**SETUP:**

1. Angular tutorial - Prat 1 -setup + project creation
2. Install Node JS : https://nodejs.org/en/download
3. check if Node & NPM installed or not : npm -v & node -v
4. Install angular cli - npm i @angular/cli
5. Check if it installed : ng version
6. https://angular.dev/tutorials/learn-angular : angular website
7. create new project : ng new angular-18-tutorial
8. GitHub repo : https://github.com/rawatankita/Angular\_18\_tutorial
9. npm install
10. ng serve --o
11. multiple port : ng serve --port 4204

-------------------------------

**Setup questions:**

✔ Which stylesheet format would you like to use? CSS [ https://developer.mozilla.org/docs/Web/CSS ] ✔ Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)? No ? Do you want to create a 'zoneless' application without zone.js (Developer Preview)?

**Answer**:

Great question! A zoneless Angular application is one that runs without zone.js, which traditionally handles change detection in Angular. This new approach is part of Angular’s Developer Preview and aims to improve performance, debugging, and compatibility.

🚀 Why Go Zoneless?

Better performance: Avoids unnecessary change detection cycles.

Smaller bundle size: Removes zone.js from your build.

Improved debugging: Cleaner stack traces.

Future-ready: Avoids monkey-patching browser APIs, which can break with newer features.

--------------------------------

**new changes in angular 18+**

no app.module.ts file : changed to app.config.ts

zoneless change detection

app.route.ts : constant file

-----------------------------

**Angular 18 tutorial part 2:**

**Project files and structure:**

package.json : (dependencies and packages)

angular.json : (project setup/ architecture file)

style.css

gitignore

tsconfig.json : (typescript configuration)

main.ts ( first file to execute when app start : bootstarap app)

index.html (Single page application one file only index.html load)

app.config.ts

--------------------------------

**package.json**

dependency : ( part of angular build)

dev dependency : (to run angular project locally and not part of your build)

-------------------------------

🧭 1. main.ts — The True Entry Point

Located in src/main.ts, this file is where Angular begins execution.

---------------------------------

**Angular 20 highlights**

You're absolutely right to notice a change in Angular 20's structure. In earlier versions, the angular.json file explicitly listed the path to index.html under the build options. But in Angular 20, the configuration has evolved.

🧾 What's New in Angular 20?

According to the official Angular workspace configuration guide:

The angular.json file still exists and controls build and development settings.

However, the index option may not be explicitly listed unless customized.

Angular CLI assumes default paths unless overridden.

🗂 Default Behavior

If you don’t see index.html in angular.json, it’s because Angular 20 uses convention over configuration:

It assumes your index.html is located at src/index.html.

Unless you change it, there's no need to declare it explicitly.

🛠 How to Customize It (If Needed)

You can still manually add or override the index file path like this:

json

"projects": {

"your-app-name": {

"architect": {

"build": {

"options": {

"index": "src/custom-index.html"

}

}

}

}

}

You're absolutely right to notice that in Angular 20, the standalone: true flag might not appear explicitly in newly generated components—and that's by design.

🧠 What's Happening in Angular 20?

Starting with Angular 19 and continuing into Angular 20:

Standalone components are now the default.

That means when you generate a component using:

ng generate component my-component

You're absolutely on point—Angular 20 has introduced a major shift in naming conventions and project structure, which is why you're not seeing app.component.ts anymore.

**🚨 What's Changed in Angular 20?**

Angular 20 has **dropped the** .component.ts **naming convention** for newly generated files. Instead:

* The main component is now simply named app.ts.
* This change aligns with Angular’s push toward **standalone components** and a cleaner, more modern file structure.

So instead of:

Code

src/app/app.component.ts

You’ll now see:

Code

src/app/app.ts

**🧩 Why This Might Cause Issues**

Some tools (like Visual Studio or older build setups) still expect the old app.component.ts file. This can lead to errors like:

"Could not find file 'app.component.ts'" or "File 'src/app/app.component.ts' is missing from the TypeScript compilation"

These are compatibility issues with tooling that hasn’t yet adapted to Angular 20’s new structure2.

**🛠 How to Fix or Work Around It**

1. **Use Angular 19 if your tooling depends on the old structure**:

bash

npm install -g @angular/cli@19

1. **Re-enable old naming conventions in Angular 20**: Update your angular.json to use legacy schematic settings:

json

"schematics": {

"@schematics/angular:component": {

"style": "css",

"skipTests": false,

"prefix": "app",

"namePattern": "component"

}

}

1. **Manually rename files** if needed and update imports accordingly.

**Components**

1. How to create component?

ng g c component

1. What is component?

In Angular, a **component** is the fundamental building block of the user interface. It encapsulates everything needed to render and manage a piece of the UI—like a button, form, or entire page.

**🧱 Anatomy of an Angular Component**

Each component typically includes:

* **TypeScript class**: Defines the logic and behaviour.
* **HTML template**: Describes what appears on the screen.
* **CSS styles**: Controls the visual appearance.
* **Selector**: A custom HTML tag used to embed the component.

1. What is standalone component?

These are default type of component which are not part of any other component until we import them. In angular 18 we have “standalone: true” option while creating component but from angular 20 onward standalone option is now by default and no need to write it explicitly in imports: [] array @component decorator.

1. What are Component Decorators?

@Component decorator in Angular is a special TypeScript annotation that tells Angular: **“This class is a component.”** It provides metadata that Angular uses to create, render, and manage the component at runtime.

**Data Binding**

1. Types of Data Bindings?

One way binding, Two-way binding, Signals

**On way Binding:**

* + From .ts to html
  + Interpolation
  + Property Binding
  + From html to .ts Event binding

**Two-way Binding:**

* + Using [(ngModel)]

**Signals (Angular 17 onwards)**

⚡ What Is a Signal?

A signal is a reactive primitive introduced in Angular 16+ that holds a value and notifies the UI when that value changes. Think of it like a smarter, leaner version of BehaviorSubject or NgRx, but built right into Angular.

✅ Why Use Signals?

1. Automatic UI Updates

When you change a signal’s value, Angular automatically updates the DOM—no need for manual change detection or async pipes.

ts

count = signal(0);

this.count.set(this.count() + 1); // UI updates instantly

2. Simpler State Management

Signals can replace complex RxJS setups for local component state. No subscriptions, no memory leaks.

3. Better Performance

Signals are fine-tuned for Angular’s rendering engine. They reduce overhead by tracking dependencies precisely—only re-rendering what’s needed.

4. Composable Logic

You can derive new signals from existing ones:

ts

fullName = computed(() => `${this.firstName()} ${this.lastName()}`);

5. Cleaner Templates

No need for async pipe or ngIf gymnastics. Just call the signal like a function:

html

<p>{{ fullName() }}</p> (called as a method)

🧠 When Should You Use Signals?

* For local component state (like form inputs, counters, toggles)
* When you want fine-grained reactivity without external libraries
* To simplify derived values and reactive expressions
* In Angular standalone components or modern apps using the latest features

**ngmodel and signal difference**

**🧠 ngModel: Classic Two-Way Binding**

* **What it is**: A directive from FormsModule that enables two-way binding between form inputs and component properties.
* **Syntax**: [(ngModel)]="value"
* **Use case**: Ideal for template-driven forms and simple input binding.
* **How it works**: Updates the component property when the user types, and vice versa.

ts

export class MyComponent {

name: string = 'Angular';

}

html

<input [(ngModel)]="name" />

<p>{{ name }}</p>

**⚡ signal: Modern Reactive Primitive**

* **What it is**: A reactive value introduced in Angular 16+ that automatically triggers UI updates when changed.
* **Syntax**: name = signal('Angular')
* **Use case**: Great for reactive state management, computed values, and fine-grained reactivity.
* **How it works**: You access the value with name() and update it with name.set('New Value').

ts

export class MyComponent {

name = signal('Angular');

}

html

<input [value]="name()" (input)="name.set($event.target.value)" />

<p>{{ name() }}</p>

**🔄 Can You Use ngModel with Signals?**

Yes! You can bind ngModel to a **writable signal**, but you need to handle the getter/setter manually:

html

<input [ngModel]="name()" (ngModelChange)="name.set($event)" />

**Directives [change behaviours of DOM element]**

* + - 1. Structural directive [ Change structure of DOM]

\*ngIf - >Add / remove element from the DOM

\*ngFor -> dynamic DOM structure update

\*ngSwitch -> Switch on basis of conditions

Angular 18+ updates : need to import commonModule separately to use directives

1. Attribute Directives [ Change style of DOM element]

ngStyle

ngClass

1. Component Directives
2. Custom Directives

Angular has **three main types of directives**, each serving a distinct purpose in how you build and control your application's behavior and layout:

**🧱 1. Component Directives**

* **Definition**: These are directives with a template. In fact, every Angular component is a directive.
* **Purpose**: They define UI blocks and encapsulate logic, styles, and templates.
* **Example**:

ts

@Component({

selector: 'app-user',

templateUrl: './user.component.html'

})

export class UserComponent {}

**🎨 2. Attribute Directives**

* **Definition**: These modify the appearance or behavior of an element.
* **Purpose**: Used to dynamically change styles, classes, or properties.
* **Common Built-in Examples**:
  + ngClass: Adds/removes CSS classes.
  + ngStyle: Applies inline styles.
  + ngModel: Enables two-way data binding.
* **Example**:

html

<div [ngClass]="{ 'highlight': isActive }"></div>

**🏗️ 3. Structural Directives**

* **Definition**: These change the DOM layout by adding or removing elements.
* **Purpose**: Control rendering based on conditions or loops.
* **Common Built-in Examples**:
  + \*ngIf: Conditionally includes a template.
  + \*ngFor: Loops over a collection.
  + \*ngSwitch: Switch-case rendering.
* **Example**:

html

<div \*ngIf="isLoggedIn">Welcome back!</div>

**🛠️ Bonus: Custom Directives**

You can create your own **attribute or structural directives** to encapsulate reusable logic.

ts

@Directive({

selector: '[appHighlight]'

})

export class HighlightDirective {

constructor(private el: ElementRef) {

el.nativeElement.style.backgroundColor = 'yellow';

}

}

For a deeper dive into Angular directives, check out the official Angular guide on directives.

Want to try building a custom directive together?

**Routing**

* What is routing?

Routing in Angular allows you to **navigate between different components or views** based on the URL path—without reloading the page. It’s the backbone of SPAs, enabling seamless transitions like going from /home to /about without a full refresh.

* What is route object?

A **route object** defines a mapping between a URL path and the component that should be displayed when that path is accessed.

const routes: Routes = [

{ path: 'home', component: HomeComponent },

{ path: 'about', component: AboutComponent }

];

Each object can include:

path: URL fragment

component: Component to render

Optional: redirectTo, pathMatch, children, canActivate, etc.

* How to create route?

Creating a route in **Angular 20** is simpler and cleaner than ever, thanks to the shift toward **standalone components** and the use of provideRouter. Here's a step-by-step guide to help you set up routing in a modern Angular 20 project:

**🛠️ Step 1: Create Components**

Use Angular CLI to generate components:

bash

ng generate component Home

ng generate component About

These will be **standalone components** by default in Angular 20.

**📦 Step 2: Define Routes**

Create a file called app.routes.ts in your src/app folder:

ts

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

import { HomeComponent } from './home/home.component';

import { AboutComponent } from './about/about.component';

export const routes: Routes = [

{ path: '', component: HomeComponent },

{ path: 'about', component: AboutComponent }

];

**⚙️ Step 3: Configure the Router**

In main.ts, use provideRouter to register your routes:

ts

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

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

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

import { routes } from './app/app.routes';

bootstrapApplication(AppComponent, {

providers: [provideRouter(routes)]

});

This replaces the older AppRoutingModule approach.

**🧭 Step 4: Add <router-outlet> to Your Template**

In app.component.html, insert:

html

<router-outlet></router-outlet>

This is where routed components will be displayed.

**✅ Optional: Add Navigation Links**

In your navbar or any component:

html

<nav>

<a routerLink="/">Home</a>

<a routerLink="/about">About</a>

</nav>

* How to create navbar?

Use routerLink to link to routes:

html

<nav>

<a routerLink="/home" routerLinkActive="active">Home</a>

<a routerLink="/about" routerLinkActive="active">About</a>

</nav>

* How to redirect from one component to another?
  1. From .ts
  2. From html

**1️⃣ From .ts (TypeScript)**

Use Angular’s Router service:

ts

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

constructor(private router: Router) {}

goToAbout() {

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

}

You can also pass route parameters:

ts

this.router.navigate(['/user', userId]);

**2️⃣ From HTML**

Use [routerLink] directive:

html

<button [routerLink]="['/about']">Go to About</button>

With parameters:

html

<a [routerLink]="['/user', userId]">View Profile</a>

**Control Flow statements (Angular 17 onwards)**

How to write if, else if, switch, for

Angular now supports a powerful and intuitive **control flow syntax** using @-prefixed blocks, introduced in Angular 17 and refined in Angular 20. These replace traditional structural directives like \*ngIf, \*ngFor, and \*ngSwitch, offering better readability and performance.

**🔁 Core Control Flow Statements in Angular**

**1. @if, @else if, @else – Conditional Rendering**

html

@if (a > b) {

<p>{{ a }} is greater than {{ b }}</p>

} @else if (b > a) {

<p>{{ a }} is less than {{ b }}</p>

} @else {

<p>{{ a }} is equal to {{ b }}</p>

}

* Cleaner than \*ngIf
* Supports multiple branches
* You can also alias values: @if (user.profile.settings.startDate; as startDate) { {{ startDate }} }

**2. @for – Looping Over Collections**

html

@for (item of items; track item.id) {

<li>{{ item.name }}</li>

} @empty {

<li>No items found.</li>

}

* Replaces \*ngFor
* track helps optimize DOM updates
* Contextual variables like $index, $even, $odd, $first, $last are available

**3. @switch, @case, @default – Switch Logic**

html

@switch (status) {

@case ('loading') {

<p>Loading...</p>

}

@case ('error') {

<p>Error occurred.</p>

}

@default {

<p>All good!</p>

}

}

* Inspired by JavaScript’s switch
* Great for rendering based on discrete states

These new control flow blocks are part of Angular’s push toward **signal-based reactivity** and **template clarity**

**We don’t have to use CommonModule with control flow statements**

**Pipe**

Angular pipes are a powerful feature that let you transform data directly in your templates—like formatting dates, numbers, or strings—without cluttering your component logic. Think of them as elegant little filters that make your UI smarter and cleaner.

🧪 What Is an Angular Pipe?

A pipe takes in data as input and returns a transformed version of that data. You use it in templates with the pipe operator |.

Example:

html

<p>{{ today | date:'fullDate' }}</p>

🧰 Built-in Pipes in Angular

Angular provides a rich set of built-in pipes out of the box:

| Pipe Name | Purpose |
| --- | --- |
| DatePipe | Formats a date value based on locale |
| CurrencyPipe | Converts a number into a currency string |
| DecimalPipe | Formats a number with decimal points |
| PercentPipe | Converts a number into a percentage |
| UpperCasePipe | Transforms text to uppercase |
| LowerCasePipe | Transforms text to lowercase |
| TitleCasePipe | Capitalizes the first letter of each word |
| SlicePipe | Extracts a portion of a string or array |
| JsonPipe | Converts an object into a JSON string (great for debugging) |
| KeyValuePipe | Converts an object or Map into an array of key-value pairs |
| AsyncPipe | Automatically subscribes to Observables or Promises |
| I18nPluralPipe | Handles pluralization based on locale |
| I18nSelectPipe | Selects a value based on a key (useful for gender or language variants) |

You can explore more on Angular’s official pipe guide.

🧪 Custom Pipes

If the built-in ones don’t cut it, you can create your own!

Steps to create a custom pipe:

1. Use the @Pipe decorator.
2. Implement the PipeTransform interface.
3. Define your transformation logic.

Example:

ts

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

@Pipe({ name: 'reverse' })

export class ReversePipe implements PipeTransform {

transform(value: string): string {

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

}

}

Then use it in your template:

html

<p>{{ 'Angular' | reverse }}</p> <!-- Output: ralugnA -->

**Types of Pipe**

* + - 1. Built-in Pipes
      2. Custom Pipes
      3. Pure vs Impure Pipes

1. ith pu

| **Type** | **Behaviour** |
| --- | --- |
| **Pure** | Executes only when input changes (default). Ideal for performance. |
| **Impure** | Executes on every change detection cycle. Use cautiously for dynamic data. |

You can mark a pipe as impure like this:

@Pipe({ name: 'impurePipe', pure: false })

🧪 Impure Pipe Example: FilterPipe

Let’s say you want to filter a list of items based on a search term, and the list might be updated dynamically.

1. Create the Pipe

ts

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

@Pipe({

name: 'filter',

pure: false // 👈 This makes it impure

})

export class FilterPipe implements PipeTransform {

transform(items: string[], searchText: string): string[] {

if (!items || !searchText) return items;

return items.filter(item =>

item.toLowerCase().includes(searchText.toLowerCase())

);

}

}

2. Use It in a Component Template

html

<input [(ngModel)]="searchText" placeholder="Search items" />

<ul>

<li \*ngFor="let item of items | filter:searchText">

{{ item }}

</li>

</ul>

⚠️ Why Mark It as Impure?

If items is a reference to a mutable array (e.g. items are added or removed without replacing the array), Angular won’t detect changes unless the pipe is impure. By setting pure: false, the pipe re-runs on every change detection cycle, ensuring the UI stays in sync.

Impure pipes can impact performance because they run frequently. Use them only when necessary, and avoid complex logic inside them.

async pipe

The async pipe in Angular is a powerful tool for handling asynchronous data streams—like **Observables** or **Promises**—directly in your templates. It automatically subscribes to the data source and updates the view when new values arrive, and it also handles unsubscribing when the component is destroyed. No manual subscription management needed!

**Forms**

Template driven forms:

More code in template less in component

More code in html

Validation is easy

ngModel

**Setup Steps**

1. **Import** FormsModule in your app module or standalone component:

ts

import { FormsModule } from '@angular/forms';

1. **Create the form in your template**:

html

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

<label>Name:</label>

<input type="text" name="name" [(ngModel)]="user.name" required>

<div \*ngIf="userForm.controls['name']?.invalid && userForm.controls['name']?.touched">

Name is required.

</div>

<label>Email:</label>

<input type="email" name="email" [(ngModel)]="user.email" required email>

<button type="submit" [disabled]="userForm.invalid">Submit</button>

</form>

1. **Handle the form in your component**:

ts

export class UserComponent {

user = {

name: '',

email: ''

};

onSubmit(form: any) {

console.log('Form Submitted!', form.value);

}

}

2. Reactive forms

Core Building Blocks

| Component | Description |
| --- | --- |
| FormControl | Represents a single input field and tracks its value and validation status |
| FormGroup | A collection of FormControls, grouped together logically |
| FormArray | A dynamic array of FormControls or FormGroups |
| Validators | Built-in or custom functions to validate form inputs |

✅ Benefits

* Full control over form logic
* Easier unit testing
* Reactive updates via observables
* Dynamic form creation

**What Is FormArray?**

FormArray is used when you need a **dynamic list of form controls**, such as:

* Multiple phone numbers
* A list of tasks
* Editable rows in a table

Unlike FormGroup, which uses named keys, FormArray uses **indexed positions**.

**What Is FormArray?**

FormArray is used when you need a **dynamic list of form controls**, such as:

* Multiple phone numbers
* A list of tasks
* Editable rows in a table

Unlike FormGroup, which uses named keys, FormArray uses **indexed positions**.

**🛠️ Example: Dynamic List of Skills**

ts

import { FormBuilder, FormGroup, FormArray, FormControl } from '@angular/forms';

export class ProfileComponent {

profileForm: FormGroup;

constructor(private fb: FormBuilder) {

this.profileForm = this.fb.group({

name: [''],

skills: this.fb.array([this.fb.control('')]) // FormArray with one control

});

}

get skills() {

return this.profileForm.get('skills') as FormArray;

}

addSkill() {

this.skills.push(this.fb.control(''));

}

removeSkill(index: number) {

this.skills.removeAt(index);

}

}

**🧾 Template**

html

<form [formGroup]="profileForm">

<input formControlName="name" placeholder="Name">

<div formArrayName="skills">

<div \*ngFor="let skill of skills.controls; let i = index">

<input [formControlName]="i" placeholder="Skill {{ i + 1 }}">

<button (click)="removeSkill(i)">Remove</button>

</div>

</div>

<button (click)="addSkill()">Add Skill</button>

</form>

**🚀 What Is FormBuilder?**

FormBuilder is a **service** provided by Angular that simplifies the creation of:

* FormControl
* FormGroup
* FormArray

Instead of manually instantiating each form control, FormBuilder lets you define your form structure in a clean, declarative way.

**🧪 Without vs With FormBuilder**

**❌ Without FormBuilder**

ts

this.profileForm = new FormGroup({

name: new FormControl(''),

email: new FormControl('')

});

**✅ With FormBuilder**

ts

constructor(private fb: FormBuilder) {

this.profileForm = this.fb.group({

name: [''],

email: ['']

});

}

See how much cleaner that is? You can also add validators directly:

ts

this.profileForm = this.fb.group({

name: ['', Validators.required],

email: ['', [Validators.required, Validators.email]]

});

**🔄 Key Methods: below**

| **Method** | **Purpose** |
| --- | --- |
| group() | Creates a FormGroup |
| control() | Creates a FormControl |
| array() | Creates a FormArray |

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

import { FormBuilder, FormGroup, FormArray, FormControl, Validators } from '@angular/forms';

TS:

@Component({

selector: 'app-profile-form',

templateUrl: './profile-form.component.html'

})

export class ProfileFormComponent {

profileForm: FormGroup;

constructor(private fb: FormBuilder) {

this.profileForm = this.fb.group({

name: this.fb.control('', Validators.required), // FormControl

email: this.fb.control('', [Validators.required, Validators.email]), // FormControl

hobbies: this.fb.array([ // FormArray

this.fb.control('Reading'),

this.fb.control('Cooking')

])

});

}

get hobbies(): FormArray {

return this.profileForm.get('hobbies') as FormArray;

}

addHobby() {

this.hobbies.push(this.fb.control(''));

}

removeHobby(index: number) {

this.hobbies.removeAt(index);

}

}

HTML:

<form [formGroup]="profileForm">

<label>Name:</label>

<input formControlName="name">

<label>Email:</label>

<input formControlName="email">

<div formArrayName="hobbies">

<label>Hobbies:</label>

<div \*ngFor="let hobby of hobbies.controls; let i = index">

<input [formControlName]="i">

<button (click)="removeHobby(i)">Remove</button>

</div>

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

</div>

<button type="submit">Submit</button>

</form>

**Http Client API Integration**

**🌐 What is HttpClient?**

**HttpClient** is a tool or library used in programming to send HTTP requests and receive HTTP responses from a server. It’s like a messenger that helps your app talk to other web services.

* In **Java**, it's part of the java.net.http package.
* In **.NET**, it's part of the System.Net.Http namespace.
* In **JavaScript**, you often use fetch() or libraries like Axios, which act as HttpClients.

**Use case:** If your app needs to get weather data from an external API, you'd use HttpClient to send a request and handle the response.

**🔁 What are GET, POST, PUT, DELETE?**

These are **HTTP methods**—ways to interact with resources on a server. Think of them as verbs in a conversation between your app and a web service:

| **Method** | **Purpose** | **Example Use Case** |
| --- | --- | --- |
| **GET** | Retrieve data | Get a list of users |
| **POST** | Send new data | Create a new user |
| **PUT** | Update existing data | Edit a user's profile |
| **DELETE** | Remove data | Delete a user account |

Each method tells the server what kind of action you want to perform.

**🧠 What is an API?**

**API** stands for **Application Programming Interface**. It’s a set of rules that lets different software systems communicate with each other.

* Think of it as a **menu in a restaurant**: the menu (API) tells you what you can order (functions), and the kitchen (server) prepares it for you.
* APIs can be **web-based**, allowing apps to talk over the internet using HTTP.

**Example:** A weather app uses an API to fetch temperature data from a weather service.

An API (Application Programming Interface) is a set of rules that lets software applications talk to each other. In Node.js, APIs are often built to:

* Handle HTTP requests (like GET, POST, PUT, DELETE)
* Perform CRUD operations (Create, Read, Update, Delete)
* Serve JSON data to clients
* Connect to databases like MongoDB or MySQL
* A **RESTful API** is a type of web API that follows the principles of **REST**—short for **Representational State Transfer**. It’s a design style for building scalable and easy-to-use web services that communicate over **HTTP**.
* Let’s break it down in a simple, relatable way:
* **🧠 What Is REST?**
* REST is like a set of rules for how clients (like browsers or mobile apps) and servers should talk to each other. It’s based on:
* **Statelessness**: Each request is independent—no memory of previous interactions.
* **Client-Server Architecture**: The client and server are separate and communicate via requests and responses.
* **Cacheability**: Responses can be cached to improve performance.
* **Uniform Interface**: Consistent structure using HTTP methods like GET, POST, PUT, DELETE.
* **🔁 RESTful API in Action**
* Imagine a RESTful API as a waiter in a restaurant:
* You (the client) ask for something using a clear request (like “GET /menu”).
* The waiter (API) goes to the kitchen (server) and brings back the result (like a list of dishes in JSON format).
* Here’s how the HTTP methods map to actions:

| * **Method** | * **Action** | * **Example Endpoint** |
| --- | --- | --- |
| * **GET** | * Read data | * /users or /users/1 |
| * **POST** | * Create data | * /users |
| * **PUT** | * Update data | * /users/1 |
| * **DELETE** | * Remove data | * /users/1 |

**Service**

**🧩 What Is a Service in Angular?**

An Angular **service** is a class that provides reusable logic or data across components. It’s where you put code that doesn’t belong directly in the UI.

**🎯 What Can We Use Services For?**

* **HTTP requests** (e.g., calling APIs)
* **Business logic** (e.g., calculations, filtering)
* **Shared data** between components
* **Authentication** and user session handling
* **Utility functions** (e.g., formatting dates)

**🛠️ Ways to Use a Service**

1. **Create a service**:

bash

ng generate service my-service

1. **Inject it into a component**:

ts

constructor(private myService: MyService) {}

1. **Use its methods**:

ts

this.myService.getData().subscribe(data => {

this.items = data;

});

**💾 Can We Store Data in a Service?**

Absolutely! Services are perfect for storing temporary data that needs to be shared across components.

**Example:**

ts

@Injectable({ providedIn: 'root' })

export class DataService {

private userData: any;

setUser(data: any) {

this.userData = data;

}

getUser() {

return this.userData;

}

}

This way, you can set data in one component and retrieve it in another.

If you want, I can help you build a mock API and connect it to an Angular service step-by-step. Just say the word!

**🌱 What Does providedIn: 'root' Mean?**

When you add this to your service like so:

ts

@Injectable({

providedIn: 'root'

})

export class MyService {

// service logic here

}

You're telling Angular:

“Make this service available globally — across the entire application — without needing to manually add it to any module.”

**🧠 When Would You NOT Use 'root'?**

If you want a service to be available **only in a specific module**, you can provide it like this:

ts

@Injectable({

providedIn: SomeFeatureModule

})

Or manually register it in the module’s providers array.

**2 ways to use service**

1. **http = inject(departmentservie)**
2. **constructor(private http : departmentservie){}**

**Reusable component**

**🧩 What Is a Reusable Component?**

A **reusable component** is a self-contained UI block that can be used in multiple places across your app — or even across different projects — without rewriting the logic or markup.

**🔧 Example:**

A ButtonComponent that accepts a label and click handler can be reused in forms, modals, dashboards, etc.

Benefits:

* Reduces code duplication
* Improves maintainability
* Promotes consistency in UI and behavior

**📥 What Is the Use of @Input()?**

@Input() allows a parent component to **pass data** into a child component.

**🔧 Example:**

ts

@Input() label: string;

In the parent:

html

<app-button [label]="'Submit'"></app-button>

Use cases:

* Passing dynamic content (titles, images, values)
* Configuring behavior (e.g., button type, visibility)
* Binding form data or state

**📤 What Is the Use of @Output()?**

@Output() lets a child component **send events or data back** to its parent using an EventEmitter.

**🔧 Example:**

ts

@Output() clicked = new EventEmitter<void>();

onClick() {

this.clicked.emit();

}

In the parent:

html

<app-button (clicked)="handleClick()"></app-button>

Use cases:

* Handling user interactions (clicks, selections)
* Communicating form changes
* Triggering actions in parent components

**🧠 When Should You Create a Reusable Component?**

Create one when:

* You repeat the same UI pattern in multiple places
* You want to isolate logic for clarity and testing
* You need configurable components (via @Input)
* You want to emit events (via @Output)
* You plan to share components across modules or projects

**Common reusable components:**

* Buttons, modals, cards, form fields, loaders, alerts, tabs