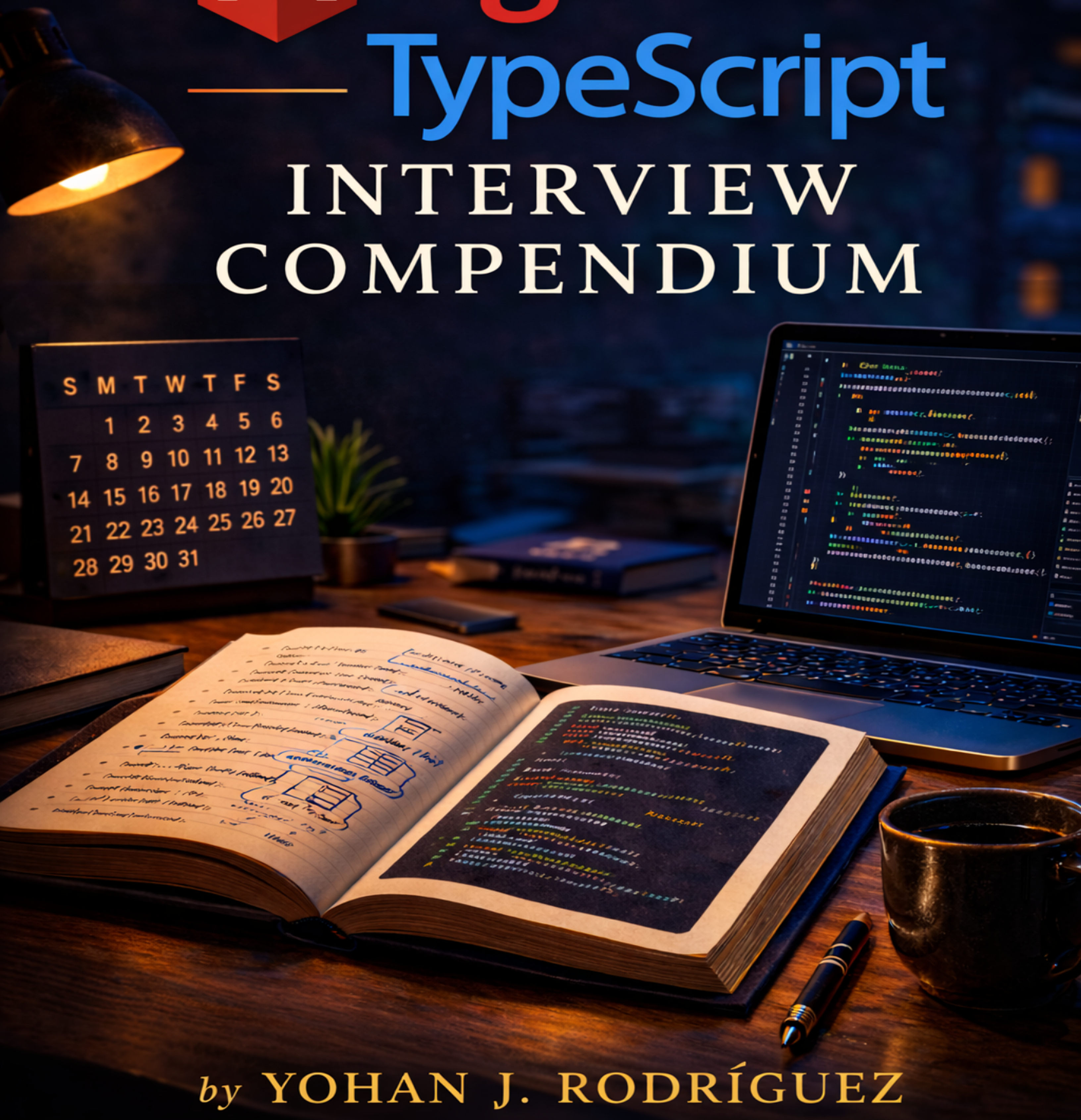
The Angular and TypeScript INTERVIEW COMPENDIUM



by YOHAN J. RODRÍGUEZ

Preface

"Preparation beats memorization in technical interviews."

The Angular and TypeScript Interview Compendium is a practical guide for developers who want to build strong interview readiness with modern Angular and advanced TypeScript. It is designed for real interview use: concise explanations, architecture tradeoff discussions, and code examples that mirror production-style patterns.

This book emphasizes what interviewers actually test: framework fundamentals, component and state architecture, deep type-system reasoning, reactivity decisions, API integration, testing strategy, performance tuning, and security reasoning. Rather than focusing on isolated trivia, each chapter connects technical decisions to maintainability, scalability, and team execution.

You will find theory questions, scenario questions, debugging prompts, and comparison questions across the chapters. The intended outcome is not just familiarity with Angular syntax, but the ability to explain why one approach is better than another under concrete constraints and type-safety requirements.

The Angular and TypeScript Interview Compendium Team

Contents

1 | Components, Templates, and Standalone Architecture

Components are the center of Angular application design and one of the highest-frequency interview topics. In senior interviews, correctness alone is not enough. You need to show design discipline: clear APIs, low coupling, testability, and predictable rendering behavior.

1.1 Component Design Goals

High-quality components optimize for:

- explicit input and output contracts,
- minimal local state surface,
- easy unit and integration testing,
- composability without hidden side effects.

1.2 Component Fundamentals

What is a component in Angular and what are its core parts?

Every Angular component is built from three core parts: the **decorator metadata**, the **template**, and the **class**. The `@Component` decorator provides metadata such as the selector, template, styles, change detection strategy, and standalone flag. The template defines the view using HTML augmented with Angular's binding syntax and control flow. The class contains the component's state, input/output declarations, lifecycle hooks, and methods that drive behavior.

A well-structured component keeps its template declarative, its class focused on a single responsibility, and its metadata explicit about dependencies. Interviewers look for candidates who can articulate this separation clearly and explain how each part contributes to testability and maintainability.

Listing 1.1: Component anatomy: metadata, template, and class

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

/**
 * Every Angular component has three core parts:
 * 1. The @Component decorator (metadata)
 * 2. The template (view)
 * 3. The class (behavior and state)
 */
@Component({
  // --- Metadata ---
  selector: 'app-notification-banner',
  standalone: true,
  changeDetection: ChangeDetectionStrategy.OnPush,

  // --- Template (view) ---
  template: `
    <div
      class="banner"
      [class.banner--warning]="severity === 'warning'"
      [class.banner--error]="severity === 'error'"
      role="alert"
    >
      <span class="banner__message">{{ message }}</span>
      <button type="button" (click)="dismissed.emit()">Dismiss</button>
    </div>
  `,

  // --- Styles (scoped to this component) ---
  styles: [
    `
      .banner {
        padding: 0.75rem 1rem;
        border-radius: 4px;
      }
      .banner--warning {
        background: #fff3cd;
      }
      .banner--error {
        background: #f8d7da;
      }
    `,
  ],
})
export class NotificationBannerComponent {
  // --- Class (behavior and state) ---

  /** The text shown to the user. */
  @Input({ required: true }) message!: string;

  /** Controls visual styling. Defaults to informational. */
  @Input() severity: 'info' | 'warning' | 'error' = 'info';

  /** Emitted when the user clicks dismiss. */
  @Output() dismissed = new EventEmitter<void>();
}
```

What is the difference between standalone and NgModule-declared components?

Before Angular 14, every component had to be declared inside an `NgModule`. The module was responsible for importing dependencies like `CommonModule`, declaring the component, and exporting it for use elsewhere. Standalone components, introduced in Angular 14 and promoted as the default from Angular 17 onward, move dependency declarations into the component itself via the `imports` array in the decorator.

The practical consequences are significant. Standalone components are self-describing: you can read a single file and know every dependency. They eliminate the indirection of shared modules where a component declared in `SharedModule` silently depends on dozens of unrelated imports. They also simplify lazy loading because you can lazy-load a single component without creating a wrapper module.

NgModule-declared components remain valid and are common in legacy codebases. Interviewers expect you to work with both and to explain the tradeoffs of each approach.

Listing 1.2: Standalone vs NgModule-declared component comparison

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

// ---------------------------------------------
// Standalone component (modern approach, Angular 14+)
// Dependencies are declared directly in the component.
// ---------------------------------------------
@Component({
  selector: 'app-status-badge',
  standalone: true,
  imports: [CommonModule],
  template: `
    <span class="badge" [ngClass]="status">{{ label }}</span>
  `,
})
export class StatusBadgeComponent {
  status: 'active' | 'inactive' = 'active';
  get label(): string {
    return this.status === 'active' ? 'Online' : 'Offline';
  }
}

// ---------------------------------------------
// NgModule-declared component (legacy approach)
// The component itself does NOT declare its imports;
// its NgModule must provide everything.
// ---------------------------------------------
@Component({
  selector: 'app-status-badge-legacy',
  template: `
    <span class="badge" [ngClass]="status">{{ label }}</span>
  `,
})
export class StatusBadgeLegacyComponent {
  status: 'active' | 'inactive' = 'active';
  get label(): string {
    return this.status === 'active' ? 'Online' : 'Offline';
  }
}

@NgModule({
  declarations: [StatusBadgeLegacyComponent],
  imports: [CommonModule],
  exports: [StatusBadgeLegacyComponent],
})
export class StatusBadgeLegacyModule {}
```

How does Angular's component lifecycle work?

Angular manages a component's lifecycle through a series of hooks that fire in a defined order. The most commonly used hooks are:

1. **ngOnChanges** — called before `ngOnInit` and whenever an `@Input` property changes. Receives a `SimpleChanges` object with previous and current values.

2. **ngOnInit** — called once after the first `ngOnChanges`. The standard place for initialization logic that depends on input values.

3. **ngDoCheck** — called during every change detection run. Used for custom change detection logic.

4. **ngAfterContentInit** — called once after Angular projects external content into the component via `ng-content`.

5. **ngAfterContentChecked** — called after every check of projected content.

6. **ngAfterViewInit** — called once after the component's view and child views are initialized.

7. **ngAfterViewChecked** — called after every check of the component's view.

8. **ngOnDestroy** — called once before Angular destroys the component. The place to clean up subscriptions, timers, and event listeners.

In interviews, the most important hooks to discuss in depth are `ngOnInit`, `ngOnChanges`, `ngAfterViewInit`, and `ngOnDestroy`. Candidates should demonstrate understanding of the hook order and practical cleanup patterns.

Listing 1.3: Lifecycle hooks in action: countdown timer component

```
import {
  AfterViewInit,
  Component,
  ElementRef,
  Input,
  OnChanges,
  OnDestroy,
  OnInit,
  SimpleChanges,
  ViewChild,
} from '@angular/core';
import { Subscription, interval } from 'rxjs';

@Component({
  selector: 'app-countdown-timer',
  standalone: true,
  template: `
    <p #display>{{ remainingSeconds }}s remaining</p>
  `,
})
export class CountdownTimerComponent
  implements OnChanges, OnInit, AfterViewInit, OnDestroy
{
  @Input({ required: true }) durationSeconds!: number;
  @ViewChild('display') displayEl!: ElementRef<HTMLParagraphElement>;

  remainingSeconds = 0;
  private tickSub?: Subscription;

  // 1. ngOnChanges - called BEFORE ngOnInit and whenever
  //    an @Input property value changes.
  ngOnChanges(changes: SimpleChanges): void {
    if (changes['durationSeconds']) {
      this.remainingSeconds = this.durationSeconds;
      this.restartTimer();
    }
  }

  // 2. ngOnInit - called once after the first ngOnChanges.
  //    Use it for initialization logic that depends on inputs.
  ngOnInit(): void {
    console.log('Component initialized with', this.durationSeconds, 'seconds');
  }

  // 3. ngAfterViewInit - called once after the component's
  //    view (and child views) have been initialized.
  ngAfterViewInit(): void {
    this.displayEl.nativeElement.setAttribute('aria-live', 'polite');
  }

  // 4. ngOnDestroy - called once right before Angular destroys
  //    the component. Clean up subscriptions and listeners here.
  ngOnDestroy(): void {
    this.tickSub?.unsubscribe();
  }

  private restartTimer(): void {
    this.tickSub?.unsubscribe();
    this.tickSub = interval(1000).subscribe(() => {
      if (this.remainingSeconds > 0) {
        this.remainingSeconds--;
      }
    });
  }
}
```

What is the difference between ngOnInit and the constructor?

The constructor is a standard TypeScript class feature that runs when the class is instantiated. At construction time, Angular has *not yet* set `@Input()` bindings, so any logic that depends on input values will see `undefined`. The constructor should be used exclusively for dependency injection and trivial field initialization.

`ngOnInit` is an Angular lifecycle hook that fires after Angular has resolved all input bindings. This is the correct place for initialization logic: loading data based on an input ID, setting up subscriptions, or computing derived state from inputs.

This distinction is one of the most frequently asked interview questions for Angular positions at all levels. A strong answer explains *why* the timing difference matters, not just *that* it exists.

Listing 1.4: Constructor vs ngOnInit: timing and appropriate usage

```
import { Component, inject, Input, OnInit } from '@angular/core';
import { LoggerService } from './logger.service';

@Component({
  selector: 'app-order-summary',
  standalone: true,
  template: `<p>Order #{{ orderId }} - {{ statusLabel }}</p>`,
})
export class OrderSummaryComponent implements OnInit {
  /**
   * The constructor runs when the class is instantiated.
   * At this point, @Input() values have NOT been set yet.
   * Use it only for dependency injection and trivial field init.
   */
  private readonly logger = inject(LoggerService);

  @Input({ required: true }) orderId!: string;

  statusLabel = '';

  /**
   * ngOnInit runs after Angular has set all @Input() bindings.
   * This is the correct place for initialization logic that
   * depends on input values, service calls, or subscriptions.
   */
  ngOnInit(): void {
    // orderId is guaranteed to be available here
    this.logger.info(`Loading order ${this.orderId}`);
    this.statusLabel = this.resolveStatus(this.orderId);
  }

  private resolveStatus(id: string): string {
    // Simplified - in practice this would call a service
    return id.startsWith('RUSH') ? 'Priority' : 'Standard';
  }
}

// -- Why this matters in interviews --
// Interviewers test whether you understand that:
// 1. The constructor is a TypeScript/ES class concept - Angular
//    has not wired up bindings yet.
// 2. ngOnInit is an Angular lifecycle hook - inputs are resolved
//    and the component is ready for initialization logic.
// 3. Putting HTTP calls or complex logic in the constructor is
//    an anti-pattern because inputs are undefined at that point.
```

What is ChangeDetectionStrategy.OnPush and when should you use it?

By default, Angular uses `ChangeDetectionStrategy.Default`, which checks every component in the tree on every change detection cycle. This is convenient but becomes expensive as the component tree grows.

`ChangeDetectionStrategy.OnPush` tells Angular to skip checking this component unless one of the following occurs:

- an `@Input()` reference changes (identity check, not deep equality),
- a DOM event originates from this component or its descendants,
- an `Observable` bound with the `async` pipe emits a new value,
- `ChangeDetectorRef.markForCheck()` is called manually.

OnPush is recommended for all presentational components and is increasingly the default for all new components in modern Angular projects. It enforces immutable data patterns: parents must pass new object references instead of mutating existing ones. When combined with Angular Signals (16+), OnPush becomes nearly automatic since signals notify the framework of changes without manual intervention.

Listing 1.5: Default vs OnPush change detection strategies

```
import {
  ChangeDetectionStrategy,
  ChangeDetectorRef,
  Component,
  inject,
  Input,
  OnInit,
} from '@angular/core';

// -- Default change detection --
// Angular checks this component on EVERY change detection cycle,
// even when none of its data changed. Fine for small apps,
// costly at scale.
@Component({
  selector: 'app-activity-log-default',
  standalone: true,
  template: `
    <ul>
      @for (entry of entries; track entry.id) {
        <li>{{ entry.message }}</li>
      }
    </ul>
  `,
  // changeDetection: ChangeDetectionStrategy.Default  (implicit)
})
export class ActivityLogDefaultComponent {
  @Input() entries: { id: number; message: string }[] = [];
}

// -- OnPush change detection --
// Angular only checks this component when:
// 1. An @Input reference changes (not mutation),
// 2. An event originates from this component or its children,
// 3. An Observable bound with `async` emits,
// 4. You manually call ChangeDetectorRef.markForCheck().
@Component({
  selector: 'app-activity-log-onpush',
  standalone: true,
  template: `
    <ul>
      @for (entry of entries; track entry.id) {
        <li>{{ entry.message }}</li>
      }
    </ul>
  `,
  changeDetection: ChangeDetectionStrategy.OnPush,
})
export class ActivityLogOnPushComponent implements OnInit {
  @Input() entries: { id: number; message: string }[] = [];

  private readonly cdr = inject(ChangeDetectorRef);

  ngOnInit(): void {
    // Example: if data arrives from a WebSocket outside NgZone,
    // you must notify Angular manually.
    // this.cdr.markForCheck();
  }
}

// -- Key interview points --
// * OnPush is recommended for all presentational components.
// * It enforces immutable data flow patterns - parent must
//   pass a NEW array/object reference, not mutate the old one.
// * Combining OnPush with Signals (Angular 16+) eliminates
//   the need for markForCheck in most cases.
```

How does OnPush change detection interact with Angular Signals?

Angular Signals provide a fine-grained reactivity system that complements `ChangeDetectionStrategy.OnPush`. When a component using `OnPush` reads a signal in its template, Angular automatically marks that component (and its ancestors) for check whenever the signal value changes.

This eliminates the need for manual `ChangeDetectorRef.markForCheck()` calls in most scenarios. Unlike RxJS Observables, which require the `async` pipe or manual subscription management to trigger change detection in `OnPush` components, signals notify the framework directly. This makes `OnPush` safer and easier to adopt as the default strategy.

Listing 1.6: OnPush component using Signals for automatic change detection

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

@Component({
  selector: 'app-signal-counter',
  standalone: true,
  template: `
    <div>
      <p>Count: {{ count() }}</p>
      <p>Doubled: {{ doubled() }}</p>
      <button (click)="increment()">Increment</button>
    </div>
  `,
  // With OnPush, Angular only re-checks this template when:
  // 1. The 'count' signal (which is read in the template) changes.
  // 2. A DOM event (like the button click) occurs.
  changeDetection: ChangeDetectionStrategy.OnPush,
})
export class SignalCounterComponent {
  count = signal(0);
  doubled = computed(() => this.count() * 2);

  increment() {
    // Updating the signal automatically marks this component
    // for check, even though we are in OnPush mode.
    this.count.update(c => c + 1);
  }
}

/**
 * Key Interview Insight:
 * Prior to Signals, using OnPush with an internal property update
 * (e.g., this.count++) would NOT trigger a UI update unless
 * the update happened inside a Zone.js event or we called
 * cdr.markForCheck(). Signals make OnPush "just work" for
 * internal state changes.
 */
```

1.3 Template Concepts and Querying

How do you use @ViewChild and @ContentChild?

`@ViewChild` queries an element, directive, or component that exists in the component's *own* template. It is available after `ngAfterViewInit`. Common uses include accessing a native DOM element for focus management, reading a child component's public API, or obtaining a `ViewContainerRef` for dynamic component creation.

`@ContentChild` queries an element, directive, or component that was *projected* into the component through `<ng-content>`. It is available after `ngAfterContentInit`. This is useful for components that accept customizable templates or slots from their consumers.

The key distinction is ownership: `@ViewChild` accesses what the component *owns*, while `@ContentChild` accesses what a *parent* projected into it. Mixing these up in interviews is a common mistake.

Listing 1.7: @ViewChild and @ContentChild: querying owned vs projected content

```
import {
  AfterContentInit,
  AfterViewInit,
  Component,
  ContentChild,
  Directive,
  ElementRef,
  TemplateRef,
  ViewChild,
} from '@angular/core';

// -- @ViewChild: queries an element/component in the
//    component's OWN template --

@Component({
  selector: 'app-search-bar',
  standalone: true,
  template: `
    <input #searchInput type="text" placeholder="Search..." />
    <button (click)="focusInput()">Focus</button>
  `,
})
export class SearchBarComponent implements AfterViewInit {
  @ViewChild('searchInput')
  inputRef!: ElementRef<HTMLInputElement>;

  ngAfterViewInit(): void {
    // The queried element is available after the view initializes
    console.log('Input element:', this.inputRef.nativeElement.tagName);
  }

  focusInput(): void {
    this.inputRef.nativeElement.focus();
  }
}

```

```
// -- @ContentChild: queries projected content supplied by
//    a PARENT component through <ng-content> --

@Directive({
  selector: '[appPanelHeader]',
  standalone: true,
})
export class PanelHeaderDirective {
  constructor(public readonly templateRef: TemplateRef<unknown>) {}
}

@Component({
  selector: 'app-expandable-panel',
  standalone: true,
  template: `
    <div class="panel">
      <header (click)="toggle()">
        <ng-container
          *ngTemplateOutlet="headerTemplate?.templateRef"
        ></ng-container>
      </header>
      @if (expanded) {
        <div class="panel__body">
          <ng-content></ng-content>
        </div>
      }
    </div>
  `,
})
export class ExpandablePanelComponent implements AfterContentInit {
  // Queries the directive applied to projected content
  @ContentChild(PanelHeaderDirective)
  headerTemplate?: PanelHeaderDirective;

  expanded = false;

  ngAfterContentInit(): void {
    if (!this.headerTemplate) {
      console.warn('ExpandablePanel: no *appPanelHeader provided');
    }
  }

  toggle(): void {
    this.expanded = !this.expanded;
  }
}

// Usage in a parent template:
// <app-expandable-panel>
//   <ng-template appPanelHeader>
//     <h3>Click to expand</h3>
//   </ng-template>
//   <p>This is the projected body content.</p>
// </app-expandable-panel>
```

What is content projection and how does ng-content work?

Content projection allows a component to accept markup from its parent and render it inside its own template using `<ng-content>`. This is Angular's equivalent of the "slot" concept in Web Components.

Angular supports three forms of content projection:

- **Single-slot projection:** a single `<ng-content>` that captures all projected content.
- **Multi-slot projection:** multiple `<ng-content>` elements with `select` attributes that match projected elements by CSS selector, attribute, or directive.
- **Conditional projection:** combining `ng-content` with structural directives or the new control flow to project content only when a condition is met.

Content projection is essential for building reusable layout components like dialogs, cards, tabs, and panels. A strong interview answer demonstrates multi-slot projection with named slots.

Listing 1.8: Content projection: single-slot, multi-slot, and conditional

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

// -- Single-slot content projection --
@Component({
  selector: 'app-card',
  standalone: true,
  template: `
    <div class="card">
      <ng-content></ng-content>
    </div>
  `,
})
export class CardComponent {}

// -- Multi-slot content projection --
// Consumers place content into named slots using the select attribute.
@Component({
  selector: 'app-dialog',
  standalone: true,
  template: `
    <div class="dialog-backdrop" role="dialog">
      <header class="dialog__header">
        <ng-content select="[dialog-title]"></ng-content>
      </header>

      <section class="dialog__body">
        <ng-content></ng-content>
      </section>

      <footer class="dialog__footer">
        <ng-content select="[dialog-actions]"></ng-content>
      </footer>
    </div>
  `,
})
export class DialogComponent {}

// -- Conditional content projection --
// Combine ng-content with ng-template to project content
// only when a condition is met.
@Component({
  selector: 'app-collapsible-section',
  standalone: true,
  template: `
    <button (click)="isOpen = !isOpen">
      {{ isOpen ? 'Collapse' : 'Expand' }}
    </button>
    @if (isOpen) {
      <div class="section__content">
        <ng-content></ng-content>
      </div>
    }
  `,
})
export class CollapsibleSectionComponent {
  @Input() isOpen = false;
}

// -- Usage example --
// <app-dialog>
//   <h2 dialog-title>Confirm Deletion</h2>
//   <p>Are you sure you want to delete this item?</p>
//   <div dialog-actions>
//     <button (click)="cancel()">Cancel</button>
//     <button (click)="confirm()">Delete</button>
//   </div>
// </app-dialog>
```

What are template reference variables?

Template reference variables are declared in the template with the `#` syntax and provide a way to reference a DOM element, directive, or component instance directly within the template. When applied to a plain HTML element, the variable references the native element. When applied to a component or directive, it references the component or directive instance (or the value specified by `exportAs`).

Template reference variables are scoped to the template. To access them in the component class, use `@ViewChild` with the variable name as a string selector. They are commonly used for form validation checks, programmatic focus management, and passing element references to method calls.

Listing 1.9: Template reference variables for form elements and directives

```
import { Component, ViewChild, ElementRef } from '@angular/core';
import { NgForm, FormsModule } from '@angular/forms';

@Component({
  selector: 'app-login-form',
  standalone: true,
  imports: [FormsModule],
  template: `
    <!-- #emailInput is a template reference variable pointing
         to the native <input> DOM element. -->
    <input
      #emailInput
      type="email"
      name="email"
      [(ngModel)]="email"
      required
    />

    <!-- #loginForm="ngForm" is a template reference variable
         pointing to the NgForm directive instance. -->
    <form #loginForm="ngForm" (ngSubmit)="onSubmit(loginForm)">
      <input
        type="password"
        name="password"
        [(ngModel)]="password"
        required
      />

      <!-- Use the template variable directly in the template
           to check form validity. -->
      <button type="submit" [disabled]="loginForm.invalid">
        Log In
      </button>
    </form>

    <!-- Access a native element to trigger an action. -->
    <button type="button" (click)="emailInput.focus()">
      Focus Email
    </button>
  `,
})
export class LoginFormComponent {
  email = '';
  password = '';

  @ViewChild('emailInput')
  emailInputRef!: ElementRef<HTMLInputElement>;

  onSubmit(form: NgForm): void {
    if (form.valid) {
      console.log('Submitting', { email: this.email });
    }
  }
}

// -- Key interview points --
// * Template reference variables are declared with # in the template.
// * They reference either the DOM element or a directive/component
//   instance (using exportAs or an explicit assignment like #f="ngForm").
// * They are scoped to the template - you cannot access them
//   outside the template unless you use @ViewChild in the class.
```

How do you conditionally render content with @if, @for, and @switch?

Angular 17 introduced built-in control flow blocks that replace the legacy structural directives `*ngIf`, `*ngFor`, and `ngSwitch`. The new syntax is block-based, does not require importing `CommonModule`, and enables better compile-time optimization.

`@if` renders content conditionally and supports `@else` blocks and the `as` keyword for local variable binding. `@for` iterates over a collection and requires a `track` expression for efficient DOM reconciliation; it also supports an `@empty` block that renders when the collection is empty. `@switch` provides multi-branch conditional rendering without requiring a wrapper element.

In interviews, candidates should show awareness of the `track` requirement in `@for` and explain why it exists (performance optimization through identity-based DOM reuse). Mentioning the migration path from legacy directives to the new syntax demonstrates practical experience.

Listing 1.10: New control flow: @if, @for, @switch with @empty

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

export interface Task {
  id: number;
  title: string;
  status: 'todo' | 'in-progress' | 'done';
  assignee?: string;
}

@Component({
  selector: 'app-task-board',
  standalone: true,
  template: `
    <!-- @if replaces *ngIf - cleaner syntax, better type narrowing -->
    @if (tasks.length === 0) {
      <p class="empty">No tasks found.</p>
    } @else {
      <h2>Task Board ({{ tasks.length }} items)</h2>

      <!-- @for replaces *ngFor - requires a track expression -->
      @for (task of tasks; track task.id) {
        <article class="task-card">
          <h3>{{ task.title }}</h3>

          <!-- @switch replaces ngSwitch - no wrapper element needed -->
          @switch (task.status) {
            @case ('todo') {
              <span class="badge badge--todo">To Do</span>
            }
            @case ('in-progress') {
              <span class="badge badge--wip">In Progress</span>
            }
            @case ('done') {
              <span class="badge badge--done">Done</span>
            }
            @default {
              <span class="badge">Unknown</span>
            }
          }

          @if (task.assignee; as assignee) {
            <p class="assignee">Assigned to: {{ assignee }}</p>
          } @else {
            <p class="assignee unassigned">Unassigned</p>
          }
        </article>
      } @empty {
        <!-- @empty block renders when the collection is empty -->
        <p>All tasks cleared!</p>
      }
    }
  `,
})
export class TaskBoardComponent {
  @Input() tasks: Task[] = [];
}

// -- Key interview points --
// * Built-in control flow (@if, @for, @switch) was introduced
//   in Angular 17 as a replacement for structural directives.
// * @for requires a `track` expression (like trackBy) - Angular
//   uses it for efficient DOM diffing.
// * @empty is a convenience block inside @for that renders when
//   the iterable is empty.
// * The new syntax is block-based, does NOT require importing
//   CommonModule, and enables better compile-time optimization.
```

What is the difference between ng-template, ng-container, and ng-content?

These three Angular elements serve distinct purposes and are frequently confused in interviews:

- **ng-template:** defines a block of template content that is *not rendered by default*. It must be explicitly stamped out using a structural directive (`*ngIf`, `*ngFor`) or `ngTemplateOutlet`. It supports context variables, making it powerful for creating reusable template fragments.
- **ng-container:** a grouping element that produces *no DOM node*. It is used to apply structural directives without introducing an extra wrapper element in the DOM. For example, applying both `*ngIf` and `*ngFor` on the same element requires an `ng-container` since only one structural directive can be applied per element.
- **ng-content:** a *projection slot* that marks where content from a parent component's template will be inserted. The content originates in the consuming component, not in the component that defines `ng-content`.

A strong answer includes a practical example showing all three working together, such as a data table component that uses `ng-content` for headers, `ng-container` for structural grouping, and `ng-template` with `ngTemplateOutlet` for customizable row rendering.

Listing 1.11: ng-template, ng-container, and ng-content in a data table

```
import { CommonModule } from '@angular/common';
import {
  Component,
  Directive,
  Input,
  TemplateRef,
} from '@angular/core';

// ----------------------------------------------
// ng-template: defines a reusable template block that is
// NOT rendered by default. It must be explicitly stamped
// out using a structural directive or ngTemplateOutlet.
// ----------------------------------------------

// ----------------------------------------------
// ng-container: a grouping element that does NOT produce
// any DOM node. Useful for applying structural directives
// without introducing extra wrapper elements.
// ----------------------------------------------

// ----------------------------------------------
// ng-content: a projection slot. It marks where content
// from a parent component's template will be inserted.
// Unlike ng-template, the projected content originates
// in the consuming component's template.
// ----------------------------------------------

@Component({
  selector: 'app-data-table',
  standalone: true,
  imports: [CommonModule],
  template: `
    <table>
      <thead>
        <tr>
          <!-- ng-content: project column headers from parent -->
          <ng-content select="[table-header]"></ng-content>
        </tr>
      </thead>
      <tbody>
        <!-- ng-container: group without adding a DOM element -->
        <ng-container *ngFor="let row of rows; let i = index">
          <tr>
            <!-- ng-template + ngTemplateOutlet: render the
                 consumer-supplied row template with context -->
            <ng-container
              *ngTemplateOutlet="rowTemplate; context: { $implicit: row, index: i }"
            ></ng-container>
          </tr>
        </ng-container>
      </tbody>
    </table>

    <!-- Fallback when no data -->
    <ng-container *ngIf="rows.length === 0">
      <!-- ng-template used as a reusable empty state -->
      <ng-container *ngTemplateOutlet="emptyState"></ng-container>
    </ng-container>

    <ng-template #emptyState>
```

```
      <p class="empty-message">No data available.</p>
    </ng-template>
  `,
})
export class DataTableComponent {
  @Input() rows: unknown[] = [];
  @Input() rowTemplate!: TemplateRef<{ $implicit: unknown; index: number }>;
}

// -- Usage in a parent component --
// <app-data-table [rows]="users" [rowTemplate]="userRow">
//   <th table-header>Name</th>
//   <th table-header>Email</th>
// </app-data-table>
//
// <ng-template #userRow let-user let-i="index">
//   <td>{{ i + 1 }}. {{ user.name }}</td>
//   <td>{{ user.email }}</td>
// </ng-template>
```

1.4 Advanced Component Patterns

How do you create dynamic components?

Dynamic component creation is used when the component to render is not known at compile time — for example, a dashboard that loads user-configured widgets, or a form builder that renders different field types based on configuration.

The modern API uses `ViewContainerRef.createComponent()`, which replaced the older `ComponentFactoryResolver` approach. You obtain a `ViewContainerRef` via `@ViewChild` on an anchor element, then call `createComponent()` with the component type. The returned `ComponentRef` provides access to the instance for setting inputs via `setInput()`, which correctly triggers `ngOnChanges`.

Key considerations for interview answers:

- Dynamic components must be standalone or part of a module available at runtime.
- Use `ComponentRef.setInput()` instead of directly assigning properties to ensure proper lifecycle hook triggering.
- Always call `ViewContainerRef.clear()` before re-rendering to prevent memory leaks.
- For lazy loading, combine with `import()` to load component code on demand.

Listing 1.12: Dynamic component creation with ViewContainerRef

```
import {
  Component,
  Input,
  Type,
  ViewChild,
  ViewContainerRef,
  OnInit,
  inject,
} from '@angular/core';

// -- Step 1: Define components that can be loaded dynamically --

@Component({
  selector: 'app-text-widget',
  standalone: true,
  template: `<p class="widget">{{ content }}</p>`,
})
export class TextWidgetComponent {
  @Input() content = '';
}

@Component({
  selector: 'app-chart-widget',
  standalone: true,
  template: `<div class="widget chart">Chart: {{ title }}</div>`,
})
export class ChartWidgetComponent {
  @Input() title = '';
}

// -- Step 2: A registry that maps widget types to components --

export interface WidgetConfig {
  type: 'text' | 'chart';
  inputs: Record<string, unknown>;
}

const WIDGET_MAP: Record<string, Type<unknown>> = {
  text: TextWidgetComponent,
  chart: ChartWidgetComponent,
};

// -- Step 3: Host component that creates widgets at runtime --

@Component({
  selector: 'app-dashboard',
  standalone: true,
  template: `
    <h2>Dashboard</h2>
    <!-- The anchor where dynamic components will be inserted -->
    <ng-container #widgetHost></ng-container>
  `,
})
export class DashboardComponent implements OnInit {
  @Input() widgets: WidgetConfig[] = [];

  @ViewChild('widgetHost', { read: ViewContainerRef, static: true })
  hostRef!: ViewContainerRef;

  ngOnInit(): void {
    this.renderWidgets();
  }

  private renderWidgets(): void {
    this.hostRef.clear();

    for (const config of this.widgets) {
      const componentType = WIDGET_MAP[config.type];
      if (!componentType) {
        console.warn(`Unknown widget type: ${config.type}`);
        continue;
      }

      // createComponent returns a ComponentRef with
      // access to the instance for setting inputs.
      const ref = this.hostRef.createComponent(componentType);

      // Set inputs programmatically
      for (const [key, value] of Object.entries(config.inputs)) {
        ref.setInput(key, value);
      }
    }
  }
}

// -- Key interview points --
// * ViewContainerRef.createComponent() is the modern API for
//   dynamic component creation (replaces ComponentFactoryResolver).
// * ComponentRef.setInput() sets inputs and triggers OnChanges,
//   matching the behavior of template-bound inputs.
// * Dynamic components must be standalone or declared in a module
//   that is available at runtime.
```

What is ViewContainerRef and how does it relate to dynamic component loading?

`ViewContainerRef` is a container where one or more views can be attached to a component. It is the core API for dynamic UI manipulation in Angular. Every component or directive has an associated `ViewContainerRef` (which can be injected), representing the location in the DOM where it exists.

When you call `vcr.createComponent(Type)`, Angular:

1. Instantiates the component type.
2. Creates a new Host View for that component.
3. Inserts the component's rendered output into the DOM at the container's location.

This is fundamentally different from simply using `@if` to show/hide a component. `ViewContainerRef` allows you to decide *which* component to load at runtime, based on data, and manage its lifecycle (inputs, outputs, and destruction) programmatically.

Listing 1.13: Advanced dynamic loading with lazy-loaded components

```
import {
  Component,
  ComponentRef,
  inject,
  Input,
  OnInit,
  ViewChild,
  ViewContainerRef
} from '@angular/core';

@Component({
  selector: 'app-dynamic-host',
  standalone: true,
  template: `
    <h3>Dynamic Component Loader</h3>
    <div #container></div>
    <button (click)="loadAdminPanel()">Load Admin Panel</button>
  `,
})
export class DynamicHostComponent {
  @ViewChild('container', { read: ViewContainerRef, static: true })
  vcr!: ViewContainerRef;

  private componentRef?: ComponentRef<any>;

  async loadAdminPanel() {
    // 1. Clear existing components to prevent leaks
    this.vcr.clear();

    // 2. Dynamic import for code splitting
    const { AdminPanelComponent } = await import('./admin-panel.component');

    // 3. Create component instance
    this.componentRef = this.vcr.createComponent(AdminPanelComponent);

    // 4. Set inputs via setInput() to trigger lifecycle hooks correctly
    this.componentRef.setInput('timestamp', Date.now());

    // 5. Handle outputs via subscriptions
    this.componentRef.instance.closed.subscribe(() => {
      this.vcr.clear();
      this.componentRef = undefined;
    });
  }
}

/**
 * Technical Detail:
 * Why use ComponentRef.setInput()?
 * In Angular 14.1+, setInput() was introduced to ensure that
 * when an input is changed on a dynamic component, the
 * ngOnChanges hook is properly triggered. Assigning to the
 * instance property directly (this.componentRef.instance.timestamp = value)
 * bypasses the framework's change tracking mechanism for OnChanges.
 */
```

When would you use ViewContainerRef instead of structural directives?

Structural directives like `@if` and `@for` are declarative and preferred for most UI logic. However, `ViewContainerRef` is required when:

- The component type to render is determined by data at runtime (e.g., a "Dashboard Widget" fetched from a database).
- You are building a low-level utility like a "Modal Service" or "Toast Service" that needs to inject components into a top-level container.
- You need to implement complex preloading or lazy-loading logic that standard routing doesn't cover.

Listing 1.14: Comparison: Declarative @if vs Programmatic ViewContainerRef

```
// --- Declarative Approach (@if) ---
// Use this when the set of possible components is fixed and
// known at compile time. It's readable, testable, and
// handles lifecycle automatically.
@Component({
  template: `
    @if (userRole === 'admin') {
      <app-admin-panel [data]="data" />
    } @else {
      <app-user-panel [data]="data" />
    }
  `
})
export class DeclarativeComponent {
  @Input() userRole = 'user';
  @Input() data: any;
}

// --- Programmatic Approach (ViewContainerRef) ---
// Use this when components are dynamic, pluggable, or
// loaded from external configuration.
@Component({
  template: `<div #container></div>`
})
export class ProgrammaticComponent {
  @ViewChild('container', { read: ViewContainerRef }) vcr!: ViewContainerRef;

  async loadWidget(widgetType: string) {
    this.vcr.clear();
    // widgetRegistry maps strings to dynamic imports
    const component = await widgetRegistry.get(widgetType);
    const ref = this.vcr.createComponent(component);
    ref.setInput('config', this.getExternalConfig());
  }
}

/**
 * Interview Summary:
 * Prefer @if/@for (Declarative) for 95% of cases.
 * Use ViewContainerRef (Programmatic) only for:
 * 1. Pluggable architectures (CMS, Dashboards)
 * 2. Library/Utility creation (Modals, Toasts)
 * 3. Extreme performance cases where you need to bypass
 *    template instantiation until a specific interaction.
 */
```

What are view encapsulation modes and when do you change them?

- **Emulated (default):** Angular rewrites CSS selectors and adds unique attributes to DOM elements to scope styles to the component. No Shadow DOM is used. This is sufficient for the vast majority of components.
- **None:** Styles are added to the global stylesheet with no scoping. Use this sparingly, typically for global theme overrides or when you need to style third-party components that cannot be reached through encapsulated styles.
- **ShadowDom:** Uses the browser's native Shadow DOM for true style isolation. Styles cannot leak in or out. Best suited for truly isolated widgets like embeddable micro-frontends or custom elements published for external consumption.

Interviewers may ask when you would change the default. The most common valid reason is `ViewEncapsulation.None` for a component that provides global CSS custom properties or needs to override deeply nested third-party component styles. Using `ShadowDom` is rare in typical Angular applications but relevant in micro-frontend architectures.

Listing 1.15: View encapsulation modes: Emulated, None, and ShadowDom

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

// -- ViewEncapsulation.Emulated (default) --
// Angular adds unique attributes to elements and rewrites CSS
// selectors to scope styles to this component only.
// No Shadow DOM is used.
@Component({
  selector: 'app-emulated-card',
  standalone: true,
  encapsulation: ViewEncapsulation.Emulated,
  template: `<div class="card">Emulated encapsulation</div>`,
  styles: [
    `
      /* Compiled to .card[_ngcontent-xyz] - scoped automatically */
      .card {
        border: 1px solid #ccc;
        padding: 1rem;
      }
    `,
  ],
})
export class EmulatedCardComponent {}

```

```
// -- ViewEncapsulation.None --
// Styles are added to the global stylesheet with NO scoping.
// Use sparingly - for global theme overrides or third-party
// component styling that cannot be reached otherwise.
@Component({
  selector: 'app-global-styles',
  standalone: true,
  encapsulation: ViewEncapsulation.None,
  template: `<div class="card card--themed">No encapsulation</div>`,
  styles: [
    `
      /* WARNING: this will affect ALL .card elements globally */
      .card--themed {
        border: 2px solid var(--primary-color, #3f51b5);
      }
    `,
  ],
})
export class GlobalStylesComponent {}

// -- ViewEncapsulation.ShadowDom --
// Uses the browser's native Shadow DOM for true style isolation.
// Styles cannot leak in or out. Requires browser support and
// makes global theming harder.
@Component({
  selector: 'app-shadow-card',
  standalone: true,
  encapsulation: ViewEncapsulation.ShadowDom,
  template: `<div class="card">Shadow DOM encapsulation</div>`,
  styles: [
    `
      /* Only applies inside this component's shadow root */
      .card {
        border: 1px dashed #999;
        padding: 1rem;
      }
    `,
  ],
})
export class ShadowCardComponent {}

// -- When to change encapsulation --
// * Emulated (default): use for nearly all components.
// * None: use for global theme sheets or to style projected
//   content from third-party libraries.
// * ShadowDom: use for truly isolated widgets (e.g., embeddable
//   micro-frontends or web components published as custom elements).
```

How do you handle component communication beyond Input/Output?

`@Input()` and `@Output()` are the standard mechanisms for parent-child communication and should be the first choice for direct relationships. However, several scenarios require alternative approaches:

- **Sibling components:** Use a shared service with `Subject` or `BehaviorSubject`. Both siblings inject the service; one publishes, the other subscribes.

- **Deeply nested descendants:** Use a shared service or Angular's dependency injection hierarchy. Avoid chaining inputs through multiple intermediate components ("prop drilling").
- **Complex state:** Use a dedicated state management solution such as NgRx, a signals-based store, or a feature-scoped facade service.
- **Cross-cutting UI events:** Use a global event bus service for truly application-wide concerns like notifications.

Avoid using `@ViewChild` to call methods on child components as a communication mechanism. This creates tight coupling and makes the parent dependent on the child's internal API. In interviews, articulate why the service-based approach is preferred for non-parent-child relationships.

Listing 1.16: Component communication: Input/Output and shared service patterns

```
import {
  Component,
  EventEmitter,
  inject,
  Injectable,
  Input,
  OnDestroy,
  Output,
} from '@angular/core';
import { BehaviorSubject, Subject, Subscription } from 'rxjs';

// ----------------------------------------------
// 1. Input/Output - parent-child communication
// ----------------------------------------------
@Component({
  selector: 'app-quantity-picker',
  standalone: true,
  template: `
    <button (click)="decrement()">-</button>
    <span>{{ value }}</span>
    <button (click)="increment()">+</button>
  `,
})
export class QuantityPickerComponent {
  @Input() value = 1;
  @Output() valueChange = new EventEmitter<number>();

  increment(): void {
    this.valueChange.emit(++this.value);
  }
  decrement(): void {
    if (this.value > 0) this.valueChange.emit(--this.value);
  }
}

// ----------------------------------------------
// 2. Shared service - sibling or distant component communication
// ---------------------------------------------
@Injectable({ providedIn: 'root' })
export class NotificationBus {
  private readonly messages$ = new Subject<string>();
  readonly notifications$ = this.messages$.asObservable();

  send(message: string): void {
    this.messages$.next(message);
  }
}

@Component({
  selector: 'app-toolbar',
  standalone: true,
  template: `<button (click)="notify()">Broadcast</button>`,
})
export class ToolbarComponent {
  private readonly bus = inject(NotificationBus);

  notify(): void {
    this.bus.send('Action triggered from toolbar');
  }
}

@Component({
  selector: 'app-sidebar',
  standalone: true,
  template: `<p>Last message: {{ lastMessage }}</p>`,
})
export class SidebarComponent implements OnDestroy {
  private readonly bus = inject(NotificationBus);
  private readonly sub: Subscription;
  lastMessage = '(none)';

  constructor() {
    this.sub = this.bus.notifications$.subscribe(
      (msg) => (this.lastMessage = msg)
    );
  }

  ngOnDestroy(): void {
    this.sub.unsubscribe();
  }
}

// -- Key interview points --
// * Input/Output is best for direct parent-child relationships.
// * A shared service with Subjects/BehaviorSubjects works for
//   sibling or deeply nested communication.
// * For complex state, prefer a dedicated state management
//   approach (NgRx, signals-based store, etc.).
// * Avoid relying on ViewChild to reach into child internals -
//   it creates tight coupling.
```

1.5 Component Architecture Patterns

What is the container/presentational pattern?

The container/presentational (or smart/dumb) pattern separates components into two categories based on their responsibilities:

Presentational components receive all data through `@Input()` and communicate user actions through `@Output()`. They have no injected services, no side effects, and use `OnPush` change detection. They are trivial to test: set inputs, check template output, assert emitted events.

Container components own the data pipeline. They inject services, manage state, coordinate async operations, and pass data down to presentational children. Their templates are thin wiring layers, not layout-heavy markup.

This pattern is not mandatory for every component, but it is a powerful scaling strategy. When a component starts mixing data fetching with rendering logic, splitting it into a container and one or more presentational components improves testability, reusability, and readability.

Listing 1.17: Container/presentational split for a product catalog

```
import {
  ChangeDetectionStrategy,
  Component,
  EventEmitter,
  inject,
  Input,
  Output,
} from '@angular/core';
import { Observable } from 'rxjs';
import { AsyncPipe } from '@angular/common';

// ---------------------------------------------
// Presentational component
// ---------------------------------------------
// * Receives data through @Input, emits events through @Output.
// * Has NO injected services and NO side effects.
// * Uses OnPush change detection for performance.
// * Easy to test: just set inputs, check template output.

export interface Product {
  id: string;
  name: string;
  price: number;
}

@Component({
  selector: 'app-product-list',
  standalone: true,
  changeDetection: ChangeDetectionStrategy.OnPush,
  template: `
    <ul>
      @for (product of products; track product.id) {
        <li>
          {{ product.name }} - {{ product.price | currency }}
          <button (click)="addToCart.emit(product.id)">Add</button>
        </li>
      }
    </ul>
  `,
})
export class ProductListComponent {
  @Input({ required: true }) products!: Product[];
  @Output() addToCart = new EventEmitter<string>();
}

// ---------------------------------------------
// Container component
// ---------------------------------------------
// * Owns the data pipeline: injects services, manages state.
// * Passes data DOWN to presentational children via inputs.
// * Reacts to child events and delegates to services.
// * Template is thin - mostly wiring, not layout.

import { ProductService } from './product.service';

@Component({
  selector: 'app-product-catalog-page',
  standalone: true,
  imports: [AsyncPipe, ProductListComponent],
  changeDetection: ChangeDetectionStrategy.OnPush,
  template: `
    @if (products$ | async; as products) {
      <app-product-list
        [products]="products"
        (addToCart)="onAddToCart($event)"
      />
    } @else {
      <p>Loading products...</p>
    }
  `,
})
export class ProductCatalogPageComponent {
  private readonly productService = inject(ProductService);
  readonly products$: Observable<Product[]> = this.productService.getAll();

  onAddToCart(productId: string): void {
    this.productService.addToCart(productId);
  }
}
```

What are host directives and how do they enable composition?

Host directives, introduced in Angular 15, allow a component to declaratively apply one or more directives to its host element. This enables a composition-over-inheritance pattern: instead of creating a base class with shared behavior, you extract behaviors into standalone directives and attach them via the `hostDirectives` array.

The component can selectively expose the directive's inputs and outputs to its consumers, or keep them internal. This provides a clean separation where the consuming template does not need to know about the underlying directives.

Common use cases include:

- adding tooltip behavior to multiple button components,
- applying consistent disabled-state styling and ARIA attributes,
- composing loading-state indicators across different container types.

Listing 1.18: Host directives: composing tooltip and disabled-state behaviors

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

// -- Reusable behavior: tooltip directive --
@Directive({
  selector: '[appTooltip]',
  standalone: true,
})
export class TooltipDirective {
  @Input('appTooltip') tooltipText = '';
  private tooltipEl?: HTMLDivElement;

  @HostListener('mouseenter')
  show(): void {
    // Simplified - production code would use an overlay service
    this.tooltipEl = document.createElement('div');
    this.tooltipEl.className = 'tooltip';
    this.tooltipEl.textContent = this.tooltipText;
    document.body.appendChild(this.tooltipEl);
  }

  @HostListener('mouseleave')
  hide(): void {
    this.tooltipEl?.remove();
  }
}

// -- Reusable behavior: disabled-state directive --
@Directive({
  selector: '[appDisableable]',
  standalone: true,
})
export class DisableableDirective {
  @Input() disabled = false;

  @HostBinding('class.is-disabled')
  get cssClass(): boolean {
    return this.disabled;
  }

  @HostBinding('attr.aria-disabled')
  get ariaDisabled(): string | null {
    return this.disabled ? 'true' : null;
  }
}

// -- Component that composes multiple directives via hostDirectives --
// Instead of manually applying [appTooltip] and [appDisableable]
// everywhere this button is used, the component declares them once.
@Component({
  selector: 'app-action-button',
  standalone: true,
  hostDirectives: [
    {
      directive: TooltipDirective,
      inputs: ['appTooltip: tooltip'],
    },
    {
      directive: DisableableDirective,
      inputs: ['disabled'],
    },
  ],
  template: `
    <button [disabled]="isDisabled" type="button">
      <ng-content></ng-content>
    </button>
  `,
})
export class ActionButtonComponent {
  @Input() isDisabled = false;
}

// Usage:
// <app-action-button tooltip="Save changes" [disabled]="isSaving">
//   Save
// </app-action-button>
//
// The tooltip and disabled-state behaviors are applied automatically
// without the consumer needing to know about the underlying directives.
```

When does a component become too large and how do you refactor it?

A component is too large when it handles unrelated responsibilities: data fetching, business logic, complex rendering, and formatting all in one place. The signal is not a specific line count but a loss of focus. Concrete warning signs include:

- The template exceeds approximately 80 lines or contains three or more nested conditionals.
- The class mixes HTTP calls, business logic, and UI state management.
- Unit tests require complex setup with many mocks because the component depends on too many services.
- Two developers frequently edit the same component for unrelated features (merge conflict hotspot).

The refactoring strategy is to split by responsibility boundaries, not arbitrary line count. Extract presentational sub-components for self-contained template fragments. Move orchestration logic into facade services. Move business calculations into pure utility functions that are independently testable.

Listing 1.19: Refactoring a monolithic component into focused pieces

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

// ----------------------------------------------
// BEFORE refactoring: a monolithic component that handles
// data fetching, business logic, and complex rendering.
// ----------------------------------------------

// @Component({
//   selector: 'app-invoice-page',
//   template: `
//     <h1>Invoice #{{ invoiceId }}</h1>
//     <div *ngIf="loading">Loading...</div>
//     <div *ngIf="!loading">
//       <div *ngFor="let line of lineItems">
//         {{ line.description }} - {{ line.amount | currency }}
//         <button (click)="removeLine(line.id)">Remove</button>
//       </div>
//       <p>Subtotal: {{ subtotal | currency }}</p>
//       <p>Tax ({{ taxRate }}%): {{ tax | currency }}</p>
//       <p>Total: {{ total | currency }}</p>
//       <button (click)="submitInvoice()">Submit</button>
//     </div>
//   `,
// })
// export class InvoicePageComponent {
//   // 200+ lines mixing HTTP calls, calculations, navigation...
// }

// ----------------------------------------------
// AFTER refactoring: split into focused pieces
// ----------------------------------------------

// 1. Presentational: renders a single line item
@Component({
  selector: 'app-invoice-line',
  standalone: true,
  changeDetection: ChangeDetectionStrategy.OnPush,
  template: `
    <div class="line-item">
      <span>{{ description }}</span>
      <span>{{ amount | currency }}</span>
      <button (click)="remove.emit()">Remove</button>
    </div>
  `,
})
export class InvoiceLineComponent {
  @Input({ required: true }) description!: string;
  @Input({ required: true }) amount!: number;
  @Output() remove = new EventEmitter<void>();
}

// 2. Presentational: renders the invoice summary
@Component({
  selector: 'app-invoice-summary',
  standalone: true,
  changeDetection: ChangeDetectionStrategy.OnPush,
  template: `
    <div class="summary">
      <p>Subtotal: {{ subtotal | currency }}</p>
      <p>Tax ({{ taxRate }}%): {{ tax | currency }}</p>
      <p><strong>Total: {{ total | currency }}</strong></p>
    </div>
  `,
})
export class InvoiceSummaryComponent {
  @Input({ required: true }) subtotal!: number;
  @Input({ required: true }) taxRate!: number;

  get tax(): number {
    return this.subtotal * (this.taxRate / 100);
  }

  get total(): number {
    return this.subtotal + this.tax;
  }
}

// 3. Container: orchestrates the page
@Component({
  selector: 'app-invoice-page',
  standalone: true,
  imports: [InvoiceLineComponent, InvoiceSummaryComponent],
  changeDetection: ChangeDetectionStrategy.OnPush,
  template: `
    @if (loading) {
      <p>Loading...</p>
    } @else {
      <h1>Invoice #{{ invoiceId }}</h1>
      @for (line of lineItems; track line.id) {
        <app-invoice-line
          [description]="line.description"
          [amount]="line.amount"
          (remove)="removeLine(line.id)"
        />
      }
      <app-invoice-summary
        [subtotal]="subtotal"
        [taxRate]="taxRate"
      />
      <button (click)="submit()">Submit</button>
    }
  `,
})
```

```typescript
export class InvoicePageComponent {
  @Input() invoiceId = '';
  loading = false;
  lineItems: { id: string; description: string; amount: number }[] = [];
  taxRate = 10;

  get subtotal(): number {
    return this.lineItems.reduce((sum, l) => sum + l.amount, 0);
  }

  removeLine(id: string): void {
    this.lineItems = this.lineItems.filter((l) => l.id !== id);
  }

  submit(): void {
    // Delegate to a service - not shown here
  }
}

// -- Refactoring signals --
// * Template exceeds ~80 lines or has 3+ nested conditionals.
// * Class mixes HTTP, business logic, and UI state.
// * Unit tests require complex setup with many mocks.
// * Split by responsibility: rendering vs orchestration vs logic.
```

How do you keep reusable components stable across teams?

Reusable components that are consumed by multiple teams require extra discipline to remain stable. The key practices are:

- **Strict input/output contracts:** Use `required: true` on mandatory inputs. Define dedicated interface types for output event payloads instead of primitive types.
- **No application-specific dependencies:** Generic UI components should not inject domain services. They should depend only on Angular core and the CDK.
- **Behavior documented through tests:** Consumers rely on tested behavior. A comprehensive test suite is the most reliable form of API documentation.
- **Semantic versioning:** When published to a shared library, breaking input/output changes require a major version bump.
- **OnPush change detection:** Enforces immutable data flow and prevents subtle bugs from mutable state.

Listing 1.20: Stable reusable pagination component with strict contracts

```typescript
import {
  ChangeDetectionStrategy,
  Component,
  EventEmitter,
  Input,
  Output,
  TemplateRef,
  ContentChild,
} from '@angular/core';
import { CommonModule } from '@angular/common';

// -- A reusable pagination component designed for cross-team stability --

export interface PageChangeEvent {
  page: number;
  pageSize: number;
}

@Component({
  selector: 'ui-pagination',
  standalone: true,
  imports: [CommonModule],
  changeDetection: ChangeDetectionStrategy.OnPush,
  template: `
    <nav aria-label="Pagination">
      <button
        [disabled]="currentPage === 1"
        (click)="goToPage(currentPage - 1)"
      >
        Previous
      </button>

      @for (page of pages; track page) {
        <button
          [class.active]="page === currentPage"
          [attr.aria-current]="page === currentPage ? 'page' : null"
          (click)="goToPage(page)"
        >
          {{ page }}
        </button>
      }

      <button
        [disabled]="currentPage === totalPages"
        (click)="goToPage(currentPage + 1)"
      >
        Next
      </button>
    </nav>
  `,
})
export class PaginationComponent {
  // -- Strict, typed input contract --
  @Input({ required: true }) totalItems!: number;
  @Input() pageSize = 10;
  @Input() currentPage = 1;

  // -- Single, well-defined output event --
  @Output() pageChange = new EventEmitter<PageChangeEvent>();

  get totalPages(): number {
    return Math.ceil(this.totalItems / this.pageSize);
  }

  get pages(): number[] {
    return Array.from({ length: this.totalPages }, (_, i) => i + 1);
  }

  goToPage(page: number): void {
    if (page < 1 || page > this.totalPages) return;
    this.currentPage = page;
    this.pageChange.emit({ page, pageSize: this.pageSize });
  }
}

// -- Stability guidelines for shared components --
//
// 1. Use ‘required: true‘ on inputs that must always be provided.
// 2. Define dedicated interfaces for output event payloads
//    (PageChangeEvent, not raw numbers) - this makes the
//    contract explicit and easier to version.
// 3. Avoid injecting application-specific services. Generic UI
//    components should depend only on Angular core and CDK.
// 4. Document behavior through unit tests that double as
//    specification - consumers rely on tested behavior.
// 5. Use semantic versioning when publishing to a shared
//    library: breaking input/output changes = major bump.
// 6. Prefer OnPush to enforce immutable data flow.
```

What are the best practices for component testing?

Component testing in Angular uses `TestBed` to create a testing module, instantiate the component, and interact with it through its fixture. Key best practices include:

1. **Use data-testid attributes** for DOM queries instead of CSS class selectors, which are fragile and change with styling.
2. **Test inputs** by setting them on the component instance before calling `fixture.detectChanges()`.
3. **Test outputs** by spying on `EventEmitter.emit()` and triggering the corresponding DOM event.
4. **Handle OnPush** by calling `fixture.detectChanges()` after every input change, or by using `fixture.componentRef.setInput()`.
5. **Prefer shallow tests:** test one component in isolation. Stub or mock child components using `ng-mocks` or manual stubs to avoid pulling in the entire component tree.
6. **Use fixture.debugElement** instead of `nativeElement.querySelector` for consistency across testing environments.

Listing 1.21: Component testing best practices with TestBed

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { By } from '@angular/platform-browser';
import {
  ChangeDetectionStrategy,
  Component,
  EventEmitter,
  Input,
  Output,
} from '@angular/core';

// -- The component under test --
@Component({
  selector: 'app-counter',
  standalone: true,
  changeDetection: ChangeDetectionStrategy.OnPush,
  template: `
    <span data-testid="count">{{ count }}</span>
    <button data-testid="increment" (click)="increment()">+</button>
    <button data-testid="reset" (click)="resetClicked.emit()">Reset</button>
  `,
})
export class CounterComponent {
  @Input() count = 0;
  @Output() countChange = new EventEmitter<number>();
  @Output() resetClicked = new EventEmitter<void>();

  increment(): void {
    this.count++;
    this.countChange.emit(this.count);
  }
}

// -- Test suite --
describe('CounterComponent', () => {
  let fixture: ComponentFixture<CounterComponent>;
  let component: CounterComponent;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      // Standalone components are imported, not declared
      imports: [CounterComponent],
    }).compileComponents();

    fixture = TestBed.createComponent(CounterComponent);
    component = fixture.componentInstance;
  });

  it('should render the initial count', () => {
    component.count = 5;
    fixture.detectChanges();

    const span = fixture.debugElement.query(
      By.css('[data-testid="count"]')
    );
    expect(span.nativeElement.textContent.trim()).toBe('5');
  });

  it('should emit countChange when increment is clicked', () => {
    component.count = 0;
    fixture.detectChanges();

    const emitSpy = jest.spyOn(component.countChange, 'emit');

    const button = fixture.debugElement.query(
      By.css('[data-testid="increment"]')
    );
    button.triggerEventHandler('click', null);

    expect(emitSpy).toHaveBeenCalledWith(1);
  });

  it('should emit resetClicked on reset button', () => {
    fixture.detectChanges();
    const emitSpy = jest.spyOn(component.resetClicked, 'emit');

    const button = fixture.debugElement.query(
      By.css('[data-testid="reset"]')
    );
    button.triggerEventHandler('click', null);

    expect(emitSpy).toHaveBeenCalled();
  });
});

// -- Best practices for component testing --
// 1. Use data-testid attributes for stable selectors.
// 2. Test inputs by setting them before detectChanges().
// 3. Test outputs by spying on EventEmitter.emit().
// 4. For OnPush components, call fixture.detectChanges()
//    after every input change.
// 5. Use fixture.debugElement.query(By.css(...)) instead
//    of nativeElement.querySelector for consistency.
// 6. Prefer shallow tests (test one component in isolation)
//    and use ng-mocks or manual stubs for child components.
```

1.6 Template Discipline

Templates should remain declarative. Heavy decision logic in templates is usually a smell and should be moved into component properties, computed signals, or dedicated view models.

Useful rules

- avoid expensive method calls directly in interpolation,
- keep structural directives readable and shallow,
- prefer computed state over repeated conditional fragments.

1.7 Standalone Components as the Modern Baseline

Standalone components simplify local reasoning because dependencies are declared near usage. This does not remove all complexity, but it reduces accidental coupling introduced by large, shared module declarations.

Interview framing

If asked "Why standalone?", strong answers mention:

- lower cognitive overhead for new contributors,
- simpler lazy loading strategy,
- easier incremental migration from older architecture.

1.8 Code Walkthroughs

Presentational standalone component

Listing 1.22: Focused presentational component with strict API contract

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

export interface UserProfileViewModel {
  id: string;
  name: string;
  email: string;
  isActive: boolean;
}

@Component({
  selector: 'app-user-profile-card',
  standalone: true,
  template: `
    <article class="card">
      <h3>{{ user.name }}</h3>
      <p>{{ user.email }}</p>
      <span>{{ user.isActive ? 'Active' : 'Inactive' }}</span>
      <button type="button" (click)="editRequested.emit(user.id)">Edit</button>
    </article>
  `,
  changeDetection: ChangeDetectionStrategy.OnPush,
})
export class UserProfileCardComponent {
  @Input({ required: true }) user!: UserProfileViewModel;
  @Output() editRequested = new EventEmitter<string>();
}
```

Container component orchestrating data and interactions

Listing 1.23: Container component coordinating state and UI events

```
import { AsyncPipe, NgFor } from '@angular/common';
import { ChangeDetectionStrategy, Component, inject } from '@angular/core';

import { UserProfileCardComponent } from './user_profile_card.component';
import { UsersFacadeService } from './users_facade.service';

@Component({
  selector: 'app-users-page',
  standalone: true,
  imports: [NgFor, AsyncPipe, UserProfileCardComponent],
  template: `
    <section>
      <h2>Users</h2>
      <app-user-profile-card
        *ngFor="let user of users$ | async"
        [user]="user"
        (editRequested)="openEditor($event)"
      />
    </section>
  `,
  changeDetection: ChangeDetectionStrategy.OnPush,
})
export class UsersPageComponent {
  private readonly facade = inject(UsersFacadeService);
  readonly users$ = this.facade.users$;

  openEditor(userId: string): void {
    this.facade.selectUser(userId);
  }
}
```

Facade service keeping domain orchestration outside presentation

Listing 1.24: Facade pattern for component-friendly feature orchestration

```
import { Injectable } from '@angular/core';
import { BehaviorSubject, Observable } from 'rxjs';

import { UserProfileViewModel } from './user_profile_card.component';

@Injectable({ providedIn: 'root' })
export class UsersFacadeService {
  private readonly usersSubject = new BehaviorSubject<UserProfileViewModel[]>([]);
  readonly users$: Observable<UserProfileViewModel[]> = this.usersSubject.asObservable();

  private readonly selectedUserId = new BehaviorSubject<string | null>(null);
  readonly selectedUserId$ = this.selectedUserId.asObservable();

  loadUsers( users: UserProfileViewModel[]): void {
    this.usersSubject.next(users);
  }

  selectUser(userId: string): void {
    this.selectedUserId.next(userId);
  }
}
```

Anti-pattern example: overcoupled component

Listing 1.25: Overcoupled component doing too much (what to avoid)

```
import { HttpClient } from '@angular/common/http';
import { Component, inject } from '@angular/core';
import { Router } from '@angular/router';

@Component({
  selector: 'app-overcoupled-users',
  standalone: true,
  template: `<div *ngFor="let user of users">{{ user.name }}</div>`,
})
export class OvercoupledUsersComponent {
  private readonly http = inject(HttpClient);
  private readonly router = inject(Router);

   users: Array<{ id: string; name: string }> = [];

  constructor() {
    this.http.get<Array<{ id: string; name: string }>>('/api/users').subscribe((result) => {
      this.users = result;
      if (result.length === 0) {
        this.router.navigate(['/empty-state']);
      }
    });
  }
}
```

Standalone route configuration

Listing 1.26: Standalone route composition for feature boundaries

```
import { Routes } from '@angular/router';

export const usersRoutes: Routes = [
  {
    path: '',
   loadComponent:() => import('./users_page.component').then((m) => m.UsersPageComponent),
  },
  {
    path: ':id',
   loadComponent:() => import('./user_details_page.component').then((m) => m.UserDetailsPageComponent),
  },
];
```

Reusable host directive for cross-cutting component behavior

Listing 1.27: Host directive composition pattern

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

@Directive({
  selector: '[appLoadingState]',
  standalone: true,
})
export class LoadingStateDirective {
  @Input('appLoadingState') isLoading = false;

  @HostBinding('class.is-loading')
  get applyLoadingClass(): boolean {
    return this.isLoading;
  }

  @HostBinding('attr.aria-busy')
  get ariaBusy(): string {
    return this.isLoading ? 'true' : 'false';
  }
}
```

1.9 Anti-Patterns to Call Out

- **God component:** one component orchestrates data loading, business rules, state mutation, and all rendering.
- **Hidden dependencies:** presentational component quietly injects global services.
- **Template complexity explosion:** deeply nested conditionals and duplicated markup.
- **Premature abstraction:** creating “shared” components before two real use cases exist.
- **Prop drilling:** chaining inputs through multiple intermediate components instead of using a service.
- **ViewChild abuse:** using `@ViewChild` to call methods on children instead of using Input/Output or services.
- **Constructor overloading:** placing initialization logic in the constructor instead of `ngOnInit` .

1.10 Key Takeaways

- Component quality is measured by boundary clarity and long-term maintainability.
- Standalone architecture is a practical default for modern Angular projects.
- Container/presentational separation is a useful scaling tool when complexity grows.
- OnPush change detection should be the default for presentational components.
- Lifecycle hooks have a defined order; `ngOnInit` and `ngOnDestroy` are the most critical to master.

- Content projection and template querying (`@ViewChild`, `@ContentChild`) are essential for building composable UI libraries.
- Host directives enable composition over inheritance for cross-cutting behaviors.
- Component testing should be shallow, input/output-focused, and use stable selectors.