(result));
```

in this example, the setTimeout() function delays execution for 2 seconds, after which the promise is resolved with a message. The execution of the rest of the application isn't blocked during this delay.

2. Observables (RxJS)

Observables are more powerful than Promises in Angular as they can handle multiple values over time, not just one. Observables are widely used in Angular for handling asynchronous operations like HTTP requests and event-based data streams.

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

export class AppComponent {
  data$: Observable<any>;

  constructor(private http: HttpClient) {}

  fetchData() {
    this.data$ = this.http.get('https://jsonplaceholder.typicode.com/posts');
  }
}
```

Common Use Cases for Asynchronous Operations in Angular

HTTP Requests: Fetching data from a backend service (commonly done using Angular's HttpClient module).

Timers/Intervals: Delaying execution or periodically executing code (e.g., with setTimeout() or setInterval()).

Event Handling: Handling user events like clicks or keypresses asynchronously.

Reactive Forms: Asynchronous form validation (e.g., validating an email with a server-side check).

Lazy Loading: Dynamically loading modules or components when needed to improve performance.

Conclusion

Synchronous operations block the execution flow, and tasks are performed one after the other.

Asynchronous operations in Angular allow tasks to happen independently without blocking the main thread. These operations are crucial for handling non-blocking tasks like HTTP requests, file I/O, or timed actions.

Angular makes heavy use of Promises and Observables (via RxJS) to manage asynchronous behavior, and developers can choose between them based on the complexity of their tasks.

What is Thread? How It execute with Async and Await

A thread is the smallest unit of processing that can be scheduled by an operating system. It represents a single sequence of execution within a process. In simpler terms, threads allow a program to perform multiple operations concurrently, making better use of system resources.

Thread States:

New: A thread that has been created but not yet started.
Runnable: A thread that is ready to run and is waiting for CPU time.
Blocked: A thread that is waiting for a resource or event (like I/O).
Terminated: A thread that has finished execution

Common Methods:

start(): Starts the thread's execution.
run(): Contains the code that runs in the new thread.
join(): Waits for the thread to die (finish execution).
sleep(milliseconds): Puts the thread to sleep for a specified time.
interrupt(): Interrupts the thread, which can be used to signal it to stop.

Interrupt(): Interrupts the thread, which can be used to signal it to stop.
Synchronization: To prevent data corruption when multiple threads access shared resources, synchronization is used.
synchronized: A keyword used to lock a method or block of code so that only one thread can access it at a time.

What is component module and services and how it creates in angular

Component

A component is a class that controls a part of the user interface (UI). It encapsulates the template (HTML), styles (CSS), and behavior (TypeScript) for that UI part.
Component:-ng generate component component-name
ng g c component-name

Module

A module is a container for a cohesive block of code dedicated to a specific application domain, workflow, or set of functionalities. It organizes related components, directives, pipes, and services.
Module:-ng generate module module-name

ng g m module-name

Services

A service is a class that provides functionality, typically for data fetching, business logic, or shared utilities. Services can be injected into components or other services using Angular's dependency injection system.
Services:-ng generate service service-name
ng g s service-name

What is lazy loading in Angular

Lazy loading is a design pattern used to defer the loading of resources until they are actually needed, which can significantly improve the performance and user experience of applications. In the context of Angular, lazy loading allows you to load feature modules on demand rather than loading them all at startup.

Benefits of Lazy Loading
Improved Load Time: Only the essential parts of the application are loaded initially, which reduces the time it takes for the app to become usable.
Reduced Memory Usage: Unused modules are not loaded into memory until needed, which can lower the overall memory footprint.
Better User Experience: Users can start interacting with the application sooner, and only experience delays when navigating to routes that require loading additional modules.

Dependency Injection In Angular

Dependency Injection (DI) in Angular is a design pattern that allows you to create services and inject them into components, directives, or other services. It promotes the separation of concerns and makes your code more modular, testable, and maintainable.
We can achieve dependency injection using services.

Structural Directive 14/10/2024

Structural directives manipulate the DOM layout by adding, removing, or replacing elements, while attribute directives modify the appearance or behavior of elements without affecting their structure. Directives in Angular are nothing but the classes that allow us to add and modify the behavior of elements. Using directives in angular we can modify the **DOM (Document Object Module)** styles, handle user functionality, and much more.

In Angular, structural directives are a type of directive that modify the layout of the DOM by adding, removing, or replacing elements based on certain conditions. We have three built-in structural directives namely : ***ngIf, *ngFor, *ngSwitch.**

Attribute Directives

In Angular, Attribute directives are used within html tags to modify the appearance of elements, components or directives. They are used for Dynamic Styling, DOM Manipulation etc.
Used as attributes in the template. eg : `[ngClass]`, `[ngStyle]`
[ngClass]=is used where we make changes suppose we have to change in this color of div there we use ngclass the function is called on the button
Note:-If we have to use directive u should have to import the common module

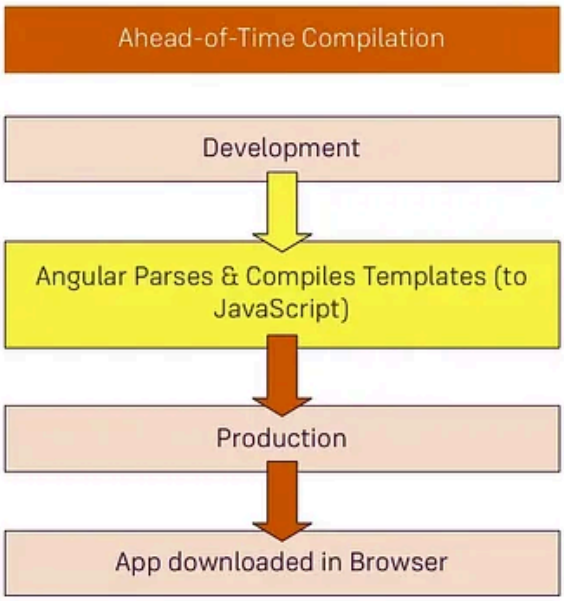
What is JIT & AOT

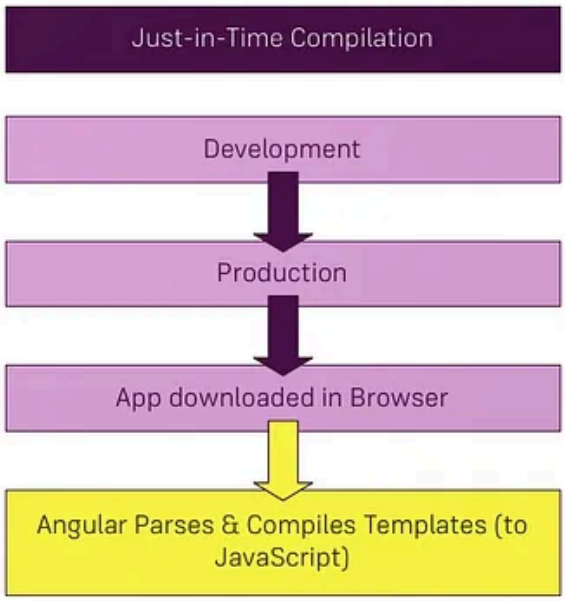
The Ahead Of Time Compiler converts all your Angular HTML and Typescript code to Javascript. That code is then downloaded by the browser and executed. In this type of compilation, the code is already compiled before it is executed for running in the browser. This compiling of the application provides a faster rendering in the browser.

JIT:-Just in time compilation compiles the code at the browser so the error we get at the browser not at compilation time so sometimes it create problem to getting the error directly at browser

Disadvantages:-gets more time to execute
Gets more error at production level

AOT:-Ahead of time compilation executes the code at the time of compilation in terminal so if it some error then we get it directly in the terminal so the performance will increase and time to execute will also be lesser. Try to use AOT always





AOT	JIT
AOT compilation is already done when you build the application. No runtime compilation occurs. No downloading of the compiler occurs.	JIT downloads the compiler and compiles the code right before it is displayed (in the browser).
AOT is **recommended** for the production mode.	JIT is more suitable for development mode since you can see and link your source code in the inspection of the browser because it compiles run time with a map file.
AOT provides a better security since all of the compilation has already occurred during the build of the application so no client side attacks or source code manipulation can be injected.	JIT is a bit riskier since all your components and JS files are compiled right into the browser which can potentially give a pathway to several XSS attacks or any kind of potential injection attacks.
Using AOT you can catch the template error during the building of your application	You can only catch the template binding error during the display time.

Forms **15/10/22**

Template Forms;-

- Used more html we add validation using html

- Used for smaller application
- If we have to add validation in html add #property into the html
- Also we are using form tag then in input we must have to add name property into it
- We can bind with the help of ngModel

Reactive Forms

Import reactive formModule
Create object of the formGroup
Add all property
Add FormGroup into the main Form with its object
In reactive forms formControlName and Name always should match

Imp things While Development

Your console always open
Check console while doing any things

Directive

In Angular, a directive is like a special instruction you can attach to elements in the HTML to change their behavior or appearance. Directives are a key way Angular adds extra features to HTML elements without having to create a whole new component.

Attribute Directives: These modify the behavior or style of an existing HTML element.

- Example: `ngClass` is an attribute directive that lets you add or remove CSS classes based on a condition.

<div [ngClass]="{'highlight': isHighlighted}">This text is conditionally highlighted.

Structural Directives: These add or remove elements from the DOM (the page structure) based on a condition.
<p *ngIf="isLoggedIn">Welcome back, user!</p>

Custom Directives: You can also create your own directives to add unique functionality to HTML elements.

For instance, you could create a custom directive to change the background color of an element when you hover over it.
Why Use Directives?

Directives are useful because they help you:

Reuse common behaviors (like showing or hiding elements).
Make your code cleaner by adding logic directly to HTML elements.
Avoid writing repetitive code, as you can apply the same directive to multiple elements.
Example in Simple Terms
Think of a directive as a tag you put on an element to give it extra powers. If HTML is the skeleton, then Angular directives are like small tools that you attach to the skeleton to make it move, change, or look a certain way based on your instructions.

Attribute Directive

Attribute directives add interactivity and styling to elements dynamically, giving you powerful control over the look and feel of your Angular applications. They can modify classes, styles, or any element properties based on conditions or interactions.

Decorators

Its design pattern that is used to modify or decorate the class without modifying the original source code
Decorators have four type
Property decorator ;- @Input @Output
Method decorator :- @HostListners
Class decorator :- @Component @ngModule they allow angular to tell the class is particular class or module
Parameter decorator :- @inject

NgRx State Management

Step 1:- add packanage ng add @ngrx/store, @ngrx/effetcs, @ngrx/store-devtools
Step 2:- Create interface in model folder
Step 3: Add store, effects devtools into the app.config.ts
Step $:- Add initial data into the reducer and export it. Add the initialdata into the app config in store where we store the temp data // basically add data into the store of app.config.ts
Step 5:- pass the data into the component constructor add render the data into the UI
Step 6:- Add the actions into the action.reducer.ts file after that it dispatch it into the button inceremnet
Step 7 :data comes into the action.ts that dispatch it into the bucket.reducer.ts
Step 8 :add your logic to add data or anything to bucket.reducer.ts
Step 9:- now render your data into the ui by adding it into the constructor
Step 10:- if you want to filter the data like showing specific data then use selector

Angular All Imporant Topic To Cover all Angular Parts

Core Angular Concepts

Components

1. Lifecycle hooks (ngOnInit, ngOnChanges, etc.)
2. Standalone components
3. Component interaction (Input, Output, EventEmitter)

Modules and Routing

1. Angular modules (@NgModule, lazy loading)
2. Routing and navigation (including route guards, child routes)

Templates and Directives

1. Structural directives (*ngIf, *ngFor)

2. Attribute directives ([ngClass], [ngStyle], custom directives)

Forms

1. Template-driven forms
2. Reactive forms
1. Validation (custom validators, async validators)

Dependency Injection (DI)

1. Providers and services (@Injectable, hierarchical DI)
2. Injection tokens

Advanced Angular Features

1. State Management
2. RxJS and Observables
3. NgRx or alternative state management libraries

HTTP Client

1. Making API calls
2. Interceptors for request/response handling
3. Error handling and retry logic
4. Performance Optimization

Component Interaction

Parent to Child Communication

1. What is the purpose of the @Input decorator in Angular?

Answer:
The @Input decorator in Angular is used to pass data from a Parent Component to a Child Component. It allows the parent to bind a property or value to a child component, enabling communication between the two components.

```
// Parent Component
<app-child [data]="parentData"></app-child>

// Child Component
@Input() data: string;
```

2. Can we pass multiple inputs to the child component? How?

Answer:
Yes, we can pass multiple inputs to a child component by defining multiple `@Input` properties in the child component. Each property can then be bound separately in the parent component.

// Parent Component

<app-child [name]="productName" [price]="productPrice"></app-child>

// Child Component

@Input() name: string;

@Input() price: number;

Key Points:

Each input property in the child corresponds to a binding in the parent.

Ensure type safety by defining the correct data types for @Input properties.

3. How do you handle a scenario where no data is passed to the child component?

Answer:
To handle this scenario, you can:

1. **Set default values** for the `@Input` properties in the child component.
2. Use `*ngIf` in the child template to display a fallback message if the data is not available.

// Child Component

@Input() data: string = 'Default Value';

// Template

<p *ngIf="data; else noData">Data: {{ data }}</p>

<ng-template #noData>No data provided.</ng-template>

Key Points:

Default values ensure the child does not break if no data is passed.

Using *ngIf prevents displaying undefined or null values.

4. What are some use cases where parent-to-child communication is the best choice?

Answer:

Parent-to-child communication is best suited for scenarios where:

Data Propagation: The parent needs to provide data for display or processing in the child, e.g., a list of items to a child table component.

Reusable Components: A reusable child component requires configurable inputs, e.g., passing configuration options or styles.

Dynamic Updates: The parent dynamically changes data, and the child reflects these updates in real time.

State Management: Simple data-sharing patterns where the parent holds the main state and passes it down to children.

3.Component to Component Communication:-Subject and Behaviour Subject

What is Observable:-

In Angular, an observable is a data stream that can emit multiple values over time. They are used to handle asynchronous operations, such as HTTP requests, event handling, and real-time data

How observables work

- Observables are part of the Reactive Extensions for JavaScript (RxJS) library.
- Observables follow the observer design pattern, where an observable produces values and observers react to them.
- You can subscribe to an observable to listen to the values it emits.
- You can unsubscribe from an observable to stop receiving values.

1. Observable and Observer in Angular: Study Notes

1.1. What is an Observable?

Definition: An Observable is a data stream that can emit multiple values over time. It is the source of data or events in the reactive programming model. In Angular, observables are commonly used with RxJS to manage asynchronous tasks such as HTTP requests, user input, or events.

Characteristics:

Lazy: It doesn't do anything until an Observer subscribes to it.

Multiple Emissions: It can emit many values over time (e.g., HTTP response, user clicks, time intervals).

Can Emit:

A single value (such as an HTTP response).

Multiple values (like a stream of user inputs).

Error (if something goes wrong).

Completion (when no more values will be emitted).

1.2. What is an Observer?

Definition: An Observer is an entity that subscribes to an Observable to receive notifications whenever the Observable emits a new value. The Observer defines how to handle emitted values with three main methods.

next(value): Handles the emitted value.

error(err): Handles any errors emitted by the Observable.

complete(): Called when the Observable completes (emits all values).

Real-Life Example:

Observer is like a YouTube viewer who subscribes to a channel to get notifications every time a new video is uploaded.

1.3. Key Terms

Observable:

A stream of data that can emit values over time.

Observable is lazy and starts emitting only when an Observer subscribes.

Observer:

A listener that subscribes to an Observable and reacts to the emitted values.

An Observer can handle next(), error(), and complete() notifications.

Subscription:

A subscription is created when an Observer subscribes to an Observable.

You can unsubscribe from the Observable if you no longer want to receive updates.

1.4. Working with Observables in Angular

Angular makes heavy use of RxJS Observables, especially when dealing with HTTP requests, user inputs, and other asynchronous operations.

Common Operators in RxJS:

map: Transforms the emitted values.

filter: Filters values based on conditions.

mergeMap: Flattens multiple inner observables.

catchError: Handles errors in the stream.

4. Real-Life Analogies to Remember

Observable = YouTube Channel

A YouTube channel (Observable) creates and posts videos (emits values) regularly.

Observer = You (The Viewer)

You (Observer) subscribe to the channel to get notifications when new videos are posted.

Subscription = Subscribing to the Channel

By subscribing, you start receiving new videos. If you unsubscribe, you stop receiving notifications.

What is a BehaviorSubject? [Youtube Video] [Medium Article]

At its core, a BehaviorSubject is a type of Observable provided by the RxJS library. Unlike traditional Observables that emit values only upon specific events, BehaviorSubject maintains the latest value it has emitted and immediately dispatches it to new subscribers upon subscription.

Difference Between Subject and BehaviorSubject in Angular (RxJS)

In RxJS (Reactive Extensions for JavaScript), both `Subject` and `BehaviorSubject` are types of **Observables** that allow multiple subscribers to listen to data. However, they have key differences in how they handle emitted values and how they behave with new subscribers.

Here's a detailed breakdown of the differences:

---

1. Subject

Definition:

- A **Subject** is a type of Observable that allows you to **multicast** values to multiple subscribers.
- It **doesn't have an initial value**, and it only emits the values to subscribers that **subscribe after** a value is emitted.

Key Points:

- **No initial value**: A `Subject` does not have a default or initial value when it's created.
- **Doesn't cache previous values**: If you subscribe to a `Subject` after a value has been emitted, the new subscriber will **not** receive any previous values, only the values that are emitted after subscription.
- **Multiple subscribers**: Any value emitted by the `Subject` will be passed to **all active subscribers** at that time.

Real-Life Analogy:

- **Subject** is like a **group chat** where each message you send only appears to people who are currently in the chat. If someone joins the chat later, they won't see any past messages.

Example:

```
import { Subject } from 'rxjs';

const subject = new Subject();

// First subscriber

subject.subscribe(value => console.log('Subscriber 1:', value));

// Emitting values

subject.next('Hello');

// Second subscriber

subject.subscribe(value => console.log('Subscriber 2:', value));

// Emitting another value

subject.next('World');
```

Output:

```
Subscriber 1: Hello

Subscriber 1: World

Subscriber 2: World
```

- **Explanation**: The second subscriber only gets the value emitted after it subscribes. It doesn't get `'Hello'` because it subscribes after that value was emitted.

---

2. BehaviorSubject

Definition:

- A **BehaviorSubject** is a **specialized version** of `Subject` that **requires an initial value** and **always emits the current value** to new subscribers, even if they subscribe after a value has been emitted.

Key Points:

- **Initial value**: You must provide an initial value when creating a `BehaviorSubject`.
- **Cache the last emitted value**: A `BehaviorSubject` stores the **most recent** value and will immediately send that value to any new subscriber, regardless of when they subscribe.
- **Value-based subscription**: The `BehaviorSubject` always has the **latest value** which will be emitted to new subscribers when they subscribe.

Real-Life Analogy:

- **BehaviorSubject** is like a **news feed** that always shows the most recent post to anyone who joins the feed. Even if you join late, you'll see the latest post immediately.

Example:

```
import { BehaviorSubject } from 'rxjs';

const behaviorSubject = new BehaviorSubject('Initial Value');

// First subscriber

behaviorSubject.subscribe(value    =>    console.log('Subscriber    1:',
value));

// Emitting values

behaviorSubject.next('New Value');

// Second subscriber

behaviorSubject.subscribe(value    =>    console.log('Subscriber    2:',
value));

// Emitting another value

behaviorSubject.next('Another New Value');
```

Output:

```
Subscriber 1: Initial Value

Subscriber 1: New Value
```

```
Subscriber 2: New Value

Subscriber 1: Another New Value

Subscriber 2: Another New Value
```

- **Explanation**: The second subscriber gets the latest value (`'New Value'`) immediately upon subscribing. It doesn't miss the previously emitted values like with a regular `Subject`.

---

3. Key Differences Between Subject and BehaviorSubject

Feature	Subject	BehaviorSubject
Initial Value	No initial value (undefined initially).	Requires an initial value.
Emitted Values	Doesn't remember previous values. Only emits new values to new subscribers.	Remembers and emits the **last emitted value** to new subscribers.
New Subscriber Behavior	A new subscriber only receives values emitted **after** they subscribe.	A new subscriber immediately receives the **last emitted value**.
Use Case	Useful for broadcasting multiple values or events in real-time, like a simple message bus.	Useful for managing state, where you want subscribers to always have the current value (e.g., form inputs, settings, etc.).

---

4. When to Use Each

- **Use a `Subject`**:
  - When you need to **broadcast events or messages** to multiple observers, but you don't care about the last emitted value.
  - Example: Broadcasting clicks, or broadcasting the status of an ongoing process (where the last emitted value is not important).
- **Use a `BehaviorSubject`**:
  - When you need to **remember the last emitted value** and send that value to any new subscribers.
  - Example: Managing state (e.g., current user data, form values, application settings), where you want the most recent state to be available for new subscribers.

Ng-template and Ng-decorator [youtube]

1. Understand ng-template and ng-container (Theoretical Overview)

ng-template

Definition: ng-template is an Angular directive that defines a template for later rendering. It is not rendered immediately but can be rendered dynamically in certain scenarios. It is used for structural directives (like *ngIf, *ngFor, *ngSwitch).

Purpose: ng-template helps in creating reusable chunks of DOM that Angular can render when needed. It enables lazy loading or deferred rendering of a portion of the view.

Example Usage:

```
<ng-template #myTemplate>
  <div>This content is inside the template!</div>
</ng-template>
<button (click)="showTemplate = !showTemplate">Toggle Template</button>
<ng-container *ngIf="showTemplate; else myTemplate">
```

```
  <div>Template is not shown directly.</div>

</ng-container>
```

ng-container

Definition: ng-container is a structural directive that doesn't add extra DOM elements but acts as a wrapper for other elements. It’s often used when you need to apply directives without introducing additional HTML nodes.

Purpose: It allows you to group elements without changing the DOM structure, and it's useful when you need to apply Angular structural directives (*ngIf, *ngFor, etc.) to multiple elements or components without cluttering the DOM.

Example Usage:

```
<ng-container *ngIf="condition">

  <p>This content will be displayed only if the condition is true</p>

  <button>Click me</button>

</ng-container>
```

2. Learn ng-template and ng-container Practically

A. Set Up Your Angular Project

To start working with ng-template and ng-container, you’ll need to have an Angular project. Here’s how to set one up:

Install Angular CLI:

```
npm install -g @angular/cli
```

Create a new Angular project:

```
ng new angular-ng-template-container

cd angular-ng-template-container
```

Serve the project:

```
ng serve
```

B. Implement ng-template and ng-container in Components

Using ng-template: You can use ng-template to defer rendering content, create reusable sections, and display content conditionally.

Example:

```
<!-- app.component.html -->

<div>

  <button (click)="showTemplate = !showTemplate">Toggle Template</button>

  <ng-template #myTemplate>

    <div>Template content is now rendered!</div>

  </ng-template>

  <ng-container *ngIf="showTemplate; else myTemplate">

    <div>This is an alternative content</div>

  </ng-container>

</div>
```

Component logic:

typescript

Copy code

```
export class AppComponent {

  showTemplate = false;
```

}Using ng-container: ng-container doesn’t render an additional element but allows you to apply directives to multiple child elements.

Example:

```
<!-- app.component.html -->

<div>
```

```
  <ng-container *ngIf="showMessage">
    <h2>Welcome User</h2>
    <p>This content is displayed conditionally with ng-container.</p>
  </ng-container>
</div>
```

Component logic:

typescript

```
export class AppComponent {
  showMessage = true;
}
```

C. Experiment with Complex Examples

Try combining both ng-template and ng-container in more complex scenarios like displaying a list of items, conditional rendering, and using them with *ngIf and *ngFor.

For instance:

```
<div *ngIf="items.length > 0; else noItems">
  <ul>
    <ng-container *ngFor="let item of items">
      <li>{{ item }}</li>
    </ng-container>
  </ul>
</div>


<ng-template #noItems>
  <p>No items available</p>
```

</ng-template>

3. Study Practical Use-Cases

Look into Angular documentation and tutorials on how these directives are used in real-world applications, such as:

Lazy loading components.

Dynamic content rendering in templates.

Complex UI components that need conditional rendering without adding extra markup.

4. Explore More Advanced Topics

After learning the basics, move on to more advanced topics like:

Dynamic template rendering: Using ng-template with ViewContainerRef and TemplateRef.

Structural directives with ng-template: Create custom structural directives.

Performance optimizations: Minimize unnecessary DOM manipulations.

5. Resources to Learn

Angular Documentation:

Tutorials and Blogs: Look for blog posts and tutorials that show how ng-template and ng-container are used in Angular applications. Sites like dev.to, Medium, and StackBlitz are great places to find examples.

YouTube: Watch video tutorials on Angular by experienced developers. Channels like "Academind", "Traversy Media", and "Programming with Mosh" have tutorials on Angular fundamentals and advanced topics.