What is zone.js?

• zone.js is a library that creates an execution context (called a "zone") to keep track of asynchronous operations (like setTimeout, HTTP requests, events, etc.).

• It allows Angular to know when to update the UI automatically, without you having to manually trigger change detection.

Why is it imported in polyfills.ts?

• The line import 'zone.js'; in your polyfills.ts ensures that zone.js is loaded before your Angular app starts.

• This is required for Angular’s change detection to work properly.

• Without zone.js, Angular would not know when to update the view after async operations.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

How does Angular use zone.js?

• Whenever you perform an async operation (like an HTTP call, timer, or user event), zone.js notifies Angular.

• Angular then runs change detection to update the UI with any new data or changes.

Do you need to use zone.js directly?

• In most cases, no. Angular handles everything for you.

• You only need to interact with it directly if you want to optimize performance (e.g., run code outside Angular’s zone to avoid unnecessary change detection) using Angular’s NgZone service.

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

constructor(private ngZone: NgZone) {}

doSomething() {

// This will trigger Angular change detection

this.ngZone.run(() => {

// your code here

});

// This will NOT trigger Angular change detection

this.ngZone.runOutsideAngular(() => {

// your code here

});

}

What is a Decorator in Angular?

A decorator is a special function in TypeScript (and Angular) that adds extra features or metadata to classes, methods, properties, or parameters. In Angular, decorators tell Angular how to treat a class or property.

Common Angular Decorators with Examples

1. @Component

Marks a class as an Angular component and provides its metadata (selector, template, etc.).

@Component

Turns a class into an Angular component.

Defines the component’s HTML, CSS, and selector.

Used to build UI parts of your app.

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

@Component({

selector: 'app-hello',

template: '<h1>Hello, Angular!</h1>'

})

export class HelloComponent {}

2. @NgModule

Defines an Angular module, which groups components, directives, and services.

Groups related components, directives, and services.

Helps organize and structure your Angular app.

Declares what parts belong to the module and what it uses.

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

@NgModule({

declarations: [HelloComponent],

imports: [],

bootstrap: [HelloComponent]

})

export class AppModule {}

3. @Injectable

Marks a class as available for dependency injection (for services).

Makes a class available for dependency injection.

Used mainly for services.

Allows Angular to create and share service instances.

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

@Injectable({

providedIn: 'root'

})

export class DataService {}

4. @Input and @Output

Used in components to pass data in and out.

@Input

Passes data from parent to child component.

Used as a property decorator in child components.

Enables dynamic and reusable components.

@Output

Sends events from child to parent component.

Uses EventEmitter to notify the parent.

Enables communication and interaction between components.

Scenario:

You have a parent component that displays a user’s name and a child component with a button to update the name.

1. Parent to Child (@Input)

Parent Component (parent.component.ts):

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

@Component({

selector: 'app-parent',

template: `

<h2>Parent Component</h2>

<app-child [userName]="name"></app-child>

`

})

export class ParentComponent {

name = 'Alice';

}

Child Component (child.component.ts):

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

@Component({

selector: 'app-child',

template: `

<h3>Child Component</h3>

<p>User Name from Parent: {{ userName }}</p>

`

})

export class ChildComponent {

@Input() userName: string;

}

2. Child to Parent (@Output)

Parent Component (parent.component.ts):

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

@Component({

selector: 'app-parent',

template: `

<h2>Parent Component</h2>

<p>Current Name: {{ name }}</p>

<app-child [userName]="name" (nameChanged)="onNameChanged($event)"></app-child>

`

})

export class ParentComponent {

name = 'Alice';

onNameChanged(newName: string) {

this.name = newName;

}

}

Child Component (child.component.ts):

import { Component, Input, Output, EventEmitter } from '@angular/core';

@Component({

selector: 'app-child',

template: `

<h3>Child Component</h3>

<p>User Name from Parent: {{ userName }}</p>

<button (click)="changeName()">Change Name to Bob</button>

`

})

export class ChildComponent {

@Input() userName: string;

@Output() nameChanged = new EventEmitter<string>();

changeName() {

this.nameChanged.emit('Bob');

}

}

How it works:

The parent passes the name to the child using [userName]="name".

The child displays the name and has a button to change it.

When the button is clicked, the child emits a new name to the parent using (nameChanged)="onNameChanged($event)".

The parent updates its name property, and the change is reflected in both components.

import { Component, Input, Output, EventEmitter } from '@angular/core';

@Component({

selector: 'app-child',

template: '<button (click)="notifyParent()">Click me</button>'

})

export class ChildComponent {

@Input() childData: string;

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

notifyParent() {

this.childClicked.emit();

}

}

5. @HostListener and @HostBinding

Used in directives to listen to events or bind properties on the host element.

@HostListener

Listens for events on the host element.

Runs a method when the event occurs.

Useful for handling user actions like clicks or mouse movements.

@HostBinding

Binds a property or attribute to the host element.

Dynamically updates the host element’s properties.

Useful for changing classes, styles, or attributes from code.

import { Directive, HostListener, HostBinding } from '@angular/core';

@Directive({

selector: '[appHighlight]'

})

export class HighlightDirective {

@HostBinding('style.backgroundColor') bgColor: string;

@HostListener('mouseenter') onMouseEnter() {

this.bgColor = 'yellow';

}

@HostListener('mouseleave') onMouseLeave() {

this.bgColor = null;

}

}

What are Angular Lifecycle Hooks?

Lifecycle hooks are special methods in Angular that let you run code at specific moments in a component’s life (creation, update, destruction).

**Key Differences**

|  |  |  |
| --- | --- | --- |
| **Hook** | **When it Runs** | **Common Use Case** |
| ngOnInit | After component is initialized | Fetch data, setup logic |
| ngOnChanges | When input properties change | React to input changes |
| ngDoCheck | During every change detection run | Custom change detection |
| ngAfterContentInit | After content (ng-content) is projected | Interact with projected content |
| ngAfterContentChecked | After every check of projected content | Respond to content changes |
| ngAfterViewInit | After component’s view is initialized | Access view/child elements |
| ngAfterViewChecked | After every check of the component’s view | Respond to view changes |
| ngOnDestroy | Just before component is destroyed | Cleanup, unsubscribe |

Code :

parent.component.html

<h2>Parent Component</h2>

<input [(ngModel)]="userId" placeholder="Enter User ID" />

<app-child [userId]="userId">

<span #projectedContent>Projected content from parent!</span>

</app-child>

child.component.ts

import {

Component, Input, Output, EventEmitter, SimpleChanges,

OnInit, OnChanges, DoCheck, AfterContentInit, AfterContentChecked,

AfterViewInit, AfterViewChecked, OnDestroy, ContentChild, ViewChild, ElementRef

} from '@angular/core';

@Component({

selector: 'app-child',

template: `

<div>

<h3>Child Component</h3>

<p>User ID from parent: {{ userId }}</p>

<ng-content></ng-content>

<input #inputRef type="text" placeholder="Focus me after view init" />

</div>

`

})

export class ChildComponent implements

OnInit, OnChanges, DoCheck, AfterContentInit, AfterContentChecked,

AfterViewInit, AfterViewChecked, OnDestroy {

@Input() userId: string;

@ContentChild('projectedContent') projected: ElementRef;

@ViewChild('inputRef') input: ElementRef;

timer: any;

constructor() {

console.log('Constructor: ChildComponent created');

}

ngOnChanges(changes: SimpleChanges) {

console.log('ngOnChanges:', changes);

}

ngOnInit() {

console.log('ngOnInit: Initialization logic here');

this.timer = setInterval(() => {

console.log('Timer running in child...');

}, 2000);

}

ngDoCheck() {

console.log('ngDoCheck: Custom change detection');

}

ngAfterContentInit() {

console.log('ngAfterContentInit: Projected content initialized:', this.projected?.nativeElement.textContent);

}

ngAfterContentChecked() {

console.log('ngAfterContentChecked: Projected content checked');

}

ngAfterViewInit() {

console.log('ngAfterViewInit: View initialized');

this.input.nativeElement.focus();

}

ngAfterViewChecked() {

console.log('ngAfterViewChecked: View checked');

}

ngOnDestroy() {

clearInterval(this.timer);

console.log('ngOnDestroy: Cleanup done, timer cleared');

}

}

Imports

Component, Input, Output, EventEmitter: For creating a component and handling data/event communication.

SimpleChanges: Used in ngOnChanges to detect changes in input properties.

Lifecycle interfaces: (OnInit, OnChanges, etc.) Let you use Angular lifecycle hooks.

ContentChild, ViewChild, ElementRef: For accessing projected content and view elements.

@Component Decorator

Defines the component’s selector (app-child) and its template (HTML).

Class Properties

@Input() userId: Receives a value from the parent component.

@ContentChild('projectedContent') projected: Gets a reference to content projected from the parent (using <ng-content>).

@ViewChild('inputRef') input: Gets a reference to the input element in the component’s own template.

timer: Stores a timer ID for later cleanup.

Lifecycle Hooks

constructor(): Runs when the component is created. Logs a message.

ngOnChanges(): Runs whenever an input property (like userId) changes. Logs the changes.

ngOnInit(): Runs once after the component is initialized. Starts a timer that logs every 2 seconds.

ngDoCheck(): Runs during every change detection cycle. Logs a message.

ngAfterContentInit(): Runs once after projected content is initialized. Logs the projected content’s text.

ngAfterContentChecked(): Runs after every check of projected content. Logs a message.

ngAfterViewInit(): Runs once after the component’s view is initialized. Focuses the input element.

ngAfterViewChecked(): Runs after every check of the component’s view. Logs a message.

ngOnDestroy(): Runs just before the component is destroyed. Clears the timer and logs a message

What is Rxjs : Key points: RxJS is a library for working with async data streams in JavaScript.

RxJS provides the Observable type, which represents a stream of data that can be observed over time.

It includes operators (like map, filter, merge, switchMap) to transform, combine, and manage streams.

RxJS is widely used in Angular for handling async operations, such as HTTP requests and event handling.

RxJS (Reactive Extensions for JavaScript) is a library for reactive programming using observables. It allows you to work with asynchronous data streams (such as events, HTTP requests, or user input) in a powerful, flexible way.

RxJS provides the Observable type, which represents a stream of data that can be observed over time.

It includes operators (like map, filter, merge, switchMap) to transform, combine, and manage streams.

RxJS is widely used in Angular for handling async operations, such as HTTP requests and event handling.

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

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

import { Observable } from 'rxjs';

@Component({

selector: 'app-users',

template: `

<h3>User List</h3>

<ul>

<li \*ngFor="let user of users">{{ user.name }}</li>

</ul>

`

})

export class UsersComponent implements OnInit {

users: any[] = [];

constructor(private http: HttpClient) {}

ngOnInit() {

// This returns an Observable of user data

this.http.get<any[]>('https://jsonplaceholder.typicode.com/users')

.subscribe(data => {

this.users = data; // Assign the data to the users array

});

}

}

RxJS (Reactive Extensions for JavaScript) is a library for composing asynchronous and event-based programs using observable sequences and operators. It provides a powerful way to work with streams of data—such as user input, web requests, or real-time events—by allowing you to create, transform, and combine these streams in a declarative and reactive manner. RxJS is widely used in Angular for handling asynchronous operations and managing complex data flows.

React to data as it arrives over time" means you can handle and process new pieces of data (like user actions, messages, or server responses) whenever they happen, instead of waiting for all data at once. RxJS helps you do this easily by using streams and observables.

**Signals ?**

Signals in Angular are a new way to manage and react to state changes in your application. They provide a simple, reactive variable that automatically updates the UI or triggers logic whenever its value changes.

Key points:

Signals are like special variables that notify Angular when their value changes.

They make UI updates automatic and more efficient—no need for manual subscriptions or change detection.

Signals are best for local component state and simple reactivity, not for complex async streams.

Local component state means the data or variables that belong only to a specific component and are used to control its behavior or display. This state is not shared with other components.

Examples:

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

@Component({

selector: 'app-modal',

template: `

<button (click)="open()">Open Modal</button>

<div \*ngIf="isOpen()" class="modal">

<p>This is a modal dialog!</p>

<button (click)="close()">Close</button>

</div>

`,

styles: [`.modal { background: #eee; padding: 20px; border: 1px solid #ccc; }`]

})

export class ModalComponent {

isOpen = signal(false); // Signal for modal open/close state

open() {

this.isOpen.set(true);

}

close() {

this.isOpen.set(false);

}

}

Signals

For local/component state (small, simple, UI-focused).

Minimal setup, easy to use.

Automatic UI updates when state changes.

Best for: toggles, counters, form fields, and state that doesn’t need to be shared widely.

NgRx

For global/app-wide state (complex, shared, or persistent data).

Requires setup: actions, reducers, selectors, effects.

Powerful for large apps with many components needing the same data.

Best for: authentication, user profiles, shopping carts, and data shared across many parts of the app.

Managing global application state means keeping track of data that needs to be shared and accessed by multiple components across your entire app. This includes things like:

User authentication status (logged in or not)

User profile information

Shopping cart contents

Theme or language settings

Instead of each component having its own copy, the global state is stored in one place, and any component can read or update it. This ensures consistency and makes it easier to manage complex data flows in large applications.