# Introduction to Angular

## What is Angular?

**Angular** is a **TypeScript-based**, **open-source** front-end web application framework developed by **Google**.

It allows developers to build **scalable**, **single-page applications (SPAs)** with clean architecture, reusable components, and strong tooling support.

### Key Features

| **Feature** | **Description** |
| --- | --- |
| **Component-based** | UI is built using reusable and isolated components |
| **TypeScript-first** | Angular apps are written in TypeScript, enabling type safety and tooling |
| **MVVM Architecture** | Separates view (HTML) and logic (TypeScript) for clean code organization |
| **Reactive Forms & Signals** | Supports both form control and state reactivity with Signals (new in v16+) |
| **Routing System** | Built-in router for navigation and lazy loading |
| **Dependency Injection (DI)** | Services and shared logic are injected where needed for better testability |
| **HttpClient** | For calling REST APIs and backend services |
| **Built-in Tools** | Angular CLI, testing, AOT compiler, and more |

## Building Blocks of Angular

| **Concept** | **Description** |
| --- | --- |
| **Modules** | Group related components and features (optional in standalone apps) |
| **Components** | Reusable building blocks for UI (@Component) |
| **Templates** | HTML code with Angular directives and bindings |
| **Services** | Shared logic and APIs injected via Dependency Injection (@Injectable) |
| **Directives** | Extend or modify DOM behavior (e.g., \*ngIf, \*ngFor) |
| **Pipes** | Transform data in the UI (e.g., date, currency, custom pipes) |
| **Routing** | Handle page navigation using RouterModule or provideRouter() |
| **HttpClient** | Communicate with REST APIs and servers |

## Angular CLI

Angular comes with a powerful command-line interface (@angular/cli) for:

* Scaffolding new apps: ng new
* Generating components/services: ng generate component
* Running the app: ng serve
* Building for production: ng build

## Example: Basic Angular Component

| @Component({  selector: 'app-welcome',  template: `<h1>Hello, {{ name }}!</h1>`  })  export class WelcomeComponent {  name = 'Angular';  } |
| --- |

## Real-World Use Cases

* Dashboards and Admin Panels
* E-commerce frontends
* Content Management Systems
* Enterprise apps with complex forms and workflows
* Progressive Web Apps (PWAs)

## **Angular Versioning & Releases**

* Angular uses **semantic versioning** (e.g., v17.0.0)
* **Major updates** every 6 months (long-term support provided)
* Recent versions focus on:  
  + **Standalone APIs** (no NgModules)
  + **Signals** for reactivity
  + **Improved SSR (Server Side Rendering)**
  + **Zoneless change detection (experimental)**

## Summary

Angular is a complete framework for building **robust, testable, and scalable** front-end applications — perfect for enterprise and large-scale web apps.

| **Term** | **Summary** |
| --- | --- |
| Angular | Modern frontend framework by Google |
| Component | Core building block for UI |
| TypeScript | Angular's primary language |
| CLI | Command-line tool for fast development |
| SPA | Single Page Application |
| DI | Inject shared services for scalability |

# What’s new in Angular19

​Angular 19 introduces several significant enhancements aimed at improving performance, reactivity, and developer experience.

## Signals: Enhanced Reactivity

Angular's Signals API, introduced in earlier versions, has been stabilized in Angular 19, providing a more intuitive and fine-grained approach to state management. Key additions include:​

* **linkedSignal()**: Creates writable signals that automatically update based on dependencies, simplifying state synchronization.
* **resource() and rxResource()**: Facilitate asynchronous data fetching within the reactive system, integrating seamlessly with Angular's HttpClient.
* **afterRenderEffect()**: A new lifecycle hook that runs effects after the component has rendered, ensuring DOM updates occur at safe intervals.
* **Signal-based APIs**: Stable support for signal-based input, output, model, and query bindings, enhancing component communication.​

These enhancements promote a more predictable and efficient reactivity model, reducing the need for manual subscriptions and boilerplate code. ​

## **Zoneless Angular: Improved Performance**

Angular 19 continues the transition towards a zoneless architecture, removing the dependency on Zone.js for change detection. Benefits include:

* **Performance Gains**: Eliminates unnecessary global change detection cycles, leading to faster updates and reduced CPU usage.​
* **Smaller Bundle Sizes**: Removing Zone.js reduces the application's payload, improving load times.​
* **Enhanced Debugging**: Provides clearer stack traces and more predictable change detection behavior.​

To enable zoneless mode:

1. Remove zone.js from your project dependencies and polyfills.​
2. Configure your application with provideExperimentalZonelessChangeDetection() in app.config.ts.​
3. Use Signals or manual triggers like ChangeDetectorRef.markForCheck() to manage change detection.

## **Server-Side Rendering (SSR) Enhancements**

Angular 19 introduces several SSR improvements to optimize performance and user experience:​

* **Incremental Hydration**: Allows components to hydrate lazily based on user interactions or visibility, reducing initial load times.​
* **Event Replay**: Captures user events during the hydration gap and replays them once the necessary JavaScript is loaded, ensuring seamless interactivity.​
* **Route-Level Render Modes**: Provides granular control over rendering strategies per route—choose between server-side rendering, client-side rendering, or prerendering.

Example configuration:

| export const serverRouteConfig: ServerRoute[] = [  { path: '/login', mode: RenderMode.Server },  { path: '/dashboard', mode: RenderMode.Client },  { path: '/\*\*', mode: RenderMode.Prerender },  ]; |
| --- |

These features collectively enhance the performance and flexibility of Angular applications, particularly for large-scale or SEO-sensitive projects.

Angular 19 marks a significant step towards a more modern, efficient, and developer-friendly framework, aligning with contemporary web development practices.​

## 

## **Introduction to TypeScript**

### **What is TypeScript?**

* **TypeScript** is a **superset of JavaScript** developed by Microsoft.
* It adds **static typing**, **interfaces**, and **compile-time checking** to JavaScript.
* TypeScript code compiles (or transpiles) down to **plain JavaScript** for execution in browsers or Node.js.

### **Why Use TypeScript?**

* **Improved Developer Experience:**
  + Autocomplete, type inference, and IntelliSense.
* **Early Error Detection:**
  + Catches type-related bugs at compile time.
* **Scalability:**
  + Better suited for large, complex applications.
* **Code Readability & Maintainability:**
  + Clear contracts and structure with types and interfaces.
* **Supports Modern JavaScript:**
  + Uses ES6+ features and compiles down for compatibility.

### **Key Features**

* **Static Typing:**
  + Declaring types for variables, function parameters, and return values.
* **Interfaces and Types:**
  + Define custom types and object shapes.
* **Type Inference:**
  + Automatically deduces types when not explicitly stated.
* **Enums, Tuples, Generics:**
  + Advanced type features to enhance structure.
* **Class and Module Support:**
  + Aligns with object-oriented programming (OOP) and modular architecture.

### **TypeScript vs JavaScript**

| **Feature** | **JavaScript** | **TypeScript** |
| --- | --- | --- |
| Typing | Dynamic | Static (optional) |
| Error Checking | Runtime | Compile-time |
| Interfaces & Generics | No | Yes |
| Tooling | Basic | Rich IDE Support |
| Learning Curve | Easy | Slightly Steeper |

### **Basic TypeScript Example**

| function greet(name: string): string {  return `Hello, ${name}`;  }  let user = "Alice";  console.log(greet(user)); |
| --- |

### **How to Get Started**

**Install TypeScript:**

| npm install -g typescript |
| --- |

**Compile a .ts file:**

| tsc hello.ts |
| --- |

**Run with Node.js:**

| node hello.js |
| --- |

### **Popular Use Cases**

* Frontend frameworks (Angular uses TS by default)
* Backend with Node.js (e.g., NestJS)
* Enterprise-scale applications
* Codebases requiring strong maintainability

## 

## **TypeScript: Basic Types**

TypeScript provides several **primitive and custom types** to improve code quality and readability.

### **1. number**

Represents all numeric values — integers and floating-point numbers.

| let age: number = 30;  let price: number = 99.99; |
| --- |

### **2. string**

Used for text data. Can use single, double, or backtick quotes.

| let firstName: string = "Alice";  let greeting: string = `Hello, ${firstName}`; |
| --- |

### **3. boolean**

Represents true/false values.

| let isActive: boolean = true; |
| --- |

### **4. null and undefined**

These are valid types. Can be assigned explicitly.

| let nothing: null = null;  let notDefined: undefined = undefined; |
| --- |

Note: strictNullChecks affects how these types are handled.

### **5. any**

Disables type checking. Use sparingly (usually for gradual typing).

| let data: any = "Could be anything";  data = 5; // Still valid |
| --- |

### **6. unknown**

Similar to any, but safer. Must be type-checked before usage.

| let value: unknown = "Hello";  if (typeof value === "string") {  console.log(value.toUpperCase());  } |
| --- |

### **7. array**

Represents a list of items of the same type.

| let scores: number[] = [90, 85, 88];  let names: Array<string> = ["Alice", "Bob"]; |
| --- |

### **8. tuple**

Fixed-length array with known types.

| let person: [string, number] = ["John", 28]; |
| --- |

### **9. enum**

Defines a set of named constants (numeric or string).

| enum Direction {  Up,  Down,  Left,  Right  }  let move: Direction = Direction.Up; |
| --- |

### **10. object**

Any non-primitive value.

| let user: { name: string; age: number } = {  name: "Emma",  age: 25  }; |
| --- |

### **11. void**

Used when a function **does not return a value**.

| function logMessage(msg: string): void {  console.log(msg);  } |
| --- |

### **12. never**

Used for functions that **never return** (e.g., always throw errors).

| function throwError(message: string): never {  throw new Error(message);  } |
| --- |

### **Type Inference**

TypeScript can automatically infer types:

| let city = "Kuala Lumpur"; // Inferred as string |
| --- |

## **Functions and Type Annotations**

TypeScript enhances JavaScript functions with **strong typing**, enabling safer and more predictable code.

### **Function Syntax with Type Annotations**

You can define:

* **Parameter types**
* **Return type**

| function add(x: number, y: number): number {  return x + y;  } |
| --- |

### **Why Use Type Annotations?**

* Prevents type-related bugs
* Improves code readability
* Enables better IDE support (autocompletion, refactoring)

### **Optional Parameters**

Use ? to indicate a parameter is optional.

| function greet(name: string, title?: string): string {  return title ? `${title} ${name}` : `Hello, ${name}`;  } |
| --- |

### **Default Parameters**

Provide a default value for parameters.

| function multiply(a: number, b: number = 2): number {  return a \* b;  } |
| --- |

### **Rest Parameters**

Accept multiple arguments as an array.

| function sumAll(...numbers: number[]): number {  return numbers.reduce((acc, num) => acc + num, 0);  } |
| --- |

### **Anonymous and Arrow Functions**

Functions can be typed when stored in variables:

| const divide = (x: number, y: number): number => x / y; |
| --- |

Or:

| const subtract: (a: number, b: number) => number = function (a, b) {  return a - b;  }; |
| --- |

### **Function Type Aliases**

Use type to define reusable function signatures:

| type MathOperation = (a: number, b: number) => number;  const multiply: MathOperation = (a, b) => a \* b; |
| --- |

### **Void and Never Return Types**

* **void**: Function does not return a value
* **never**: Function never returns (e.g., throws error)

| function log(msg: string): void {  console.log(msg);  }  function fail(message: string): never {  throw new Error(message);  } |
| --- |

**Function Overloading**

TypeScript supports multiple signatures for the same function:

| function combine(a: number, b: number): number;  function combine(a: string, b: string): string;  function combine(a: any, b: any): any {  return a + b;  } |
| --- |

## **Interfaces and Type Aliases**

Both **Interfaces** and **Type Aliases** are used to define the shape of objects and custom types in TypeScript.

### **What is an Interface?**

An **interface** defines a contract for object structure.

| interface Person {  name: string;  age: number;  isStudent?: boolean; // Optional property  } |
| --- |

Usage:

| const user: Person = {  name: "Alice",  age: 25  }; |
| --- |

### **Interface Features**

* Supports **optional properties** using ?
* Supports **readonly** properties

| interface Product {  readonly id: number;  name: string;  price: number;  } |
| --- |

* Supports **method definitions**

### **Interface Extension**

Interfaces can **extend** other interfaces to create complex types.

| interface Employee extends Person {  jobTitle: string;  } |
| --- |

### **What is a Type Alias?**

A **type alias** assigns a name to any type — primitive, union, object, etc.

| type ID = string | number;  type User = {  username: string;  email: string;  }; |
| --- |

### **Type Alias Features**

* Can represent **union**, **intersection**, and **tuple** types

| type Status = "active" | "inactive";  type Coordinates = [number, number];  type AdminUser = User & { role: "admin" }; |
| --- |

**Interface vs Type Alias**

| **Feature** | **Interface** | **Type Alias** |
| --- | --- | --- |
| Object structure | Yes | Yes |
| Extension (inheritance) | Yes (extends) | Yes(&) |
| Union & primitive types | No | Yes |
| Declaration merging | Yes | No |
| Recommended for | Object shapes | Complex/union types |

### **Example: Interface vs Type Alias**

| // Interface  interface Car {  brand: string;  year: number;  }  // Type Alias  type CarType = {  brand: string;  year: number;  }; |
| --- |

Both are valid — choice depends on use case and team preference.

### **When to Use Which?**

* Use **interface** when working with objects/classes and want extendability.
* Use **type alias** when needing unions, intersections, or complex custom types.

## **Classes in TypeScript**

TypeScript builds on JavaScript’s class-based object-oriented programming (OOP) model, adding **strong typing**, **access modifiers**, and **interfaces support**.

### **What is a Class?**

A **class** is a blueprint for creating objects with shared structure and behavior.

| class Person {  name: string;  age: number;  constructor(name: string, age: number) {  this.name = name;  this.age = age;  }  greet(): string {  return `Hello, my name is ${this.name}`;  }  } |
| --- |

Usage:

| const person1 = new Person("Alice", 25);  console.log(person1.greet()); |
| --- |

### **Access Modifiers**

| **Modifier** | **Description** |
| --- | --- |
| public | Accessible from anywhere (default) |
| private | Only accessible within the class |
| protected | Accessible within the class and subclasses |
| readonly | Value can only be assigned once |

| class Car {  public brand: string;  private speed: number;  protected fuel: number;  readonly vin: string;  constructor(brand: string, vin: string) {  this.brand = brand;  this.vin = vin;  this.speed = 0;  this.fuel = 100;  }  } |
| --- |

### **Getters and Setters**

Encapsulate internal properties with custom access logic.

| class Account {  private \_balance: number = 0;  get balance(): number {  return this.\_balance;  }  set balance(amount: number) {  if (amount >= 0) {  this.\_balance = amount;  }  }  } |
| --- |

### **Inheritance (Extending Classes)**

Classes can inherit properties and methods from other classes using extends.

| class Animal {  move(): void {  console.log("Animal is moving");  }  }  class Dog extends Animal {  bark(): void {  console.log("Woof!");  }  }  const dog = new Dog();  dog.move(); // Inherited  dog.bark(); |
| --- |

### **Abstract Classes**

Used as base classes. Cannot be instantiated directly. Must contain at least one abstract method.

| abstract class Shape {  abstract getArea(): number;  printArea(): void {  console.log("Area:", this.getArea());  }  }  class Circle extends Shape {  constructor(public radius: number) {  super();  }  getArea(): number {  return Math.PI \* this.radius \*\* 2;  }  } |
| --- |

### **Implementing Interfaces**

Classes can implement one or more interfaces to ensure specific structure.

| interface Logger {  log(message: string): void;  }  class ConsoleLogger implements Logger {  log(message: string): void {  console.log("LOG:", message);  }  } |
| --- |

### **Static Members**

Static properties and methods belong to the class, not instances.

| class MathUtil {  static PI = 3.14;  static square(x: number): number {  return x \* x;  }  }  console.log(MathUtil.PI);  console.log(MathUtil.square(5)); |
| --- |

### **Summary**

| **Feature** | **Description** |
| --- | --- |
| Class | Template for creating objects |
| Constructor | Initializes class properties |
| Access Modifiers | Control visibility of members |
| Inheritance | Allows sharing and overriding behavior |
| Abstract Class | Base class with abstract methods |
| Interface | Structure enforcement on class |
| Static Members | Belong to class, not object instances |

## **Working with Modules in TypeScript**

Modules help **organize code into separate files** for maintainability, reusability, and scalability. TypeScript supports **ES Modules (ESM)** using import/export.

### **What is a Module?**

A **module** is any file that contains a top-level import or export statement.  
 Each module has its own **scope**, preventing variable conflicts.

### **Exporting Code**

#### **1. Named Exports**

You can export multiple items from a file using named exports.

| // mathUtils.ts  export const PI = 3.14;  export function add(a: number, b: number): number {  return a + b;  } |
| --- |

#### **2. Default Export**

You can export a single default value (function, class, object).

| // logger.ts  export default function log(message: string): void {  console.log("Log:", message);  } |
| --- |

### **Importing Code**

#### **1. Named Imports**

| import { PI, add } from './mathUtils';  console.log(PI);  console.log(add(5, 3)); |
| --- |

#### **2. Default Import**

| import log from './logger';  log("This is a message"); |
| --- |

#### **3. Aliasing Imports**

| import { add as addNumbers } from './mathUtils';  addNumbers(10, 20); |
| --- |

### **Re-exporting**

You can re-export from another module:

| export \* from './mathUtils'; // Re-exports everything  export { add } from './mathUtils'; // Re-export specific item |
| --- |

### **Organizing Large Projects**

* Group related features in folders
* Use index.ts to simplify imports

| // utils/index.ts  export \* from './mathUtils';  export \* from './logger'; |
| --- |

Then import in one line:

| import { add, log } from './utils'; |
| --- |

### **Module Resolution**

TypeScript resolves modules based on:

* File names and extensions
* Configuration in tsconfig.json (e.g. baseUrl, paths)

Example:

| {  "compilerOptions": {  "baseUrl": "./src",  "paths": {  "@utils/\*": ["utils/\*"]  }  }  } |
| --- |

Then:

| import { add } from '@utils/mathUtils'; |
| --- |

### **Common Errors**

| **Issue** | **Reason** | **Fix** |
| --- | --- | --- |
| Cannot find module | Wrong path | Check relative or aliased path |
| is not exported | Not exported properly | Use export keyword |
| Mixed default and named exports | Import incorrectly | Match syntax with how it’s exported |

### **Summary**

| **Concept** | **Purpose** |
| --- | --- |
| export / import | Share code across files/modules |
| Named exports | Multiple exports from a module |
| Default export | Single main export |
| Re-export | Aggregate and forward exports |
| tsconfig.json | Controls module resolution |

## Lab: Introduction to ES6

#### **Objective:**

In this lab, you'll learn and practice the key features introduced in ECMAScript 6 (ES6). By the end of this session, you should be able to understand and use the main enhancements in ES6, including let and const, arrow functions, template literals, destructuring, modules, and more.

### **Lab Instructions**

Follow the steps below to explore each ES6 feature. For each section, create a separate JavaScript file (feature1.js, feature2.js, etc.) to keep things organized.

### **1. let and const (Block-Scoped Variables)**

1. **Create a file called** let-const.js.
2. **Experiment with let and const** by declaring variables with both keywords.

Example code:

| let x = 10;  const y = 20;  if (true) {  let x = 30; // Block scope  console.log("Inside block:", x); // 30  }  console.log("Outside block:", x); // 10 |
| --- |

1. **Test const immutability** by attempting to reassign a const variable and observe the error.
2. **Practice**: Try declaring variables inside and outside of different blocks to observe the scope behavior.

**Expected Output:** Understand the scope of let and const compared to var, and observe immutability with const.

### **2. Arrow Functions**

1. **Create a file called** arrow-functions.js.
2. **Convert a regular function to an arrow function.**

Example code:

| const greet = (name) => `Hello, ${name}!`;  console.log(greet("Alice")); |
| --- |

1. **Use implicit return**: Write a one-liner arrow function that returns the square of a number.
2. **Experiment with this context**: Create an object with a method and test this inside an arrow function and a regular function.

**Expected Output:** Recognize the syntax of arrow functions, understand implicit returns, and observe how this behaves in arrow functions vs. regular functions.

### **3. Template Literals**

1. **Create a file called** template-literals.js.
2. **Write a string** using template literals with both single-line and multi-line content.

Example code:

| const name = "John";  const message = `Hello, ${name}!  Welcome to ES6.`;  console.log(message); |
| --- |

1. **Experiment with expressions** inside template literals, such as calculations or function calls.

**Expected Output:** Understand how template literals work and see how they allow for easier string interpolation and multi-line strings.

### **4. Destructuring**

1. **Create a file called** destructuring.js.
2. **Destructure an object** into individual variables.

Example code:

| const person = { name: "Alice", age: 25, city: "Paris" };  const { name, age } = person;  console.log(name, age); |
| --- |

1. **Destructure an array** into variables.

Example:

| const numbers = [10, 20, 30];  const [first, second] = numbers;  console.log(first, second); |
| --- |

1. **Practice**: Try swapping two variables using array destructuring.

**Expected Output:** Learn how destructuring simplifies extracting data from arrays and objects.

### **5. Default Parameters**

1. **Create a file called** default-parameters.js.
2. **Define a function with default parameters**.

Example code:

| const greet = (name = "Guest") => `Hello, ${name}!`;  console.log(greet()); // "Hello, Guest!"  console.log(greet("Alice")); // "Hello, Alice!" |
| --- |

1. **Test with different numbers of arguments** to see how defaults work.

**Expected Output:** Understand how default parameters can make functions more flexible.

### **6. Spread and Rest Operators**

1. **Create a file called** spread-rest.js.
2. **Experiment with the spread operator** to clone and merge arrays or objects.

Example code:

| const arr1 = [1, 2, 3];  const arr2 = [...arr1, 4, 5];  console.log(arr2); // [1, 2, 3, 4, 5] |
| --- |

1. **Use the rest parameter** in a function to handle multiple arguments.

Example:

| const sum = (...numbers) => numbers.reduce((acc, val) => acc + val, 0);  console.log(sum(1, 2, 3)); // 6 |
| --- |

1. **Practice**: Try using spread with objects, and pass a varying number of arguments into a function using the rest parameter.

**Expected Output:** Understand the use of spread for copying/merging and rest for handling multiple arguments.

### **7. Classes**

1. **Create a file called** classes.js.
2. **Define a class with a constructor and methods**.

Example code:

| class Animal {  constructor(name) {  this.name = name;  }  speak() {  console.log(`${this.name} makes a noise.`);  }  }  const dog = new Animal("Dog");  dog.speak(); |
| --- |

1. **Create a subclass** to demonstrate inheritance.

Example:

| class Dog extends Animal {  speak() {  console.log(`${this.name} barks.`);  }  }  const myDog = new Dog("Buddy");  myDog.speak(); |
| --- |

1. **Practice**: Add more properties and methods to the class.

**Expected Output:** Learn how to use ES6 classes to create and extend objects.

### **8. Modules**

1. **Create two files:** math.js and app.js.
2. **Define a simple module** in math.js.

Example code (math.js):

| export const add = (a, b) => a + b;  export const subtract = (a, b) => a - b; |
| --- |

1. **Import and use the module** in app.js.

Example code (app.js):

| import { add, subtract } from './math.js';  console.log(add(5, 3)); // 8  console.log(subtract(5, 3)); // 2 |
| --- |

1. **Practice**: Try exporting/importing a default export.

**Expected Output:** Understand how ES6 modules allow you to organize and reuse code.

## **Lab: Introduction to TypeScript**

### **Objective**

The objective of this lab is to get familiar with TypeScript basics, understand how to set up a TypeScript environment, and learn key TypeScript features like types, interfaces, and classes.

### **Part 1: Setting Up TypeScript**

**Initialize a Node Project**Open a terminal and create a new directory for your project:

| mkdir typescript-intro  cd typescript-intro  npm init -y |
| --- |

**Install TypeScript**Install TypeScript globally or as a development dependency:

| npm install -D typescript |
| --- |

**Create a tsconfig.json**Initialize a TypeScript configuration file:

| npx tsc --init |
| --- |

1. This creates a tsconfig.json file with default settings, which allows you to customize your TypeScript project. For now, leave the file as it is.

### **Part 2: Basic Types**

**Create a TypeScript File**Create a file named index.ts:

| touch index.ts |
| --- |

**Exploring Basic Types**Add the following code to index.ts to explore some basic types:

| // Primitive types  let isActive: boolean = true;  let age: number = 30;  let name: string = "TypeScript";  // Array  let numbers: number[] = [1, 2, 3, 4, 5];  // Tuple  let person: [string, number] = ["Alice", 25];  // Enum  enum Direction {  Up,  Down,  Left,  Right,  }  let move: Direction = Direction.Up;  // Any  let randomValue: any = "Could be anything";  console.log(isActive, age, name, numbers, person, move, randomValue); |
| --- |

**Compile and Run the Code**Compile the TypeScript file:

| npx tsc |
| --- |

This will generate an index.js file. Run it using Node:

| node index.js |
| --- |

1. **Task**Experiment by changing variable values to different types and observe TypeScript’s error messages.

### **Part 3: Functions and Type Annotations**

**Adding Type Annotations to Functions**Modify index.ts to add functions with type annotations:

| // Function with parameter types and return type  function add(a: number, b: number): number {  return a + b;  }  console.log(add(5, 10)); // Should output 15 |
| --- |

**Optional and Default Parameters**Modify index.ts to explore optional and default parameters:

| function greet(name: string, greeting: string = "Hello"): string {  return `${greeting}, ${name}!`;  }  console.log(greet("Alice"));  console.log(greet("Bob", "Hi")); |
| --- |

1. **Task**Add a new function called multiply that takes two numbers and returns their product. Test it by calling multiply with different arguments.

### **Part 4: Interfaces and Type Aliases**

**Defining Interfaces**Add an interface to define a Person structure:

| interface Person {  name: string;  age: number;  isActive?: boolean; // Optional property  }  let user: Person = {  name: "Charlie",  age: 28,  };  console.log(user); |
| --- |

**Type Aliases**Create a type alias for a Point object with x and y coordinates:

| type Point = {  x: number;  y: number;  };  let point: Point = {  x: 10,  y: 20,  };  console.log(point); |
| --- |

1. **Task**Create an interface Car with properties make, model, and year. Create an object of type Car and print it to the console.

### **Part 5: Classes**

**Creating a Class**Create a Person class with a constructor and a method:

| class Person {  name: string;  age: number;  constructor(name: string, age: number) {  this.name = name;  this.age = age;  }  greet() {  return `Hello, my name is ${this.name}`;  }  }  const person1 = new Person("Alice", 30);  console.log(person1.greet()); |
| --- |

**Adding Inheritance**Extend the Person class to create a Student subclass:

| class Student extends Person {  studentId: number;  constructor(name: string, age: number, studentId: number) {  super(name, age);  this.studentId = studentId;  }  study() {  return `${this.name} is studying.`;  }  }  const student1 = new Student("Bob", 20, 12345);  console.log(student1.greet());  console.log(student1.study()); |
| --- |

1. **Task**Create a Teacher class that extends Person and adds a subject property. Implement a teach method and create an instance of Teacher.

### **Part 6: Working with Modules**

**Creating and Importing Modules**Create two new files, mathUtils.ts and main.ts. Move the add and multiply functions into mathUtils.ts and export them:

| // mathUtils.ts  export function add(a: number, b: number): number {  return a + b;  }  export function multiply(a: number, b: number): number {  return a \* b;  } |
| --- |

**Import Functions**In main.ts, import the functions from mathUtils.ts and use them:

| // main.ts  import { add, multiply } from "./mathUtils";  console.log(add(3, 7));  console.log(multiply(4, 5)); |
| --- |

**Compile and Run**Compile all files and run main.ts:

| npx tsc  node main.js |
| --- |

1. **Task**Add a subtract function in mathUtils.ts and test it in main.ts.

### **Summary**

In this lab, you have:

* Set up a basic TypeScript environment.
* Explored TypeScript types, functions, interfaces, and classes.
* Worked with modules to organize code.

## Angular Architecture Overview

Angular is a component-based front-end framework structured around several core building blocks:

### **Core Building Blocks**

* **Components** – UI units that control parts of the screen.
* **Templates** – Define HTML layout and bind to component logic.
* **Directives** – Extend HTML with custom behavior.
* **Services** – Provide reusable business logic, injected via Dependency Injection.
* **Routing** – Manages navigation and views.
* **Modules / Standalone APIs** – Organize code and manage dependencies.

## **NgModules (Traditional Module-based Architecture)**

### **What is an NgModule?**

An NgModule (@NgModule) is a container that groups related components, directives, pipes, and services. It helps with:

* Declaring what components belong to this context.
* Importing other modules' functionality.
* Controlling the scope of features.
* Providing services at module or app level.

### **Pros:**

* Well-structured for large apps.
* Mature ecosystem with third-party modules.
* Angular CLI scaffolding uses this by default.

### **Cons:**

* Boilerplate-heavy: multiple declarations and imports needed.
* Difficult to tree-shake unused modules.
* Harder learning curve due to the indirect structure.

## **Standalone Components (Introduced in Angular 14+)**

### **What is a Standalone Component?**

A **standalone component** is a self-contained component that does **not require** inclusion in an NgModule. Declared using:

| @Component({  standalone: true,  imports: [CommonModule, RouterModule],  ...  }) |
| --- |

### **Pros:**

* Reduces boilerplate — no need for NgModules.
* Simplifies lazy loading and testing.
* Encourages modular thinking.
* Easier learning curve for beginners.

### **Cons:**

* Still being adopted by many libraries.
* Transitioning legacy apps from NgModules can be challenging.

## **Modules vs Standalone Components – Comparison Table**

| **Feature** | **NgModules** | **Standalone Components** |
| --- | --- | --- |
| Declaration | Uses @NgModule | Uses standalone: true in @Component |
| Dependency Management | Via imports in @NgModule | Via imports in @Component |
| Learning Curve | Steeper | Simpler |
| Boilerplate | Higher | Lower |
| Lazy Loading | Via routing modules | Direct and cleaner |
| Third-Party Library Support | Mature and fully supported | Still growing |
| Ideal Use Case | Large, legacy, or enterprise apps | Modern, lightweight apps |

**When to Use What?**

* **Use NgModules** if:  
  + You're working on an existing project using modules.
  + You need compatibility with libraries expecting module-based setup.
  + You want centralized configuration for features.
* **Use Standalone Components** if:  
  + You're starting a new project.
  + You want a simpler, more modular approach.
  + You aim for tree-shaking and performance optimization.

## **Hybrid Approach (Recommended in Angular 15+)**

Angular allows combining both:

* Use **NgModules** for global/shared configurations.
* Use **standalone components** for individual features and lazy-loaded routes.

This modular evolution enables Angular to support both traditional enterprise architecture and modern web app practices.

## 

## **Angular CLI with Standalone Apps**

Angular CLI (Command Line Interface) is a powerful tool to scaffold, develop, test, and build Angular applications. As of Angular v14 and beyond, CLI fully supports **standalone components**, allowing for more modular, lightweight applications without relying on traditional NgModules.

### **Key Benefits of Using Angular CLI with Standalone Apps**

* **Faster bootstrapping** with standalone: true
* **Simplified structure**: no need to create NgModules for each feature
* **Out-of-the-box tooling support**: routing, lazy loading, testing still work seamlessly
* **CLI commands updated** to support standalone flag and routing options

## **Creating a Standalone App**

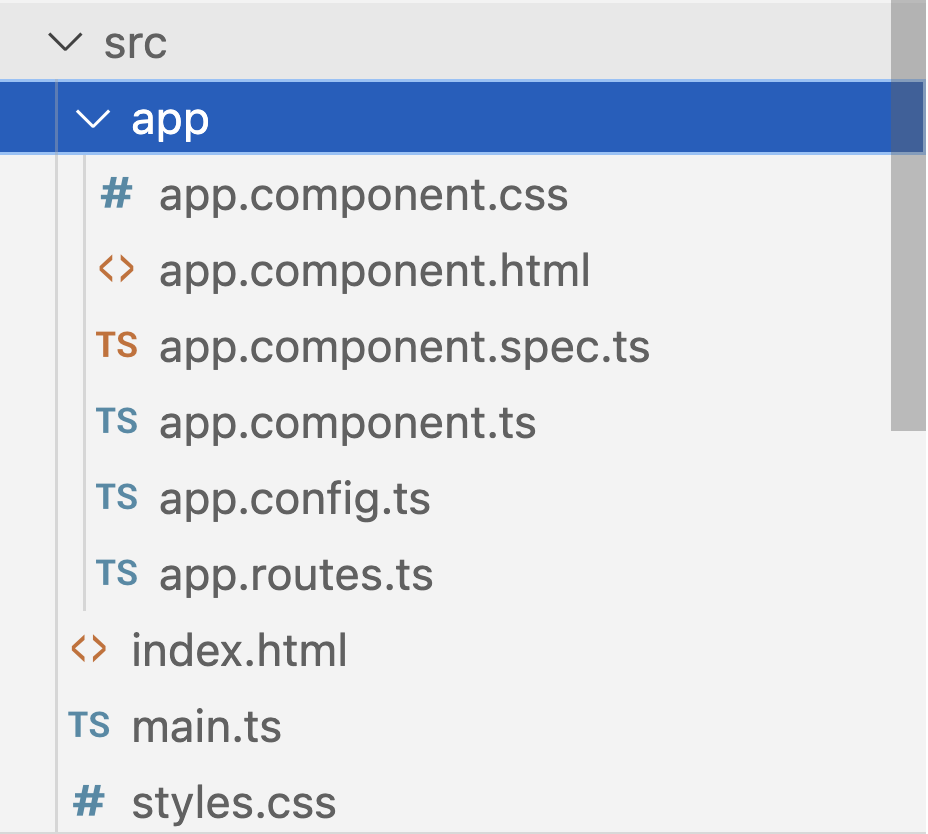
To create a new Angular project **with standalone APIs**, use the CLI with the --standalone flag:

| ng new my-standalone-app --standalone |
| --- |

This:

* Sets up the app without an AppModule
* Bootstraps AppComponent directly via main.ts
* Uses provideRouter() and importProvidersFrom() in main.ts

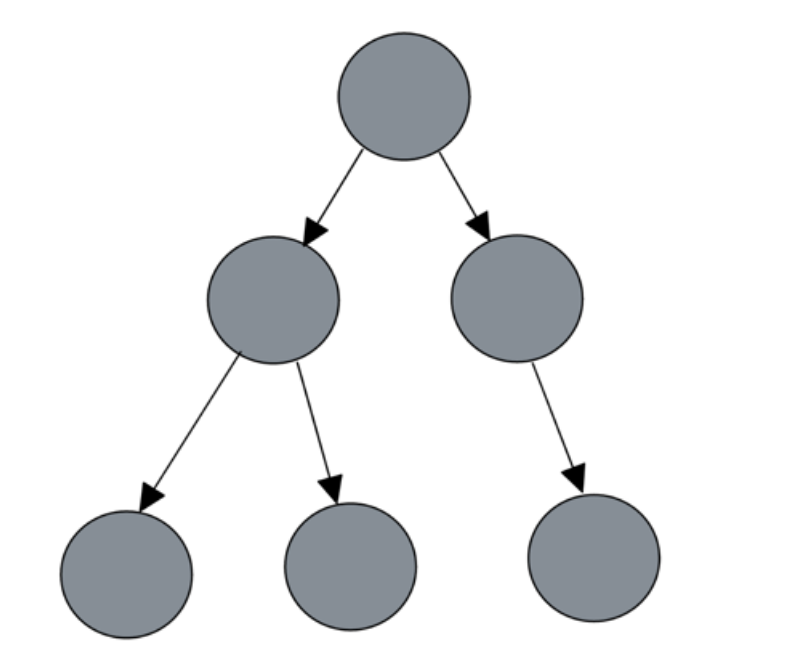
## **Folder Structure of a Standalone App**



## **Example main.ts in Standalone App**

| import { bootstrapApplication } from '@angular/platform-browser';  import { appConfig } from './app/app.config';  import { AppComponent } from './app/app.component';  bootstrapApplication(AppComponent, appConfig)  .catch((err) => console.error(err)); |
| --- |

## **Generating Standalone Components via CLI**



Components are the foundational building blocks for any Angular application. Each component has three parts:

* TypeScript class
* HTML template
* CSS styles

You can create a standalone component using:

| ng generate component example --standalone |
| --- |

Or shorthand:

| ng g c example --standalone |
| --- |

This creates:

* example.component.ts with standalone: true
* Automatically adds necessary imports in the component

## **Configuring Routing in Standalone Apps**

Define routes using Route[] array in app.routes.ts:

| import { Routes } from '@angular/router';  import { HomeComponent } from './home/home.component';  export const appRoutes: Routes = [  { path: '', component: HomeComponent },  ]; |
| --- |

Use provideRouter(appRoutes) in main.ts to enable routing.

## **Testing Support**

* Angular CLI continues to support **unit tests** and **end-to-end tests** using the same structure.
* Use TestBed.configureTestingModule() and import standalone components directly.

Example:

| TestBed.configureTestingModule({  imports: [MyStandaloneComponent],  }); |
| --- |

## **Lazy Loading Standalone Components**

Lazy loading is cleaner with standalone components:

| {  path: 'about',  loadComponent: () => import('./about/about.component').then(m => m.AboutComponent)  } |
| --- |

No need for loadChildren or a separate module.

## **Migrating to Standalone with CLI**

Use Angular CLI with **ng generate** and **standalone flags** to refactor existing features one by one:

| ng generate component dashboard --standalone |
| --- |

Then update routes and providers accordingly.

## **Summary: Best Practices**

| **Practice** | **Tip** |
| --- | --- |
| Use --standalone flag | When generating components, pages, and new projects |
| Organize routes in a separate file | For cleaner and scalable routing |
| Use bootstrapApplication() | Instead of AppModule for app initialization |
| Prefer provideRouter() & provideHttpClient() | For global service registration |
| Combine with signals and zoneless | For cutting-edge performance and architecture |

Angular CLI remains a robust tool for modern Angular development, and with full standalone support, it enables faster, cleaner, and more modular apps.

## 

## Angular Folder Structure Walkthrough

When you create an Angular project using the CLI (ng new), the tool generates a standardized directory structure to help organize your codebase. Understanding this structure is crucial for

**Default Project Structure**

| my-angular-app/  ├── e2e/ # End-to-end testing setup (optional in Angular 15+)  ├── node\_modules/ # Installed dependencies  ├── src/ # Main application source code  │ ├── app/ # Application-specific code  │ ├── assets/ # Static assets (images, fonts, etc.)  │ ├── environments/ # Environment-specific config files  │ ├── index.html # Main HTML file  │ ├── main.ts # Entry point of the app  │ ├── styles.css # Global styles  │ └── ... # Angular-specific config files  ├── angular.json # Angular project configuration  ├── package.json # Project metadata and dependencies  ├── tsconfig.json # TypeScript configuration  └── README.md # Project documentation |
| --- |

**src/app/ Folder (Core of Your Application)**

This is where your application lives.

**For Traditional (NgModule-based) Architecture:**

| src/app/  ├── app.module.ts # Root Angular module  ├── app.component.ts # Root component (with template, style, logic)  ├── app.component.html  ├── app-routing.module.ts # Centralized routing configuration  ├── shared/ # Shared components, pipes, directives  ├── core/ # Singleton services, guards, interceptors  ├── features/ # Feature modules (e.g., dashboard, users, etc.)  └── services/ # Reusable services (optional if not using core/) |
| --- |

**For Standalone Component Architecture:**

| src/app/  ├── app.component.ts # Root component with `standalone: true`  ├── app.routes.ts # Routes defined using `Routes[]`  ├── dashboard/ # Feature folder with standalone component(s)  │ ├── dashboard.component.ts  │ └── dashboard.component.html  ├── shared/ # Shared standalone components, pipes, directives  ├── services/ # Injectable services  └── guards/ # Route guards (optional) |
| --- |

**src/environments/ Folder**

Contains environment-specific configuration files used at build time.

| environments/  ├── environment.ts # Default (development)  └── environment.prod.ts # Production config |
| --- |

Use these to store different base URLs, feature flags, or settings:

| export const environment = {  production: false,  apiBaseUrl: 'http://localhost:3000/api'  }; |
| --- |

**src/assets/ Folder**

Stores static files like:

* Images
* Icons
* JSON files
* Fonts

These are served as-is at runtime and can be referenced in templates or styles:

| <img src="assets/logo.png"> |
| --- |

**angular.json**

Controls the overall build configuration, including:

* File replacements for environments
* Styles and scripts
* Output paths
* Assets copying
* Builder configurations (e.g., SSR, prerender)

**main.ts**

This is the **entry point** of your app.

In module-based apps:

| platformBrowserDynamic().bootstrapModule(AppModule) |
| --- |

In standalone apps:

| bootstrapApplication(AppComponent, { providers: [...] }); |
| --- |

**styles.css or styles.scss**

Global styles applied across the app.

You can:

* Import theme styles
* Define root variables
* Include font links

## **Recommended Folder Practices**

| **Folder/Feature** | **Best Practice** |
| --- | --- |
| features/ | Group components by feature rather than type |
| shared/ | Use for pure UI components, pipes, directives shared globally |
| core/ | Use for singleton services (e.g., Auth, API Interceptors) |
| routes/ or app.routes.ts | Use central file for routing if using standalone setup |
| guards/, models/ | Keep reusable logic types and guards organized |

## **Summary**

* Angular CLI provides a structured and scalable project layout.
* Use **feature-based** organization for larger apps.
* Prefer **standalone components** for modular, lazy-loadable architecture.
* Keep **shared and core logic** separated for clarity and reusability.

This structure helps streamline development across teams and makes future maintenance easier.

## 

## **Interpolation in Angular**

### **What is Interpolation?**

**Interpolation** in Angular is a technique to **bind component data to the HTML template** using double curly braces ({{ ... }}).

It allows you to **display dynamic values** (e.g., strings, numbers, expressions) in the UI.

## **Syntax**

| <p>Hello, {{ name }}!</p> |
| --- |

The value inside {{ }} is an **expression** evaluated by Angular.

## **Common Use Cases**

| **Use Case** | **Example** |
| --- | --- |
| Display a string property | {{ userName }} |
| Call a method | {{ getFullName() }} |
| Evaluate an expression | {{ age > 18 ? 'Adult' : 'Minor' }} |
| Math operations | {{ price \* quantity }} |
| Date/time formatting (via method or pipe) | {{ getDate() }} or `{{ today |

## **Example Component**

| @Component({  selector: 'app-user',  template: `  <h2>Welcome, {{ firstName }} {{ lastName }}!</h2>  <p>Age: {{ age }}</p>  <p>Status: {{ age >= 18 ? 'Adult' : 'Minor' }}</p>  `  })  export class UserComponent {  firstName = 'Alice';  lastName = 'Ng';  age = 21;  } |
| --- |

## **What You Can’t Do in Interpolation**

| **Not Allowed** | **Why** |
| --- | --- |
| Assignments (=) | Interpolation is read-only |
| Control structures (if, for) | Use Angular directives instead (\*ngIf, \*ngFor) |
| Access to global variables | Limited to component context |

## **Behind the Scenes**

Angular:

1. Evaluates the expression inside {{ }}
2. Re-renders the DOM when the value changes
3. Uses **change detection** to keep the UI in sync

## **Best Practices**

* Keep expressions simple and fast (avoid heavy logic)
* Prefer calling methods only when they return static values
* Avoid using functions that do calculations or trigger side effects in interpolation

## **Summary**

| **Feature** | **Description** |
| --- | --- |
| Syntax | {{ expression }} |
| Purpose | Display dynamic data from the component |
| Evaluation | One-way binding (view updates from data) |
| Scope | Expression must exist in the component class |

## **Angular’s New Control Flow Syntax**

As of **Angular 17**, Angular introduces a **new built-in control flow syntax** using the @ symbol, replacing older structural directives like \*ngIf, \*ngFor, and \*ngSwitch. This new syntax is more powerful, readable, and aligned with modern frontend frameworks.

## **Why New Control Flow?**

* **More intuitive and readable**: Syntax closer to JavaScript logic.
* **Improved performance**: Built into the Angular compiler (not directive-based).
* **Better debugging**: Errors and logic are easier to trace.
* **Supports block scoping**: Variables declared in one block don't leak outside.

## **@if – Conditional Rendering**

### **Syntax:**

| @if (isLoggedIn) {  <p>Welcome, user!</p>  } @else {  <p>Please log in.</p>  } |
| --- |

### **Benefits:**

* No need for \*ngIf or ng-template.
* Clear separation between @if and @else blocks.
* No need for <ng-container> wrappers.

## **@for – Looping Over Data**

### **Syntax:**

| @for (user of users; track user.id) {  <li>{{ user.name }}</li>  } |
| --- |

### **Features:**

* Native support for **tracking by identity** (track user.id)

Supports index, first, last, etc.:

| @for (item of items; let i = $index) {  <p>Item {{ i }}: {{ item }}</p>  } |
| --- |

### **Replaces:**

| \*ngFor="let item of items; let i = index" |
| --- |

## **@switch, @case, @default**

### **Syntax:**

| @switch (status) {  @case ('pending') {  <p>Loading...</p>  }  @case ('success') {  <p>Data loaded!</p>  }  @default {  <p>Unknown status</p>  }  } |
| --- |

## **Block-Scoped Variables**

Variables declared with let inside control blocks are scoped only to that block, avoiding accidental overrides.

| @for (product of products; let index = $index) {  <div>Product #{{ index }}: {{ product.name }}</div>  } |
| --- |

## **Comparison with Legacy Syntax**

| **Feature** | **Old Syntax** | **New Syntax** |
| --- | --- | --- |
| If condition | \*ngIf="x" | @if (x) {} |
| If-else | \*ngIf... else template | @if... @else |
| Loop | \*ngFor | @for (x of y) |
| Switch-case | \*ngSwitch, \*ngSwitchCase | @switch... @case |
| Nesting | Uses <ng-container> | Clean nested blocks |

## **Summary**

| **Feature** | **Description** |
| --- | --- |
| @if, @else | Inline conditional blocks |
| @for | Cleaner, enhanced looping with tracking |
| @switch, @case, @default | Declarative multi-branch logic |
| Benefits | Better readability, performance, and maintainability |

This new syntax is a major step in making Angular more expressive and modern while improving performance and DX (developer experience).

## 

## **Lab 1: Install Angular CLI**

### **Objective:**

By the end of this lab, you will have installed the Angular CLI globally on your machine, enabling you to create and manage Angular projects.

### **Prerequisites:**

* Ensure **Node.js** and **npm** are installed:

You can check by running:

| node -v  npm -v |
| --- |

* + If not installed, download from<https://nodejs.org>

### **Step-by-Step Instructions**

#### **Step 1: Open Terminal / Command Prompt**

* Windows: Use **Command Prompt** or **PowerShell**
* macOS/Linux: Use **Terminal**

#### **Step 2: Install Angular CLI Globally**

Type the following command to install Angular CLI globally:

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

* The -g flag installs it **globally**, so you can use Angular CLI in any project folder.

#### **Step 3: Verify the Installation**

After installation, check the installed version:

| ng version |
| --- |

Expected output will show Angular CLI version and other related packages.

### **Notes:**

You may need to run the install command as an administrator or use sudo on macOS/Linux:

| sudo npm install -g @angular/cli |
| --- |

* If you get permission errors on macOS/Linux, consider using **Node Version Manager (nvm)** to manage permissions and node versions.

### **Outcome:**

You have successfully installed Angular CLI and are ready to create Angular applications using commands like:

| ng new my-app  cd my-app  ng serve |
| --- |

## 

## **Lab 2: Scaffold a New Angular Standalone Project**

### **Objective:**

Create a new Angular 19 application using the standalone component architecture via the Angular CLI.

### **Prerequisites:**

Angular CLI installed globally

Confirm with:

| ng version |
| --- |

Node.js and npm installed

Confirm with:

| node -v  npm -v |
| --- |

### **Step-by-Step Instructions**

#### **Step 1: Open Terminal / Command Prompt**

Use the terminal appropriate to your OS:

* Windows: Command Prompt or PowerShell
* macOS/Linux: Terminal

#### **Step 2: Run the Scaffold Command**

Run the following command to generate a new standalone Angular project:

| ng new angular19-app --standalone |
| --- |

This will:

* Create a new project named angular19-app
* Use the standalone component API (no AppModule)
* Automatically configure routing and structure for standalone mode

#### **Step 3: Choose Configuration Options**

During setup, you’ll be prompted to choose:

1. **Add Angular routing?** → Yes
2. **Which stylesheet format?** → Choose one (CSS, SCSS, etc.)

The CLI will scaffold the project with standalone component support.

#### **Step 4: Navigate into Your Project**

| cd angular19-app |
| --- |

#### **Step 5: Serve the Application**

Run the app locally to verify the setup:

| ng serve |
| --- |

Then open your browser and visit:  
<http://localhost:4200>

### **Resulting Folder Highlights**

| angular19-app/  ├── src/  │ ├── app/  │ │ ├── app.component.ts // Standalone root component  │ │ └── app.routes.ts // Route configuration  │ ├── main.ts // Uses bootstrapApplication() |
| --- |

### **Outcome:**

You’ve successfully created an Angular 19 project using the **standalone component architecture** and verified that it runs correctly in the browser.

## 

## **Lab 2: Practice Interpolation in Angular**

### **Objective:**

Learn how to use **interpolation ({{ }})** in Angular to:

* Display string and number variables
* Evaluate expressions
* Call methods in the template

### **Step-by-Step Instructions**

### **Step 1: Create a New Component**

In your Angular project directory, generate a standalone component:

| ng generate component components/interpolation-demo --standalone |
| --- |

### **Step 2: Update the Component Class**

Open src/app/components/interpolation-demo/interpolation-demo.component.ts and modify the class:

| import { Component } from '@angular/core';  import { CommonModule } from '@angular/common';  @Component({  selector: 'app-interpolation-demo',  standalone: true,  imports: [CommonModule],  templateUrl: './interpolation-demo.component.html'  })  export class InterpolationDemoComponent {  firstName = 'Alice';  lastName = 'Tan';  age = 22;  quantity = 3;  pricePerItem = 50;  get fullName(): string {  return `${this.firstName} ${this.lastName}`;  }  getTotalPrice(): number {  return this.quantity \* this.pricePerItem;  }  } |
| --- |

### **Step 3: Add Interpolation to the Template**

Open interpolation-demo.component.html and add the following:

| <h2>Hello, {{ firstName }}!</h2>  <p>Your full name is: {{ fullName }}</p>  <p>You are {{ age }} years old.</p>  <p>Status: {{ age >= 18 ? 'Adult' : 'Minor' }}</p>  <hr />  <h3>🛒 Shopping Cart</h3>  <p>Quantity: {{ quantity }}</p>  <p>Price per item: RM{{ pricePerItem }}</p>  <p>Total price (expression): RM{{ quantity \* pricePerItem }}</p>  <p>Total price (method): RM{{ getTotalPrice() }}</p> |
| --- |

### **Step 4: Render the Component in main.ts**

In src/main.ts, replace the component if needed:

| import { bootstrapApplication } from '@angular/platform-browser';  import { InterpolationDemoComponent } from './app/components/interpolation-demo/interpolation-demo.component';  bootstrapApplication(InterpolationDemoComponent); |
| --- |

### **Step 5: Run the App**

| ng serve |
| --- |

Open<http://localhost:4200>

* See your name and age displayed dynamically
* Watch computed values and expressions update
* Confirm that values are displayed using {{ }}

**Optional Enhancements**

* Change the values in the component class and observe updates.
* Add a "toggle name" method and bind it to a button (bonus).
* Add a currentDate variable and show it using {{ currentDate | date:'fullDate' }}.

**Outcome:**

You’ve successfully:

* Used **interpolation** to display dynamic data in the UI
* Evaluated **expressions and method calls** in the template
* Built a standalone Angular component using **interpolation**

## **Lab 3: Build a Basic Standalone Component with @if and @for**

### **Objective:**

Create a standalone component that:

* Displays a list of items using @for
* Shows a conditional message using @if based on the list content

### **Step-by-Step Instructions**

#### **Step 1: Open Your Project Directory**

If you haven’t yet, navigate to your Angular project folder:

| cd angular19-app |
| --- |

#### **Step 2: Generate a Standalone Component**

Use the Angular CLI to generate a new standalone component:

| ng generate component item-list --standalone |
| --- |

This creates a new component with standalone: true automatically added.

#### **Step 3: Add the Component to Your Routes**

Edit src/app/app.routes.ts and add the route for ItemListComponent:

| import { Routes } from '@angular/router';  import { ItemListComponent } from './item-list/item-list.component';  export const appRoutes: Routes = [  { path: '', component: ItemListComponent }  ]; |
| --- |

#### **Step 4: Update the Component Logic**

Open src/app/item-list/item-list.component.ts and update it as follows:

| import { Component } from '@angular/core';  import { CommonModule } from '@angular/common';  @Component({  selector: 'app-item-list',  standalone: true,  imports: [CommonModule],  templateUrl: './item-list.component.html',  })  export class ItemListComponent {  items = ['Apple', 'Banana', 'Orange'];  } |
| --- |

#### **Step 5: Use @if and @for in the Template**

Edit src/app/item-list/item-list.component.html:

| <h2>My Fruits List</h2>  @if (items.length > 0) {  <ul>  @for (item of items; let i = $index) {  <li>{{ i + 1 }}. {{ item }}</li>  }  </ul>  } @else {  <p>No items found.</p>  } |
| --- |

#### **Step 6: Run the App**

| ng serve |
| --- |

Then open your browser and go to:

<http://localhost:4200>

You should see a list of fruits. Try emptying the list to see the @else condition.

### **Outcome:**

You successfully created a **standalone Angular component** and used the new Angular 17+ **control flow syntax (@if, @for)** to render content conditionally and dynamically.

## **Lab Activity: Use Signals for Simple State (Counter Example)**

### **Objective:**

Learn how to use Angular’s reactive **signals API** to manage state by building a simple counter component.

### **Step-by-Step Instructions**

#### **Step 1: Open Your Project**

Make sure you are inside your standalone Angular project directory:

| cd angular19-app |
| --- |

#### **Step 2: Generate a New Standalone Component**

| ng generate component counter --standalone |
| --- |

#### **Step 3: Add the Counter Route**

Update src/app/app.routes.ts to include the new CounterComponent:

| import { Routes } from '@angular/router';  import { CounterComponent } from './counter/counter.component';  export const appRoutes: Routes = [  { path: '', component: CounterComponent }  ]; |
| --- |

#### **Step 4: Update Component Logic Using Signals**

Edit src/app/counter/counter.component.ts:

| import { Component, computed, signal } from '@angular/core';  import { CommonModule } from '@angular/common';  @Component({  selector: 'app-counter',  standalone: true,  imports: [CommonModule],  templateUrl: './counter.component.html',  })  export class CounterComponent {  // Signal to hold the counter state  count = signal(0);  // Optional: computed signal to determine if count is even or odd  isEven = computed(() => this.count() % 2 === 0);  increment() {  this.count.update(value => value + 1);  }  decrement() {  this.count.update(value => value - 1);  }  reset() {  this.count.set(0);  }  } |
| --- |

#### **Step 5: Build the Template with Bindings**

Edit src/app/counter/counter.component.html:

| <h2> Counter Example with Signals</h2>  <p>Current Count: {{ count() }}</p>  <p \*ngIf="isEven()">The number is even </p>  <p \*ngIf="!isEven()">The number is odd </p>  <button (click)="increment()">➕ Increment</button>  <button (click)="decrement()">➖ Decrement</button>  <button (click)="reset()"> Reset</button> |
| --- |

#### **Step 6: Run Your App**

| ng serve |
| --- |

Navigate to:  
<http://localhost:4200>

Test the counter functionality and observe how the view reacts instantly to state changes.

### **Bonus Tip:**

You can replace the \*ngIf with the new @if syntax if using Angular 17+:

| @if (isEven()) {  <p>The number is even </p>  } @else {  <p>The number is odd </p>  } |
| --- |

### **Outcome:**

You’ve built a simple **signal-powered counter** using Angular’s new reactivity model. You now understand how to:

* Create and update **signals**
* Use **computed** for derived state
* Bind signals in the template

## 

## **Property Binding in Angular**

### **What is Property Binding?**

**Property binding** is a technique in Angular that allows you to **set the value of an HTML element property** dynamically from the component.

It uses **square brackets** [] syntax to bind a DOM property to a **component class property**.

## **Syntax**

| [elementProperty]="componentProperty" |
| --- |

### **Example:**

| <img [src]="imageUrl" /> |
| --- |

This binds the <img> element’s src attribute to the imageUrl property in the component.

## **Common Use Cases**

| **HTML Property** | **Binding Example** | **Purpose** |
| --- | --- | --- |
| src | <img [src]="profileImage"> | Bind image URL |
| href | <a [href]="linkUrl"> | Dynamic anchor link |
| disabled | <button [disabled]="isDisabled"> | Enable/disable elements |
| value | <input [value]="username"> | Pre-fill form fields |
| class | <div [class]="dynamicClass"> | Bind dynamic CSS class |
| style | <div [style.color]="textColor"> | Set inline style dynamically |

## **Behind the Scenes**

Angular evaluates the **right-hand expression** and updates the **left-hand DOM property** automatically when data changes, using **change detection**.

It’s a **one-way data binding**:  
 Component -> Template

## **Property Binding vs Attribute Binding**

| **Concept** | **Property Binding** | **HTML Attribute Binding** |
| --- | --- | --- |
| Syntax | [property]="value" | attribute="value" |
| Binding Target | DOM property | HTML attribute |
| Reactivity | Yes (dynamic updates) | No (static) |
| Example | <input [value]="name"> | <input value="John"> |

Example: <input [disabled]="true"> disables the input, while <input disabled="false"> still disables it (HTML treats any disabled as true).

## **Best Practices**

* Use property binding for all **dynamic values** in the DOM.
* Don’t mix static and dynamic bindings (e.g., avoid src="{{ url }}").
* Prefer **property binding over interpolation** for setting element properties.

## **Component Example**

### **app.component.ts**

| export class AppComponent {  userName = 'Alice';  isSubmitDisabled = true;  profilePic = 'assets/user.jpg';  } |
| --- |

### **App.component.html**

| <h2>Hello, {{ userName }}</h2>  <img [src]="profilePic" alt="Profile Picture">  <button [disabled]="isSubmitDisabled">Submit</button> |
| --- |

## **Summary**

| **Feature** | **Description** |
| --- | --- |
| Purpose | Dynamically bind component data to element properties |
| Syntax | [property]="expression" |
| Data flow | One-way: Component ➡️ View |
| Common usage | src, href, disabled, value, style, etc. |

## **Event Handling in Angular**

### **What is Event Handling?**

**Event handling** in Angular refers to responding to user interactions in the UI, such as **clicks**, **typing**, **form submissions**, and more.

Angular uses **event binding** to connect **DOM events** (like click, input, submit) to **methods in your component**.

## **Syntax**

| (element event)="componentMethod($event)" |
| --- |

The $event variable gives you access to the DOM event object.

## **Common Event Bindings**

| **Event Type** | **Usage Example** | **Triggered By** |
| --- | --- | --- |
| click | <button (click)="onClick()"> | Button click |
| input | <input (input)="onInput($event)"> | User typing |
| keyup | <input (keyup.enter)="onEnter()"> | Enter key pressed |
| submit | <form (submit)="onSubmit()"> | Form submission |
| change | <select (change)="onSelect($event)"> | Dropdown value changed |
| mouseenter | <div (mouseenter)="hover()"> | Mouse hover |

## **Component Example**

### **app.component.ts**

| export class AppComponent {  count = 0;  name = '';  increment() {  this.count++;  }  onInput(event: Event) {  const inputElement = event.target as HTMLInputElement;  this.name = inputElement.value;  }  } |
| --- |

### **app.component.html**

| <h2>Click Counter: {{ count }}</h2>  <button (click)="increment()">Click me</button>  <hr />  <input (input)="onInput($event)" placeholder="Type your name" />  <p>Hello, {{ name }}!</p> |
| --- |

## **Behind the Scenes**

* Angular listens to DOM events using the (event) syntax.
* When the event occurs, it triggers the method in the component.
* The view updates automatically using **change detection**.

## **Best Practices**

* Use **arrow functions sparingly** in templates to avoid performance issues.
* Prefer **separate methods** in the component for clarity.
* Use $event only when you need to access event details (e.g., input value, target).
* Combine with @Output() and EventEmitter for **child-to-parent** communication.

## **Comparison with Other Bindings**

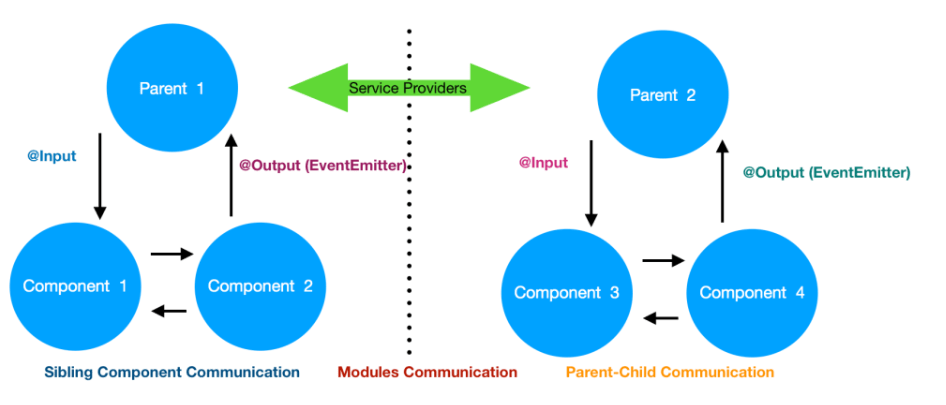
| **Binding Type** | **Syntax** | **Purpose** |
| --- | --- | --- |
| **Property** | [prop]="value" | Pass data from component → view |
| **Event** | (event)="method" | Trigger method on DOM event |
| **Two-way** | [(ngModel)]="val" | Sync data both ways (Forms) |

## **Summary**

| **Concept** | **Description** |
| --- | --- |
| Event Binding | Connects UI events to component methods |
| Syntax | (eventName)="methodName($event)" |
| Usage | Clicks, input, key presses, mouse events, etc. |
| Data Flow | View → Component (opposite of property binding) |

## **Angular Component Interaction**

In Angular, components often need to communicate with each other — typically from **parent to child** (passing data) and from **child to parent** (sending events).



## **@Input() – Passing Data from Parent to Child**

### **Purpose:**

Allows a parent component to bind values to a child component’s properties.

### **Syntax in Child Component:**

| import { Component, Input } from '@angular/core';  @Component({  selector: 'app-child',  template: `<p>Hello, {{ name }}!</p>`  })  export class ChildComponent {  @Input() name!: string;  } |
| --- |

### **Usage in Parent Template:**

| <app-child [name]="'Alice'"></app-child> |
| --- |

* Use @Input() for one-way data binding: parent -> child

## **@Output() – Sending Events from Child to Parent**

### **Purpose:**

Allows a child component to **emit events** that a parent can listen to.

### **Syntax in Child Component:**

| import { Component, Output, EventEmitter } from '@angular/core';  @Component({  selector: 'app-child',  template: `<button (click)="sendMessage()">Send</button>`  })  export class ChildComponent {  @Output() message = new EventEmitter<string>();  sendMessage() {  this.message.emit('Hello from Child!');  }  } |
| --- |

### **Usage in Parent Template:**

| <app-child (message)="handleMessage($event)"></app-child> |
| --- |

| handleMessage(msg: string) {  console.log(msg); // Output: Hello from Child!  } |
| --- |

* Use @Output() and EventEmitter for child → parent communication.

## **Example: Combined @Input and @Output**

**Child Component:**

| @Component({  selector: 'app-counter',  template: `  <p>Count: {{ count }}</p>  <button (click)="increment()">Increment</button>  `  })  export class CounterComponent {  @Input() count = 0;  @Output() countChange = new EventEmitter<number>();  increment() {  this.countChange.emit(this.count + 1);  }  } |
| --- |

**Parent Component:**

| <app-counter [count]="parentCount" (countChange)="parentCount = $event"></app-counter> |
| --- |

| parentCount = 5; |
| --- |

## **Best Practices**

| **Practice** | **Recommendation** |
| --- | --- |
| Naming @Input() | Use clear and intuitive names |
| Naming @Output() | Use eventName or eventNameChange convention |
| Avoid logic in templates | Handle event logic in the component class |
| Two-way binding (optional) | Combine @Input() and @Output() with [(...)] syntax |

## **Two-Way Binding**

Angular allows simplified two-way binding with custom components using:

| <app-counter [(count)]="parentCount"></app-counter> |
| --- |

This works only if:

* You define @Input() count
* You define @Output() countChange

## **Summary**

| **Direction** | **Technique** | **Syntax** |
| --- | --- | --- |
| Parent → Child | @Input() | [property]="value" |
| Child → Parent | @Output() + EventEmitter | (event)="handler($event)" |
| Two-way binding | @Input() + @Output() + [(...)] | [(property)]="value" |

## 

## **Signals Deep Dive in Angular**

Angular **Signals** provide a modern, reactive way to manage state in your components. They enable fine-grained reactivity with automatic change tracking and efficient updates.

## **1. Signals Overview**

### **What is a Signal?**

A **Signal** is a reactive primitive that holds a value and notifies dependents when it changes.

| import { signal } from '@angular/core';  const count = signal(0);  count(); // get value → 0  count.set(5); // update value |
| --- |

## **2. computed() – Derive State from Other Signals**

### **Purpose:**

Create a **read-only signal** whose value is derived from other signals.

| const count = signal(2);  const double = computed(() => count() \* 2);  double(); // → 4  count.set(4);  double(); // → 8 (auto-updated) |
| --- |

### **Key Points:**

* Automatically tracks dependencies.
* Recalculates only when necessary.
* Used for **derived values**, e.g., filters, totals, validation.

## **3. effect() – React to Signal Changes**

### **Purpose:**

Run **side effects** (e.g., logging, DOM updates, API calls) when one or more signals change.

| const count = signal(0);  effect(() => {  console.log('Count changed to:', count());  });  count.set(3); // Logs: "Count changed to: 3" |
| --- |

### **Best Practices:**

* Avoid using effect() for heavy computations.
* Ideal for **triggering effects** outside Angular’s rendering cycle.
* Cleanup happens automatically when the effect is destroyed.

**Signals vs Other Reactivity Approaches**

| **Feature** | **Signal** | **RxJS Observable** | **@Input() / @Output()** |
| --- | --- | --- | --- |
| Synchronous | Yes | No (async) | Yes |
| Push-based | Yes | Yes | Yes |
| Easy to use | Very | Steep learning curve | Moderate |
| Lifecycle-aware | Yes | Yes | Yes |
| Lazy updates | Yes (computed, effect) | Manual unsub needed | Yes |

## 

## **Real-World Use Cases**

| **Scenario** | **Use** |
| --- | --- |
| Form validation rules | computed() |
| Updating the DOM or triggering events | effect() |
| Showing filtered lists or totals | computed() |
| Triggering localStorage/API calls on change | effect() |

## **Summary**

| **Signal Tool** | **Description** | **Use Case Example** |
| --- | --- | --- |
| signal() | Holds a reactive value | State (e.g., count, user) |
| computed() | Creates a value based on other signals | Derived data (e.g., total price) |
| effect() | Triggers side-effects when signals change | Log, API call, DOM update |

Angular Signals simplify reactive state management and improve performance with a clean, composable, and synchronous API — ideal for modern UI development.

## 

## **Angular Services & Dependency Injection (DI)**

### **What is a Service?**

A **Service** in Angular is a class used to encapsulate reusable logic that can be shared across components. Examples include:

* API communication
* Business logic
* State management
* Utility functions

## **What is Dependency Injection (DI)?**

**Dependency Injection** is a design pattern where a class receives its dependencies from an external source (the Angular injector), rather than creating them itself.

Angular’s DI system:

* Creates and manages service instances
* Injects services wherever needed (components, other services)
* Controls service scope and lifecycle

## **Creating a Service**

Use the Angular CLI to generate a service:

| ng generate service user |
| --- |

This creates:

| src/app/user.service.ts |
| --- |

| import { Injectable } from '@angular/core';  @Injectable({  providedIn: 'root' // Registers the service globally (singleton)  })  export class UserService {  getUserName() {  return 'Alice';  }  } |
| --- |

## **Injecting a Service into a Component**

You can use constructor injection to consume a service in a component:

| import { Component } from '@angular/core';  import { UserService } from './user.service';  @Component({  selector: 'app-profile',  template: `<p>Welcome, {{ username }}</p>`  })  export class ProfileComponent {  username: string;  constructor(private userService: UserService) {  this.username = this.userService.getUserName();  }  } |
| --- |

The Angular injector automatically provides an instance of UserService.

## **Service Registration Options**

| **Method** | **Scope** | **Use Case** |
| --- | --- | --- |
| @Injectable({ providedIn: 'root' }) | Singleton across entire app | Default and recommended |
| @Injectable({ providedIn: 'any' }) | New instance per lazy-loaded module | Feature isolation |
| providers: [MyService] in component | New instance for that component only | Per-component behavior, testing |

## **Service-to-Service Injection**

| @Injectable({ providedIn: 'root' })  export class AuthService {  constructor(private http: HttpClient) {}  } |
| --- |

## **Testing with DI**

Angular’s TestBed makes it easy to mock and test services:

| TestBed.configureTestingModule({  providers: [UserService]  });  const service = TestBed.inject(UserService); |
| --- |

## **Benefits of Angular DI**

* **Reusability**: Services can be shared across components.
* **Encapsulation**: Keeps logic out of the UI layer.
* **Scalability**: Easier to manage dependencies in large apps.
* **Testability**: Services can be easily mocked and tested.

**Summary**

| **Concept** | **Description** |
| --- | --- |
| Service | A class for business or reusable logic |
| Injectable | Decorator that makes a class injectable |
| DI | Angular automatically injects dependencies |
| ProvidedIn | Controls service scope (app, module, component) |
| Use Cases | APIs, auth, utilities, state sharing |

## 

## **Angular Lifecycle with Signals**

When working with **signals** in Angular, you may want to run side effects or cleanup logic tied to the component lifecycle — especially during setup and teardown. Angular provides tools like effect() and DestroyRef to handle this reactively and cleanly.

## **1. effect() – Reacting to Signal Changes**

### **Purpose:**

effect() lets you define **reactive side effects** that automatically run when signals they depend on change.

### **Example:**

| import { signal, effect } from '@angular/core';  const count = signal(0);  effect(() => {  console.log('Count is now:', count());  }); |
| --- |

Runs once immediately, then re-runs whenever count() changes.

## **Side Effects in Components**

You often use effect() in a component's constructor or lifecycle setup code.

| @Component({  ...  })  export class CounterComponent {  count = signal(0);  constructor() {  effect(() => {  console.log('Counter:', this.count());  });  }  } |
| --- |

## **2. DestroyRef – Tied to Component Lifecycle**

### **Purpose:**

DestroyRef allows you to **clean up** reactive effects when the component is destroyed, preventing memory leaks.

### **Usage Example:**

| import { Component, DestroyRef, effect, inject } from '@angular/core';  @Component({  ...  })  export class ExampleComponent {  value = signal(0);  destroyRef = inject(DestroyRef);  constructor() {  const cleanupEffect = effect(() => {  console.log('Value changed:', this.value());  });  this.destroyRef.onDestroy(() => cleanupEffect.destroy());  }  } |
| --- |

### **What Happens:**

* effect() starts tracking reactive changes.
* When the component is destroyed, cleanupEffect.destroy() is called automatically.

## **Best Practices for Lifecycle Management with Signals**

| **Feature** | **Purpose** | **When to Use** |
| --- | --- | --- |
| effect() | Run side effects on signal change | Logging, fetch, update UI |
| DestroyRef | Clean up resources when component is destroyed | Unsubscribing, canceling effects |
| inject() | Access DestroyRef without constructor injection | For cleaner, signal-based setup |

## **Signals + Lifecycle Tips**

* **Do not use ngOnDestroy manually if you’re using DestroyRef with inject()**.
* **Avoid side effects in templates** – use effect() instead.
* **Chain DestroyRef with effect()** to keep components clean and reactive.

## **Summary**

| **Tool** | **Description** |
| --- | --- |
| effect() | Sets up reactive side effects for signal changes |
| DestroyRef | Registers teardown logic when component is destroyed |
| inject() | Retrieves Angular services like DestroyRef outside of constructor |

Signals combined with effect() and DestroyRef provide a **declarative, memory-safe, and lifecycle-aware** approach to reactive programming in Angular.

## 

## **Angular Directives Overview**

### **What is a Directive?**

A **directive** is a class that allows you to attach behavior to elements in the DOM. Angular provides both **built-in** and **custom** directives to enhance HTML with dynamic behavior.

## **Types of Directives in Angular**

| **Type** | **Description** | **Examples** |
| --- | --- | --- |
| Structural | Alters the **DOM structure** | \*ngIf, \*ngFor, \*ngSwitch |
| Attribute | Alters the **appearance or behavior** of an element | ngClass, ngStyle, ngModel |
| Custom | User-defined behavior as directive classes | appHighlight, appTooltip |

## **Structural Directives**

Structural directives modify the **layout by adding/removing elements**.

### **Common Structural Directives:**

#### **\*ngIf**

Conditionally includes a template block in the DOM.

| <p \*ngIf="isLoggedIn">Welcome!</p> |
| --- |

Can be used with else:

| <p \*ngIf="isLoggedIn; else guest">Welcome!</p>  <ng-template #guest><p>Please log in.</p></ng-template> |
| --- |

#### **\*ngFor**

Repeats a template block for each item in a list.

| <li \*ngFor="let item of items">{{ item }}</li> |
| --- |

Supports local variables:

| <li \*ngFor="let item of items; let i = index">#{{ i }} - {{ item }}</li> |
| --- |

#### **\*ngSwitch**

Displays elements based on a matching condition.

| <div [ngSwitch]="role">  <p \*ngSwitchCase="'admin'">Admin Panel</p>  <p \*ngSwitchCase="'user'">User Dashboard</p>  <p \*ngSwitchDefault>Access Denied</p>  </div> |
| --- |

## **Attribute Directives**

Attribute directives change the **appearance or behavior** of elements.

### **Common Attribute Directives:**

#### **ngClass**

Dynamically adds or removes CSS classes.

| <p [ngClass]="{ active: isActive }">Toggle Me</p> |
| --- |

#### **ngStyle**

Applies dynamic inline styles.

| <p [ngStyle]="{ color: isRed ? 'red' : 'black' }">Styled Text</p> |
| --- |

#### **ngModel**

Two-way data binding for form inputs.

| <input [(ngModel)]="username"> |
| --- |

Requires importing FormsModule.

## **Custom Directives**

You can build your own directive for reusable logic:

| @Directive({  selector: '[appHighlight]'  })  export class HighlightDirective {  constructor(el: ElementRef) {  el.nativeElement.style.backgroundColor = 'yellow';  }  } |
| --- |

Usage in template:

| <p appHighlight>This will be highlighted</p> |
| --- |

## **Summary Table**

| **Directive** | **Type** | **Purpose** |
| --- | --- | --- |
| \*ngIf | Structural | Show/hide DOM elements |
| \*ngFor | Structural | Loop over data |
| \*ngSwitch | Structural | Conditional branching |
| ngClass | Attribute | Add/remove CSS classes |
| ngStyle | Attribute | Add inline styles |
| ngModel | Attribute | Two-way data binding (forms) |
| appYourCustom | Custom | Reusable behavior attached to element |

## **Best Practices**

* Use **attribute directives** for visual/UI behavior changes.
* Use **structural directives** for DOM structure control.
* Prefer **custom directives** for reusable patterns (e.g., tooltips, validation).
* Don’t mix multiple structural directives on the same element.

## **Pipes and Custom Pipes**

### **What is a Pipe?**

* A **pipe** in Angular is a **function** that transforms **template output**.
* Used with the | (pipe) symbol in HTML templates.
* Pipes are **pure by default** – they only update when inputs change.

**Example:**

| <p>{{ name | uppercase }}</p> |
| --- |

### **Common Built-in Pipes**

| **Pipe** | **Description** |
| --- | --- |
| uppercase / lowercase | Changes text case |
| date | Formats date/time |
| currency | Formats a number as currency |
| percent | Formats a number as percentage |
| slice | Extracts a section of a string or array |
| json | Converts object to JSON string |

### **Advantages of Pipes**

* Cleaner and more readable templates
* Encapsulate display logic
* Reusable across components

## **Custom Pipes in Angular 19**

### **When to Use Custom Pipes?**

* When built-in pipes don’t cover your formatting or transformation needs.
* Example: masking phone numbers, formatting IDs, or filtering lists.

### **How to Create a Custom Pipe**

Generate with CLI:

| ng generate pipe customPipeName |
| --- |

1. Implement PipeTransform interface:

| import { Pipe, PipeTransform } from '@angular/core';  @Pipe({  name: 'reverse'  })  export class ReversePipe implements PipeTransform {  transform(value: string): string {  return value.split('').reverse().join('');  }  } |
| --- |

1. Use it in template:

| <p>{{ 'hello' | reverse }}</p> <!-- Output: olleh --> |
| --- |

### **Pure vs Impure Pipes**

| **Type** | **Description** |
| --- | --- |
| **Pure** (default) | Called only when input reference changes |
| **Impure** (pure: false) | Called on every change detection cycle – can impact performance |

### **Best Practices**

* Keep pipes **stateless** and **pure** for performance.
* Avoid using pipes for **complex logic** — use services or components instead.
* Use **impure pipes cautiously**; prefer them only for real-time data changes.

## 

## **Lab: Property Binding in Angular**

### **Objective:**

Learn how to use **property binding** ([property]="expression") to:

* Dynamically set element attributes (e.g. src, value, disabled)
* Control styles and classes based on component data

### **Step-by-Step Instructions**

### **Step 1: Generate a Standalone Component**

| ng generate component components/property-binding-demo --standalone |
| --- |

### **Step 2: Update Component Class**

Open property-binding-demo.component.ts:

| import { Component } from '@angular/core';  import { CommonModule } from '@angular/common';  @Component({  selector: 'app-property-binding-demo',  standalone: true,  imports: [CommonModule],  templateUrl: './property-binding-demo.component.html'  })  export class PropertyBindingDemoComponent {  userName = 'Alice';  profileImage = 'https://i.pravatar.cc/150?img=3';  isButtonDisabled = false;  inputPlaceholder = 'Enter your name...';  backgroundColor = 'lightblue';  boxClass = 'rounded';  } |
| --- |

### **Step 3: Create the Template with Property Bindings**

Open property-binding-demo.component.html and add:

| <h2>Property Binding Demo</h2>  <!-- Image Binding -->  <img [src]="profileImage" alt="User Photo" width="150" />  <!-- Input Binding -->  <p><strong>Name Input:</strong></p>  <input [placeholder]="inputPlaceholder" [value]="userName" />  <!-- Button Binding -->  <p><strong>Action Button:</strong></p>  <button [disabled]="isButtonDisabled">Submit</button>  <!-- Style and Class Binding -->  <p><strong>Styled Box:</strong></p>  <div [style.background-color]="backgroundColor" [class]="boxClass" style="padding: 1rem;">  This box has dynamic background and class.  </div> |
| --- |

### **Step 4: Bootstrap the Component**

In main.ts:

| import { bootstrapApplication } from '@angular/platform-browser';  import { PropertyBindingDemoComponent } from './app/components/property-binding-demo/property-binding-demo.component';  bootstrapApplication(PropertyBindingDemoComponent); |
| --- |

### **Step 5: Run the App**

| ng serve |
| --- |

Open your browser to<http://localhost:4200>

You should see:

* A dynamically loaded image
* An input field pre-filled with a name and placeholder
* A button that’s either enabled or disabled
* A styled box with color and rounded edges from class

### **Optional Enhancements**

* Toggle isButtonDisabled value with a button click
* Bind boxClass to 'rounded shadow' for multiple classes
* Add title binding to the image: [title]="userName"

### **Outcome:**

You have successfully:

* Applied **property binding** to elements
* Bound **image src**, **input value**, **button disabled**, **style**, and **class**
* Built a dynamic and reactive UI using Angular’s binding syntax

## **Lab Activity: Practice Event Binding in Angular**

### **Objective:**

Learn how to use Angular **event binding** to:

* Handle button clicks
* Capture input values
* Respond to keyboard events

### **Step-by-Step Instructions**

### **Step 1: Generate a Standalone Component**

| ng generate component components/event-binding-demo --standalone |
| --- |

### **Step 2: Update Component Class**

Open src/app/components/event-binding-demo/event-binding-demo.component.ts:

| import { Component } from '@angular/core';  import { CommonModule } from '@angular/common';  @Component({  selector: 'app-event-binding-demo',  standalone: true,  imports: [CommonModule],  templateUrl: './event-binding-demo.component.html'  })  export class EventBindingDemoComponent {  counter = 0;  name = '';  message = '';  incrementCounter() {  this.counter++;  }  updateName(event: Event) {  const input = event.target as HTMLInputElement;  this.name = input.value;  }  showKeyPress(event: KeyboardEvent) {  this.message = `You pressed: ${event.key}`;  }  reset() {  this.counter = 0;  this.name = '';  this.message = '';  }  } |
| --- |

### **Step 3: Create the HTML Template**

Open event-binding-demo.component.html and add the following:

| <h2>🖱️ Event Binding Demo</h2>  <!-- Button Click Event -->  <p><strong>Counter:</strong> {{ counter }}</p>  <button (click)="incrementCounter()"> Increment</button>  <hr />  <!-- Input Event -->  <p><strong>Enter your name:</strong></p>  <input (input)="updateName($event)" [value]="name" placeholder="Type your name" />  <p>Hello, {{ name }}!</p>  <hr />  <!-- Keyboard Event -->  <p><strong>Keyboard Activity:</strong></p>  <input (keyup)="showKeyPress($event)" placeholder="Press a key..." />  <p>{{ message }}</p>  <hr />  <!-- Reset Button -->  <button (click)="reset()">🔄 Reset</button> |
| --- |

### **Step 4: Bootstrap the Component**

In main.ts:

| import { bootstrapApplication } from '@angular/platform-browser';  import { EventBindingDemoComponent } from './app/components/event-binding-demo/event-binding-demo.component';  bootstrapApplication(EventBindingDemoComponent); |
| --- |

### **Step 5: Run the App**

| ng serve |
| --- |

Visit<http://localhost:4200>

### **What to Test:**

* Clicking the **"Increment"** button increases the counter
* Typing in the name input updates the greeting in real time
* Pressing any key in the keyup field shows which key was pressed
* Clicking **"Reset"** clears all values

### **Outcome:**

You’ve successfully:

* Handled **button click**, **input**, and **keyboard** events
* Used Angular’s **event binding syntax** (event)="method($event)"
* Built interactive features using **component methods**

## **Lab Activity: Create a Counter with Signals and Effects**

### **Objective:**

Build a standalone Angular component that uses:

* signal() for reactive state
* effect() to monitor state changes
* Buttons to update the state (increment, decrement, reset)

### **Step-by-Step Instructions**

#### **Step 1: Create a New Standalone Component**

From your project root directory:

| ng generate component counter-signal --standalone |
| --- |

#### **Step 2: Add Route to the New Component**

In src/app/app.routes.ts, import and register the new component:

| import { Routes } from '@angular/router';  import { CounterSignalComponent } from './counter-signal/counter-signal.component';  export const appRoutes: Routes = [  { path: '', component: CounterSignalComponent }  ]; |
| --- |

**Step 3: Implement Signal and Effect Logic**

Open src/app/counter-signal/counter-signal.component.ts and update it as follows:

| import { Component, signal, effect } from '@angular/core';  import { CommonModule } from '@angular/common';  @Component({  selector: 'app-counter-signal',  standalone: true,  imports: [CommonModule],  templateUrl: './counter-signal.component.html',  })  export class CounterSignalComponent {  // Reactive state  count = signal(0);  constructor() {  // Reactive effect that runs when the count changes  effect(() => {  console.log('Count changed to:', this.count());  });  }  increment() {  this.count.update(value => value + 1);  }  decrement() {  this.count.update(value => value - 1);  }  reset() {  this.count.set(0);  }  } |
| --- |

#### **Step 4: Update the Template with Buttons**

Open src/app/counter-signal/counter-signal.component.html and add the following:

| <h2> Signal Counter</h2>  <p>Current Count: {{ count() }}</p>  <button (click)="increment()">Increment</button>  <button (click)="decrement()">Decrement</button>  <button (click)="reset()">Reset</button> |
| --- |

#### **Step 5: Run the App**

Start the development server:

| ng serve |
| --- |

Visit:<http://localhost:4200>

Try clicking the buttons and observe:

* The count updates in the UI
* The console logs each new count value (from effect())

### **Outcome:**

You successfully:

* Created a **reactive counter** using Angular Signals
* Used effect() to observe changes and run side effects
* Applied reactive patterns in a clean, modern Angular component

## 

## **Build a Parent-Child Component with Data Flow**

### **Objective:**

Create two standalone components — a **parent** and a **child** — and enable data flow:

* **Parent to Child** using @Input()
* **Child to Parent** using @Output() and EventEmitter

### **Step-by-Step Instructions**

### **Step 1: Generate the Components**

In your Angular project directory:

| ng generate component parent --standalone  ng generate component child --standalone |
| --- |

### **Step 2: Define the Route to the Parent Component**

Update src/app/app.routes.ts:

| import { Routes } from '@angular/router';  import { ParentComponent } from './parent/parent.component';  export const appRoutes: Routes = [  { path: '', component: ParentComponent }  ]; |
| --- |

### **Step 3: Update the Child Component**

Edit src/app/child/child.component.ts:

| import { Component, EventEmitter, Input, Output } from '@angular/core';  import { CommonModule } from '@angular/common';  @Component({  selector: 'app-child',  standalone: true,  imports: [CommonModule],  templateUrl: './child.component.html',  })  export class ChildComponent {  @Input() messageFromParent!: string;  @Output() notifyParent = new EventEmitter<string>();  sendMessageToParent() {  this.notifyParent.emit('Hello from Child!');  }  } |
| --- |

Then update the child component template child.component.html:

| <h3>Child Component</h3>  <p>Received: {{ messageFromParent }}</p>  <button (click)="sendMessageToParent()">Send to Parent</button> |
| --- |

### **Step 4: Update the Parent Component**

Edit src/app/parent/parent.component.ts:

| import { Component } from '@angular/core';  import { CommonModule } from '@angular/common';  import { ChildComponent } from '../child/child.component';  @Component({  selector: 'app-parent',  standalone: true,  imports: [CommonModule, ChildComponent],  templateUrl: './parent.component.html',  })  export class ParentComponent {  parentMessage = 'Hello from Parent!';  childResponse = '';  handleChildEvent(event: string) {  this.childResponse = event;  }  } |
| --- |

Then edit the template parent.component.html:

| <h2> Parent Component</h2>  <app-child  [messageFromParent]="parentMessage"  (notifyParent)="handleChildEvent($event)"  ></app-child>  <p>Message from child: {{ childResponse }}</p> |
| --- |

### **Step 5: Run the App**

Start the server:

| ng serve |
| --- |

Go to<http://localhost:4200>

Expected behavior:

* The parent passes a message to the child.
* The child displays it and can send a message back via a button.
* The parent listens for the child’s event and displays the response.

**Outcome:**

You’ve successfully:

* Built a **parent-child relationship** using @Input() and @Output()
* Demonstrated two-way communication between components

## **Lab Activity: Create and Inject a Logging or Data Service**

### **Objective:**

Learn how to:

* Create a service using Angular CLI
* Inject it into a component using Angular’s DI system
* Use it to log messages or provide shared data

### **Step-by-Step Instructions**

### **Step 1: Generate the Logging Service**

Use the Angular CLI to create a new service:

| ng generate service logging |
| --- |

This creates a file: src/app/logging.service.ts

### **Step 2: Implement Logging Logic**

Edit src/app/logging.service.ts:

| import { Injectable } from '@angular/core';  @Injectable({  providedIn: 'root' // Makes it a singleton service available app-wide  })  export class LoggingService {  log(message: string) {  const timestamp = new Date().toISOString();  console.log(`[${timestamp}] ${message}`);  }  } |
| --- |

### **Step 3: Inject and Use the Service in a Component**

Let’s use it in a component like AppComponent or your own:

Edit src/app/app.component.ts:

### 

| import { Component } from '@angular/core';  import { LoggingService } from './logging.service';  @Component({  selector: 'app-root',  standalone: true,  imports: [],  template: `  <h1>Angular Logging Service Demo</h1>  <button (click)="logAction()">Click to Log</button>  `  })  export class AppComponent {  constructor(private logger: LoggingService) {}  logAction() {  this.logger.log('Button was clicked!');  }  } |
| --- |

### **Step 4: Run the App**

| ng serve |
| --- |

Navigate to:<http://localhost:4200>

* Click the **“Click to Log”** button
* Check the browser console — you should see the timestamped log message

### **Bonus: Try a Shared Data Service**

If you'd like to try a **data service** instead of logging:

| @Injectable({ providedIn: 'root' })  export class DataService {  private data: string = '';  setData(value: string) {  this.data = value;  }  getData() {  return this.data;  }  } |
| --- |

You can inject it into multiple components to **share state**.

**Outcome:**

You have successfully:

* Created a reusable Angular **service**
* Injected it using Angular’s **dependency injection system**
* Used it inside a component to perform logic

## 

## **Lab Activity: Use @for, @if, and @switch in Angular Templates**

### **Objective:**

Use Angular 17+ control flow syntax to:

* Loop through a list using @for
* Conditionally show content using @if and @else
* Display status-based content using @switch

### **Step-by-Step Instructions**

### **Step 1: Generate a New Standalone Component**

| ng generate component status-list --standalone |
| --- |

### **Step 2: Register the Component in Routes**

In src/app/app.routes.ts, add:

| import { Routes } from '@angular/router';  import { StatusListComponent } from './status-list/status-list.component';  export const appRoutes: Routes = [  { path: '', component: StatusListComponent }  ]; |
| --- |

### **Step 3: Define Data and State**

Open src/app/status-list/status-list.component.ts and update:

| import { Component } from '@angular/core';  import { CommonModule } from '@angular/common';  @Component({  selector: 'app-status-list',  standalone: true,  imports: [CommonModule],  templateUrl: './status-list.component.html',  })  export class StatusListComponent {  users = [  { name: 'Alice', status: 'active' },  { name: 'Bob', status: 'inactive' },  { name: 'Charlie', status: 'pending' }  ];  showList = true;  } |
| --- |

### **Step 4: Use @for, @if, and @switch in the Template**

Edit src/app/status-list/status-list.component.html:

### 

| <h2> User Status List</h2>  <!-- Toggle Button -->  <button (click)="showList = !showList">  {{ showList ? 'Hide' : 'Show' }} List  </button>  <!-- @if directive -->  @if (showList) {  <ul>  <!-- @for directive -->  @for (user of users; let i = $index) {  <li>  {{ i + 1 }}. {{ user.name }} —  <!-- @switch directive -->  @switch (user.status) {  @case ('active') {  <strong style="color: green">Active</strong>  }  @case ('inactive') {  <strong style="color: gray">Inactive</strong>  }  @case ('pending') {  <strong style="color: orange">Pending Approval</strong>  }  @default {  <strong style="color: red">Unknown</strong>  }  }  </li>  }  </ul>  } @else {  <p>The list is hidden.</p>  } |
| --- |

### **Step 5: Run the App**

| ng serve |
| --- |

Visit:<http://localhost:4200>

* The list of users should be shown using @for.
* You can hide/show the list using the button (@if, @else).
* Each user's status is styled using @switch, @case, and @default.

### **Outcome:**

You have successfully used:

* @for to iterate through data
* @if and @else to conditionally render blocks
* @switch and @case to selectively render based on values

## **Lab: Using and Creating Pipes in Angular 19**

### **Objective:**

* Understand how to use built-in pipes in Angular templates.
* Create and apply a custom pipe for data transformation.

### **Part 1: Using Built-in Pipes**

#### **1. Create a New Component (if needed)**

| ng generate component pipe-demo |
| --- |

#### **2. Update the Template (pipe-demo.component.html)**

| <h2>Angular Pipes Demo</h2>  <p><strong>Original:</strong> {{ name }}</p>  <p><strong>Uppercase:</strong> {{ name | uppercase }}</p>  <p><strong>Lowercase:</strong> {{ name | lowercase }}</p>  <p><strong>Date Format:</strong> {{ today | date: 'fullDate' }}</p>  <p><strong>Currency:</strong> {{ price | currency:'MYR':'symbol':'1.2-2' }}</p> |
| --- |

#### **3. Update the Class (pipe-demo.component.ts)**

| export class PipeDemoComponent {  name = 'Angular 19 Pipes';  today = new Date();  price = 1234.56;  } |
| --- |

**Check output in browser.** Observe how data is formatted with pipes.

### **Part 2: Create a Custom Pipe**

#### **1. Generate a Pipe**

| ng generate pipe reverse |
| --- |

#### **2. Implement the Logic (reverse.pipe.ts)**

| import { Pipe, PipeTransform } from '@angular/core';  @Pipe({  name: 'reverse'  })  export class ReversePipe implements PipeTransform {  transform(value: string): string {  return value.split('').reverse().join('');  }  } |
| --- |

#### **3. Use the Pipe in Template**

Add to your pipe-demo.component.html:

| <p><strong>Reversed:</strong> {{ name | reverse }}</p> |
| --- |

**Expected Output:** sepip 91 ralugnA if name is "Angular 19 pipes"

### **Challenge Task (Optional)**

* Create a custom pipe called maskPhone that turns 0123456789 into 012-\*\*\*-6789.
* Try using a pipe with @for to transform list values.

### **Recap**

In this lab, you:

* Applied multiple **built-in pipes**.
* Created a **custom pipe** using PipeTransform.
* Used custom logic to transform data in the view.

# **Angular Router: Modern Standalone Approach**

Angular’s Router is the backbone of navigation in single-page applications (SPAs). Starting with Angular 14 and supercharged in Angular 16+, the framework now supports **standalone routing APIs** — removing the need for NgModule-based routing setup.

## **1. provideRouter() – Setting Up Standalone Routing**

**What is provideRouter()?** It’s the modern way to register routes in standalone Angular applications — replacing the old RouterModule.forRoot() setup.

### **Example: Basic Setup**

**main.ts**

| import { bootstrapApplication } from '@angular/platform-browser';  import { provideRouter } from '@angular/router';  import { AppComponent } from './app/app.component';  import { appRoutes } from './app/app.routes';  bootstrapApplication(AppComponent, {  providers: [provideRouter(appRoutes)],  }); |
| --- |

**app.routes.ts**

| import { Routes } from '@angular/router';  import { HomeComponent } from './home/home.component';  export const appRoutes: Routes = [  { path: '', component: HomeComponent },  ]; |
| --- |

**Note:** Routes are defined as an array of objects — no module boilerplate required.

## **2. Defining Standalone Routes**

Each route can link to a **component**, **lazy-loaded module**, or **redirect**.

### **Component Route**

| { path: 'about', component: AboutComponent } |
| --- |

### **Redirect Route**

| { path: '', redirectTo: 'home', pathMatch: 'full' } |
| --- |

### **Wildcard Route (404)**

| { path: '\*\*', component: NotFoundComponent } |
| --- |

### **Route Parameters (Dynamic Segments)**

You can define dynamic segments using :paramName.

| { path: 'user/:id', component: UserProfileComponent } |
| --- |

To access the route parameter (id in this case) inside UserProfileComponent, inject ActivatedRoute:

| constructor(private route: ActivatedRoute) {}  ngOnInit() {  const userId = this.route.snapshot.paramMap.get('id');  } |
| --- |

**Optional Parameters** Optional params can be handled via query params:

| { path: 'search', component: SearchComponent } |
| --- |

URL: /search?term=angular

Access via:

| this.route.queryParamMap.subscribe(params => {  const term = params.get('term');  }); |
| --- |

## **3. Lazy Loading Standalone Components**

**Goal:** Load parts of your app only when needed, reducing initial bundle size.

### **Lazy Load a Standalone Component**

| {  path: 'dashboard',  loadComponent: () => import('./dashboard/dashboard.component').then(m => m.DashboardComponent)  } |
| --- |

**Notes:**

* No need for loadChildren or NgModule.
* Works directly with provideRouter().

## **4. Lazy Loading with Feature Routes**

For feature modules or nested routes:

| {  path: 'admin',  loadChildren: () => import('./admin/admin.routes').then(m => m.adminRoutes)  } |
| --- |

**admin.routes.ts**

| import { Routes } from '@angular/router';  import { AdminHomeComponent } from './admin-home.component';  export const adminRoutes: Routes = [  { path: '', component: AdminHomeComponent },  ]; |
| --- |

## **5. Router Directives (in Templates)**

| <nav>  <a routerLink="/">Home</a>  <a routerLink="/about">About</a>  <a [routerLink]="['/user', userId]">My Profile</a>  </nav>  <router-outlet></router-outlet> |
| --- |

Works exactly the same in standalone components!

## **Advantages of the Standalone Routing Approach**

| **Feature** | **Benefit** |
| --- | --- |
| provideRouter() | No NgModules needed |
| Lazy loading standalone components | Faster load, smaller bundles |
| Clean route definitions | More modular and readable code |
| No boilerplate | Simpler, cleaner architecture |
| Full support for route parameters | Easily pass and retrieve dynamic values |

## **Summary**

| **Concept** | **Description** |
| --- | --- |
| provideRouter() | Registers app routes without NgModules |
| appRoutes | Central array of route definitions |
| loadComponent() | Enables lazy-loading of standalone components |
| routerLink, router-outlet | Directives for navigation and view rendering |
| Route Parameters | Use :id, :slug, etc. for dynamic routing |
| Standalone-ready | Full routing power with no module dependencies |

Angular’s standalone router setup with provideRouter() simplifies routing while giving you full flexibility — including support for route parameters, lazy loading, and dynamic segments — all with less boilerplate.

## 

## **Angular Route Guards Overview**

**Route Guards** are interfaces that let you control navigation in your Angular app. They determine whether a route can be:

* **Entered** (canActivate)
* **Left** (canDeactivate)
* (Others include canLoad, canActivateChild, resolve, etc.)

Guards are typically implemented as **services** and applied to routes.

## **canActivate – Control Access Before Entering a Route**

### **Purpose:**

Used to check **if a user is allowed to access** a route. Often used for authentication, authorization, or preconditions.

### **Implementation:**

#### **1. Create a Guard Service:**

| ng generate guard auth |
| --- |

#### **2. Define the Guard Logic:**

| import { Injectable } from '@angular/core';  import { CanActivate, Router } from '@angular/router';  @Injectable({ providedIn: 'root' })  export class AuthGuard implements CanActivate {  constructor(private router: Router) {}  canActivate(): boolean {  const isLoggedIn = localStorage.getItem('token') !== null;  if (!isLoggedIn) {  this.router.navigate(['/login']);  return false;  }  return true;  }  } |
| --- |

#### **3. Apply to a Route:**

| {  path: 'dashboard',  canActivate: [AuthGuard],  loadComponent: () => import('./dashboard.component').then(m => m.DashboardComponent)  } |
| --- |

Navigation is **blocked** if canActivate returns false.

## **canDeactivate – Control Exit from a Route**

### **Purpose:**

Used to check **if it’s safe to leave a component**, typically to prevent unsaved changes from being lost.

### **Step-by-Step:**

#### **1. Create a Guard Service:**

| ng generate guard unsaved |
| --- |

#### **2. Create an Interface in Your Component:**

| export interface CanComponentDeactivate {  canDeactivate: () => boolean | Observable<boolean>;  } |
| --- |

#### **3. Implement in Your Component:**

| export class EditProfileComponent implements CanComponentDeactivate {  hasUnsavedChanges = true;  canDeactivate(): boolean {  return !this.hasUnsavedChanges || confirm('Discard changes?');  }  } |
| --- |

#### **4. Define the Guard Logic:**

| import { Injectable } from '@angular/core';  import { CanDeactivate } from '@angular/router';  import { CanComponentDeactivate } from './edit-profile.component';  @Injectable({ providedIn: 'root' })  export class UnsavedChangesGuard implements CanDeactivate<CanComponentDeactivate> {  canDeactivate(component: CanComponentDeactivate): boolean {  return component.canDeactivate();  }  } |
| --- |

#### **5. Apply to Route:**

| {  path: 'edit',  canDeactivate: [UnsavedChangesGuard],  loadComponent: () => import('./edit-profile.component').then(m => m.EditProfileComponent)  } |
| --- |

If the user has unsaved changes, they’ll be prompted before leaving the page.

**Comparison Table**

| **Guard Type** | **Purpose** | **Example Use Case** |
| --- | --- | --- |
| canActivate | Block/allow route entry | Auth check before login |
| canDeactivate | Confirm route exit | Unsaved form detection |

## **Best Practices**

* Keep guard logic **simple and reusable** (e.g., move logic to auth service).

Combine multiple guards using arrays:

| canActivate: [AuthGuard, PermissionGuard] |
| --- |

* Return Observable<boolean> for async checks (e.g., API verification).
* Always test guards with **unit tests** for route scenarios.

## **Summary**

| **Feature** | **Description** |
| --- | --- |
| canActivate | Controls access before loading a route |
| canDeactivate | Confirms whether navigation away is allowed |
| Guards as services | Use @Injectable({ providedIn: 'root' }) |
| Usage | Applied in route configuration |

Route guards are a powerful part of Angular's router, letting you **secure routes**, **protect data**, and **guide user behavior** at critical navigation points.

## 

## **Angular Forms Overview**

Angular offers two main strategies for building and managing forms:

1. **Template-driven Forms**
2. **Reactive Forms**

Both are part of Angular’s @angular/forms package but serve different use cases and developer preferences.

## **Template-driven Forms**

### **Key Features:**

* Declarative, HTML-first approach
* Simpler and easier to use for small to medium forms
* Uses directives like ngModel, required, #ref

### **Example:**

| <form #userForm="ngForm" (ngSubmit)="onSubmit(userForm)">  <input name="username" [(ngModel)]="username" required />  <button type="submit">Submit</button>  </form> |
| --- |

### **Characteristics:**

* Defined mostly in the template
* Uses FormsModule
* Automatic tracking of form state and validation
* Ideal for simpler use cases

1. **Reactive Forms**

### **Key Features:**

* Programmatic, code-driven approach
* More control, flexibility, and scalability
* Strongly typed and testable

### **Example:**

| form = new FormGroup({  username: new FormControl('', [Validators.required])  }); |
| --- |

| <form [formGroup]="form" (ngSubmit)="onSubmit()">  <input formControlName="username" />  <button type="submit">Submit</button>  </form> |
| --- |

### **Characteristics:**

* Defined in the component class
* Uses ReactiveFormsModule
* Explicit form state and validation logic
* Suitable for dynamic, complex, or large forms

**Comparison Table**

| **Feature** | **Template-driven Forms** | **Reactive Forms** |
| --- | --- | --- |
| Approach | Declarative (HTML-driven) | Imperative (code-driven) |
| Setup | Easy, less code | More boilerplate, but flexible |
| Module Required | FormsModule | ReactiveFormsModule |
| Validation | Template-based | Programmatic (validators in code) |
| Control Creation | Automatic via ngModel | Manual via FormControl / FormGroup |
| Dynamic Form Controls | Harder | Easier and well-supported |
| Testability | Moderate | Excellent |
| Preferred Use Case | Simple forms, quick setup | Complex, dynamic, or scalable forms |

## **When to Use**

| **Scenario** | **Recommended Form Type** |
| --- | --- |
| Simple form with few fields | Template-driven |
| Large form with dynamic fields | Reactive |
| Requires fine-grained validation logic | Reactive |
| Minimal setup and HTML-first preference | Template-driven |
| Unit testing and explicit state management | Reactive |

**Coexistence**

Angular allows **mixing** both types in the same app, but it’s recommended to stick to **one pattern per component** to avoid confusion.

## **Summary**

| **Form Type** | **Summary Description** |
| --- | --- |
| Template-driven | Simple, declarative forms using ngModel and HTML |
| Reactive | Flexible, powerful forms managed via component code |

## **Angular Form Validation Overview**

Angular provides robust tools to validate user input in forms. Validation can be applied in two ways:

* **Synchronous Validation**: Runs instantly on input change (e.g., required, minlength)
* **Asynchronous Validation**: Runs with a delay, usually after HTTP requests (e.g., checking if username exists)

**1. Synchronous Validators**

These validators return a result **immediately**. Built-in validators include:

### **Common Sync Validators:**

| **Validator** | **Description** |
| --- | --- |
| Validators.required | Field must not be empty |
| Validators.minLength(x) | Minimum number of characters |
| Validators.maxLength(x) | Maximum number of characters |
| Validators.pattern(x) | Must match a given RegExp |
| Validators.email | Valid email format |

### **Example (Reactive Form):**

| this.form = new FormGroup({  username: new FormControl('', [Validators.required, Validators.minLength(4)])  }); |
| --- |

### **Example (Template-driven):**

| <input name="username" [(ngModel)]="username" required minlength="4" /> |
| --- |

## **2. Asynchronous Validators**

These return **a Promise or Observable**, and are used for checks like:

* Is email already registered?
* Is username available?

### **Example Async Validator:**

| function usernameExistsValidator(userService: UserService): AsyncValidatorFn {  return (control: AbstractControl): Observable<ValidationErrors | null> => {  return userService.checkUsername(control.value).pipe(  map(isTaken => (isTaken ? { usernameTaken: true } : null))  );  };  } |
| --- |

### **Usage in Reactive Form:**

| this.form = new FormGroup({  username: new FormControl(  '',  [Validators.required],  [usernameExistsValidator(this.userService)]  )  }); |
| --- |

Async validators are passed as the **third parameter** of FormControl.

## **Validity State Properties**

| **Property** | **Description** |
| --- | --- |
| valid | All validators pass |
| invalid | At least one validator fails |
| pending | Async validation in progress |
| pristine | Field not modified yet |
| dirty | Field has been modified |
| touched | Field was focused and blurred |
| untouched | Field was never touched |

## **Showing Validation Errors (Reactive Form)**

| <input formControlName="email" />  <div \*ngIf="form.get('email')?.errors?.['required']">  Email is required  </div> |
| --- |

## **Showing Validation Errors (Template-driven Form)**

| <input name="email" [(ngModel)]="email" #emailRef="ngModel" required />  <div \*ngIf="emailRef.invalid && emailRef.touched">  Email is required  </div> |
| --- |

## **Best Practices**

* Group multiple error checks using custom functions.
* Use statusChanges observable to monitor validation status.
* Apply async validators sparingly to avoid performance issues.
* Always provide user-friendly messages for validation errors.

## **Summary**

| **Type** | **Description** | **Returns** |
| --- | --- | --- |
| Synchronous | Runs instantly with user input | null or error |
| Asynchronous | Checks with server or delay | Observable/Promise |
| Built-in Validators | Required, MinLength, Email, Pattern | Angular core |
| Custom Validators | Business logic-specific (e.g. age > 18) | User-defined |

## 

## **Signals in Angular Forms**

### **What’s New?**

With Angular's **Signal-based reactivity model**, forms can now be managed in a **fine-grained, reactive, and declarative** way — as an alternative (or complement) to traditional template-driven or reactive forms.

Introduced in Angular 17+ and evolving further in Angular 18+, **Signals in Forms** allow developers to:

* Use signal() to represent form state
* Apply computed() for derived/validated values
* Trigger side effects with effect()
* Avoid template-based tracking and manual subscriptions

## **Key Concepts**

| **Signal Tool** | **Role in Forms** |
| --- | --- |
| signal() | Holds current form field values |
| computed() | Tracks derived or validated form state |
| effect() | Runs on change: validation, logging, etc. |
| set() / update() | Updates values programmatically |

## **Basic Example: Signal-based Form Fields**

| import { signal, computed, effect } from '@angular/core';  export class ProfileComponent {  firstName = signal('');  lastName = signal('');  fullName = computed(() => `${this.firstName()} ${this.lastName()}`);  constructor() {  effect(() => {  console.log('Full Name:', this.fullName());  });  }  submit() {  console.log('Submitting:', {  firstName: this.firstName(),  lastName: this.lastName()  });  }  } |
| --- |

### **Template:**

| <input [value]="firstName()" (input)="firstName.set($event.target.value)" />  <input [value]="lastName()" (input)="lastName.set($event.target.value)" />  <p>Full Name: {{ fullName() }}</p>  <button (click)="submit()">Submit</button> |
| --- |

## **Validation with Signals**

You can create a validation signal using computed():

| isFirstNameValid = computed(() => this.firstName().length >= 2);  isFormValid = computed(() => this.isFirstNameValid() && this.lastName().length > 0); |
| --- |

### **Template Example:**

| <p \*ngIf="!isFirstNameValid()">First name must be at least 2 characters</p>  <button [disabled]="!isFormValid()">Submit</button> |
| --- |

## **Benefits of Using Signals in Forms**

| **Feature** | **Benefit** |
| --- | --- |
| Fine-grained reactivity | Only recomputes what's needed, better performance |
| Declarative validation | Cleaner, reusable computed expressions |
| No FormControl boilerplate | Avoids FormGroup, FormBuilder, etc. |
| Native integration | Easy to use alongside @for, @if, and effects |

## **Experimental Forms API (Future Direction)**

Angular team is exploring **Signal-based Form APIs**, where field-level reactivity and validation are powered by signals without needing full FormControl trees.

| const nameField = formSignal('', {  validators: [requiredValidator, minLengthValidator(3)],  }); |
| --- |

This would support:

* Signal-based form groups
* Derived validation states
* Form event effects

Still experimental — but promising for the future of form management in Angular.

## **Summary**

| **Concept** | **Usage Example** |
| --- | --- |
| signal() | firstName = signal('') |
| computed() | isValid = computed(() => value().length > 2) |
| effect() | effect(() => console.log(value())) |
| Form benefits | Lightweight, reactive, testable, readable |

## **When to Use Signals in Forms?**

Ideal for:

* Simple to moderately complex forms
* Apps using standalone components
* Developers favoring composability and reactivity

Avoid if:

* You need full Angular form features like FormArray, nested groups
* You're integrating with existing reactive or template-driven forms

## 

## **Lab Activity: Set Up Routing with provideRouter()**

### **Objective:**

Learn how to configure and use Angular's **standalone router** with provideRouter() for navigation between components in a standalone app.

### **Step-by-Step Instructions**

### **Step 1: Generate a Standalone Angular Project**

| ng new routing-demo --standalone  cd routing-demo |
| --- |

Choose:

* Add Angular routing: **Yes**
* Styles: Your preferred option (e.g., CSS)

Angular CLI will scaffold a standalone-ready app with routing support.

### **Step 2: Create Home and About Components**

Generate two standalone components:

| ng generate component home --standalone  ng generate component about --standalone  ng generate component user --standalone |
| --- |

user will be used to demonstrate **route parameters** (e.g., /user/123).

### **Step 3: Define Routes in app.routes.ts**

Locate and update src/app/app.routes.ts with the following content:

| import { Routes } from '@angular/router';  import { HomeComponent } from './home/home.component';  import { AboutComponent } from './about/about.component';  import { UserComponent } from './user/user.component';  export const appRoutes: Routes = [  { path: '', component: HomeComponent },  { path: 'about', component: AboutComponent },  { path: 'user/:id', component: UserComponent }, // Route with parameter  { path: '\*\*', redirectTo: '' } // Wildcard fallback route  ]; |
| --- |

### **Step 4: Register Routes with provideRouter() in main.ts**

Open src/main.ts and make sure it contains:

| import { bootstrapApplication } from '@angular/platform-browser';  import { provideRouter } from '@angular/router';  import { AppComponent } from './app/app.component';  import { appRoutes } from './app/app.routes';  bootstrapApplication(AppComponent, {  providers: [provideRouter(appRoutes)]  }); |
| --- |

### **Step 5: Add <router-outlet> to AppComponent**

Update src/app/app.component.html to include routing UI:

| <h1>Angular Standalone Routing Demo</h1>  <nav>  <a routerLink="/">Home</a> |  <a routerLink="/about">About</a>  <a [routerLink]="['/user', 101]">User 101</a>  </nav>  <hr />  <router-outlet></router-outlet> |
| --- |

routerLink is used for navigation, and router-outlet displays the route content.

### **Step 6: Use Route Parameters in UserComponent**

Edit src/app/user/user.component.ts to read the id from the URL:

| import { Component, OnInit } from '@angular/core';  import { ActivatedRoute } from '@angular/router';  @Component({  selector: 'app-user',  standalone: true,  template: `  <h2>User Detail</h2>  <p>User ID: {{ userId }}</p>  `  })  export class UserComponent implements OnInit {  userId: string | null = null;  constructor(private route: ActivatedRoute) {}  ngOnInit() {  this.userId = this.route.snapshot.paramMap.get('id');  }  } |
| --- |

### **Step 7: Serve the App**

| ng serve |
| --- |

Open your browser and navigate to<http://localhost:4200>

* Verify that clicking **Home** and **About** updates the view.
* Try navigating to an unknown route like /abc and see if it redirects.

### **Outcome:**

You have successfully:

* Defined routes using appRoutes: Routes
* Registered the router using provideRouter() in main.ts
* Built a navigation menu using routerLink
* Loaded components dynamically via <router-outlet>

## 

## **Lab Activity: Build a Form for User Input and Validate It**

### **Objective:**

Create a user registration form using **Reactive Forms**, validate the input fields, and display error messages.

### **Step-by-Step Instructions**

### **Step 1: Create a Standalone Component for the Form**

| ng generate component user-form --standalone |
| --- |

### **Step 2: Add Routing to the Form Page**

In src/app/app.routes.ts, add:

| import { Routes } from '@angular/router';  import { UserFormComponent } from './user-form/user-form.component';  export const appRoutes: Routes = [  { path: '', component: UserFormComponent }  ]; |
| --- |

### **Step 3: Import ReactiveFormsModule**

Open user-form.component.ts and update:

| import { Component } from '@angular/core';  import { FormBuilder, FormGroup, Validators, ReactiveFormsModule } from '@angular/forms';  import { CommonModule } from '@angular/common';  @Component({  selector: 'app-user-form',  standalone: true,  imports: [CommonModule, ReactiveFormsModule],  templateUrl: './user-form.component.html'  })  export class UserFormComponent {  form: FormGroup;  constructor(private fb: FormBuilder) {  this.form = this.fb.group({  name: ['', [Validators.required, Validators.minLength(3)]],  email: ['', [Validators.required, Validators.email]],  age: ['', [Validators.required, Validators.min(18)]]  });  }  onSubmit() {  if (this.form.valid) {  console.log('Form submitted:', this.form.value);  } else {  this.form.markAllAsTouched(); // Show validation errors  }  }  } |
| --- |

### **Step 4: Create the Form Template**

In user-form.component.html, add:

| <h2> User Registration Form</h2>  <form [formGroup]="form" (ngSubmit)="onSubmit()">  <!-- Name Field -->  <label>Name:</label>  <input type="text" formControlName="name" />  <div \*ngIf="form.get('name')?.touched && form.get('name')?.invalid">  <small \*ngIf="form.get('name')?.errors?.['required']">Name is required.</small>  <small \*ngIf="form.get('name')?.errors?.['minlength']">Name must be at least 3 characters.</small>  </div>  <!-- Email Field -->  <label>Email:</label>  <input type="email" formControlName="email" />  <div \*ngIf="form.get('email')?.touched && form.get('email')?.invalid">  <small \*ngIf="form.get('email')?.errors?.['required']">Email is required.</small>  <small \*ngIf="form.get('email')?.errors?.['email']">Enter a valid email.</small>  </div>  <!-- Age Field -->  <label>Age:</label>  <input type="number" formControlName="age" />  <div \*ngIf="form.get('age')?.touched && form.get('age')?.invalid">  <small \*ngIf="form.get('age')?.errors?.['required']">Age is required.</small>  <small \*ngIf="form.get('age')?.errors?.['min']">Must be at least 18 years old.</small>  </div>  <br />  <button type="submit">Submit</button>  </form> |
| --- |

### **Step 5: Run the App**

| ng serve |
| --- |

Go to<http://localhost:4200>

* Try submitting the form without entering values
* Fill in each field and watch the validation messages disappear
* Check the console for form data upon successful submission

### **Outcome:**

You have successfully:

* Built a standalone component with a form
* Used **Reactive Forms** and **Validators**
* Provided real-time validation feedback in the UI
* Logged the form data upon successful submission

## 

## **Lab Activity: Use Signals to Handle Form State**

### **Objective:**

Learn how to use **Angular Signals** (signal, computed, effect) to manage and validate form state without using FormGroup or FormControl.

### **Step-by-Step Instructions**

### **Step 1: Generate a Standalone Component**

| ng generate component signal-form --standalone |
| --- |

### **Step 2: Add the Route**

In src/app/app.routes.ts:

| import { Routes } from '@angular/router';  import { SignalFormComponent } from './signal-form/signal-form.component';  export const appRoutes: Routes = [  { path: '', component: SignalFormComponent }  ]; |
| --- |

### **Step 3: Create Form State with Signals**

In src/app/signal-form/signal-form.component.ts, set up the form state:

| import { Component, signal, computed, effect } from '@angular/core';  import { CommonModule } from '@angular/common';  @Component({  selector: 'app-signal-form',  standalone: true,  imports: [CommonModule],  templateUrl: './signal-form.component.html'  })  export class SignalFormComponent {  // Form signals  name = signal('');  email = signal('');  age = signal('');  // Validation logic using computed  isNameValid = computed(() => this.name().trim().length >= 3);  isEmailValid = computed(() => /^[\w.-]+@[a-zA-Z\d.-]+\.[a-zA-Z]{2,}$/.test(this.email()));  isAgeValid = computed(() => Number(this.age()) >= 18);  // Entire form validity  isFormValid = computed(() =>  this.isNameValid() && this.isEmailValid() && this.isAgeValid()  );  constructor() {  effect(() => {  console.log('Form state changed:', {  name: this.name(),  email: this.email(),  age: this.age()  });  });  }  submit() {  if (this.isFormValid()) {  alert('Form submitted successfully!');  console.log('Submitted data:', {  name: this.name(),  email: this.email(),  age: this.age()  });  } else {  alert('Form has errors.');  }  }  } |
| --- |

### **Step 4: Create the Form Template**

In src/app/signal-form/signal-form.component.html:

| <h2> Signal-Based User Form</h2>  <form (ngSubmit)="submit()" novalidate>  <!-- Name Field -->  <label>Name:</label>  <input  type="text"  [value]="name()"  (input)="name.set($event.target.value)"  />  <div \*ngIf="!isNameValid()">Name must be at least 3 characters.</div>  <!-- Email Field -->  <label>Email:</label>  <input  type="email"  [value]="email()"  (input)="email.set($event.target.value)"  />  <div \*ngIf="!isEmailValid()">Please enter a valid email.</div>  <!-- Age Field -->  <label>Age:</label>  <input  type="number"  [value]="age()"  (input)="age.set($event.target.value)"  />  <div \*ngIf="!isAgeValid()">Age must be 18 or older.</div>  <br />  <button type="submit" [disabled]="!isFormValid()">Submit</button>  </form> |
| --- |

### **Step 5: Run the App**

| ng serve |
| --- |

Go to:<http://localhost:4200>

Test:

* Input validation in real-time
* Submit button enabling/disabling
* Console logging and alert on submit

### **Outcome:**

You have successfully:

* Managed form state using **Angular Signals**
* Validated fields using computed()
* Reacted to changes using effect()
* Submitted form data conditionally based on form validity

## 

## **Lab Activity: Add Route Guards to a Form Wizard**

### **Objective:**

Add route protection to a multi-step form (wizard) using:

* canActivate to restrict access if prerequisites are not met
* canDeactivate to warn users before leaving incomplete steps

### **Sample Wizard Route Structure**

Assume your wizard has 3 steps:

| const appRoutes: Routes = [  { path: '', redirectTo: 'step1', pathMatch: 'full' },  { path: 'step1', component: Step1Component },  { path: 'step2', component: Step2Component, canActivate: [Step1Guard] },  { path: 'step3', component: Step3Component, canDeactivate: [UnsavedChangesGuard] },  ]; |
| --- |

### **Step-by-Step Instructions**

### **Step 1: Generate Guards**

| ng generate guard step1  ng generate guard unsaved-changes |
| --- |

### **Step 2: Create a Shared Form State Service**

| In src/app/services/form-state.service.ts:  import { Injectable, signal } from '@angular/core';  @Injectable({ providedIn: 'root' })  export class FormStateService {  completedSteps = signal<number>(1); // Tracks completed steps  hasUnsavedChanges = signal<boolean>(false); // For canDeactivate  markStepComplete(step: number) {  if (step >= this.completedSteps()) {  this.completedSteps.set(step);  }  }  setUnsaved(value: boolean) {  this.hasUnsavedChanges.set(value);  }  } |
| --- |

### **Step 3: Implement canActivate Guard for Step 2**

In src/app/step1.guard.ts:

| import { Injectable } from '@angular/core';  import { CanActivate, Router } from '@angular/router';  import { FormStateService } from './services/form-state.service';  @Injectable({ providedIn: 'root' })  export class Step1Guard implements CanActivate {  constructor(private state: FormStateService, private router: Router) {}  canActivate(): boolean {  if (this.state.completedSteps() >= 1) {  return true;  }  alert('You must complete Step 1 first!');  this.router.navigate(['/step1']);  return false;  }  } |
| --- |

### **Step 4: Implement canDeactivate Guard for Step 3**

In src/app/unsaved-changes.guard.ts:

| import { Injectable } from '@angular/core';  import { CanDeactivate } from '@angular/router';  import { FormStateService } from './services/form-state.service';  @Injectable({ providedIn: 'root' })  export class UnsavedChangesGuard implements CanDeactivate<unknown> {  constructor(private state: FormStateService) {}  canDeactivate(): boolean {  if (this.state.hasUnsavedChanges()) {  return confirm('You have unsaved changes. Are you sure you want to leave?');  }  return true;  }  } |
| --- |

### **Step 5: Use the Service in Your Step Components**

#### **In Step1Component:**

| constructor(private state: FormStateService) {}  completeStep() {  this.state.markStepComplete(1);  } |
| --- |

#### **In Step3Component:**

| constructor(private state: FormStateService) {}  onInputChange() {  this.state.setUnsaved(true);  }  submitForm() {  this.state.setUnsaved(false);  // Proceed to next action  } |
| --- |

### **Step 6: Register Routes with Guards**

Update app.routes.ts:

| import { Step1Guard } from './step1.guard';  import { UnsavedChangesGuard } from './unsaved-changes.guard';  export const appRoutes: Routes = [  { path: '', redirectTo: 'step1', pathMatch: 'full' },  { path: 'step1', component: Step1Component },  { path: 'step2', component: Step2Component, canActivate: [Step1Guard] },  { path: 'step3', component: Step3Component, canDeactivate: [UnsavedChangesGuard] }  ]; |
| --- |

### **Step 7: Test the Wizard**

Run the app:

| ng serve |
| --- |

* Try going to /step2 before completing /step1 — it should block access.
* Make changes in /step3 and try navigating away — it should show a confirmation prompt.

### **Outcome:**

You have successfully:

* Created a multi-step form (wizard)
* Used canActivate to restrict access to future steps
* Used canDeactivate to warn users about unsaved data
* Used Signals to manage shared form state across components

## 

## **Angular HttpClient & provideHttpClient**

### **Purpose:**

Angular’s HttpClient is used to perform HTTP requests like **GET**, **POST**, **PUT**, **DELETE**, etc., for connecting your app to APIs and servers.

Starting from Angular 15+, and improved in Angular 16+, you can now provide HttpClient using **provideHttpClient()** instead of importing HttpClientModule—a more modular and tree-shakable approach designed for **standalone apps**.

## **Traditional Setup (Before Angular 15)**

You had to import HttpClientModule in the root module:

| @NgModule({  imports: [HttpClientModule]  })  export class AppModule {} |
| --- |

## **Modern Setup with provideHttpClient() (Standalone API)**

### **main.ts (Standalone App Bootstrapping)**

| import { bootstrapApplication } from '@angular/platform-browser';  import { provideHttpClient } from '@angular/common/http';  import { AppComponent } from './app/app.component';  bootstrapApplication(AppComponent, {  providers: [provideHttpClient()]  }); |
| --- |

This eliminates the need for HttpClientModule, making the app more lightweight and modular.

## **Example: Using HttpClient in a Service**

### **Step 1: Create a Service**

| import { Injectable } from '@angular/core';  import { HttpClient } from '@angular/common/http';  import { Observable } from 'rxjs';  @Injectable({ providedIn: 'root' })  export class UserService {  constructor(private http: HttpClient) {}  getUsers(): Observable<any> {  return this.http.get('https://jsonplaceholder.typicode.com/users');  }  } |
| --- |

## **Example: Using the Service in a Standalone Component**

| import { Component, inject, OnInit } from '@angular/core';  import { CommonModule } from '@angular/common';  import { UserService } from './user.service';  @Component({  selector: 'app-user-list',  standalone: true,  imports: [CommonModule],  template: `  <h2>Users</h2>  <ul>  <li \*ngFor="let user of users">{{ user.name }}</li>  </ul>  `  })  export class UserListComponent implements OnInit {  private userService = inject(UserService);  users: any[] = [];  ngOnInit() {  this.userService.getUsers().subscribe((data) => {  this.users = data;  });  }  } |
| --- |

## **Optional Enhancements with provideHttpClient**

You can customize the HTTP behavior using configuration helpers:

| import { withInterceptors } from '@angular/common/http';  bootstrapApplication(AppComponent, {  providers: [  provideHttpClient(  withInterceptors([  (req, next) => {  const cloned = req.clone({ setHeaders: { 'X-App': 'DemoApp' } });  return next(cloned);  }  ])  )  ]  }); |
| --- |

## **Benefits of provideHttpClient()**

| **Feature** | **Benefit** |
| --- | --- |
| Tree-shakable | Smaller bundles, faster loading |
| No need for HttpClientModule | Clean module-free architecture |
| Fully compatible with signals | Works well in modern reactive setups |
| Interceptors still supported | Flexible for global request customization |

**Summary**

| **Concept** | **Description** |
| --- | --- |
| HttpClient | Core Angular service for HTTP communication |
| provideHttpClient() | Modern alternative to HttpClientModule |
| Use in standalone apps | Directly in main.ts during bootstrap |
| Works with services | Injected just like before (constructor or inject()) |

## 

## **Fetching Data in Angular with HttpClient**

### **Purpose:**

Angular's HttpClient provides a powerful and type-safe API to make HTTP requests to REST APIs or any backend server.

* Available via the @angular/common/http package
* Works seamlessly with **RxJS Observables**
* Supports **interceptors**, **error handling**, **headers**, and more

## **Setup (Modern Approach for Standalone Apps)**

In main.ts:

| import { provideHttpClient } from '@angular/common/http';  bootstrapApplication(AppComponent, {  providers: [provideHttpClient()]  }); |
| --- |

If you are using a module-based app, import HttpClientModule in AppModule instead.

## **1. GET – Fetch Data**

### **Use Case:**

Retrieve data from the server (e.g., list of users)

| this.http.get<User[]>('https://api.example.com/users')  .subscribe(data => {  console.log('Users:', data);  }); |
| --- |

## **2. POST – Create Data**

### **Use Case:**

Send new data to the server (e.g., submitting a form)

| const newUser = { name: 'Alice', age: 25 };  this.http.post('https://api.example.com/users', newUser)  .subscribe(response => {  console.log('User created:', response);  }); |
| --- |

## **3. PUT – Update Existing Data**

### **Use Case:**

Replace data for an existing record

| const updatedUser = { id: 1, name: 'Alice B', age: 26 };  this.http.put(`https://api.example.com/users/1`, updatedUser)  .subscribe(response => {  console.log('User updated:', response);  }); |
| --- |

## **4. PATCH – Partial Update**

### **Use Case:**

Update only specific fields of a record

| this.http.patch(`https://api.example.com/users/1`, { age: 27 })  .subscribe(response => {  console.log('User partially updated:', response);  }); |
| --- |

## **5. DELETE – Remove Data**

### **Use Case:**

Delete a specific record

| this.http.delete(`https://api.example.com/users/1`)  .subscribe(() => {  console.log('User deleted');  }); |
| --- |

## **Common Options (Headers, Params)**

You can pass options such as headers and query params:

| this.http.get('https://api.example.com/users', {  headers: { Authorization: 'Bearer token' },  params: { page: 1, limit: 10 }  }); |
| --- |

## **Observables & Error Handling**

Use .pipe() with catchError and retry from RxJS:

| import { catchError, retry } from 'rxjs/operators';  import { throwError } from 'rxjs';  this.http.get('/api/data')  .pipe(  retry(2),  catchError(error => {  console.error('Error fetching data', error);  return throwError(() => error);  })  )  .subscribe(data => console.log(data)); |
| --- |

## **Summary of HTTP Methods**

| **Method** | **Purpose** | **Idempotent** | **Angular Method** |
| --- | --- | --- | --- |
| GET | Read data | Yes | http.get() |
| POST | Create new data | No | http.post() |
| PUT | Update/replace data | Yes | http.put() |
| PATCH | Partial update | Yes | http.patch() |
| DELETE | Remove data | Yes | http.delete() |

## **Best Practices**

* Use **services** to organize all API calls.
* Use **RxJS operators** for error handling and retry logic.
* Add **loading states** and **error messages** in the UI.
* Secure requests with **interceptors** for auth tokens.
* Prefer **typed responses** using Angular interfaces.

## 

## **Signals with Observables in Angular**

### **Why Combine Signals with Observables?**

Angular offers both:

* **Signals** for synchronous, fine-grained reactivity
* **Observables (RxJS)** for async data streams like HTTP, user input, and timers

To integrate them smoothly, Angular provides utility functions like toSignal() and fromObservable().

## **toSignal() – Convert Observable to Signal**

### **Purpose:**

Convert an **Observable** into a **Signal**, allowing the signal to automatically update when the observable emits new values.

### **Syntax:**

| import { toSignal } from '@angular/core/rxjs-interop';  const data$ = this.http.get<User[]>('/api/users');  const data = toSignal(data$); |
| --- |

### **Features:**

* Automatically subscribes to the observable
* Updates the signal value reactively
* Optional default value and error handling

### **With default value:**

| const users = toSignal(data$, { initialValue: [] }); |
| --- |

### **With error fallback:**

| const users = toSignal(data$, {  initialValue: [],  requireSync: false,  transformError: () => []  }); |
| --- |

## **fromObservable() – Create a Signal Wrapper for Observable**

### **Purpose:**

Creates a **Signal** from an **Observable**, but preserves the lazy behavior of the observable — doesn't subscribe until used.

### **Syntax:**

| import { fromObservable } from '@angular/core/rxjs-interop';  const count$ = interval(1000);  const countSignal = fromObservable(count$); |
| --- |

fromObservable() is **experimental** as of Angular 17+ and may behave differently than toSignal().

## **Use Cases**

| **Scenario** | **Use** |
| --- | --- |
| Use HTTP data as signal in template | toSignal() |
| Convert BehaviorSubject to signal | toSignal() |
| Use signal-based state inside RxJS pipe | effect() + signal() |

## **Example: HTTP Observable to Signal**

| export class UserListComponent {  users = toSignal(this.http.get<User[]>('/api/users'), {  initialValue: []  });  } |
| --- |

### **Template:**

| <ul>  @for (user of users()) {  <li>{{ user.name }}</li>  }  </ul> |
| --- |

## **Example: Manual Observable Bridge**

| user$ = new BehaviorSubject('Alice');  userSignal = toSignal(this.user$, { initialValue: 'Loading...' }); |
| --- |

## **Example: Signal to Observable**

If you want to use a signal inside an observable pipe:

| signalValue = signal(42);  const double$ = computed(() => this.signalValue() \* 2).asObservable(); |
| --- |

Use .asObservable() if you need to interop the other way (signal → observable).

## **Summary**

| **Function** | **Direction** | **Description** |
| --- | --- | --- |
| toSignal() | Observable → Signal | Creates a reactive signal from an observable |
| fromObservable() | Observable → Lazy Signal | Lazy conversion, experimental |
| .asObservable() | Signal → Observable | Converts a signal back to an observable |

## **Best Practices**

* Use toSignal() for integrating HTTP or event streams into your reactive components.
* Wrap signal state in computed() for derived values.
* Use effect() when combining signal and observable side effects.

## 

## **Error Handling & Retry Logic in Angular**

### **Purpose:**

Ensure that your Angular app gracefully handles network/API errors and optionally retries failed requests when appropriate.

Angular uses **HttpClient** for making HTTP calls and **RxJS operators** for handling async data, including retries and error responses.

## **1. Basic Error Handling with catchError**

### **catchError is an RxJS operator used to handle errors from observables.**

### **Example:**

| this.http.get('/api/users')  .pipe(  catchError(error => {  console.error('Error occurred:', error);  return throwError(() => error);  })  )  .subscribe({  next: data => console.log(data),  error: err => console.log('Handled in subscribe:', err)  }); |
| --- |

Always return a new observable using throwError() or a fallback observable in catchError.

## **2. Retrying Failed Requests with retry**

### **retry(n) automatically re-attempts the request n times on failure.**

### **Example:**

| this.http.get('/api/data')  .pipe(  retry(3), // Retry up to 3 times  catchError(err => {  console.error('Failed after retries:', err);  return throwError(() => err);  })  )  .subscribe(); |
| --- |

Use retry() only for **transient errors** like network issues — not for logic/validation errors.

## **3. Controlled Retry with retryWhen**

Use retryWhen() for custom retry logic with delays or limits.

### **Example:**

| import { retryWhen, delay, take, scan } from 'rxjs/operators';  this.http.get('/api/data')  .pipe(  retryWhen(errors =>  errors.pipe(  scan((acc, error) => {  if (acc >= 2) throw error;  return acc + 1;  }, 0),  delay(1000) // Delay 1s before each retry  )  ),  catchError(err => {  console.error('Error after retryWhen:', err);  return throwError(() => err);  })  )  .subscribe(); |
| --- |

## **4. Error Handling in Services**

Use a centralized error handler inside your service:

| private handleError(error: HttpErrorResponse) {  if (error.status === 0) {  console.error('Network error:', error.error);  } else {  console.error(`Backend returned code ${error.status}`, error.error);  }  return throwError(() => new Error('Something went wrong. Please try again.'));  } |
| --- |

Then apply it to each call:

| this.http.get('/api/users').pipe(catchError(this.handleError)); |
| --- |

## **5. Fallback Value Example**

Instead of throwing an error, you can return default values:

| this.http.get('/api/settings')  .pipe(  catchError(() => of({ theme: 'light', notifications: true }))  )  .subscribe(settings => console.log('Settings:', settings)); |
| --- |

## **Useful RxJS Operators**

| **Operator** | **Purpose** |
| --- | --- |
| catchError | Catch and handle errors |
| retry(n) | Retry failed request n times |
| retryWhen | Retry with custom logic (e.g. delay) |
| throwError | Emit a new error observable |
| of() | Return fallback/default observable |

## **Best Practices**

* Always **handle HTTP errors** with catchError in services.
* Show user-friendly messages via toasts/dialogs for errors.
* Avoid infinite retry loops; use retryWhen with take() or scan() to limit retries.
* Use **interceptors** for global error handling (e.g., token expiry).
* Log errors to a server or monitoring service in production.

## **Summary**

| **Concept** | **Use Case** |
| --- | --- |
| catchError() | Gracefully handle API failures |
| retry(n) | Retry temporary failures (e.g. network) |
| retryWhen() | Customize retry behavior with logic/delay |
| fallback values | Return defaults instead of throwing error |
| Global Handling | Use interceptors or centralized handler |

## 

## **Working with async Pipe + Signals in Angular**

### **Purpose:**

Angular now supports two powerful paradigms for managing reactive data:

* **Signals** – For fine-grained, synchronous reactivity
* **Observables** – For async data streams (used with async pipe)

Understanding how to use each—and how to **combine them effectively**—is key to writing clean, reactive Angular code.

## **1. async Pipe Overview**

### **What is the async pipe?**

A **template pipe** that subscribes to an Observable (or Promise) and renders its latest value.

### **Usage:**

| <p>Welcome, {{ user$ | async }}</p> |
| --- |

### **Benefits:**

* Automatically unsubscribes on component destroy
* Clean template usage without manual subscribe()

**2. Signals Overview**

### **What is a Signal?**

A **synchronous, reactive primitive** introduced in Angular 16+ that holds a value and notifies dependents on change.

### **Usage:**

| const name = signal('Alice'); |
| --- |

| <p>Hello, {{ name() }}</p> |
| --- |

## **Differences Between async Pipe and Signals**

| **Feature** | **async Pipe (Observable)** | **signal()** |
| --- | --- | --- |
| Async support | Yes (e.g., HTTP, intervals) | No (for sync state only) |
| Lifecycle mgmt | Auto-subscribe/unsubscribe | Not needed (always live) |
| Template usage | `{{ value$ | async }}` |
| Push-based | Yes | Yes |
| Use cases | API data, streams | UI state, forms, flags |

## **Interoperability: Signals + Observables**

### **Convert Observable → Signal:**

Use toSignal() from @angular/core/rxjs-interop:

| import { toSignal } from '@angular/core/rxjs-interop';  const user$ = this.http.get('/api/user');  const user = toSignal(user$, { initialValue: null }); |
| --- |

| <p>{{ user()?.name }}</p> |
| --- |

Benefit: This allows you to **remove async pipe** and use the signal instead.

### **Convert Signal → Observable:**

You can call .asObservable() on a signal to use it in an RxJS pipeline:

| const counter = signal(0);  const counter$ = counter.asObservable(); |
| --- |

## **When to Use Each**

| **Scenario** | **Recommendation** |
| --- | --- |
| HTTP/API data | Use Observable + async |
| Local UI state (e.g., toggle, form fields) | Use signal() |
| Derived/formatted values | Use computed() |
| Bridging async to signals | Use toSignal() |

## **Summary**

| **Concept** | **Usage Example** |
| --- | --- |
| async pipe | `{{ data$ |
| signal() | const count = signal(0); {{ count() }} |
| toSignal() | Convert Observable to reactive Signal |
| .asObservable() | Convert Signal to Observable |
| Composition | Use both together to manage UI and data |

## **Best Practices**

* Prefer signal() for **form state, toggles, derived values**
* Use async pipe for **async streams** like HTTP, websockets, intervals
* Use toSignal() when bridging observable-based services into signal-based UI
* Avoid unnecessary conversions between signals and observables unless needed for reusability

## 

## **Lab Activity: Fetch Data from an API using Angular**

### **Objective:**

Use Angular's HttpClient to fetch a list of users or posts from an API and display it in a component. Bonus: Convert the observable into a signal for modern reactivity.

### **Step-by-Step Instructions**

### **Step 1: Create a Standalone Component**

| ng generate component api-data --standalone |
| --- |

### **Step 2: Add Route to Component**

Update src/app/app.routes.ts:

| import { Routes } from '@angular/router';  import { ApiDataComponent } from './api-data/api-data.component';  export const appRoutes: Routes = [  { path: '', component: ApiDataComponent }  ]; |
| --- |

### **Step 3: Provide HttpClient in main.ts**

Ensure this is present in src/main.ts:

| import { provideHttpClient } from '@angular/common/http';  bootstrapApplication(AppComponent, {  providers: [provideHttpClient()]  }); |
| --- |

### **Step 4: Create a Data Service**

Create src/app/services/user.service.ts:

| import { Injectable } from '@angular/core';  import { HttpClient } from '@angular/common/http';  import { Observable } from 'rxjs';  @Injectable({ providedIn: 'root' })  export class UserService {  constructor(private http: HttpClient) {}  getUsers(): Observable<any[]> {  return this.http.get<any[]>('https://jsonplaceholder.typicode.com/users');  }  } |
| --- |

### **Step 5: Fetch and Display the Data (Using async pipe)**

Edit api-data.component.ts:

| import { Component } from '@angular/core';  import { CommonModule } from '@angular/common';  import { UserService } from '../services/user.service';  import { Observable } from 'rxjs';  @Component({  selector: 'app-api-data',  standalone: true,  imports: [CommonModule],  templateUrl: './api-data.component.html'  })  export class ApiDataComponent {  users$: Observable<any[]>;  constructor(private userService: UserService) {  this.users$ = this.userService.getUsers();  }  } |
| --- |

In api-data.component.html:

| <h2>User List</h2>  <ul>  <li \*ngFor="let user of users$ | async">  {{ user.name }} ({{ user.email }})  </li>  </ul> |
| --- |

### **Optional Bonus: Convert Observable to Signal**

In api-data.component.ts, update:

| import { toSignal } from '@angular/core/rxjs-interop';  users = toSignal(this.userService.getUsers(), { initialValue: [] }); |
| --- |

And update the HTML:

| <ul>  @for (user of users()) {  <li>{{ user.name }} ({{ user.email }})</li>  }  </ul> |
| --- |

Ensure you're using Angular 17+ and @for is supported.

### **Step 6: Run the App**

| ng serve |
| --- |

Go to<http://localhost:4200> and verify:

* User data is fetched and rendered in the component
* Optional: Signal-based version updates automatically with reactive rendering

### **Outcome:**

You have successfully:

* Used HttpClient to fetch data from a public API
* Rendered it using the async pipe or signals
* Learned how to convert observable data into a reactive signal

## 

## **Lab Activity: Show Loading State and Handle Errors in Angular**

### **Objective:**

* Show a **loading indicator** while data is being fetched
* Display a **friendly error message** if the request fails
* Use HttpClient, RxJS, and optionally **Signals** for reactivity

### **Step-by-Step Instructions**

### **Step 1: Update or Create the Data Component**

Assuming you already have a UserService with getUsers():

### **api-data.component.ts**

| import { Component, signal, effect } from '@angular/core';  import { CommonModule } from '@angular/common';  import { UserService } from '../services/user.service';  import { catchError, finalize } from 'rxjs/operators';  import { of } from 'rxjs';  @Component({  selector: 'app-api-data',  standalone: true,  imports: [CommonModule],  templateUrl: './api-data.component.html'  })  export class ApiDataComponent {  users = signal<any[]>([]);  loading = signal<boolean>(false);  error = signal<string | null>(null);  constructor(private userService: UserService) {  this.fetchUsers();  }  fetchUsers() {  this.loading.set(true);  this.error.set(null);  this.userService.getUsers()  .pipe(  catchError(err => {  this.error.set('Failed to load users. Please try again later.');  return of([]); // fallback value  }),  finalize(() => this.loading.set(false))  )  .subscribe(data => {  this.users.set(data);  });  }  } |
| --- |

### **Step 2: Update the Component Template**

### **api-data.component.html**

| <h2> User List</h2>  <!-- Loading State -->  <p \*ngIf="loading()">Loading users...</p>  <!-- Error State -->  <p \*ngIf="error()" style="color: red;">{{ error() }}</p>  <!-- Data Display -->  <ul \*ngIf="!loading() && !error() && users().length > 0">  @for (user of users()) {  <li>{{ user.name }} ({{ user.email }})</li>  }  </ul>  <!-- No Results -->  <p \*ngIf="!loading() && !error() && users().length === 0">  No users found.  </p> |
| --- |

### **Step 3: Test the App**

| ng serve |
| --- |

* Check that "Loading users..." is shown initially.
* If the request fails (e.g., disconnect network), see the error message.
* Once data loads, see the list of users.

### **Bonus Ideas**

* Replace the <p> loading message with a spinner component
* Add a "Retry" button that calls fetchUsers() again
* Use a loading skeleton or shimmer effect with \*ngIf="loading()"

### **Outcome:**

You have successfully:

* Managed loading state using signal<boolean>()
* Handled API errors gracefully with catchError
* Displayed feedback in the UI using Angular template bindings

## 

# **Build a Task Tracker App in Angular 19**

## **Objective**

By the end of this lab, you will build a fully functional Task Tracker application with:

* **Add**, **delete**, **update**, and **view** tasks
* **Mark tasks as completed**
* **Routing** for task list, create, and detail pages
* **Reactive state** using Angular **Signals**
* **Persistence** using **Local Storage**
* **Separation of concerns** using a dedicated **TaskService**

## **Folder Structure**

| src/app/  ├── components/  │ ├── task-list/  │ ├── task-form/  │ └── task-detail/  ├── services/  │ ├── task.service.ts  │ ├── local-storage.service.ts  │ └── task.model.ts  ├── app.routes.ts  └── main.ts |
| --- |

## **Step-by-Step Instructions**

### **Step 1: Create the Angular App**

| ng new task-tracker --standalone --routing=true  cd task-tracker |
| --- |

Generate standalone components:

| ng generate component components/task-list --standalone  ng generate component components/task-form --standalone  ng generate component components/task-detail --standalone |
| --- |

### **Step 2: Create Task Model and Local Storage Service**

**src/app/services/task.model.ts**

| export interface Task {  id: number;  title: string;  completed: boolean;  } |
| --- |

**src/app/services/local-storage.service.ts**

| import { Injectable } from '@angular/core';  import { Task } from './task.model';  const TASKS\_KEY = 'tasks';  @Injectable({ providedIn: 'root' })  export class LocalStorageService {  getTasks(): Task[] {  return JSON.parse(localStorage.getItem(TASKS\_KEY) || '[]');  }  saveTasks(tasks: Task[]): void {  localStorage.setItem(TASKS\_KEY, JSON.stringify(tasks));  }  } |
| --- |

### **Step 3: Implement the Task CRUD Service**

**src/app/services/task.service.ts**

| import { Injectable, signal } from '@angular/core';  import { Task } from './task.model';  import { LocalStorageService } from './local-storage.service';  @Injectable({ providedIn: 'root' })  export class TaskService {  private tasks = signal<Task[]>([]);  constructor(private storage: LocalStorageService) {  this.tasks.set(this.storage.getTasks());  }  getTasks() {  return this.tasks.asReadonly();  }  getTaskById(id: number): Task | undefined {  return this.tasks().find(t => t.id === id);  }  addTask(task: Task) {  const newTask = { ...task, id: Date.now() };  this.updateTasks([...this.tasks(), newTask]);  }  deleteTask(id: number) {  this.updateTasks(this.tasks().filter(t => t.id !== id));  }  updateTask(updatedTask: Task) {  const updated = this.tasks().map(t => t.id === updatedTask.id ? updatedTask : t);  this.updateTasks(updated);  }  toggleComplete(id: number) {  const updated = this.tasks().map(t =>  t.id === id ? { ...t, completed: !t.completed } : t  );  this.updateTasks(updated);  }  private updateTasks(tasks: Task[]) {  this.tasks.set(tasks);  this.storage.saveTasks(tasks);  }  } |
| --- |

### **Step 4: Define Routes**

**src/app/app.routes.ts**

| import { Routes } from '@angular/router';  import { TaskListComponent } from './components/task-list/task-list.component';  import { TaskDetailComponent } from './components/task-detail/task-detail.component';  export const routes: Routes = [  { path: '', component: TaskListComponent },  { path: 'task/:id', component: TaskDetailComponent }  ]; |
| --- |

### **Step 5: Build the Task List Component**

**task-list.component.ts**

| import { Component } from '@angular/core';  import { CommonModule } from '@angular/common';  import { RouterModule } from '@angular/router';  import { TaskFormComponent } from '../task-form/task-form.component';  import { TaskService } from '../../services/task.service';  @Component({  selector: 'app-task-list',  standalone: true,  imports: [CommonModule, RouterModule, TaskFormComponent],  templateUrl: './task-list.component.html'  })  export class TaskListComponent {  tasks = this.taskService.getTasks();  constructor(private taskService: TaskService) {}  addTask(task: any) {  this.taskService.addTask(task);  }  deleteTask(id: number) {  this.taskService.deleteTask(id);  }  toggleComplete(taskId: number) {  this.taskService.toggleComplete(taskId);  }  } |
| --- |

**task-list.component.html**

| <h2>Task Tracker</h2>  <app-task-form (submitTask)="addTask($event)"></app-task-form>  <ul>  @for (task of tasks()) {  <li>  <input type="checkbox" [checked]="task.completed" (change)="toggleComplete(task.id)" />  <a [routerLink]="['/task', task.id]">{{ task.title }}</a>  <button (click)="deleteTask(task.id)">Delete</button>  </li>  }  </ul> |
| --- |

### **Step 6: Build the Task Form Component**

**task-form.component.ts**

| import { Component, EventEmitter, Output } from '@angular/core';  import { CommonModule } from '@angular/common';  import { ReactiveFormsModule, FormBuilder, Validators } from '@angular/forms';  @Component({  selector: 'app-task-form',  standalone: true,  imports: [CommonModule, ReactiveFormsModule],  templateUrl: './task-form.component.html'  })  export class TaskFormComponent {  @Output() submitTask = new EventEmitter();  form = this.fb.group({  title: ['', Validators.required]  });  constructor(private fb: FormBuilder) {}  onSubmit() {  if (this.form.valid) {  this.submitTask.emit({ ...this.form.value, completed: false });  this.form.reset();  }  }  } |
| --- |

**task-form.component.html**

| <form [formGroup]="form" (ngSubmit)="onSubmit()">  <input formControlName="title" placeholder="New task..." />  <button type="submit">Add</button>  <div \*ngIf="form.get('title')?.invalid && form.get('title')?.touched">  Title is required.  </div>  </form> |
| --- |

### **Step 7: Build the Task Detail Component**

**task-detail.component.ts**

| import { Component } from '@angular/core';  import { CommonModule } from '@angular/common';  import { ActivatedRoute, Router, RouterModule } from '@angular/router';  import { TaskService } from '../../services/task.service';  import { Task } from '../../services/task.model';  @Component({  selector: 'app-task-detail',  standalone: true,  imports: [CommonModule, RouterModule],  templateUrl: './task-detail.component.html'  })  export class TaskDetailComponent {  task: Task | undefined;  constructor(  private route: ActivatedRoute,  private router: Router,  private taskService: TaskService  ) {  const id = Number(this.route.snapshot.paramMap.get('id'));  this.task = this.taskService.getTaskById(id);  }  markCompleted() {  if (this.task && !this.task.completed) {  this.taskService.updateTask({ ...this.task, completed: true });  this.router.navigateByUrl('/');  }  }  deleteTask() {  if (this.task) {  this.taskService.deleteTask(this.task.id);  this.router.navigateByUrl('/');  }  }  } |
| --- |

**task-detail.component.html**

| <div \*ngIf="task">  <h2>{{ task.title }}</h2>  <p>Status: {{ task.completed ? 'Completed' : 'Incomplete' }}</p>  <button (click)="markCompleted()" [disabled]="task.completed">Mark as Completed</button>  <button (click)="deleteTask()">Delete</button>  </div> |
| --- |

### **Step 8: Bootstrap with Router**

**main.ts**

| import { bootstrapApplication } from '@angular/platform-browser';  import { provideRouter } from '@angular/router';  import { routes } from './app/app.routes';  import { TaskListComponent } from './app/components/task-list/task-list.component';  bootstrapApplication(TaskListComponent, {  providers: [provideRouter(routes)]  }); |
| --- |

## **Run and Test**

| ng serve |
| --- |

Go to http://localhost:4200

* Add tasks using the form
* Toggle completion from list
* Click a task to view detail
* Mark as completed or delete from detail view
* Refresh to confirm Local Storage persistence

## **Outcome**

You have successfully built a **modular Angular 19 Task Tracker App** with:

* Standalone Components
* Routing & Signals
* Local Storage Persistence
* Full CRUD via a Service Layer

# **Angular Deployment**

## **1. Production Build**

Before deploying, you must build your Angular app for production:

| ng build --configuration production |
| --- |

**What happens during ng build --configuration production?**

* Minifies and uglifies JS/CSS
* Tree-shakes unused code
* Generates dist/ folder with static files
* Optimizes for performance (e.g., inlining critical CSS, etc.)

## **2. Deployment Options**

### **2.1. Static Hosting (Single Page Application - SPA)**

#### **What to deploy:**

Upload contents of dist/your-app-name/ to your hosting provider.

#### **Common Static Hosts:**

| **Platform** | **Deployment Method** |
| --- | --- |
| **GitHub Pages** | ng deploy --base-href=/repo-name/ |
| **Firebase** | firebase deploy |
| **Netlify** | Drag & drop or connect Git repo |
| **Vercel** | Git integration with auto-build |
| **AWS S3 + CloudFront** | Manual or scripted with CLI |
| **Azure Static Web Apps** | GitHub integration |

#### **Handling 404 in SPA:**

Set up fallback to index.html in routing (important for Angular routing):

Example for **Netlify**:

| /\_redirects  /\* /index.html 200 |
| --- |

### **2.2. Using Angular CLI Deploy (optional)**

Install deployment packages:

| ng add angular-cli-ghpages # for GitHub Pages |
| --- |

Then deploy:

| ng deploy |
| --- |

## **3. Environment Configuration**

Angular uses environment.ts files for different builds:

* src/environments/environment.ts (default)
* src/environments/environment.prod.ts (for production)

| export const environment = {  production: true,  apiUrl: 'https://api.myapp.com'  }; |
| --- |

You can access environment variables in code:

| import { environment } from '../environments/environment'; |
| --- |

## **4. Server-Side Rendering (SSR) / Angular Universal**

### **Why use SSR?**

* SEO friendly
* Faster initial load time
* Pre-renders the page on the server

### **Setup:**

| ng add @angular/ssr  npm run build:ssr  npm run serve:ssr |
| --- |

Deploying SSR apps:

* Use Node.js-capable server (e.g., Vercel, Firebase Functions, Heroku, or a custom server like Express)
* SSR generates a server bundle alongside browser build

**5. CI/CD Pipelines (GitHub Actions, GitLab, etc.)**

**Example GitHub Actions Workflow:**

| name: Deploy Angular App  on:  push:  branches: [main]  jobs:  build:  runs-on: ubuntu-latest  steps:  - uses: actions/checkout@v3  - uses: actions/setup-node@v3  with:  node-version: 18  - run: npm install  - run: npm run build -- --configuration production  - uses: peaceiris/actions-gh-pages@v3  with:  github\_token: ${{ secrets.GITHUB\_TOKEN }}  publish\_dir: ./dist/your-app |
| --- |

## **6. Dockerizing Angular App**

### **Dockerfile:**

| # Build Stage  FROM node:18-alpine as build  WORKDIR /app  COPY . .  RUN npm install  RUN npm run build -- --configuration production  # Serve Stage  FROM nginx:alpine  COPY --from=build /app/dist/your-app /usr/share/nginx/html  EXPOSE 80 |
| --- |

### **Build & Run:**

| docker build -t angular-app .  docker run -p 8080:80 angular-app |
| --- |

## **7. Best Practices**

* Always use --configuration production
* Use a CDN (e.g., Cloudflare) for caching and asset delivery
* Set up a fallback route for SPAs
* Monitor performance (use Lighthouse)
* Test after build using: http-server dist/your-app
* Secure your headers (CSP, HSTS) in server config

# **Lab Exercise: Deploying an Angular App**

**Objective:**  
Learn how to build and deploy an Angular 19 application using production configurations. You’ll deploy to both local and cloud-based static hosting (e.g., Firebase or Netlify).

## **Part 1: Build the App for Production**

### **Step 1: Open your project**

| cd my-app |
| --- |

### **Step 2: Run a production build**

| ng build --configuration production |
| --- |

**What to expect:**

* Files generated inside dist/my-app/
* Minified .js, .css, and index.html

## **Part 2: Serve the App Locally**

To test the production build before deployment:

### **Step 1: Install a static server**

| npm install -g http-server |
| --- |

### **Step 2: Serve the dist/ folder**

| http-server dist/my-app |
| --- |

### **Step 3: Open browser**

Visit: <http://localhost:8080>  
  
Confirm the app is working as expected.

## **Part 3: Deploy to Firebase Hosting**

### **Step 1: Install Firebase CLI (if not already)**

| npm install -g firebase-tools |
| --- |

### **Step 2: Log in to Firebase**

| firebase login |
| --- |

### **Step 3: Initialize Firebase in your project**

| firebase init |
| --- |

* Choose **Hosting**
* Use dist/my-app as the **public directory**
* Configure as a **single-page app** → Yes
* Don’t overwrite index.html if asked

### **Step 4: Deploy**

| firebase deploy |
| --- |

You’ll get a public hosting URL like:  
 https://your-project-id.web.app

## **Part 4 (Optional): Deploy to Netlify**

### **Step 1: Install Netlify CLI**

| npm install -g netlify-cli |
| --- |

### **Step 2: Login**

| netlify login |
| --- |

### **Step 3: Deploy**

| netlify deploy |
| --- |

* Set publish directory: dist/my-app
* Choose manual or auto deploy
* Netlify gives you a live preview link

To go live:

| netlify deploy --prod |
| --- |

## **Extra: GitHub Pages Deployment (Optional)**

### **Step 1: Add CLI deploy package**

| ng add angular-cli-ghpages |
| --- |

### **Step 2: Deploy to GitHub Pages**

| ng deploy --base-href=/your-repo-name/ |
| --- |

Live at: https://your-username.github.io/your-repo-name/

## **Lab Summary**

| **Task** | **Status** |
| --- | --- |
| Production Build Created | Yes |
| Local Server Tested | Yes |
| Firebase Deployed | Yes |
| Netlify / GitHub Pages | Yes(optional) |