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

The Angular CLI

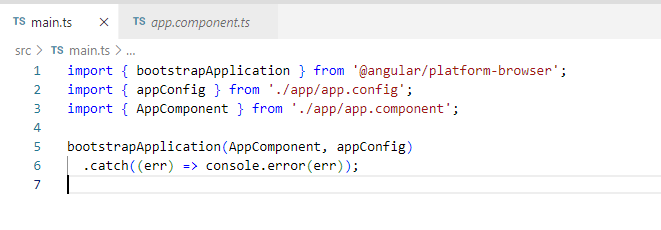
The Angular CLI is a command-line interface tool which allows you to scaffold, develop, test, deploy, and maintain Angular applications directly from a command shell.

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.

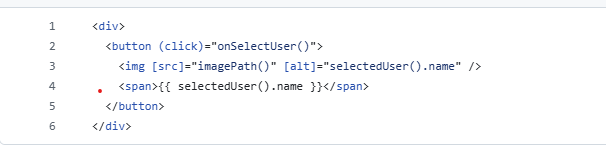
It help to remove old Angular Change Detection approach via Zone.js.

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.

**Key Differences:**

* set: Directly assigns a new value to the signal, overwriting the old value.
* update: Modifies the signal's value based on the current value using a callback.
* mutate: Alters the value in-place (particularly useful for mutable data like objects or arrays).

**When to Use Each:**

* Use **set** when you want to replace the entire state.
* Use **update** when you need to update the state based on its current value.
* Use **mutate** when you want to modify the state without replacing the entire value (especially useful with mutable objects/arrays).

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

input

Initializer API

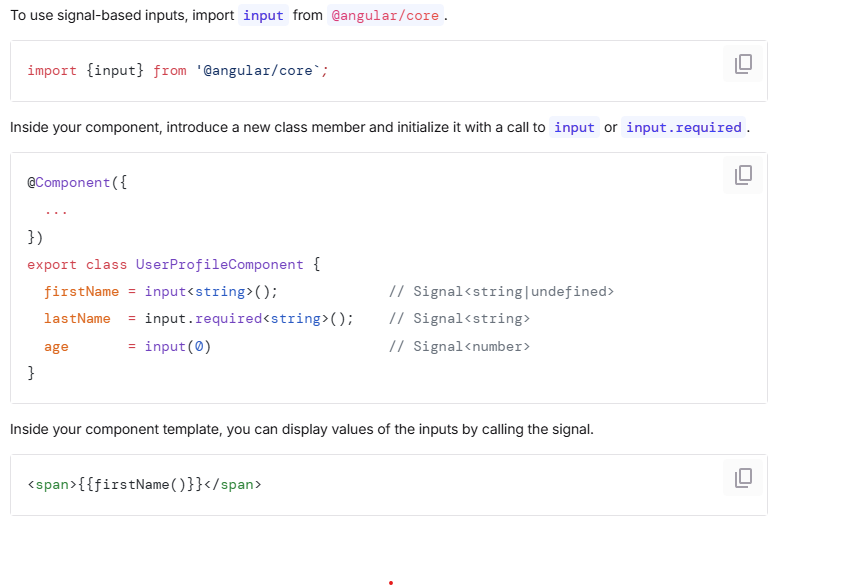
The [input](https://angular.dev/api/core/input) function allows declaration of Angular inputs in directives and components.

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" \l "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)]

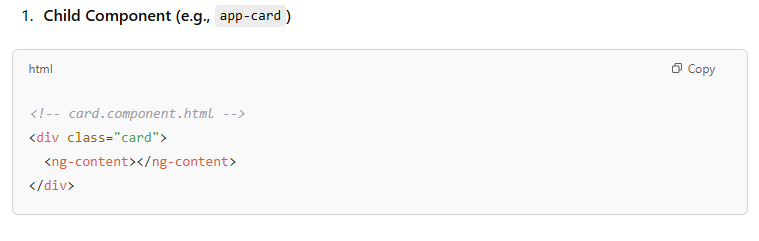
ngModel – FormsModule - @angular/forms;

<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.

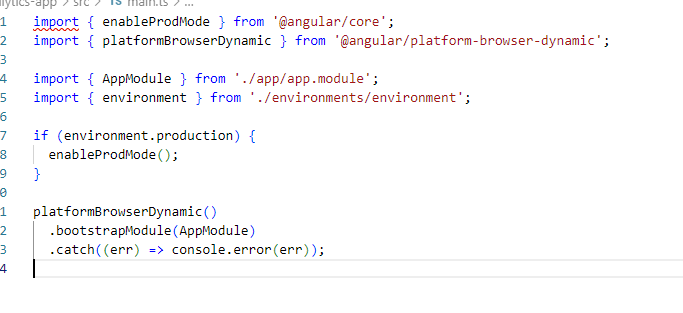
**Benefits of Standalone Components:**

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:

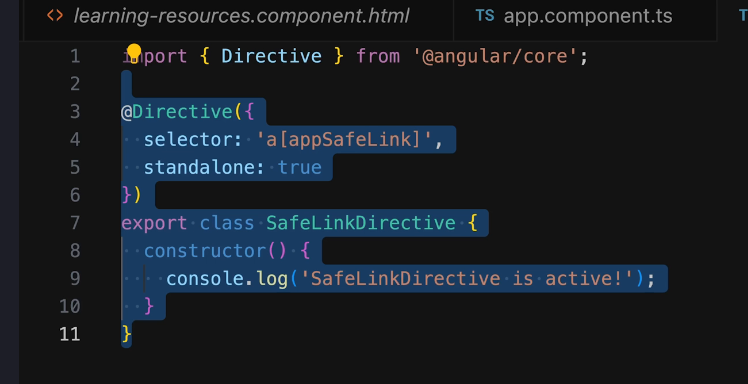


A screenshot of a computer code

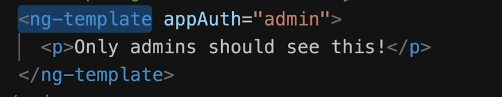
AI-generated content may be incorrect.

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/).

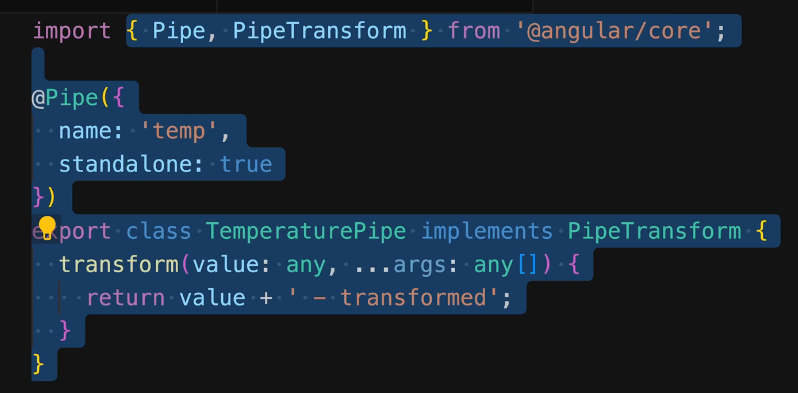
**Key Differences**

* **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;

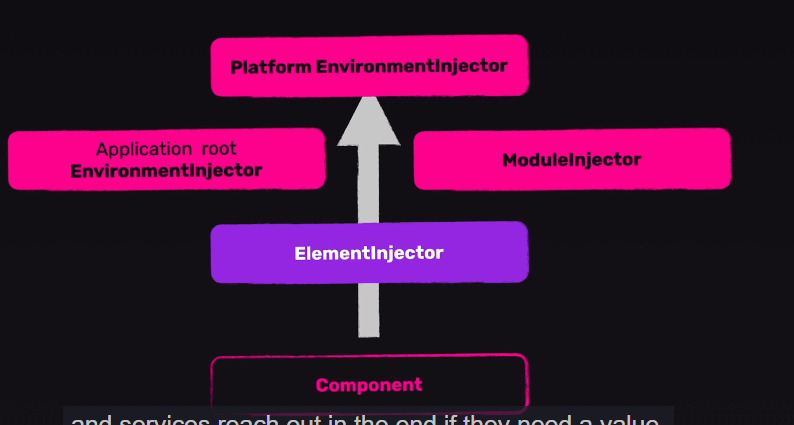
}

}

**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.

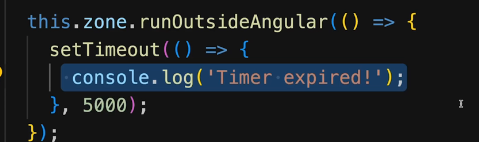
------------------------------------------------------------------------------------------------------------------

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

Every component has a **view encapsulation** setting that determines how the framework scopes a component's styles. There are three view encapsulation modes: Emulated, ShadowDom, and None. You can specify the mode in the @Component decorator:

@Component({ ..., encapsulation: ViewEncapsulation.None,})export class ProfilePhoto { }

check

[ViewEncapsulation.Emulated](https://angular.dev/guide/components/styling#viewencapsulationemulated)

By default, Angular uses emulated encapsulation so that a component's styles only apply to elements defined in that component's template. In this mode, the framework generates a unique HTML attribute for each component instance, adds that attribute to elements in the component's template, and inserts that attribute into the CSS selectors defined in your component's styles.

[ViewEncapsulation.None](https://angular.dev/guide/components/styling#viewencapsulationnone)

This mode disables all style encapsulation for the component. Any styles associated with the component behave as global styles.

[ViewEncapsulation.ShadowDom](https://angular.dev/guide/components/styling#viewencapsulationshadowdom)

This mode scopes styles within a component by using [the web standard Shadow DOM API](https://developer.mozilla.org/docs/Web/Web_Components/Using_shadow_DOM). When enabling this mode, Angular attaches a shadow root to the component's host element and renders the component's template and styles into the corresponding shadow tree.

This mode strictly guarantees that *only* that component's styles apply to elements in the component's template. Global styles cannot affect elements in a shadow tree and styles inside the shadow tree cannot affect elements outside of that shadow tree.

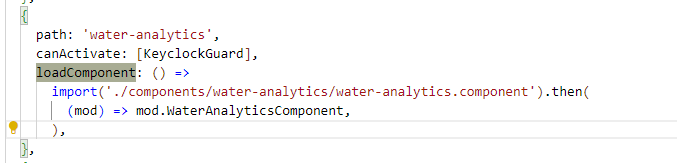
<router-outlet> - import {RouterOutlet} from @angular/router;

Import {RouterLink} from ‘@angular/router’;

<a routerLink=’/tasks’> It will override the default href behaviour

routerLinkActive=”selected” To highlight the selected router link.

Lazy loading Component



**loadChildren (Lazy Loading of Modules)**

* **Purpose**: Used for lazy loading **modules** in Angular.
* **How it works**: When loadChildren is used, Angular will only load the module when it is required (i.e., when the user navigates to a route associated with that module).

**loadComponent (Lazy Loading of Components)**

* **Purpose**: Introduced in Angular 14, loadComponent is used for lazy loading individual **components**, without needing to lazy-load an entire module.
* **How it works**: When loadComponent is used, Angular will only load the specified component when it's required. This allows developers to lazy-load **just a component** instead of an entire module.

Deferrable views, also known as @defer blocks, reduce the initial bundle size of your application by deferring the loading of code that is not strictly necessary for the initial rendering of a page. This often results in a faster initial load and improvement in Core Web Vitals (CWV), primarily Largest Contentful Paint (LCP) and Time to First Byte (TTFB).

[Which dependencies are deferred?](https://angular.dev/guide/templates/defer#which-dependencies-are-deferred)

Components, directives, pipes, and any component CSS styles can be deferred when loading an application.

In order for the dependencies within a @defer block to be deferred, they need to meet two conditions:

1. **They must be standalone.** Non-standalone dependencies cannot be deferred and are still eagerly loaded, even if they are inside of @defer blocks.
2. **They cannot be referenced outside of**@defer**blocks within the same file.** If they are referenced outside the @defer block or referenced within ViewChild queries, the dependencies will be eagerly loaded.

@defer {

<large-component />

}

In web development, **CSR (Client-Side Rendering)** and **SSR (Server-Side Rendering)** are two distinct approaches for rendering web pages. They determine where the content is generated and rendered: on the **client** (the user's browser) or the **server**.

Here's a detailed breakdown of the differences between CSR and SSR:

**1. Client-Side Rendering (CSR)**

* **Definition**: In Client-Side Rendering, the browser (client) is responsible for rendering the content. The server sends a minimal HTML file with references to JavaScript files. The browser then downloads and runs the JavaScript, which dynamically generates the content on the page.
* **How It Works**:
  1. When a user visits the website, the server sends a basic HTML skeleton (without much content).
  2. JavaScript is downloaded, and once it runs, it fetches the data (often via APIs) and populates the page.
  3. After this, the user sees the fully rendered page on their browser.
* **Examples**: Single Page Applications (SPA) built with frameworks like **React**, **Angular**, and **Vue.js** are common examples of CSR.
* **Advantages of CSR**:
  1. **Faster Subsequent Loads**: Once the JavaScript is loaded, navigation between pages is faster because only data is requested and no new HTML pages are fetched.
  2. **Rich Interactivity**: CSR allows for highly interactive, dynamic user interfaces, with JavaScript handling most of the logic on the client side.
  3. **Separation of Concerns**: Allows the frontend to be decoupled from the backend, enabling more flexibility in the development process.
* **Disadvantages of CSR**:
  1. **Initial Load Time**: The first time a user visits the page, the browser must download and execute the JavaScript, which can make the initial load slower.
  2. **SEO Challenges**: Since search engines typically rely on server-generated HTML to index pages, CSR can pose challenges for SEO (though modern techniques like prerendering and server-side APIs can help).
  3. **JavaScript Dependency**: If the user has JavaScript disabled or if there's an issue with the script, the page may not render correctly.

**2. Server-Side Rendering (SSR)**

* **Definition**: In Server-Side Rendering, the server is responsible for generating the HTML for the page. The server sends a fully rendered HTML page to the browser, so the content is available immediately, without waiting for JavaScript to render it.
* **How It Works**:
  1. When the user requests a page, the server processes the request and generates the full HTML for that page, embedding data (often from a database) directly into the HTML.
  2. The server sends this fully rendered HTML to the user's browser.
  3. Once the page is loaded in the browser, JavaScript can take over to make the page interactive (Hydration).
* **Examples**: Traditional multi-page applications (MPAs) and frameworks like **Next.js** (React), **Nuxt.js** (Vue.js), and **Angular Universal** (Angular) allow SSR.
* **Advantages of SSR**:
  1. **Faster Initial Load**: The user sees the content immediately because the page is already rendered when it arrives at the browser.
  2. **Better SEO**: Since the page content is fully rendered on the server, search engines can easily index the content, improving SEO.
  3. **Reduced JavaScript Dependency**: Users with JavaScript disabled or poor network conditions can still access the content because the HTML is fully rendered.
* **Disadvantages of SSR**:
  1. **Slower Subsequent Navigation**: After the initial load, the user might experience slower navigation as each new page requires a new request to the server to generate the HTML.
  2. **Increased Server Load**: Since the server has to render the HTML for each request, it can become a bottleneck if traffic spikes, potentially leading to slower response times or performance issues.
  3. **Less Rich Interactivity**: Since the rendering happens on the server, the client is dependent on JavaScript to make the page dynamic after the initial load.

**Key Differences Between CSR and SSR:**

| **Aspect** | **Client-Side Rendering (CSR)** | **Server-Side Rendering (SSR)** |
| --- | --- | --- |
| **Where Rendering Happens** | In the browser (client-side) | On the server (server-side) |
| **Initial Load Time** | Slower initial load (because JavaScript needs to be downloaded and executed) | Faster initial load (fully rendered HTML is sent from the server) |
| **SEO** | SEO can be harder (JavaScript-heavy sites may be hard to index) | Better SEO (content is fully rendered when sent to the browser) |
| **User Experience** | Fast subsequent navigation due to client-side rendering (SPA) | Slower subsequent navigation (each new page request requires a full page reload) |
| **Server Load** | Lighter server load after the initial page load (because most rendering happens in the browser) | Higher server load because each request results in a full HTML render |
| **Interactivity** | Rich interactivity (JavaScript handles most logic and updates) | Limited interactivity until JavaScript hydrates the page |

**When to Use CSR:**

* **Single Page Applications (SPA)** where you want a highly interactive user interface and fast navigation after the initial load.
* Projects that prioritize rich, dynamic user experiences.
* When SEO is not a critical concern or when you can handle SEO issues with techniques like prerendering.

**When to Use SSR:**

* When SEO is a high priority, such as for marketing websites, blogs, e-commerce sites, etc.
* Websites where you want to provide a fast initial load and where content should be immediately available to the user.
* When you need to optimize performance for users with poor network conditions or devices.

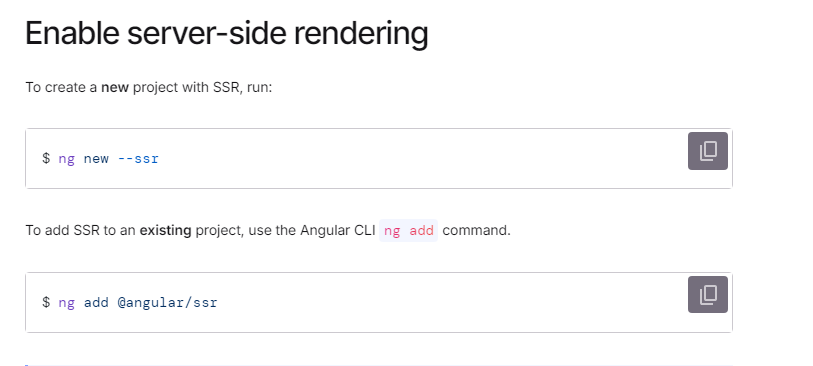
**Hybrid Approach (SSG + CSR):**

Some modern frameworks combine both CSR and SSR for a hybrid approach, often referred to as **Static Site Generation (SSG) with Hydration**. For example, **Next.js** (React) allows you to pre-render pages on the server (SSR or SSG) and then hydrate them on the client side to make them interactive (CSR).

**Conclusion:**

* **CSR** is great for interactive, dynamic applications like SPAs, where subsequent page loads are fast, and SEO isn't the primary concern.
* **SSR** is beneficial for SEO-heavy websites, where initial page load time is critical, and content must be available immediately to search engines and users.

Each approach has its advantages and trade-offs, and the choice between them depends on the specific needs of your application. Many modern frameworks allow you to combine both strategies for the best of both worlds.



[What is hydration](https://angular.dev/guide/hydration#what-is-hydration)

Hydration is the process that restores the server-side rendered application on the client. This includes things like reusing the server rendered DOM structures, persisting the application state, transferring application data that was retrieved already by the server, and other processes.

In Angular 17, the effect() function is part of the **Angular Signals** API, introduced to simplify state management and reactive programming. It allows you to create reactive computations or side effects that automatically re-run when the dependencies change. This is part of Angular’s efforts to make the framework more reactive and declarative.

Here’s a basic explanation and usage example:

**Overview of effect()**

The effect() function in Angular allows you to perform an action (like updating the UI, making API calls, etc.) whenever specific reactive state changes. It automatically tracks the dependencies of the state variables within the effect and re-executes when any of them change.

**Basic Usage Example:**

typescript

Copy

import { effect, signal } from '@angular/core';

// Define signals for reactive state

const count = signal(0);

// Define an effect

effect(() => {

// This effect will run whenever the 'count' signal changes

console.log('The count value is: ', count());

});

// Modify the signal's value

count.set(1); // This will trigger the effect and log the new count value

**Key Points:**

* signal() is used to define reactive state, similar to useState in React.
* effect() is used to execute side effects (e.g., logging, DOM updates, API calls) whenever a dependent signal changes.
* effect() automatically tracks dependencies, so you don’t need to manually manage which signals are involved.

**Advantages of effect():**

1. **Automatic Dependency Tracking**: The effect automatically tracks the signals you access within it and re-runs when they change.
2. **Declarative**: You define the logic in a clean, declarative way instead of manually managing change detection or subscriptions.
3. **Simplified Reactive Programming**: It helps with simpler and more intuitive management of side effects and reactivity.

**Advanced Example with Dependencies:**

typescript

Copy

import { effect, signal } from '@angular/core';

// Signals for reactive state

const price = signal(100);

const taxRate = signal(0.2);

const totalPrice = signal(() => price() \* (1 + taxRate()));

// Define an effect to log the total price whenever price or taxRate changes

effect(() => {

console.log('The total price is: ', totalPrice());

});

// Modify state values

price.set(120); // This will trigger the effect and log the new total price

taxRate.set(0.25); // This will also trigger the effect again

In this case:

* The totalPrice is a derived signal based on price and taxRate.
* The effect() tracks these signals, and whenever either price or taxRate changes, it recalculates the totalPrice and logs it.

**Conclusion:**

effect() in Angular 17 makes it easier to handle reactivity and side effects. It integrates naturally into Angular’s reactive paradigm and simplifies state management without the need for complex manual tracking. It’s a powerful tool for building dynamic, reactive applications in Angular.

On success navigate to

Router: Angular Router

A screen shot of a computer code

AI-generated content may be incorrect.

In Angular 17, HTTP interceptors are powerful tools used to handle HTTP requests and responses globally in an Angular application. They are especially useful for tasks like adding authentication tokens, logging, or modifying the request/response before they are sent or received.

**Real-World Example: Handling Authentication with HTTP Interceptor**

Let’s walk through an example of how to use an HTTP interceptor to add an authentication token to outgoing HTTP requests and handle responses globally.

**Steps:**

1. **Create an HTTP Interceptor Service**: This service will be responsible for intercepting HTTP requests and responses.
2. **Provide the Interceptor in Angular Module**: We need to register the interceptor in the AppModule so that it’s available globally.
3. **Use the Interceptor to Add Authentication Token**: In this example, we will add a JWT (JSON Web Token) to the Authorization header for every request.
4. **Handle Errors or Other Modifications**: Optionally, we can modify responses globally, like handling unauthorized errors.

**1. Creating the HTTP Interceptor**

First, let's create an HTTP Interceptor.

typescript

Copy

// auth-interceptor.service.ts

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

import { HttpInterceptor, HttpRequest, HttpHandler, HttpEvent } from '@angular/common/http';

import { Observable } from 'rxjs';

import { catchError, tap } from 'rxjs/operators';

import { AuthService } from './auth.service'; // Assuming an AuthService is present to handle authentication

@Injectable()

export class AuthInterceptorService implements HttpInterceptor {

constructor(private authService: AuthService) {}

intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

// Get the authentication token from the AuthService

const token = this.authService.getToken(); // AuthService should return JWT token

// Clone the request and add the Authorization header with the token

let clonedRequest = req;

if (token) {

clonedRequest = req.clone({

setHeaders: {

Authorization: `Bearer ${token}`,

}

});

}

// Optionally handle responses globally

return next.handle(clonedRequest).pipe(

tap(event => {

// You can add any logging or processing here if needed

console.log('HTTP request success:', event);

}),

catchError((error) => {

// Global error handling (e.g., for unauthorized error)

if (error.status === 401) {

console.error('Unauthorized, please login again');

// Optionally redirect to login page

}

throw error; // Rethrow the error after handling

})

);

}

}

**2. Register the Interceptor in the App Module**

Now, register the interceptor in the AppModule by adding it to the HTTP\_INTERCEPTORS array.

typescript

Copy

// app.module.ts

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

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

import { HttpClientModule, HTTP\_INTERCEPTORS } from '@angular/common/http';

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

import { AuthInterceptorService } from './auth-interceptor.service';

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

HttpClientModule

],

providers: [

{

provide: HTTP\_INTERCEPTORS,

useClass: AuthInterceptorService,

multi: true

}

],

bootstrap: [AppComponent]

})

export class AppModule { }

**3. Creating an AuthService (Assuming JWT Token Management)**

For the above example, we need an AuthService to manage the authentication token. Here’s a simple version of that service:

typescript

Copy

// auth.service.ts

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

@Injectable({

providedIn: 'root'

})

export class AuthService {

// For simplicity, storing the token in localStorage (you can use any method you prefer)

private tokenKey = 'auth\_token';

constructor() { }

// Method to get the JWT token

getToken(): string | null {

return localStorage.getItem(this.tokenKey);

}

// Method to set the JWT token (e.g., after login)

setToken(token: string): void {

localStorage.setItem(this.tokenKey, token);

}

// Method to clear the JWT token (e.g., after logout)

clearToken(): void {

localStorage.removeItem(this.tokenKey);

}

}

**4. Handling Errors in the Interceptor**

In the AuthInterceptorService, we handle errors using the catchError operator. In this case, if the server responds with a 401 Unauthorized status code, we can show an error message or redirect the user to the login page.

typescript

Copy

catchError((error) => {

if (error.status === 401) {

console.error('Unauthorized, please login again');

// Optionally redirect to login page or handle re-authentication

}

throw error; // Rethrow the error after handling

})

**5. Test the Interceptor**

Now, whenever an HTTP request is made (e.g., a GET or POST request), the AuthInterceptorService will automatically append the Authorization header with the JWT token (if available).

For example:

typescript

Copy

// some.component.ts

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

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

@Component({

selector: 'app-some',

templateUrl: './some.component.html',

styleUrls: ['./some.component.css']

})

export class SomeComponent {

constructor(private http: HttpClient) {}

makeApiCall() {

this.http.get('https://api.example.com/protected-endpoint').subscribe(

(response) => {

console.log('Response received:', response);

},

(error) => {

console.error('Error occurred:', error);

}

);

}

}

**Summary**

* **HTTP Interceptors** allow you to modify requests and responses globally.
* In this example, we used an **AuthInterceptor** to add a JWT token to outgoing requests and handled error responses globally (like 401 Unauthorized).
* Interceptors can also be used to log HTTP traffic, handle retries, or modify response data.

This pattern is quite useful in real-world applications where authorization and global error handling need to be standardized across all API requests.