Advantages of Angular

1. Organized Front-End Structure

- Angular is built on a **component-based architecture**, where each part of the user interface is a reusable component.
- Components work together within **modules**, which help in organizing the application efficiently.
- **Services** in Angular help manage business logic and data sharing across different components.

2. Powerful and Full-Featured Framework

- Unlike some other front-end libraries that focus only on the view layer (like React), Angular is a **full-fledged framework**.
- It provides built-in features like forms, validation, dependency injection, and server communication, making it ideal for **large-scale web applications**.

3. All-in-One Solution

- Angular comes with **essential tools** out of the box, eliminating the need to install external libraries:
    - **Router**: Manages navigation between different views/pages.
    - **HTTP Client**: Handles API requests efficiently.
    - **RxJS & Observables**: Enables reactive programming for handling asynchronous operations like fetching data from an API.

4. Optimized for Single Page Applications (SPAs)

- SPAs load a single HTML page and dynamically update content, enhancing **speed and user experience**.
- Angular's built-in **routing** and **data binding** mechanisms make it well-suited for SPAs.

5. Utilizes MVC (Module, View, Controller) Design Pattern

- Follows a structured approach where:
    - **Model** manages application data.
    - **View** handles the user interface.
    - **Controller** connects the model and view, processing user interactions.

6. TypeScript-Powered

- Angular is built with **TypeScript**, a superset of JavaScript that provides:
    - **Static Typing**: Helps catch errors during development.
    - **ES6+ Features**: Supports modern JavaScript features like **classes, arrow functions, and modules**.

7. Powerful CLI (Command Line Interface)

- The Angular CLI allows developers to **quickly scaffold** an application.
- It can generate **components, services, directives, and pipes** with a single command, speeding up development.

Steps to Install Angular

To install and set up an Angular project, follow these steps:

---

1. Install Node.js and npm

Angular requires **Node.js** and **npm (Node Package Manager)**.
📌 **Check if Node.js is installed**
Open a terminal (Command Prompt or PowerShell) and run:

```
node -v
```

If Node.js is not installed, download and install the **LTS version** from:
🔗 https://nodejs.org/

After installation, verify the version:

```
npm -v
```

---

2. Install Angular CLI (Command Line Interface)

Angular CLI helps create and manage Angular projects.
Run the following command to install it globally:

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

To check if it's installed correctly, run:

```
ng version
```

---

3. Create a New Angular Project

Use the following command to generate a new project:

```
ng new my-angular-app
```

📌 **Options during setup:**

- It will ask for routing setup (`Yes` or `No`).
- Choose the preferred **CSS preprocessor** (CSS, SCSS, etc.).

After completion, navigate to the project directory:

```
cd my-angular-app
```

---

4. Run the Angular Development Server

Start the development server using:

```
ng serve
```

This will start the application and provide a local development URL.
Open a browser and visit:

```
http://localhost:4200/
```

---

5. Optional: Install Dependencies

If needed, install additional Angular packages like **Bootstrap** or **Material UI**:

```
npm install bootstrap
```

Then, import it in `angular.json` or `styles.css`.

Angular Project Structure

When you create a new Angular project using `ng new my-angular-app`, the generated project follows a well-defined structure. Below is a breakdown of the key folders and files:

📂 Project Structure Overview

```
my-angular-app/
│── node_modules/
│── src/
│   ├── app/
│   │   ├── app.component.html
│   │   ├── app.component.ts
│   │   ├── app.component.css
│   │   ├── app.module.ts
│   │   ├── components/
│   │   ├── services/
│   │   ├── models/
│   │   ├── pipes/
│   │   ├── directives/
│   ├── assets/
│   ├── environments/
│   ├── favicon.ico
│   ├── index.html
│   ├── main.ts
│   ├── styles.css
│   ├── polyfills.ts
│── angular.json
│── package.json
│── tsconfig.json
│── README.md
```

📂 Detailed Explanation of Key Files & Folders

1. `node_modules/`

- Contains all installed npm dependencies.
- Avoid making changes manually.
- Generated when you run `npm install`.

2. `src/` (Main Source Folder)

This is where the actual Angular application code is written.

📌 Inside `src/` directory:

📁 `app/` (Main Application Folder)

Contains the core functionality of the application.

- **`app.component.ts`** → Root component of the application.
- **`app.module.ts`** → Main module that declares components and imports other modules.
- **`components/`** → Folder for reusable UI components.
- **`services/`** → Contains services for business logic and API calls.
- **`models/`** → Contains TypeScript interfaces and models.
- **`pipes/`** → Custom pipes for data transformation.
- **`directives/`** → Custom directives to modify UI behavior.

📁 `assets/`

- Stores static files like images, icons, fonts, etc.
- Anything placed here will be available in the app using relative paths.

📁 `environments/`

- Contains files for different environments:
  - `environment.ts` → Used in development.
  - `environment.prod.ts` → Used in production.
- Useful for storing API URLs and environment-specific settings.

`index.html`

- Main HTML file for the Angular app.
- Contains the `<app-root>` tag where the Angular app is injected.

`main.ts`

- The entry point of the application.

- Bootstraps the **AppModule**.

`styles.css`

- Global CSS styles for the entire application.

`polyfills.ts`

- Contains code to ensure compatibility with older browsers.

📂 Configuration Files

These files are located at the root of the project.

`angular.json`

- Configuration file for the Angular CLI.
- Manages build settings, file paths, and style/script imports.

`package.json`

- Defines dependencies and scripts for the project.
- Running `npm install` installs all required dependencies listed here.

`tsconfig.json`

- TypeScript configuration file.

`README.md`

- Documentation for the project.

📌 Summary

- **`app/`** → Contains components, services, and modules.
- **`assets/`** → Stores static files like images and fonts.
- **`index.html`** → Main HTML file where Angular app loads.
- **`main.ts`** → Entry point of the app.
- **`angular.json`** → CLI configuration file.
- **`package.json`** → Manages dependencies.

Working Flow of Execution in an Angular Application

When an Angular application starts, the execution follows a well-defined flow from **bootstrap** to rendering components. Let's go step by step:

1️⃣ `main.ts` (Entry Point)

- The execution starts with `main.ts`, which **bootstraps the Standalone Component**.
- Instead of `platformBrowserDynamic().bootstrapModule(AppModule)`, it now uses:

```
import { bootstrapApplication } from '@angular/platform-browser';
import { AppComponent } from './app/app.component';

bootstrapApplication(AppComponent)
  .catch(err => console.error(err));
```

📌 Key Difference:

- Uses `bootstrapApplication(AppComponent)`, meaning `AppComponent` is directly bootstrapped without needing `app.module.ts`.

2️⃣ `app.component.ts` (Root Component)

- Since the project uses **Standalone Components**, `AppComponent` itself must specify `standalone: true`:

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

@Component({
  selector: 'app-root',
  standalone: true,   // ✅ Marks this as a Standalone Component
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'My Angular App';
}
```

📌 Key Difference:

- The `@Component()` decorator has `standalone: true`, meaning this component doesn't need to be declared inside a module.

3️⃣ `app.component.ts` (Root Component)

- `app.component.ts` is the main component that gets rendered first.
- Example:

  ```
  @Component({
    selector: 'app-root',
  ```

```
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'my-angular-app';
}
```

- The `selector: 'app-root'` is used in `index.html` to load this component.

4⃣ `index.html` (Main HTML File)

- This is the main HTML file where Angular injects the application.
- It contains:

```
html
CopyEdit
<body>
  <app-root></app-root>  <!-- Angular loads AppComponent here -->
</body>
```

- Angular replaces `<app-root>` with `app.component.html`.

5⃣ `app.component.html` (Root Component View)

- This is the template for `AppComponent` and defines what gets displayed on the page.
- Example:

```
html
CopyEdit
<h1>Welcome to {{ title }}!</h1>
<app-header></app-header> <!-- Custom components -->
<app-footer></app-footer>
```

6⃣ Other Components Execution

- If other components (e.g., `HeaderComponent`, `FooterComponent`) are declared in `app.module.ts`, they will be loaded inside `app.component.html` wherever they are used.

7⃣ Services & Dependency Injection

- Services (located in `services/`) provide shared data and logic across components.
- Example of a service:

```
@Injectable({ providedIn: 'root' })
```

```
export class DataService {
  getData() { return 'Hello from service!'; }
}
```

- Services are injected into components using **Dependency Injection**:

```
constructor(private dataService: DataService) {}
```

8⃣ Routing Execution (If Used)

- If the app uses Angular routing (`app-routing.module.ts`), it controls navigation.
- Example:

```
const routes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutComponent },
];
```

- When a user visits `/about`, Angular loads `AboutComponent`.

🚀 Pipes in Angular

Pipes in Angular are used to **transform** data before displaying it in the template. They are similar to filters in other frameworks and are used with the `|` (pipe) symbol.

🔹 Built-in Pipes in Angular

Angular provides several built-in pipes:

Pipe	Description	Example & Output
`uppercase`	Converts text to uppercase	`"hello"
`lowercase`	Converts text to lowercase	`"HELLO"
`titlecase`	Capitalizes the first letter of each word	`"angular pipes"
`date`	Formats date	`today
`currency`	Formats number as currency	`5000
`percent`	Converts number to percentage format	`0.75
`json`	Converts an object to a JSON string	`{name: "Dharmik"}
`slice`	Extracts a section of a string/array	`"Angular"

🔹 Using Pipes in Angular Templates

```
<p>Uppercase: {{ 'hello world' | uppercase }}</p>

<!-- Output: HELLO WORLD -->
```

```
<p>Lowercase: {{ 'ANGULAR' | lowercase }}</p>

<!-- Output: angular -->

<p>Title Case: {{ 'angular is fun' | titlecase }}</p>

<!-- Output: Angular Is Fun -->

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

<!-- Output: Friday, March 7, 2025 -->

<p>Currency: {{ 1000 | currency:'INR' }}</p>

<!-- Output: ₹1,000.00 -->

<p>Percentage: {{ 0.85 | percent }}</p>

<!-- Output: 85% -->

<p>JSON Format: {{ {name: 'Dharmik', role: 'Developer'} | json }}</p>

<!-- Output: {"name":"Dharmik","role":"Developer"} -->
```

◆ app.component.ts

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  today: Date = new Date();
}
```

🔹 Custom Pipe in Angular

You can create your own pipe using `@Pipe` decorator.

Step 1: Generate a Pipe

Run this command:

```
ng generate pipe custom
```

or manually create `custom.pipe.ts`:

Step 2: Define Custom Pipe (`custom.pipe.ts`)

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

@Pipe({
  name: 'reverseText'  // Pipe name
})
export class ReverseTextPipe implements PipeTransform {
  transform(value: string): string {
    return value.split('').reverse().join('');
  }
}
```

Step 3: Use It in a Template

Register the pipe in the component (if using **standalone components**) or in `app.module.ts` (if using **NgModules**).

✅ If Using Standalone Components

```
import { Component } from '@angular/core';
import { ReverseTextPipe } from './custom.pipe';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [ReverseTextPipe],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  message = 'Hello Angular';
}
```

✅ If Using `NgModule`

In `app.module.ts`:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { ReverseTextPipe } from './custom.pipe';  // ✅ Import Custom Pipe

@NgModule({
  declarations: [
    AppComponent,
    ReverseTextPipe  // ✅ Add Pipe Here
  ],
  imports: [
    BrowserModule
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Step 4: Use in `app.component.html`

```
<p>Reversed Text: {{ 'Angular' | reverseText }}</p>
```

✅ **Output:** `"ralugnA"`

Angular Routing in Standalone Components (Latest Version)

Since you are using **the latest Angular version with standalone components**, let's go step by step to set up routing properly.

1⃣ Install and Configure Routing (If Not Installed)

If you haven't set up routing yet, run:

```
ng add @angular/router
```

2⃣ Define Routes in `app.routes.ts`

Since you're using standalone components, create a new file `app.routes.ts`:

📌 **`app.routes.ts`**

```
import { Routes } from '@angular/router';
import { HomeComponent } from './home.component';
import { AboutComponent } from './about.component';
import { ContactComponent } from './contact.component';

export const routes: Routes = [
  { path: '', component: HomeComponent }, // Default route
  { path: 'about', component: AboutComponent },
  { path: 'contact', component: ContactComponent },
];
```

3⃣ Create Standalone Components

Each component must be **standalone** so it can work properly in the latest Angular version.

📌 **Example: `home.component.ts`**

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';

@Component({
  selector: 'app-home',
  standalone: true,
  imports: [CommonModule],
  template: `<h2>Home Page</h2><p>Welcome to the Home Page!</p>`,
})
export class HomeComponent {}
```

📌 **Example: `about.component.ts`**

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';

@Component({
  selector: 'app-about',
  standalone: true,
  imports: [CommonModule],
  template: `<h2>About Page</h2><p>This is the About Page.</p>`,
})
export class AboutComponent {}
```

📌 **Example: `contact.component.ts`**

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';

@Component({
  selector: 'app-contact',
  standalone: true,
  imports: [CommonModule],
  template: `<h2>Contact Page</h2><p>Contact us at
support@example.com</p>`,
})
export class ContactComponent {}
```

4⃣ Modify `app.component.ts` to Load Routes

📌 **`app.component.ts`**

```
import { Component } from '@angular/core';
import { RouterModule } from '@angular/router';
import { routes } from './app.routes';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterModule.forRoot(routes)], // ✅ Register Routes
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Angular Routing Example';
}
```

5⃣ Modify `app.component.html` to Use Router

📌 **`app.component.html`**

```
<h1>{{ title }}</h1>

<!-- Navigation Links -->
<nav>
  <a routerLink="/">Home</a> |
  <a routerLink="/about">About</a> |
  <a routerLink="/contact">Contact</a>
</nav>

<!-- This will load different components based on the route -->
<router-outlet></router-outlet>
```

6️⃣ Start the Application

Run:

```
ng serve
```

✅ Now, you can navigate to:

- **Home Page:** `http://localhost:4200/`
- **About Page:** `http://localhost:4200/about`
- **Contact Page:** `http://localhost:4200/contact`

🎯 Summary

1. **Define Routes in `app.routes.ts`** using standalone components.
2. **Create Standalone Components** (e.g., Home, About, Contact).
3. **Modify `app.component.ts`** to register routes using `RouterModule.forRoot(routes)`.
4. **Modify `app.component.html`** to use `<router-outlet>` and `<a routerLink>`.
5. **Run `ng serve`** and test the routes.

Directives in Angular (Latest Version with Standalone Components)

Directives in Angular are special markers (attributes) on HTML elements that modify their behavior or appearance.

There are **3 types of directives**:

1. **Structural Directives** (modify the DOM) → `*ngIf`, `*ngFor`, `*ngSwitch`
2. **Attribute Directives** (modify element behavior/style) → `[ngClass]`, `[ngStyle]`, `ngModel`
3. **Custom Directives** (user-defined behavior)

1️⃣ Structural Directives (Modify the DOM)

◆ `*ngIf` → Condition-based rendering

✅ Example: Show/hide content based on condition

```
<p *ngIf="isLoggedIn">Welcome, User!</p>
<button (click)="isLoggedIn = !isLoggedIn">Toggle Login</button>
```

📌 How it works?

- If `isLoggedIn` is `true`, the `<p>` is displayed.
- If `false`, it is removed from the DOM.

◆ `*ngFor` → Loop through lists

✅ Example: Display a list of items

```
<ul>
  <li *ngFor="let item of items">{{ item }}</li>
</ul>
```

📌 Component:

```
items: string[] = ['Angular', 'React', 'Vue'];
```

□ Output:

- Angular
- React
- Vue

◆ `*ngSwitch` → Multiple conditions

✅ Example: Show different messages based on a value

```
<div [ngSwitch]="role">
  <p *ngSwitchCase="'admin'">Admin Panel</p>
  <p *ngSwitchCase="'user'">User Dashboard</p>
  <p *ngSwitchDefault>Guest View</p>
</div>
```

📌 Component:

```
role: string = 'user';
```

□ Output:

✅ `User Dashboard`

2️⃣ Attribute Directives (Modify Element Behavior/Style)

📌 Angular's Built-in Attribute Directives

1️⃣ **`ngClass`** - Apply classes dynamically.
2️⃣ **`ngStyle`** - Apply styles dynamically.
3️⃣ **`hidden`** - Hide elements.
4️⃣ **`disabled`** - Disable elements dynamically.

1️⃣ `ngClass` - Dynamic Class Binding

Example:

```
<p [ngClass]="{ 'active': isActive, 'highlight': isNotActive }">
  This is a dynamic paragraph!
</p>

<button (click)="toggleClass()">Toggle Classes</button>
```

Component Logic:

```
export class AppComponent {
  isActive: boolean = true;
  isNotActive: boolean = false;

  toggleClass() {
    this.isActive = !this.isActive;
    this.isNotActive = !this.isNotActive;
  }
}
```

CSS:

```
.active {
  font-weight: bold;
  color: green;
}

.highlight {
  background-color: yellow;
}
```

🔹 What happens?

- When `isActive = true`, the **text is bold and green**.
- When `isNotActive = true`, the **background turns yellow**.
- Clicking the button toggles the classes dynamically.

2️⃣ `ngStyle` - Dynamic Style Binding

Example:

```
<p [ngStyle]="{'color': textColor, 'font-size': fontSize}">
  This text changes color and size dynamically!
</p>

<button (click)="changeStyle()">Change Style</button>
```

Component Logic:

```
export class AppComponent {
  textColor: string = 'blue';
  fontSize: number = 16;

  changeStyle() {
    this.textColor = this.textColor === 'blue' ? 'red' : 'blue';
    this.fontSize = this.fontSize === 16 ? 24 : 16;
  }
}
```

◈ What happens?

- Text changes **color** between blue and red.
- Font size toggles **between 16px and 24px**.

3️⃣ `hidden` - Hide Elements Dynamically

Example:

```
<p [hidden]="isHidden">This text is conditionally hidden.</p>
<button (click)="toggleVisibility()">Toggle Visibility</button>
```

Component Logic:

```
export class AppComponent {
  isHidden: boolean = false;

  toggleVisibility() {
    this.isHidden = !this.isHidden;
  }
}
```

◆ What happens?

- Clicking the button **hides or shows** the paragraph dynamically.

4️⃣ `disabled` - Disable Elements Dynamically

Example:

```
<input type="text" placeholder="Type something..." [disabled]="isDisabled">
<button (click)="toggleDisable()">Enable/Disable Input</button>
```

Component Logic:

```
export class AppComponent {
  isDisabled: boolean = true;

  toggleDisable() {
    this.isDisabled = !this.isDisabled;
  }
}
```

3️⃣ Custom Directives (User-defined Directives)

Custom Attribute Directive in Angular (Step-by-Step Guide)

A **custom attribute directive** allows you to modify the behavior or appearance of an element dynamically. Unlike **structural directives**, it does **not** add or remove elements but changes their properties, styles, or events.

📌 Step 1: Create a Custom Directive

Let's create a directive called `HighlightDirective` to **change the background color of an element when hovered**.

Run this command in the terminal to generate the directive:

```
ng generate directive highlight
```

OR

```
ng g d highlight
```

This will create two files:

- `highlight.directive.ts` (inside the `src/app` folder)
- It will also automatically **register** the directive in `app.module.ts` (if using an older Angular version).

📌 Step 2: Implement the Directive (`highlight.directive.ts`)

Modify the generated directive file as follows:

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

@Directive({
  selector: '[appHighlight]' // Custom directive name
})
export class HighlightDirective {
  @Input() highlightColor: string = 'yellow'; // Default color

  constructor(private el: ElementRef) {}

  // Event listener when the mouse enters the element
  @HostListener('mouseenter') onMouseEnter() {
    this.el.nativeElement.style.backgroundColor = this.highlightColor;
  }

  // Event listener when the mouse leaves the element
  @HostListener('mouseleave') onMouseLeave() {
    this.el.nativeElement.style.backgroundColor = 'transparent';
  }
}
```

Explanation of Code

1. **`@Directive({ selector: '[appHighlight]' })`**
   - Declares this class as an Angular **directive**.
   - The `selector` means that when we apply `appHighlight` to an element, this directive will be applied.
2. **`constructor(private el: ElementRef) {}`**
   - The `ElementRef` service gives us access to the **DOM element** where the directive is applied.
3. **`@HostListener('mouseenter')`**
   - Listens for the `mouseenter` event (when the user hovers over the element).
   - Changes the background color when hovered.
4. **`@HostListener('mouseleave')`**
   - Listens for the `mouseleave` event (when the mouse moves away).
   - Resets the background color when the mouse leaves.
5. **`@Input() highlightColor: string = 'yellow';`**
   - Allows us to set a **custom color** when using the directive.

📌 Step 3: Use the Directive in a Component

Now, apply this directive to any HTML element.

Modify `app.component.html`

```
<h2 appHighlight highlightColor="lightblue">Hover over me to see the
effect!</h2>
```

```
<p appHighlight highlightColor="lightgreen">This text also has a custom
directive.</p>
<button appHighlight highlightColor="pink">Hover over this button!</button>
```

How It Works

✓ When the user **hovers** over the element, the background changes to the `highlightColor`.
✓ When the user **moves the mouse away**, the background returns to normal.

📌 Step 4: Ensure the Directive is Available

Since you're using the **latest Angular version** with **standalone components**, you need to import the directive where you're using it.

Modify `app.component.ts`:

```
import { Component } from '@angular/core';
import { HighlightDirective } from './highlight.directive'; // Import the
directive

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [HighlightDirective], // Register the directive
  templateUrl: './app.component.html',
})
export class AppComponent {
  title = 'Custom Directive Example';
}
```

🎯 Summary

Directive Type	Example	Purpose
Structural Directives	`*ngIf`, `*ngFor`, `*ngSwitch`	Modify DOM elements dynamically
Attribute Directives	`[ngClass]`, `[ngStyle]`, `ngModel`	Change element behavior/styles
Custom Directives	`appHighlight`	Create custom functionality

Example: "To-Do List with Visibility Control"

This example dynamically **shows, hides, and loops through a list of tasks** using structural directives (`*ngIf`, `*ngFor`, and `*ngSwitch`).

✓ TypeScript (app.component.ts)

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {
  title = 'To-Do List Example';
  showList = true; // Controls visibility
  tasks = [
    { name: 'Complete Angular Project', status: 'completed' },
    { name: 'Study for Exams', status: 'pending' },
    { name: 'Buy Groceries', status: 'in-progress' },
    { name: 'Exercise', status: 'completed' }
  ];

  toggleList() {
    this.showList = !this.showList; // Toggle visibility
  }
}
```

✓ HTML (app.component.html)

```
<h1>{{ title }}</h1>

<!-- Toggle Button -->
<button (click)="toggleList()">
  {{ showList ? 'Hide' : 'Show' }} To-Do List
</button>

<!-- Check if List is Visible -->
<div *ngIf="showList; else noList">
  <h2>Tasks:</h2>
  <ul>
    <li *ngFor="let task of tasks">
      <strong>{{ task.name }}</strong> -
      <span [ngSwitch]="task.status">
        <span *ngSwitchCase="'completed'" style="color:
green;">Completed</span>
        <span *ngSwitchCase="'in-progress'" style="color: orange;">In
Progress</span>
        <span *ngSwitchCase="'pending'" style="color: red;">Pending</span>
        <span *ngSwitchDefault>□ Unknown</span>
      </span>
    </li>
  </ul>
</div>

<!-- Template if List is Hidden -->
<ng-template #noList>
  <p style="color: gray;">To-Do List is Hidden. Click "Show" to display
it.</p>
```

```
</ng-template>
```

📌 Forms in Angular

Forms in Angular are essential for collecting and managing user input. Angular provides two types of forms:

1⃣ **Template-driven Forms** (Simpler, for small forms)
2⃣ **Reactive Forms** (More powerful, for complex forms with validations)

🔹 1⃣ Template-driven Forms (Easy Approach)

✅ Uses **`FormsModule`**
✅ Uses `ngModel` for **two-way data binding**
✅ Ideal for **simple forms**

📝 Example: Basic Template-driven Form

👉 **Steps:** 1⃣ Import `FormsModule`
2⃣ Bind input fields using `ngModel`
3⃣ Handle form submission

📌 app.module.ts

```typescript
typescript
CopyEdit
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms'; // Import FormsModule
import { AppComponent } from './app.component';

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, FormsModule], // Add FormsModule here
  bootstrap: [AppComponent]
})
export class AppModule {}
```

📌 app.component.html

```html
html
CopyEdit
<h2>Template-driven Form</h2>
<form #userForm="ngForm" (ngSubmit)="submitForm(userForm)">
  <label>Name:</label>
  <input type="text" name="name" [(ngModel)]="user.name" required>

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

  <button type="submit" [disabled]="!userForm.valid">Submit</button>
</form>

<p *ngIf="submitted">Form Submitted! Name: {{ user.name }}, Email: {{
user.email }}</p>
```

📌 app.component.ts

```
typescript
CopyEdit
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  user = { name: '', email: '' };
  submitted = false;

  submitForm(form: any) {
    this.submitted = true;
    console.log('Form Data:', form.value);
  }
}
```

✅ **`ngModel`** binds input fields to the component data.
✅ **`#userForm="ngForm"`** gives access to form properties like `valid`.

📌 Best Example: Consuming API in Angular (Latest Version)

We will create a **User Management System** where we can:
✅ Fetch users from an API (**GET**)
✅ Add a new user (**POST**)
✅ Update an existing user (**PUT**)
✅ Delete a user (**DELETE**)

👉 **We will use JSONPlaceholder API (`https://jsonplaceholder.typicode.com/users`) for demonstration.**

📌 Step 1: Set Up Angular & Install Dependencies

If you haven't already created an Angular project, run:

```
ng new api-demo --standalone
cd api-demo
```

👉 Since you're using **Angular Standalone Components**, we don't have `app.module.ts`. We will configure `HttpClient` in **`main.ts`** instead.

Now, install `json-server` (for a mock API if needed):

```
npm install -g json-server
```

📌 Step 2: Setup `HttpClient`

Modify `main.ts` to enable API calls:

✅ **`main.ts`**

```
import { bootstrapApplication } from '@angular/platform-browser';
import { provideHttpClient } from '@angular/common/http';
import { AppComponent } from './app/app.component';

bootstrapApplication(AppComponent, {
  providers: [provideHttpClient()]
});
```

🔹 **Why?** `provideHttpClient()` allows us to use `HttpClient` for making API requests.

📌 Step 3: Create an API Service

Now, let's create a **service to handle API calls**.

Run this command:

```
ng g service services/api
```

✅ **`api.service.ts`** (Create a service to handle API operations)

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

@Injectable({
  providedIn: 'root'
})
export class ApiService {
  private apiUrl = 'https://jsonplaceholder.typicode.com/users'; // Dummy
API

  constructor(private http: HttpClient) {}

  // Fetch all users (GET)
  getUsers(): Observable<any> {
    return this.http.get(this.apiUrl);
  }
  // Fetch a single user (GET)
  getUser(id: number): Observable<any> {
    return this.http.get(`${this.apiUrl}/${id}`);
  }

  // Add a new user (POST)
  addUser(user: any): Observable<any> {
    return this.http.post(this.apiUrl, user);
  }

  // Update user (PUT)
  updateUser(id: number, user: any): Observable<any> {
    return this.http.put(`${this.apiUrl}/${id}`, user);
  }

  // Delete user (DELETE)
  deleteUser(id: number): Observable<any> {
    return this.http.delete(`${this.apiUrl}/${id}`);
  }
}
```

🔹 Why create a service?

- Centralized API calls (makes `app.component.ts` cleaner)
- Follows best practices for Angular

📌 Step 4: Use API in a Component

Now, let's modify `app.component.ts` to use our service.

✅ **`app.component.ts`**

```
import { Component } from '@angular/core';
import { ApiService } from './services/api.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {
  users: any[] = [];
  newUser = { name: '', email: '' };

  constructor(private apiService: ApiService) {}

  ngOnInit() {
    this.fetchUsers();
  }

  // Fetch all users
  fetchUsers() {
    this.apiService.getUsers().subscribe((data) => {
      this.users = data;
    });
  }
  // Add a new user
  addUser() {
    if (this.newUser.name && this.newUser.email) {
      this.apiService.addUser(this.newUser).subscribe((user) => {
        this.users.push(user);
        this.newUser = { name: '', email: '' }; // Reset form
      });
    }
  }

  // Update a user
  updateUser(id: number) {
    const updatedUser = { name: 'Updated User', email:
'updated@example.com' };
    this.apiService.updateUser(id, updatedUser).subscribe((user) => {
      this.users = this.users.map((u) => (u.id === id ? user : u));
    });
  }

  // Delete a user
  deleteUser(id: number) {
    this.apiService.deleteUser(id).subscribe(() => {
      this.users = this.users.filter((user) => user.id !== id);
    });
  }
}
```

🔹 Explanation:

- Fetches users from the API when the app loads (`ngOnInit()`).
- Adds, updates, and deletes users using `ApiService`.

📌 Step 5: Display Data in HTML

✅ `app.component.html`

```
<h2>Users List</h2>
<ul>
  <li *ngFor="let user of users">
    {{ user.name }} ({{ user.email }})
    <button (click)="updateUser(user.id)">Update</button>
    <button (click)="deleteUser(user.id)">Delete</button>
  </li>
</ul>

<h2>Add New User</h2>
<input type="text" placeholder="Name" [(ngModel)]="newUser.name">
<input type="email" placeholder="Email" [(ngModel)]="newUser.email">
<button (click)="addUser()">Add User</button>
```

🔹 Explanation:

- **Lists users from the API** (`*ngFor`).
- **Adds users dynamically** via input fields and `ngModel`.

- **Updates and deletes users** when buttons are clicked.

📌 Step 6: Run the Angular App

✓ 1⃣ Start Mock API Server

If you're using a local mock API (`json-server`), run:

```
json-server --watch db.json
```

Otherwise, Angular will fetch data from `https://jsonplaceholder.typicode.com/users`.

✓ 2⃣ Start Angular App

Run:

```
ng serve
```

Visit `http://localhost:4200`.