Angular is a web framework that empowers developers to build fast, reliable applications.

The Angular CLI

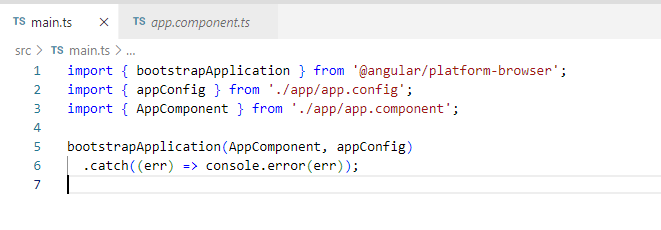
The Angular CLI is a command-line interface tool which allows you to scaffold, develop, test, deploy, and maintain Angular applications directly from a command shell.

ng new angular-sample

Local development server :

npm run start -> ng serve

angular.json – configuration file for the setting provide for angular cli

Bootstraps an instance of an Angular application and renders a standalone component as the application's root component.

A screenshot of a computer code

AI-generated content may be incorrect.

Standalone : True is automatically set for Angular 19+

Ng generate component <Component-Name>

String interpolation: {{ selectedUser.name}}

Property binding: [src] = “selectedUser.avatar”

Event handler: <button (click)="onSelectUser()"></button>

Change detection in Angular is the mechanism by which the framework keeps the view (UI) synchronized with the model (data). Angular automatically checks the component tree to see if any changes have occurred in the data, and if so, it updates the DOM to reflect those changes. This process is crucial for keeping the application’s UI consistent with the underlying data.

Here’s a breakdown of how change detection works in Angular:

**1. How Change Detection Works:**

Angular uses a **change detection cycle** to determine when to update the view. It checks for changes in the component’s data, and when it detects a change, it re-renders the affected components.

* **Components and Directives:** Each component and directive in Angular has a change detection mechanism that ensures its view stays in sync with the model.
* **Change Detection Strategies:** There are two strategies to control change detection:
  + **Default (CheckAlways):** Angular runs change detection for all components in the component tree, which is useful for most cases but can be performance-intensive if there are many components.
  + **OnPush:** This strategy tells Angular to run change detection only when specific conditions change, such as when an input property of the component changes, or an event is triggered within the component. This can improve performance significantly by reducing unnecessary checks.

**2. Change Detection Mechanism:**

* **Zone.js:** Angular relies on Zone.js for automatic change detection. Zone.js listens for asynchronous events (like HTTP requests, timeouts, or user interactions) and triggers change detection automatically when an event occurs.
* **Dirty Checking:** Angular performs a process called dirty checking to compare the old value of a property with the new value and check whether any updates are needed.
* **Change Detection Tree:** Angular builds a tree of components, and each node is checked for changes. When a change is detected, Angular updates the view accordingly.

**3. Triggering Change Detection Manually:**

You can manually trigger change detection if needed, especially in cases where Angular doesn’t detect changes automatically. For instance, when you interact with data outside Angular’s zone (like via third-party libraries), Angular may not be aware of changes.

* **ChangeDetectorRef:** You can inject the ChangeDetectorRef service and manually call methods like:
  + markForCheck(): Marks the component and its ancestors for change detection on the next cycle.
  + detectChanges(): Forces Angular to check the component and its children for changes immediately.
  + detach(): Detaches the component from the change detection tree, preventing Angular from checking it.
  + reattach(): Reattaches the component to the change detection tree.
* **NgZone:** You can also use Angular's NgZone to run code inside or outside Angular's zone. This can help in situations where you want Angular to be aware of changes or deliberately ignore them.

**4. Change Detection Lifecycle:**

Angular triggers change detection at various points in the component lifecycle:

* **ngOnChanges**: Called when any data-bound input properties change.
* **ngDoCheck**: Called during every change detection cycle, allowing you to implement custom change detection logic.
* **ngAfterViewChecked**: Called after Angular has checked the component’s view for changes.

**5. Performance Considerations:**

* **Use OnPush Strategy:** If you know that a component only needs to update in response to specific events (e.g., when an input property changes), consider using the OnPush strategy.
* \**Track By in ngFor:* When using \*ngFor, use the trackBy function to track changes to items in the list more efficiently, reducing unnecessary DOM manipulations.

**Example:**

typescript

Copy

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

@Component({

selector: 'app-hero',

template: `

<div>

<h2>{{ hero.name }}</h2>

</div>

`,

changeDetection: ChangeDetectionStrategy.OnPush

})

export class HeroComponent {

hero = { name: 'Superman' };

changeName() {

this.hero.name = 'Batman';

}

}

In this example, the HeroComponent uses OnPush change detection. Angular will only check the component for changes when the hero input property changes or when an event is triggered within the component.

**6. Common Pitfalls:**

* **Async operations:** If you're using async operations (like HTTP requests or timers) that update the model, make sure change detection is triggered to reflect those updates.
* **Detached components:** If a component is detached from the change detection tree, it will not update, even if its data changes.
* **Complexity of operations in ngDoCheck:** Overuse of ngDoCheck can lead to performance issues, so be mindful of its impact on the change detection cycle.

Change detection is a fundamental part of Angular, and understanding how it works can help you optimize performance and ensure your app is responsive.

Top of Form

Bottom of Form

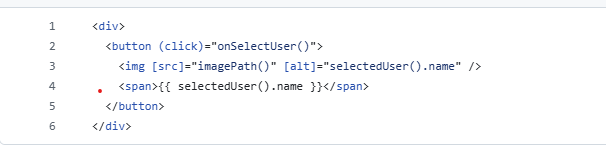
Signals: Introduced in Angular 17.

A signal is an object that stores a value(any type of value, including nested objects)

When a value is changed in a signal, angular will be notified about that change and then angular will make changes to the web app to all those places where that value is changed.



Computed() -> When using computed function angular automatically analyzes whether you are reading the value of a signal in the arrow function. If that’s the case then Angular sets up a subscription to that Signal that’s being used in there and only will recompute the value when the underlying signal value is changed



To set properties on your angular component from html we need to use @Input



A screenshot of a computer code

AI-generated content may be incorrect.

@Input({‘required’: true}) name!: string;

input

Initializer API

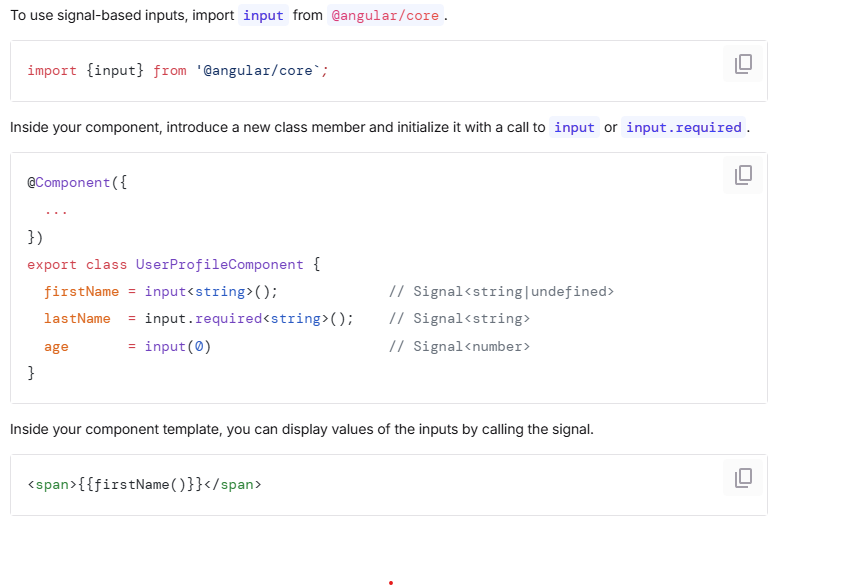
The [input](https://angular.dev/api/core/input) function allows declaration of Angular inputs in directives and components.

There are two variants of inputs that can be declared:

1. **Optional inputs** with an initial value.
2. **Required inputs** that consumers need to set.

By default, the [input](https://angular.dev/api/core/input) function will declare optional inputs that always have an initial value. Required inputs can be declared using the [input.required()](https://angular.dev/api/core/input#required()) function.

Inputs are signals. The values of an input are exposed as a [Signal](https://angular.dev/api/core/Signal). The signal always holds the latest value of the input that is bound from the parent.



Input signals are always read only

@Output() select = new EventEmitter();

onSelectUser(){

this.select.emit();

}

output()- It is creating an EventEmitter not a signal which was getting created when we use input();

ng g c component-name –skip-tests

@for(user of users){

//Alternative of ngFor

}

@if (selectedUser){

}@else{

}

Import {type Task} from ‘./task.model’;

2 way binding : banana in a box [(ngModel)]

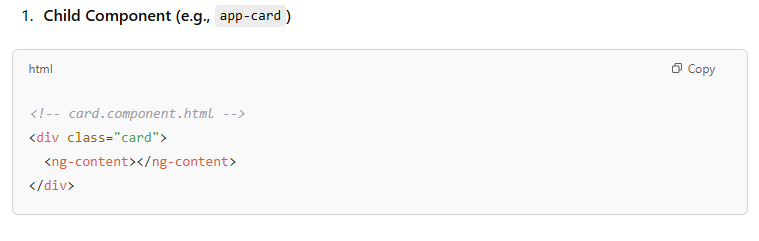
ngModel – FormsModule - @angular/forms;

<form (ngSubmit)=”onSubmit()”/>

Content Projection:

ng-content is a directive in Angular, used for content projection. It allows you to pass dynamic content into a component from its parent. Essentially, it acts like a placeholder for content inside a component template, and whatever content is inside the tags of that component will be inserted into the <ng-content> directive in the component's template.

When you wrap a component in angular with a markup It will not keep the contents of that markup and replace it with the html of the wrapper component



A white rectangular object with black text

AI-generated content may be incorrect.

Result



Import {DatePipe} from ‘@angular/common’;  
<p>{{currentDate | date}} </p>

Constructor(private tasksService: TasksService){}

// another way   
import {inject} from ‘@angular/core’;

private tasksService = inject(TasksService)

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

In Angular, **standalone components** were introduced in **Angular 14** to simplify the creation of components that do not depend on Angular modules (@NgModule). This feature allows you to create components that can be used independently without being part of an NgModule, making them easier to test, reuse, and share across applications.

**Key Features of Standalone Components:**

1. **No NgModule Required**: Standalone components don’t need to be declared in an NgModule.
2. **Simplified Imports**: Standalone components can import other standalone components or directives directly, without needing to declare them in a module.
3. **Reusability**: It enhances the ability to create components that can be reused easily across different parts of an application.

**Example: Creating a Standalone Component in Angular**

1. **Create a Standalone Component**:

bash

Copy

ng generate component standalone-button --standalone

The --standalone flag generates a component that is ready to be used independently.

1. **Modify the Standalone Component**:

Here’s an example of a basic standalone component that renders a button:

typescript

Copy

// standalone-button.component.ts

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

@Component({

selector: 'app-standalone-button',

template: `<button (click)="onClick()">Click Me!</button>`,

styleUrls: ['./standalone-button.component.css'],

standalone: true // Specifies this is a standalone component

})

export class StandaloneButtonComponent {

onClick() {

alert('Button clicked!');

}

}

Notice the standalone: true property in the @Component decorator. This marks the component as standalone.

1. **Using the Standalone Component**:

In Angular 14+, you can use this standalone component directly in other components, without needing to import it into an NgModule.

For example, let's use this StandaloneButtonComponent in a simple app.

typescript

Copy

// app.component.ts

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

import { StandaloneButtonComponent } from './standalone-button/standalone-button.component';

@Component({

selector: 'app-root',

template: `<app-standalone-button></app-standalone-button>`,

standalone: true,

imports: [StandaloneButtonComponent], // Import the standalone component here

})

export class AppComponent {}

In this example:

* The AppComponent imports the StandaloneButtonComponent in the imports array.
* You can now use the <app-standalone-button> selector directly in the AppComponent template.

1. **Run the Application**:

Run your Angular application:

bash

Copy

ng serve

The standalone button component should now be displayed in your app, and when clicked, it will show an alert.

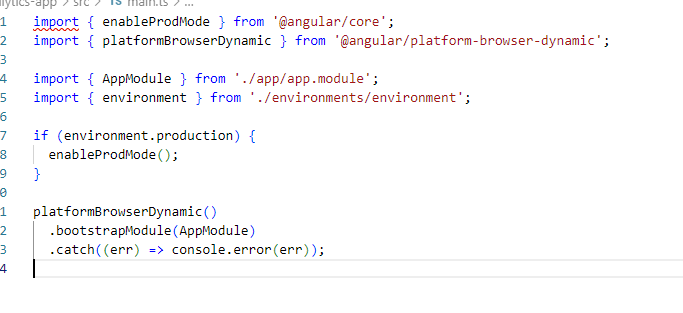
**Benefits of Standalone Components:**

1. **No NgModules**: You don’t need to worry about declaring components in NgModules, which simplifies code.
2. **Better Composition**: You can easily compose your UI by importing only the components you need.
3. **Easier to Share**: Standalone components can be shared and reused across different Angular projects without worrying about module dependencies.

**Key Points:**

* Standalone components can be used without being part of a module.
* Components are declared with standalone: true.
* You import other standalone components directly in the imports array of the component using it.

Old Way:

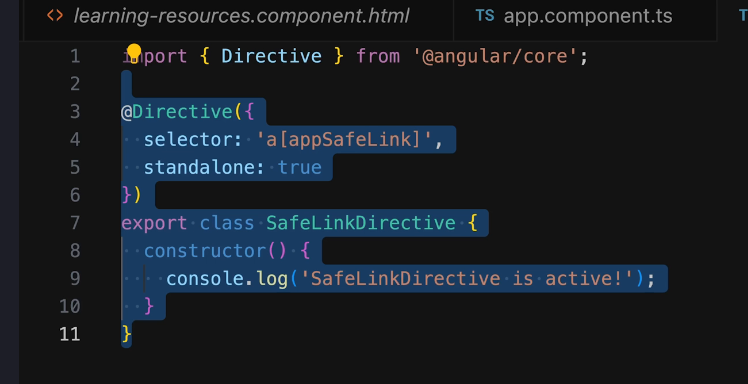


A screenshot of a computer code

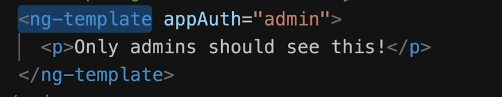
AI-generated content may be incorrect.

Directives – Are enhancements for elements (custom or built-in).

* Attribute – ngModel, ngClass, ngStyle
* Structural - ngIf, ngFor, ngSwitch prefix with \*
* Built in
* Custom



Ng-template – It will be used to display template when the flag is true .



**How They Work Together**

When you use ngIf with an else block, Angular internally uses ng-template to manage the conditional rendering. [The shorthand syntax \*ngIf="condition; else elseBlock" is expanded by Angular into a more explicit form using ng-template1](https://angular.io/api/common/NgIf)[2](https://www.freecodecamp.org/news/everything-you-need-to-know-about-ng-template-ng-content-ng-container-and-ngtemplateoutlet-4b7b51223691/).

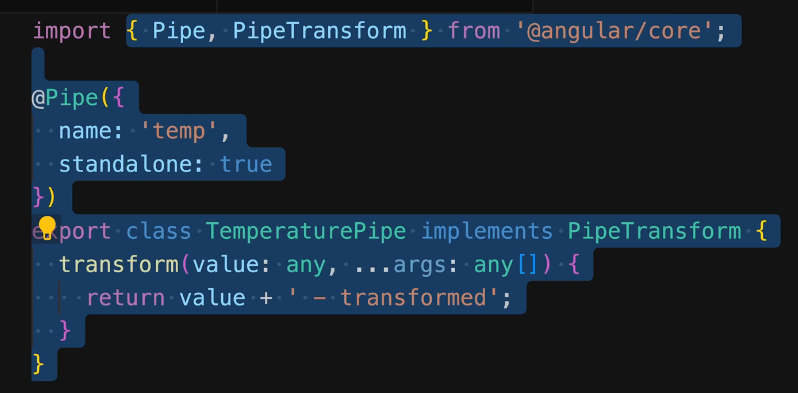
**Key Differences**

* **Rendering**: ngIf directly controls the rendering of elements based on a condition, while ng-template defines a template that can be used and reused in different parts of the application.
* **Syntax**: ngIf uses a shorthand syntax for conditional rendering, whereas ng-template is used for defining templates that are not rendered by default.

Pipes: Pipes transform the way data is displayed.

Date, Decimal, currency etc.

Custom Pipe:



**Pure Pipes**

* **State Management:** Pure pipes are stateless, meaning they don't depend on any internal state. They only rely on the input they receive.
* **Change Detection:** Angular calls pure pipes only when the inputs change. This makes pure pipes very efficient in terms of performance.
* **Usage:** Ideal for most scenarios where the transformation doesn't depend on complex state or side effects. Examples include transforming strings, numbers, or simple arrays.
* **Example:**

typescript

@Pipe({

name: 'purePipe'

})

export class PurePipe implements PipeTransform {

transform(value: any, ...args: any[]): any {

// Transformation logic

return transformedValue;

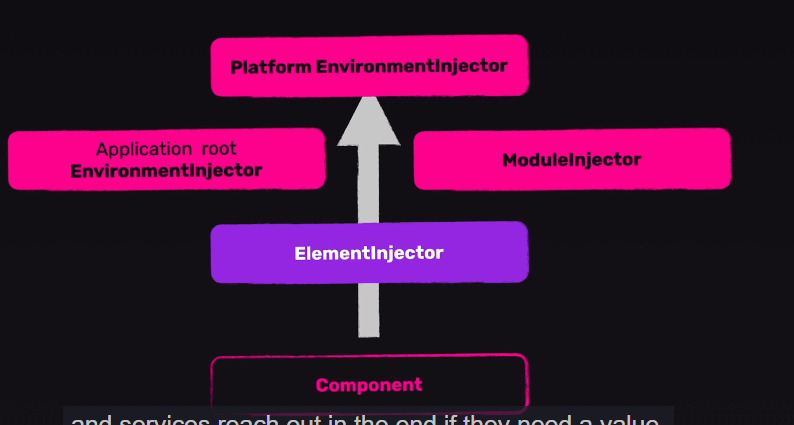
}

}

**Impure Pipes**

* **State Management:** Impure pipes can maintain internal state and might depend on more than just their input. They could have side effects.
* **Change Detection:** Angular calls impure pipes for every change detection cycle, regardless of whether the inputs have changed. This can impact performance.
* **Usage:** Useful when the transformation is dependent on external or complex state, or when dealing with real-time data updates.

Services allow you to share logic and data across the application.



@Injectable({

providedIn: ‘root’

}) => Element Injector

Module Injector->Inside providers property of @Module

Application Root Injector: Inside providers property of @NgModule

PlatformEnvironmentInjector: Inside providers property of main.ts.

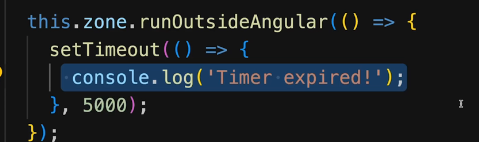
------------------------------------------------------------------------------------------------------------------

Change Detection Mechanism:

Angular build the component tree of your application and wrap it inside a zone provided by zone.js. This zone inform angular about potential events(user events, expired timers) that are going on the page that could potentially lead to the changes that are reflected on the screen.

For example if a user clicks a button, angular gets notified that about that event and then it’s start the change detection process and visit all components in the entire application and check for the changes if there a change in value then Angular updates the real DOM with that value.

Zone pollution – To avoid unnecessary Change detection in angular we can use NgZone to run certain code outside the angular CD cycle.



ChangeDetectionStrategy.OnPush - > It will improve the Angular application performance optimization by limiting the number of change detection cycle triggered. It will only get triggered when the input value to that component change, or any change in the nested components or manually change detection is triggered.

Private cdref = inject(ChangeDetectorRef);

Observables:

RxJS is a library of object produces and controls a stream of data.

To consume an observable we need to subscribe it.

Subscribe will have 3 callbacks:

* 1. Next
  2. Complete
  3. Error

Clean up observable in ngOnDestroy

Subjects are also observables but we take care of emitting values from them instead of observables where we are only consuming the values.

Convert Signal to Observable using – toObservable part of rxjs-interop;

Convert Observable to Signal using – toSignal part of rxjs-interop;

Add HttpClient to Angular App

1. import { NgModule } from '@angular/core';
2. import { FormsModule } from '@angular/forms';
3. import { provideHttpClient } from '@angular/common/http';
5. @NgModule({
6. declarations: [
7. AppComponent,
8. PlacesComponent,
9. // ... etc
10. ],
11. imports: [BrowserModule, FormsModule],
12. providers: [provideHttpClient()],
13. bootstrap: [AppComponent],
14. })
15. export class AppModule {}

HttpInterceptor

1. import {
2. HttpEvent,
3. HttpHandler,
4. HttpInterceptor,
5. HttpRequest,
6. } from '@angular/common/http';
7. import { Observable } from 'rxjs';
9. @Injectable()
10. class LoggingInterceptor implements HttpInterceptor {
11. intercept(req: HttpRequest<unknown>, handler: HttpHandler): Observable<HttpEvent<any>> {
12. console.log('Request URL: ' + req.url);
13. return handler.handle(req);
14. }
15. }

Setup

1. providers: [
2. { provide: HTTP\_INTERCEPTORS, useClass: LoggingInterceptor, multi: true }
3. ]

Form Handling

1. Template Driven – Set up via HTML template
2. Reactive Form – Setup via Typescript code.

Template Driven Form using – ngModel as the attribute to the html elements.

ngModel , ngForm is part of FormsModule.

Form Validation in Template Driven form using HTML5 validatiors required, maxLength, minLength