Angular

A Complete 14-Chapter Course

Topics covered:

TypeScript & CLI · Components & Templates · Data Binding · Directives · Communication

Services & DI · Pipes · Template & Reactive Forms · Routing · HTTP & RxJS

Lifecycle Hooks · Standalone & Signals · Testing

Exercises: 42 hands-on challenges with sample solutions

Format: A4 · Dark-theme code examples · React-comparison tables

Table of Contents

CHAPTER 1 OF 14

What Angular Is, TypeScript Primer, CLI Setup

CHAPTER 1

What Angular Is, TypeScript Primer, CLI Setup

A different philosophy from React — a full framework instead of a library, TypeScript instead of plain JavaScript

React is deliberately a library: it renders UI, and everything else — routing, forms, HTTP, state management — is a separate choice (React Router, Zustand, fetch/React Query, all covered as add-ons across the React course). **Angular** is a full **framework**: routing, forms, dependency injection, and an HTTP client are all built in from day one, with one official, opinionated way to do each. It's also written in and built around **TypeScript** — JavaScript with optional type annotations — rather than plain JavaScript.

Just Enough TypeScript to Get Started

```
let age: number = 35;
let name: string = "Philip";

interface Person {
  name: string;
  age: number;
  email?: string;  // the ? marks this property optional
}

function greet(person: Person): string {
  return `Hello, ${person.name}!`;
}
```

`: type` annotations are the core addition — declaring what a variable, parameter, or return value is allowed to be, checked while writing code rather than only discovered when it crashes at runtime. An `interface` describes the shape an object must have; `greet`'s parameter being typed as `Person` means passing an object missing `name` or `age` is flagged immediately by the editor, before the code ever runs. Classes already exist in plain JavaScript (JS Intermediate Chapter 3) — TypeScript just lets a class's properties and methods carry the same type annotations.

TYPESCRIPT COMPILES TO JAVASCRIPT

None of this runs directly in a browser — TypeScript is compiled down to plain JavaScript before it does, with the type annotations stripped out entirely (they exist purely to help while writing and reading the code). Angular's tooling

> handles this compilation step automatically; there's no separate command to remember.

Installing the Angular CLI and Creating a Project

```
# install once, globally
npm install -g @angular/cli

# create a new project
ng new my-app
cd my-app
ng serve
```

`ng new my-app` asks a few setup questions (routing, stylesheet format) and scaffolds a complete starter project — Angular's CLI does considerably more for you upfront than Vite's React template did. `ng serve` starts a local dev server (usually `http://localhost:4200`) with live reload, the same role `npm run dev` played throughout the React course.

A Generated Project's Anatomy

- `src/main.ts` — the entry point; bootstraps the root component into the page.
- `src/app/app.component.ts` — the root component's logic (a TypeScript class).
- `src/app/app.component.html` — that same component's template (its markup) — Angular keeps logic and markup in separate files by default, unlike JSX combining both.
- `src/app/app.component.css` — styles scoped to just this component.
- `src/app/app.component.spec.ts` — a test file, generated automatically alongside every component (covered properly in Chapter 14).

A First Component

```ts
// greeting.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-greeting',
  standalone: true,
  templateUrl: './greeting.component.html',
})
export class GreetingComponent {
  name = 'Philip';
}
```

```html
<!-- greeting.component.html -->
<h1>Hello, {{ name }}!</h1>
```

`@Component({...})` is a **decorator** — metadata attached to the class right above it, telling Angular how to treat `GreetingComponent`: `selector` is the HTML tag name (`<app-greeting>`) used to place it elsewhere, and `templateUrl` points to its separate HTML file. `{{ name }}` in the template is **interpolation** — Angular's equivalent of JSX's `{name}`, inserting the class property's current value directly into the rendered output.

STANDALONE: TRUE IS THE MODERN DEFAULT

Older Angular tutorials build everything around `NgModule` — a separate declaration file grouping components together. Recent Angular versions (and a fresh `ng new` project today) default to **standalone components** instead, each one self-contained and importable directly without belonging to a module — the approach used throughout this entire course. If older documentation or Stack Overflow answers reference `NgModule`, that's the previous convention, not a mistake in what's taught here.

	REACT	ANGULAR
Category	Library (you choose the rest)	Full framework (routing/forms/HTTP built in)
Language	JavaScript (+ optional TS)	TypeScript by default
Markup	JSX — embedded in the same file as logic	Separate template file (or inline) per component
Embedding a value	`{value}`	`{{ value }}`

Coding Challenges

CHALLENGE 1

Install the Angular CLI, create a new project, and modify the default AppComponent so its template shows "Welcome, [your name]!" using interpolation of a class property.

View solution

CHALLENGE 2

Write a Person interface (name: string, age: number, email optional) and a function describePerson(person: Person): string returning a sentence describing them, called with at least one object missing the optional property.

View solution

CHALLENGE 3

Use the CLI (ng generate component) to create a new standalone component with its own template showing some static content, then use its selector tag inside AppComponent's template to display it.

View solution

CHAPTER 1 QUICK REFERENCE

- **Angular** — a full framework (routing/forms/HTTP built in), TypeScript-first
- **: type** annotations, **interface** — TypeScript's core additions over plain JS
- **ng new** / **ng serve** — CLI project creation and dev server, Angular's equivalent of Vite's commands
- **@Component({...})** — a decorator configuring a class as an Angular component
- **{{ value }}** — interpolation, Angular's equivalent of JSX's `{value}`
- **standalone: true** — the modern default, used throughout this course instead of older NgModule-based setup
- **Next chapter:** components and templates in depth

CHAPTER 2 OF 14

Components and Templates

CHAPTER 2

Components and Templates

Authoring a component's markup, scoping its styles, and composing several components into one page

Chapter 1's `GreetingComponent` used a separate template file. This chapter covers the rest of a component's anatomy — inline templates and styles as an alternative, how Angular keeps one component's CSS from leaking into another's, and nesting components together to build an actual page, the direct equivalent of composing JSX components in React (Fundamentals Chapter 9).

Inline Template and Styles

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

@Component({
  selector: 'app-badge',
  standalone: true,
  template: `<span class="badge">New</span>`,
  styles: [`.badge { background: gold; padding: 2px 8px; border-radius: 8px; }`],
})
export class BadgeComponent {}
```

`template` (a backtick string, allowing multiple lines) and `styles` (an array of strings) are direct alternatives to `templateUrl` / `styleUrl` — fine for a small enough component that separate files would be overkill. Larger components generally still use separate `.html` / `.css` files, which most editors syntax-highlight more usefully than a string embedded in TypeScript.

Style Encapsulation — Styles Don't Leak Between Components

```ts
// alert.component.ts
@Component({
  selector: 'app-alert',
  standalone: true,
  template: `<p class="box">Careful!</p>`,
  styles: [`.box { background: red; }`],
})
export class AlertComponent {}

// info.component.ts — a completely different .box, same class name
@Component({
  selector: 'app-info',
  standalone: true,
  template: `<p class="box">FYI</p>`,
  styles: [`.box { background: blue; }`],
})
export class InfoComponent {}
```

Both components use a class called `.box`, with completely different styling — and both render correctly, with no collision. By default, Angular scopes each component's styles so they only ever apply to that component's own template, achieved by attaching unique generated attributes behind the scenes. This is functionally similar to what CSS Modules or styled-components give a React project deliberately — Angular does it automatically, with no extra setup, for every component.

Composing Components Together

```ts
// app.component.ts
import { Component } from '@angular/core';
import { HeaderComponent } from './header/header.component';
import { FooterComponent } from './footer/footer.component';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [HeaderComponent, FooterComponent],
  templateUrl: './app.component.html',
})
export class AppComponent {}
```

```html
<!-- app.component.html -->
<app-header></app-header>
<main>
  <h1>Main content</h1>
</main>
<app-footer></app-footer>
```

Every standalone component used inside another's template must be listed in that component's own `imports` array — the explicit step Chapter 1's third challenge already required. `<app-header></app-header>` and `<app-footer></app-footer>` in the template then place those components exactly where written, the same nesting idea as composing `<Header />` and `<Footer />` in a React JSX tree.

```
AppComponent
├── HeaderComponent      // <app-header>
├── (main content, written directly in app.component.html)
└── FooterComponent      // <app-footer>
```

A FORGOTTEN IMPORT FAILS DIFFERENTLY THAN IN REACT

Forgetting to add a component to another's `imports` array doesn't throw a JavaScript import error — Angular's compiler instead reports that it doesn't recognize the unknown HTML tag (`app-header` isn't a known element), since as far as the template compiler is concerned, an unimported selector is indistinguishable from a typo or a real, unrecognized HTML element.

Coding Challenges

CHALLENGE 1

Create a component using inline template and styles (no separate .html/.css files) rendering a styled "Beta" tag, and use it inside AppComponent.

View solution

CHALLENGE 2

Build three components (Header, Sidebar, Footer), each with simple static content, and compose all three inside AppComponent's template to form a basic page layout.

View solution

CHALLENGE 3

Build two sibling components, each using the same CSS class name internally but styled completely differently, and use both inside AppComponent — confirming both render with their own correct styling and neither affects the other.

View solution

CHAPTER 2 QUICK REFERENCE

- **template** / **styles** — inline alternatives to `templateUrl` / `styleUrl`, fine for small components
- Angular **scopes each component's styles** automatically — identical class names in different components never collide
- A component used in another's template must be listed in that component's **imports** array
- A missing import shows as an "unrecognized element" error, not a JS import error
- Composing components by nesting their selector tags is the same idea as nesting JSX components in React
- **Next chapter:** data binding — property binding, event binding, and two-way binding with `ngModel`

CHAPTER 3 OF 14

Data Binding

CHAPTER 3

Data Binding — Property, Event, and Two-Way Binding

Connecting a component's class properties to its template in both directions

Interpolation (`{{ value }}` , Chapter 2) only inserts text content. Angular has three more binding forms covering everything else — setting a DOM property directly, responding to events, and the special case of keeping a form field and a class property in sync automatically.

Property Binding — [property]="expression"

```
<button [disabled]="isSaving">Save</button>
<img [src]="photoUrl" />
```

Square brackets around an attribute name — `[disabled]` , `[src]` — bind that DOM property directly to a class property or expression, evaluated as real TypeScript/JavaScript rather than treated as a plain string. `[disabled]="isSaving"` sets the button's actual `disabled` property to whatever `isSaving` currently is (a real boolean) — the same underlying need as React's curly-brace rule for non-string prop values (Fundamentals Chapter 2), just expressed with brackets instead.

Event Binding — (event)="handler()"

```
// counter.component.ts
@Component({
  selector: 'app-counter',
  standalone: true,
  template: `
    <p>Count: {{ count }}</p>
    <button (click)="increment()">+1</button>
  `,
})
export class CounterComponent {
  count = 0;

  increment() {
    this.count++;
  }
}
```

Parentheses around an event name — `(click)` — call the given expression whenever that event fires, the direct equivalent of React's `onClick={handler}`. `increment()` is a regular method on the class; `this.count++` updates the property directly — there's no separate setter function the way `useState` requires, since Angular's change detection automatically re-renders the template after any event handler runs.

Accessing the Event Object — $event

```
<input (input)="onInput($event)" />
```

```
onInput(event: Event) {
  const value = (event.target as HTMLInputElement).value;
  console.log(value);
}
```

`$event` is a special template variable holding the actual DOM event object — passing it explicitly to the handler is the same idea as React's automatic event-object argument (Fundamentals Chapter 4), just opted into rather than implicit. The `as HTMLInputElement` cast is TypeScript-specific: `event.target` is typed generically as `EventTarget`, so accessing its `.value` property requires telling the compiler exactly what kind of element it actually is.

Two-Way Binding — [(ngModel)]

```ts
// requires FormsModule, imported directly into the component
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'app-name-form',
  standalone: true,
  imports: [FormsModule],
  template: `
    <input [(ngModel)]="name" />
    <p>Hello, {{ name }}!</p>
  `,
})
export class NameFormComponent {
  name = '';
}
```

`[(ngModel)]="name"` — sometimes called "banana in a box," combining property binding's square brackets with event binding's parentheses — keeps the input's value and the `name` property synchronized automatically in both directions: typing updates `name`, and changing `name` elsewhere in code updates the input. This is the same end result as React's controlled input pattern (`value` + `onChange`, Fundamentals Chapter 7), collapsed into one directive instead of two explicit bindings.

NGMODEL NEEDS FORMSMODULE IMPORTED — EVERY TIME

`[(ngModel)]` only works once `FormsModule` is added to the component's own `imports` array — forgetting it produces a template compile error about `ngModel` not being a known property, the same category of "unrecognized" error from Chapter 2's missing-component-import warning, just for a built-in directive instead of a custom component this time.

Class and Style Bindings

```ts
<div [class.active]="isActive">...</div>
<div [style.color]="isError ? 'red' : 'black'">...</div>
```

`[class.active]` toggles a single CSS class on or off based on a boolean expression, and `[style.color]` binds one specific inline style property directly — the same conditional-styling need React handles with a template literal or object passed to `className` / `style` (Fundamentals Chapter 5).

ANGULAR	REACT EQUIVALENT
`{{ value }}`	`{value}`
`[property]="expr"`	`property={expr}`

ANGULAR	REACT EQUIVALENT
`(event)="handler()"`	`onEvent={handler}`
`[(ngModel)]="prop"`	`value={prop} onChange={...}`

Coding Challenges

CHALLENGE 1

Build a component with a boolean isLocked property and a button whose disabled state is property-bound to it, plus a second button (event-bound) that toggles isLocked.

View solution

CHALLENGE 2

Build a click counter component using event binding, with separate +1, -1, and Reset buttons, matching the equivalent React counter from Fundamentals Chapter 3.

View solution

CHALLENGE 3

Build a component with a text input two-way bound via ngModel to a name property, displaying "Hello, [name]!" live below it as the user types. Remember FormsModule.

View solution

CHAPTER 3 QUICK REFERENCE

- **[property]="expr"** — binds a real DOM property directly, not a plain string
- **(event)="handler()"** — calls an expression when that event fires
- **$event** — the actual event object, passed explicitly to a handler when needed
- **[(ngModel)]="prop"** — two-way binding; requires `FormsModule` in the component's `imports`
- **[class.x]** / **[style.y]** — bind a single class or style property conditionally
- No setter function needed — Angular's change detection re-renders after any event handler runs automatically
- **Next chapter:** built-in directives — *ngIf, *ngFor, ngClass, ngStyle

CHAPTER 4 OF 14

Built-in Directives

CHAPTER 4

Built-in Directives

Conditional rendering and lists directly in the template — Angular's answer to JSX's ternaries and .map()

React handles conditionals and lists with plain JavaScript embedded in JSX — ternaries, `&&`, and `.map()` (Fundamentals Chapters 5 and 6). Angular instead provides **directives**: special attributes recognized by the template compiler that add, remove, or repeat elements. This chapter covers the modern built-in control-flow syntax (`@if`, `@for`) plus the attribute directives for classes and styles.

@if — Conditional Rendering

```
@if (isLoggedIn) {
  <p>Welcome back!</p>
} @else {
  <p>Please log in.</p>
}
```

The `@if` block (introduced in Angular 17 as the modern built-in control flow) renders its contents only when the condition is true, with an optional `@else` block — the direct equivalent of React's ternary or `&&`, but reading more like the `if/else` statement it actually is. Unlike React's conditional rendering, which keeps the element in the JSX expression, `@if` genuinely adds or removes the element from the DOM entirely based on the condition.

@IF/@FOR REPLACED *NGIF/*NGFOR

Older Angular code (and most tutorials written before late 2023) uses the structural directives `*ngIf="condition"` and `*ngFor="let x of items"` as attributes on an element. They still work, but the newer `@if` / `@for` block syntax is now the recommended default — cleaner to read, and built into the template compiler rather than requiring `CommonModule` to be imported. This course uses the modern block syntax throughout; recognize the older `*ng` forms when you see them in existing code.

@for — Rendering a List

```
<ul>
  @for (item of items; track item.id) {
    <li>{{ item.name }}</li>
  }
</ul>
```

`@for` repeats its block once per array item — the equivalent of React's `.map()`. The `track item.id` clause is required, and serves exactly the same purpose as React's `key` prop (Fundamentals Chapter 6): it tells Angular how to identify each item across re-renders so it can update the DOM efficiently when the list changes, rather than rebuilding everything. Where React's `key` is optional-but-warned, Angular's `track` is mandatory.

@empty and the Loop Variables

```
@for (todo of todos; track todo.id) {
  <li>{{ todo.text }}</li>
} @empty {
  <li>No todos yet.</li>
}
```

An optional `@empty` block renders when the array has no items — neatly handling the "empty list" case that React's Todo project (Project 1) had to write as a separate conditional. Inside a `@for`, contextual variables like `$index`, `$first`, `$last`, and `$even` are also available for free — `{{ $index }}` gives the current item's position, for instance.

@switch — Multiple Cases

```
@switch (status) {
  @case ('loading') { <p>Loading...</p> }
  @case ('error') { <p>Something went wrong.</p> }
  @default { <p>Ready.</p> }
}
```

`@switch` handles three-or-more named cases cleanly — the same job React's lookup-object pattern did (Fundamentals Chapter 5), expressed as template-level branching. This is a natural fit for the loading/error/success status pattern that ran throughout the React projects.

ngClass and ngStyle — Multiple Classes or Styles at Once

```
<div [ngClass]="{ active: isActive, disabled: isDisabled }">...</div>
<div [ngStyle]="{ color: textColor, 'font-size.px': size }">...</div>
```

Chapter 3's `[class.active]` toggles one class; `[ngClass]` takes an object toggling several at once, each key a class name and each value a boolean deciding whether it applies. `[ngStyle]` does the same for multiple inline styles. Both require `NgClass` / `NgStyle` (or `CommonModule`) in the component's `imports` — unlike the new `@` control-flow blocks, these attribute directives still need importing.

@IF REMOVES FROM THE DOM — IT DOESN'T JUST HIDE

A `@if` that's false removes its element from the DOM completely, which also *destroys* any component inside it (and its state). If an element only needs to be visually hidden while keeping its state alive, a plain `[style.display]` or `[hidden]` binding is the right tool instead — `@if` is for genuinely adding/removing, not merely showing/hiding.

ANGULAR	REACT EQUIVALENT
`@if (x) { } @else { }`	Ternary / `&&`
`@for (x of items; track x.id) { }`	`items.map(...)` + `key`
`@empty { }`	A separate "empty list" conditional
`@switch / @case`	Lookup object for many states
`[ngClass]="{ ... }"`	Conditional `className`

Coding Challenges

CHALLENGE 1

Build a component with a boolean isLoggedIn property toggled by a button, using @if/@else to show either "Welcome back!" or "Please log in." accordingly.

View solution

CHALLENGE 2

Given an array of objects ({ id, name }), render them as a list with @for (using track on the id), including an @empty block showing "No items" when the array is cleared.

View solution

CHALLENGE 3

Build a component with a status property ("loading"/"error"/"success") cycled by a button, using @switch to render different content per status, and apply [ngClass] to color the message based on the same status.

View solution

CHAPTER 4 QUICK REFERENCE

- **@if / @else** — conditional rendering; genuinely adds/removes the element from the DOM
- **@for (x of items; track x.id)** — list rendering; `track` is mandatory (React's `key` equivalent)
- **@empty** — renders when the array is empty; **$index/$first/$last/$even** available inside `@for`
- **@switch / @case / @default** — clean branching for three-or-more named cases
- **[ngClass]** / **[ngStyle]** — toggle several classes/styles at once via an object (needs importing)
- The modern `@` blocks replaced the older `*ngIf` / `*ngFor` structural directives
- **Next chapter:** component communication — @Input, @Output, and EventEmitter

CHAPTER 5 OF 14

Component Communication

CHAPTER 5

Component Communication — @Input, @Output, EventEmitter

Passing data into a child and emitting events back up — Angular's equivalent of props and callback props

React passes data down through props and back up by passing functions down as props (Fundamentals Chapters 2 and 4). Angular splits these into two explicit, separately-named mechanisms: **@Input** for data flowing *into* a child, and **@Output** (with an **EventEmitter**) for events flowing *out* of a child back to its parent.

@Input — Passing Data Into a Child

```ts
// user-card.component.ts
import { Component, Input } from '@angular/core';

@Component({
  selector: 'app-user-card',
  standalone: true,
  template: `<h3>{{ name }}</h3><p>Age: {{ age }}</p>`,
})
export class UserCardComponent {
  @Input() name = '';
  @Input() age = 0;
}
```

```html
<!-- parent template -->
<app-user-card [name]="'Philip'" [age]="35"></app-user-card>
```

The `@Input()` decorator marks a property as one the parent is allowed to set — the equivalent of declaring a prop in React. The parent passes it using property binding from Chapter 3: `[name]="'Philip'"`. Note the binding evaluates as an expression, so a literal string needs inner quotes (`"'Philip'"`), whereas `[age]="35"` passes a real number — the same string-vs-expression distinction as React's quotes-vs-curly-braces rule for props.

@Output and EventEmitter — Emitting an Event Up

```ts
// todo-item.component.ts
import { Component, Input, Output, EventEmitter } from '@angular/core';

@Component({
  selector: 'app-todo-item',
  standalone: true,
  template: `
    <li>
      {{ text }}
      <button (click)="deleted.emit(id)">Delete</button>
    </li>
  `,
})
export class TodoItemComponent {
  @Input() id = 0;
  @Input() text = '';
  @Output() deleted = new EventEmitter<number>();
}
```

```html
<!-- parent template -->
<app-todo-item
  [id]="todo.id"
  [text]="todo.text"
  (deleted)="onDelete($event)"
></app-todo-item>
```

An `@Output()` property is an `EventEmitter` — the child calls `.emit(value)` on it to send data upward, and the parent listens with event binding (parentheses) exactly like a DOM event: `(deleted)="onDelete($event)"`, where `$event` holds whatever value was emitted. This is the structural equivalent of React's "pass a function down, child calls it" pattern (Project 1's todo delete), but Angular models it as a custom *event* the child raises rather than a callback function it receives. The `<number>` on `EventEmitter<number>` is a TypeScript generic, declaring the type of value this event carries.

> **THE NAMING CONVENTION MIRRORS DOM EVENTS**
>
> Because a parent listens to an `@Output` with the same `(eventName)` syntax used for native events like `(click)`, custom outputs are conventionally named as plain past-tense or noun events — `deleted`, `saved`, `valueChanged` — not `onDelete`. From the parent's side, `(deleted)="..."` then reads naturally alongside `(click)="..."`, as if the child were a built-in element raising its own events.

A Two-Way Binding Output Convention

If a component has an `@Input()` called `value` and an `@Output()` called `valueChange` (the input name plus `Change`), Angular lets a parent use the same `[(banana-in-a-box)]` two-way syntax from Chapter 3 on it — `[(value)]="something"`. This is exactly how the built-in `[(ngModel)]` works under the hood, and it's the standard way to build a custom component that supports two-way binding, rather than anything special baked into the framework.

INPUTS ARE NOT TRULY READ-ONLY THE WAY REACT PROPS ARE

A child *can* technically reassign an `@Input()` property's value locally, unlike React's strictly read-only props — but doing so is strongly discouraged, since the parent has no idea the value changed and the two will silently disagree. Treat inputs as read-from-parent only: to communicate a change back, emit an `@Output` event and let the parent update the source of truth, keeping the same one-way-data-flow discipline React enforces structurally.

ANGULAR	REACT EQUIVALENT
`@Input() name`	A prop
`[name]="value"`	Passing that prop
`@Output() saved = new EventEmitter()`	A callback prop (e.g. `onSave`)
`this.saved.emit(data)`	Calling that callback: `onSave(data)`
`(saved)="handle($event)"`	Passing the callback: `onSave={handle}`

Coding Challenges

CHALLENGE 1

Build a MovieCard component with @Input properties for title and year, rendering them as "Title (Year)". Use it three times in a parent with three different movies.

View solution

CHALLENGE 2

Build a TodoItem component with @Input id/text and an @Output deleted EventEmitter. The parent holds an array of todos, renders one TodoItem each, and removes the matching todo when a delete event is emitted.

View solution

CHALLENGE 3

Build a Counter child component with an @Input count and an @Output countChange EventEmitter, then have the parent use two-way binding [(count)] on it so changing the count inside the child updates the parent's value.

View solution

CHAPTER 5 QUICK REFERENCE

- **@Input() prop** — declares a property a parent can set (Angular's "prop")
- Parent passes it with property binding: `[prop]="value"`
- **@Output() ev = new EventEmitter<T>()** — declares a custom event the child can emit
- Child raises it with `this.ev.emit(data)`; parent listens with `(ev)="handler($event)"`
- Name outputs as events (`deleted`, `saved`), not `onX` — they read like native DOM events
- An `@Input value` + `@Output valueChange` pair enables `[(value)]` two-way binding
- **Next chapter:** services and dependency injection — sharing logic and state beyond parent/child

CHAPTER 6 OF 14

Services and Dependency Injection

CHAPTER 6

Services and Dependency Injection

Sharing logic and state across components without passing it through every layer — Angular's most distinctive feature

The React course needed Context (Intermediate Chapter 2) and eventually Zustand/Redux (Advanced Chapter 1) to share state across distant components without prop drilling. Angular has a single, built-in answer baked into the framework from the start: a **service** — an ordinary class holding shared logic or state — combined with **dependency injection (DI)**, which hands that service to any component that asks for it. This is arguably Angular's defining feature.

A Service Is Just a Class

```
// counter.service.ts
import { Injectable } from '@angular/core';

@Injectable({ providedIn: 'root' })
export class CounterService {
  private count = 0;

  increment() {
    this.count++;
  }

  getCount() {
    return this.count;
  }
}
```

The `@Injectable({ providedIn: 'root' })` decorator marks this class as something Angular's DI system can provide, and `'root'` means a **single shared instance** exists for the entire app — every component that asks for `CounterService` gets that same one instance, making it a natural place to hold shared state. (A service can also be provided at a narrower scope, but app-wide `'root'` is the common default.)

Injecting a Service Into a Component

```ts
// counter.component.ts
import { Component, inject } from '@angular/core';
import { CounterService } from './counter.service';

@Component({
  selector: 'app-counter',
  standalone: true,
  template: `
    <p>Count: {{ counter.getCount() }}</p>
    <button (click)="counter.increment()">+1</button>
  `,
})
export class CounterComponent {
  counter = inject(CounterService);
}
```

`inject(CounterService)` asks Angular's DI system for the service — the component never creates it with `new CounterService()` itself. That distinction is the whole point of dependency injection: the component *declares what it needs*, and the framework *supplies it*, automatically handing over the same shared `'root'` instance. Any other component injecting `CounterService` shares the exact same count, with no props, no Context provider, and nothing passed between them.

> **INJECT() VS CONSTRUCTOR INJECTION**
>
> Older Angular code requests services through the constructor instead: `constructor(private counter: CounterService) {}`. Both achieve identical results — the newer `inject()` function (used throughout this course) reads more cleanly and works in more places, but recognize the constructor form when you encounter it in existing codebases; it remains fully supported.

Why This Replaces Prop Drilling and Context

In React, sharing one piece of state between a header badge and a distant cart page required lifting state up and then either prop drilling or wrapping the tree in a Context provider (the exact journey the Shopping Cart project took). In Angular, both components simply `inject()` the same service — there's no provider to wrap anything in, no value object to memoize (Intermediate Chapter 7's concern), and no tree structure that the shared state has to flow through. The service exists independently of the component tree entirely.

Services for Logic, Not Just State

```ts
// logger.service.ts — a stateless service, purely for shared behavior
@Injectable({ providedIn: 'root' })
export class LoggerService {
  log(message: string) {
    console.log(`[${new Date().toISOString()}] ${message}`);
  }
}
```

A service doesn't have to hold state — it's equally the home for shared *behavior*: logging, formatting, calculations, and (most importantly) talking to a backend API, which the HTTP client chapter builds on directly. Keeping that logic in an injectable service rather than inside components is a core Angular convention: components handle the view, services handle everything else.

A PLAIN GETCOUNT() IN A TEMPLATE WON'T ALWAYS REFLECT ASYNC CHANGES CLEANLY

Calling a method like `counter.getCount()` directly in a template works for state changed synchronously by user events (Angular re-checks the template after each event). But for state that changes asynchronously — a timer, an API response — exposing the value as an Observable and using the `async` pipe (Chapters 7 and 11), or Angular's newer signals (Chapter 13), is the more robust pattern. The plain-method approach here is the simplest starting point, deliberately, before those tools are introduced.

SHARING NEED	REACT APPROACH	ANGULAR APPROACH
Nearby components	Lift state up + props	A shared service (or @Input/@Output)
Distant components	Context, or Zustand/Redux	A service injected into both
Shared behavior/API calls	A custom hook / util module	An injectable service

Coding Challenges

CHALLENGE 1

Build a CounterService (providedIn: 'root') holding a count with increment/decrement/reset methods, and inject it into a single component that displays and changes the count via buttons.

View solution

CHALLENGE 2

Inject the same CounterService into two completely separate, unrelated components — one that increments the count and one that only displays it — confirming both reflect the same shared value with nothing passed between them.

View solution

CHALLENGE 3

Build a stateless LoggerService with a log(message) method that prefixes a timestamp, and inject it into a component that calls it from a button click.

View solution

CHAPTER 6 QUICK REFERENCE

- **A service** — an ordinary class for shared state or behavior, marked `@Injectable({ providedIn: 'root' })`
- **providedIn: 'root'** — one shared instance for the whole app
- **inject(ServiceClass)** — asks Angular's DI to supply the service; never `new` it yourself
- Two distant components injecting the same service share its state — no provider, no prop drilling, no Context
- Services are also the home for shared behavior: logging, formatting, and (next) API calls
- Older code injects via the constructor (`constructor(private x: X)`) — equivalent to `inject()`
- **Next chapter:** pipes — transforming displayed values in the template

CHAPTER 7 OF 14

Pipes

CHAPTER 7

Pipes — Built-in and Custom

Transforming a value for display, right inside the template, without changing the underlying data

In React, formatting a value for display means calling a function in the JSX — `{formatPrice(amount)}`, `{date.toLocaleDateString()}`. Angular has a dedicated template feature for this: a **pipe**, applied with the `|` symbol, transforming a value purely for display while leaving the original data untouched. Several common transformations ship built in, and writing custom ones is straightforward.

Built-in Pipes

```
<p>{{ name | uppercase }}</p>          <!-- PHILIP -->
<p>{{ price | currency:'EUR' }}</p>    <!-- €19.99 -->
<p>{{ today | date:'fullDate' }}</p>   <!-- Tuesday, June 23, 2026 -->
<p>{{ ratio | percent }}</p>           <!-- 42% -->
<p>{{ data | json }}</p>               <!-- debug: prints the object as JSON -->
```

A pipe takes the value on its left and transforms it for display. Arguments come after a colon — `currency:'EUR'` sets the currency code, `date:'fullDate'` picks a format. The original `name`, `price`, and `today` properties are never modified; only what appears on screen is transformed. The `json` pipe is especially handy while learning — it's the quickest way to dump an object's contents into the template to see its shape.

Chaining Pipes

```
<p>{{ name | uppercase | slice:0:3 }}</p>   <!-- "PHILIP" -> "PHI" -->
```

Multiple pipes can be chained with successive `|` symbols, each receiving the output of the one before it — applied left to right, the same direction as reading. Here `uppercase` runs first, then `slice` takes the first three characters of the result.

MOST BUILT-IN PIPES NEED IMPORTING

Pipes like `CurrencyPipe`, `DatePipe`, `UpperCasePipe`, and `JsonPipe` live in `@angular/common` and must be added to the component's `imports` array (individually, or via `CommonModule`) before use — the same "import what the template uses" rule as `ngClass` from Chapter 4. Forgetting produces the familiar "unknown pipe" template error.

Writing a Custom Pipe

```
// truncate.pipe.ts
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({ name: 'truncate', standalone: true })
export class TruncatePipe implements PipeTransform {
  transform(value: string, limit = 20): string {
    if (value.length <= limit) return value;
    return value.slice(0, limit) + '…';
  }
}
```

```
<!-- usage, after importing TruncatePipe into the component -->
<p>{{ longText | truncate:50 }}</p>
```

A custom pipe is a class implementing `PipeTransform` — a single `transform(value, ...args)` method returning the transformed result. The `@Pipe` decorator's `name` is what's used in the template (`| truncate`), and any extra parameters after the value become the pipe's arguments (`truncate:50` passes `50` as `limit`). It's a standalone pipe, imported into a component's `imports` array exactly like a standalone component.

A PIPE IS THE TEMPLATE-FRIENDLY VERSION OF A CUSTOM HOOK'S SPIRIT

A custom pipe and a React formatting function solve the same problem — reusable display transformation. Angular's version is declarative and lives in the template (`| truncate:50` reads cleanly inline), and being a class registered with the framework, the same pipe is trivially reusable across every component that imports it, without re-importing a utility function in each file.

The async Pipe — A Preview

```
<p>{{ user$ | async | json }}</p>
```

One built-in pipe deserves a special mention now: `async`. It subscribes to an Observable (or Promise) and renders its latest emitted value, automatically — directly addressing the async-state limitation flagged in Chapter 6's warning. It comes into its own with the HTTP client and RxJS in Chapter 11, but it's worth recognizing here as a pipe, since that's exactly what it is.

ANGULAR PIPE	REACT EQUIVALENT
`{{ x | uppercase }}`	`{x.toUpperCase()}`
`{{ x | currency:'EUR' }}`	`{formatCurrency(x)}`
`{{ x | date:'fullDate' }}`	`{x.toLocaleDateString(...)}`
`{{ x | myCustomPipe }}`	`{myFormatFunction(x)}`

Coding Challenges

CHALLENGE 1

Build a component displaying a name (uppercase pipe), a price (currency pipe), and today's date (date pipe with a readable format), remembering to import the needed pipes.

View solution

CHALLENGE 2

Write a custom TruncatePipe (with a configurable limit argument defaulting to 20, appending an ellipsis when it truncates) and use it on a long string with an explicit limit.

View solution

CHALLENGE 3

Write a custom FileSizePipe converting a number of bytes into a human-readable string (e.g. 1536 -> "1.5 KB", 2097152 -> "2 MB"), and use it on a few different byte values.

View solution

CHAPTER 7 QUICK REFERENCE

- **A pipe** — `{{ value | pipeName }}` — transforms a value for display, leaving the data unchanged
- Arguments follow a colon: `currency:'EUR'`, `date:'fullDate'`, `slice:0:3`
- Chain pipes with successive `|` symbols, applied left to right
- Built-in pipes (currency/date/uppercase/json...) live in `@angular/common` and must be imported
- **Custom pipe** — a class with `@Pipe({ name })` implementing `transform(value, ...args)`
- **async** pipe — subscribes to an Observable/Promise and renders its latest value (Chapter 11)

- **Next chapter:** template-driven forms

CHAPTER 8 OF 14

Template-Driven Forms

CHAPTER 8

Template-Driven Forms

Building forms whose logic lives mostly in the template, with validation handled by directives

Angular has two distinct, official approaches to forms — **template-driven** (this chapter) and **reactive** (next chapter). Template-driven forms keep most of the form's setup in the HTML template itself, built on the `ngModel` two-way binding from Chapter 3. They're the quicker option for simpler forms, and the closer fit to how React's controlled inputs felt.

The Setup — ngForm and ngModel Together

```ts
// signup.component.ts
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'app-signup',
  standalone: true,
  imports: [FormsModule],
  templateUrl: './signup.component.html',
})
export class SignupComponent {
  model = { name: '', email: '' };

  onSubmit() {
    console.log(this.model);
  }
}
```

```html
<!-- signup.component.html -->
<form #signupForm="ngForm" (ngSubmit)="onSubmit()">
  <input name="name" [(ngModel)]="model.name" required />
  <input name="email" [(ngModel)]="model.email" required email />
  <button type="submit" [disabled]="signupForm.invalid">Sign up</button>
</form>
```

Three pieces work together here. `#signupForm="ngForm"` is a **template reference variable** — it captures the form's auto-created `NgForm` object, giving access to its overall validity (`signupForm.invalid`). `(ngSubmit)` fires the handler on submit, already preventing the native page reload that React needed `preventDefault()` for (Fundamentals Chapter 4). Each `[(ngModel)]` input needs a `name` attribute so the form can track it as a named control.

Validation Through Directives

Validation is declared as plain attributes on the inputs — `required`, `email`, `minlength`, `maxlength`, `pattern`. Angular wires these standard-looking HTML attributes into its own validation system, tracking each control's validity and the overall form's. `[disabled]="signupForm.invalid"` then disables the submit button until every validator passes — declarative validation with no JavaScript validation code written at all.

Showing Validation Messages

```
<input name="email" #email="ngModel" [(ngModel)]="model.email" required email />

@if (email.invalid && email.touched) {
  <small>A valid email is required.</small>
}
```

A per-input reference variable — `#email="ngModel"` — captures that single control's state, exposing flags like `invalid`, `valid`, `touched` (the user has focused then left it), and `dirty` (the value has changed). Combining `invalid && touched` with the `@if` from Chapter 4 shows an error message only after the user has actually interacted with the field — not immediately on an untouched, empty form.

TOUCHED AND DIRTY MIRROR GOOD FORM UX

These control-state flags exist precisely to support the standard "don't yell at the user before they've done anything" pattern. `touched` waits until they've left a field; `dirty` waits until they've changed it. React projects had to track this kind of "has the user interacted yet" state manually — Angular's forms provide it built in, on every control.

EVERY NGMODEL INPUT NEEDS A UNIQUE NAME ATTRIBUTE

Inside a form, a `[(ngModel)]` control without a `name` attribute (or with a duplicate one) throws a runtime error — the form uses `name` as the key to register and track each control. This is easy to forget when copying the simpler standalone `ngModel` usage from Chapter 3, which worked without a `name` because it wasn't inside an `ngForm`.

ANGULAR TEMPLATE-DRIVEN	REACT CONTROLLED-FORM EQUIVALENT
`[(ngModel)]="model.x"`	`value` + `onChange`

ANGULAR TEMPLATE-DRIVEN	REACT CONTROLLED-FORM EQUIVALENT
`(ngSubmit)="onSubmit()"`	`onSubmit` + `preventDefault()`
`required / email / minlength`	Hand-written validation logic
`control.touched / .dirty`	Manually tracked "interacted" state

Coding Challenges

CHALLENGE 1

Build a template-driven form with name and email fields (both required, email also using the email validator), a model object, and a submit handler that logs the model — with the submit button disabled while the form is invalid.

View solution

CHALLENGE 2

Add per-field validation messages to the form, each shown only when that field is both invalid and touched, using a per-input #ref="ngModel" reference variable and @if.

View solution

CHALLENGE 3

Build a template-driven "create account" form with a username (required, minlength 3) and a password (required, minlength 8), showing the specific reason a field is invalid (e.g. "too short" vs "required") by checking the control's errors.

View solution

CHAPTER 8 QUICK REFERENCE

- Template-driven forms build on `[(ngModel)]` and require **FormsModule**
- **#form="ngForm"** — a template ref capturing the whole form's state (e.g. `form.invalid`)
- **(ngSubmit)** — submit handler; already prevents the native page reload
- Validation is declared as attributes: `required`, `email`, `minlength`, `pattern`
- **#field="ngModel"** — a per-control ref exposing `invalid` / `touched` / `dirty` / `errors`
- Every `ngModel` control inside a form needs a unique `name` attribute

- **Next chapter:** reactive forms — the same goals, but defined in the component class instead

CHAPTER 9 OF 14

Reactive Forms

CHAPTER 9

Reactive Forms

The same forms, defined explicitly in the component class — more code, far more control

Template-driven forms (Chapter 8) put the form's structure in the HTML, with Angular inferring the model behind the scenes. **Reactive forms** invert that: the form is defined explicitly in the component class as an object you build and control directly, with the template just connecting to it. More setup, but the form becomes a real, inspectable, programmatically-controllable object — the better fit for complex forms, dynamic fields, and custom validation.

Building a FormGroup

```
// login.component.ts
import { Component } from '@angular/core';
import { FormGroup, FormControl, Validators, ReactiveFormsModule } from '@angular/forms';

@Component({
  selector: 'app-login',
  standalone: true,
  imports: [ReactiveFormsModule],
  templateUrl: './login.component.html',
})
export class LoginComponent {
  loginForm = new FormGroup({
    email: new FormControl('', [Validators.required, Validators.email]),
    password: new FormControl('', [Validators.required, Validators.minLength(8)]),
  });

  onSubmit() {
    console.log(this.loginForm.value);
  }
}
```

A `FormGroup` is the whole form; each field is a `FormControl` with its initial value and an array of validators. Crucially, this entire structure is a plain object in the class — readable and writable from code. The validators come from `Validators` (`required`, `email`, `minLength(8)`, `pattern(...)`) rather than being template attributes, and uses `ReactiveFormsModule` instead of `FormsModule`.

Connecting the Template

```
<!-- login.component.html -->
<form [formGroup]="loginForm" (ngSubmit)="onSubmit()">
  <input formControlName="email" />
  <input formControlName="password" type="password" />
  <button type="submit" [disabled]="loginForm.invalid">Log in</button>
</form>
```

`[formGroup]="loginForm"` binds the template's form to the object built in the class, and each input declares which control it represents with `formControlName="email"`. Notice there's no `[(ngModel)]` anywhere — the form object is the single source of truth, and the inputs simply attach to its existing controls by name. `loginForm.invalid` works exactly as before for the submit button.

FormBuilder — Less Boilerplate

```
import { FormBuilder, Validators } from '@angular/forms';
import { inject } from '@angular/core';

export class LoginComponent {
  private fb = inject(FormBuilder);

  loginForm = this.fb.group({
    email: ['', [Validators.required, Validators.email]],
    password: ['', [Validators.required, Validators.minLength(8)]],
  });
}
```

`FormBuilder` — injected as a service (Chapter 6) — is a shorthand for building the same structure with less `new FormGroup` / `new FormControl` repetition: each field becomes a terse `[initialValue, validators]` array. It produces an identical `FormGroup`; it's purely the more concise and conventional way to write one, and what most real reactive-forms code uses.

Reacting to Value Changes

```
this.loginForm.controls.email.valueChanges.subscribe((value) => {
  console.log('email changed to:', value);
});
```

Every control (and the form itself) exposes a `valueChanges` **Observable** that emits each time its value changes — the core reason these are called *reactive* forms. This is what makes things like live-filtering,

dependent fields, or debounced validation natural, in a way template-driven forms can't match cleanly. Observables are covered properly in Chapter 11; this is a first glimpse of why reactive forms unlock so much.

Programmatic Control

```
this.loginForm.setValue({ email: 'a@b.com', password: 'secret123' });  // set all fields
this.loginForm.patchValue({ email: 'new@b.com' });  // set just some
this.loginForm.reset();  // clear everything back to initial
```

Because the form is a real object, it can be manipulated from code — `setValue` / `patchValue` to fill it (e.g. loading an existing record into an edit form), `reset()` to clear it. This programmatic control is the practical payoff of reactive forms, and the main reason they're preferred for anything beyond a simple contact form.

DON'T MIX THE TWO FORM APPROACHES ON THE SAME FORM

A single form should be entirely template-driven (`[(ngModel)]` + `FormsModule`) or entirely reactive (`formControlName` + `ReactiveFormsModule`) — mixing `[(ngModel)]` and `formControlName` on controls within the same form causes confusing conflicts. Pick one approach per form: template-driven for simple cases, reactive when the extra control is genuinely needed.

	TEMPLATE-DRIVEN (CH 8)	REACTIVE (CH 9)
Form defined in	The template (HTML)	The component class
Module	`FormsModule`	`ReactiveFormsModule`
Binding	`[(ngModel)]`	`formControlName`
Validators	Template attributes	`Validators` in the class
Best for	Simple forms	Complex/dynamic forms, programmatic control

Coding Challenges

CHALLENGE 1

Rebuild Chapter 8's signup form (name + email, both required, email validated) as a reactive form using FormGroup/FormControl, logging the form's value on submit and disabling the button while invalid.

View solution

CHALLENGE 2

Rewrite the same form using FormBuilder (fb.group) instead of new FormGroup/new FormControl, and add per-field validation messages reading each control's errors and touched state.

View solution

CHALLENGE 3

Build a reactive form with a "Fill demo data" button (using patchValue) and a "Reset" button (using reset()), plus subscribe to one field's valueChanges to log its value live as the user types.

View solution

CHAPTER 9 QUICK REFERENCE

- Reactive forms define the form in the **class**; require **ReactiveFormsModule**
- **FormGroup** = the whole form; **FormControl** = one field (initial value + validators)
- Template connects with **[formGroup]** and **formControlName** — no `ngModel`
- **Validators** (required/email/minLength/pattern) come from the class, not template attributes
- **FormBuilder** (`fb.group({...})`) — the concise, conventional way to build the same structure
- **valueChanges** Observable, plus **setValue/patchValue/reset** — programmatic power over the form
- Never mix `ngModel` and `formControlName` on the same form
- **Next chapter:** routing — RouterModule, route params, and guards

CHAPTER 10 OF 14

Routing

CHAPTER 10

Routing — Routes, Params, and Guards

Multiple pages in a single-page app — built in, where React needed React Router as a separate library

Routing was a separate library install in React (React Router, Intermediate Chapter 5). In Angular it's part of the framework — the same single-page navigation concepts apply directly, just with Angular's own API. This chapter covers defining routes, linking and navigating, reading URL parameters, and protecting routes with guards.

Defining Routes

```
// app.routes.ts
import { Routes } from '@angular/router';
import { HomeComponent } from './home.component';
import { AboutComponent } from './about.component';

export const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'product/:id', component: ProductDetailComponent },
  { path: '**', component: NotFoundComponent },
];
```

Routes are an array of objects mapping a `path` to a `component` — the direct equivalent of React Router's `<Route>` elements. `:id` marks a dynamic segment (matching `/product/anything`), and `'**'` is the catch-all for any unmatched URL — both familiar from React Router, just expressed as data rather than JSX. This array is wired into the app's configuration once, in `main.ts`, via `provideRouter(routes)`.

RouterOutlet and RouterLink

```ts
// app.component.ts
import { RouterOutlet, RouterLink } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, RouterLink],
  template: `
    <nav>
      <a routerLink="/">Home</a>
      <a routerLink="/about">About</a>
    </nav>
    <router-outlet></router-outlet>
  `,
})
export class AppComponent {}
```

`<router-outlet>` is the placeholder where the matched route's component renders — the equivalent of React Router's `<Outlet />` (or the `<Routes>` block itself). `routerLink="/about"` navigates without a full page reload, exactly like React's `<Link to>` — and the same warning applies: a plain `<a href>` would trigger a real reload and defeat the point.

Reading Route Parameters

```ts
// product-detail.component.ts
import { Component, inject } from '@angular/core';
import { ActivatedRoute } from '@angular/router';

export class ProductDetailComponent {
  private route = inject(ActivatedRoute);
  id = this.route.snapshot.paramMap.get('id');
}
```

`ActivatedRoute` — injected like any service (Chapter 6) — exposes the current route's parameters. `snapshot.paramMap.get('id')` reads the `:id` segment from the URL, the equivalent of React Router's `useParams()`. The `snapshot` form is the simplest; for a route that the same component stays on while only the parameter changes, subscribing to `route.paramMap` (an Observable, Chapter 11) reacts to those changes — but snapshot suffices for most cases.

Programmatic Navigation

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

export class SomeComponent {
  private router = inject(Router);

  goToProduct(id: number) {
    this.router.navigate(['/product', id]);
  }
}
```

Injecting the `Router` service gives `navigate([...])` for navigating from code — after a form submits, a login succeeds, an action completes — the equivalent of React Router's `useNavigate()`. The path is passed as an array of segments, which Angular joins into the URL.

Route Guards — Protecting a Route

```
// auth.guard.ts
import { CanActivateFn, Router } from '@angular/router';
import { inject } from '@angular/core';

export const authGuard: CanActivateFn = () => {
  const auth = inject(AuthService);
  const router = inject(Router);

  if (auth.isLoggedIn()) return true;
  return router.createUrlTree(['/login']);  // redirect instead of allowing access
};
```

```
// in the routes array
{ path: 'dashboard', component: DashboardComponent, canActivate: [authGuard] }
```

A **guard** is a function that runs before a route activates, returning `true` to allow it or redirecting otherwise — there's no built-in equivalent in React Router itself (it's typically hand-rolled with a wrapper component). Added to a route via `canActivate: [authGuard]`, this is the standard way to keep an unauthenticated user out of a protected page, redirecting them to login instead. Guards inject services freely, since they run inside Angular's DI context.

LAZY LOADING IS BUILT INTO THE ROUTE CONFIG

Angular pairs code splitting (the React equivalent was Advanced Chapter 3's `lazy` / `Suspense`) directly with routing: `{ path: 'admin', loadComponent: () => import('./admin.component').then(m => m.AdminComponent) }` loads that component's code only when its route is first visited. No separate `Suspense` boundary is needed — the router handles the loading transition itself.

ROUTE ORDER MATTERS — PUT THE WILDCARD LAST

The router matches routes top to bottom and stops at the first match, so the catch-all `'**'` route must be the *last* entry in the array — placed earlier, it would match everything and prevent any route below it from ever being reached. The same ordering discipline applied to React Router's `path="*"`.

ANGULAR	REACT ROUTER EQUIVALENT
`routes array + provideRouter`	`<Routes>` / `<Route>`
`<router-outlet>`	`<Outlet />`
`routerLink="/x"`	`<Link to="/x">`
`ActivatedRoute paramMap`	`useParams()`
`Router.navigate([...])`	`useNavigate()`
`canActivate guard`	Hand-rolled protected-route wrapper

Coding Challenges

CHALLENGE 1

Set up a 3-page app (Home, About, Contact) with a routes array, a nav using routerLink, a router-outlet, and a wildcard route showing a NotFound component for unmatched URLs.

View solution

CHALLENGE 2

Add a product/:id route. From a product list, use routerLink (or Router.navigate) to go to a detail page that reads the id via ActivatedRoute and displays the matching product.

View solution

CHALLENGE 3

Build an authGuard (CanActivateFn) backed by a simple AuthService with an isLoggedIn flag, protecting a /dashboard route — redirecting to /login when not logged in — and a button toggling the logged-in state to test both outcomes.

View solution

CHAPTER 10 QUICK REFERENCE

- **routes array** (path → component), wired via `provideRouter(routes)` — built into Angular
- **<router-outlet>** renders the matched route; **routerLink** navigates without reload
- **:id** dynamic segments read via `ActivatedRoute.snapshot.paramMap.get('id')`
- **Router.navigate([...])** — programmatic navigation from code
- **canActivate: [guard]** — a function gating a route, allowing or redirecting
- **loadComponent** in a route — lazy-loads that component's code on first visit
- The `'**'` wildcard route must be last; matching is top-to-bottom
- **Next chapter:** HTTP client and RxJS — talking to a backend with Observables

CHAPTER 11 OF 14

HTTP Client and RxJS Basics

CHAPTER 11

HTTP Client and RxJS Basics

Talking to a backend the Angular way — with Observables, not Promises

React used `fetch` returning a Promise, awaited in a `useEffect` (Intermediate Chapter 6). Angular's `HttpClient` instead returns an **Observable** — a stream of values from RxJS, the reactive library Angular is built on. The same Observable already appeared as `valueChanges` (Chapter 9) and `route.paramMap` (Chapter 10); this chapter covers it properly, alongside the HTTP client that produces them most often.

Observable vs Promise — The Core Difference

A Promise resolves **once** with a single value. An **Observable** is a stream that can emit *many* values over time (or just one, like an HTTP response) — and crucially, it does nothing at all until something **subscribes** to it. Where `await` kicks off a Promise immediately, an Observable is lazy: no subscription, no work. That laziness is what makes operators like cancellation, retrying, and debouncing possible in ways Promises can't match.

Setting Up HttpClient

```
// main.ts — provide it once for the whole app
import { provideHttpClient } from '@angular/common/http';

bootstrapApplication(AppComponent, {
  providers: [provideHttpClient()],
});
```

`provideHttpClient()` registers the HTTP client with Angular's DI system once, the same pattern as `provideRouter` (Chapter 10). After that, any service can inject `HttpClient` and make requests.

An API Call Inside a Service

```ts
// user.service.ts
import { Injectable, inject } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

interface User { id: number; name: string; }

@Injectable({ providedIn: 'root' })
export class UserService {
  private http = inject(HttpClient);

  getUsers(): Observable<User[]> {
    return this.http.get<User[]>('https://api.example.com/users');
  }
}
```

Per the Chapter 6 convention, API calls live in a service, not a component. `http.get<User[]>(url)` returns an `Observable<User[]>` — the `<User[]>` generic tells TypeScript what shape the response data will be, giving full type safety on the result. Note the service just *returns* the Observable without subscribing; the component decides when to do that.

Consuming It — Subscribe, or the async Pipe

```ts
// option A: subscribe manually in the component
export class UserListComponent implements OnInit {
  private userService = inject(UserService);
  users: User[] = [];

  ngOnInit() {
    this.userService.getUsers().subscribe((users) => {
      this.users = users;
    });
  }
}
```

```ts
// option B: let the async pipe subscribe (and unsubscribe) for you
export class UserListComponent {
  private userService = inject(UserService);
  users$ = this.userService.getUsers();  // just the Observable, not subscribed
}
```

```
<!-- template for option B -->
@if (users$ | async; as users) {
  <ul>
    @for (user of users; track user.id) {
      <li>{{ user.name }}</li>
    }
  </ul>
}
```

The `async` pipe (Chapter 7) subscribes to the Observable, hands its emitted value to the template, and — critically — **unsubscribes automatically** when the component is destroyed. This is the preferred approach: it avoids manual subscription bookkeeping entirely, and the `; as users` syntax captures the emitted array into a template variable to loop over. The naming convention `users$` (trailing `$`) marks a property as an Observable at a glance.

RxJS Operators with pipe()

```
import { map, catchError } from 'rxjs';
import { of } from 'rxjs';

getUsers(): Observable<User[]> {
  return this.http.get<User[]>(url).pipe(
    map((users) => users.filter((u) => u.name)),  // transform the stream
    catchError((err) => of([]))            // on error, emit an empty array instead
  );
}
```

An Observable's `.pipe(...)` applies **operators** — functions that transform the stream — much like chaining array methods, but for values arriving over time. `map` transforms each emitted value (here filtering the user list), and `catchError` handles failures, returning a fallback Observable (`of([])` emits a single empty array). This is RxJS's equivalent of the try/catch error handling from React's Intermediate Chapter 6, expressed as a pipeline.

THIS IS WHY REACTIVE FORMS FELT POWERFUL

Chapter 9's `valueChanges` Observable plus these operators is exactly what enables debounced, live-reacting forms: `searchControl.valueChanges.pipe(debounceTime(300), switchMap(term => this.api.search(term)))` is the entire debounced-search-with-cancellation pattern that took React a custom `useDebounce` hook plus careful race-condition handling (Projects 4 and Intermediate Ch 6) — here it's a few composed operators.

MANUAL SUBSCRIPTIONS NEED MANUAL CLEANUP

A manual `.subscribe()` (option A) that isn't unsubscribed can leak memory and keep running after a component is gone — the same category of issue as a React `useEffect` without a cleanup function (Fundamentals Chapter 8). The

`async` pipe sidesteps this entirely by cleaning up for you, which is the main reason to prefer it. When a manual subscription is genuinely needed, unsubscribe in `ngOnDestroy` (next chapter) — HTTP requests are a partial exception, as they complete after one emission, but it's safest to treat all subscriptions as needing cleanup.

ANGULAR	REACT EQUIVALENT
`http.get<T>(url)`	`fetch(url).then(r => r.json())`
`Returns an Observable`	Returns a Promise
`async pipe in template`	Manual loading/data state in `useState`
`.pipe(map, catchError)`	Transforms + try/catch
`debounceTime + switchMap`	Custom `useDebounce` + race handling

Coding Challenges

CHALLENGE 1

Set up provideHttpClient, build a service that fetches a list from a free public API returning an Observable, and a component that subscribes in ngOnInit and stores the result for display.

View solution

CHALLENGE 2

Rewrite Challenge 1's component to use the async pipe instead of a manual subscription — exposing the Observable directly (users$) and consuming it in the template with @if (users$ | async; as users).

View solution

CHALLENGE 3

Add .pipe() to the service method with a map operator transforming the data (e.g. extracting just the names) and a catchError operator returning an empty array on failure, then display the transformed result.

View solution

CHAPTER 11 QUICK REFERENCE

- **HttpClient** returns an **Observable**, not a Promise — provide it with `provideHttpClient()`

- An Observable is lazy: it does nothing until **subscribed**; it can emit many values over time
- API calls belong in a **service** (Chapter 6); the service returns the Observable unsubscribed
- **async pipe** — subscribes, renders the value, and unsubscribes automatically (the preferred way)
- **.pipe(map, catchError, ...)** — RxJS operators transforming the stream
- Convention: an Observable property ends in `$` (e.g. `users$`)
- Manual subscriptions need cleanup (`ngOnDestroy`) — the async pipe avoids that entirely
- **Next chapter:** lifecycle hooks — ngOnInit, ngOnChanges, ngOnDestroy

CHAPTER 12 OF 14

Lifecycle Hooks

CHAPTER 12

Lifecycle Hooks

Running code at specific moments in a component's life — Angular's answer to useEffect's timing

React's `useEffect` (Fundamentals Chapter 8) collapsed "run on mount," "run on update," and "clean up on unmount" into one hook differentiated by its dependency array. Angular splits these into separate, explicitly-named **lifecycle hook** methods — each a method with a fixed name that Angular calls at a specific moment. A component opts into one by implementing the matching interface and writing the method.

ngOnInit — Setup After Creation

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

export class UserListComponent implements OnInit {
  private userService = inject(UserService);
  users: User[] = [];

  ngOnInit() {
    this.userService.getUsers().subscribe((u) => (this.users = u));
  }
}
```

`ngOnInit` runs once, right after Angular has created the component and set its initial `@Input` values — the standard place for setup work like an initial data fetch (used already in Chapter 11). It's the direct equivalent of React's `useEffect(() => {...}, [])` with an empty dependency array. Implementing `OnInit` isn't strictly required for the method to run, but doing so lets TypeScript catch a misspelled `ngOnInit` — strongly recommended.

Why Not the Constructor?

A natural question: why not just do setup in the class constructor? The constructor runs when the object is first created, *before* Angular has finished wiring it up — specifically before `@Input` properties have their values. Putting data fetching or any logic depending on inputs in `ngOnInit` guarantees those inputs are ready. The

convention: the constructor is for dependency injection only (or just use `inject()`); `ngOnInit` is for actual initialization logic.

ngOnChanges — Responding to Input Changes

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

export class ChartComponent implements OnChanges {
  @Input() data: number[] = [];

  ngOnChanges(changes: SimpleChanges) {
    if (changes['data']) {
      console.log('data changed from', changes['data'].previousValue,
                  'to', changes['data'].currentValue);
    }
  }
}
```

`ngOnChanges` runs whenever an `@Input` value changes (and once initially, before `ngOnInit`) — the rough equivalent of `useEffect` with a specific prop in its dependency array. The `SimpleChanges` argument is an object keyed by which inputs changed, each entry holding `previousValue` and `currentValue` — useful when a component needs to react to *which* input changed and by how much, like recomputing a chart only when its data prop actually changes.

ngOnDestroy — Cleanup Before Removal

```
import { OnDestroy } from '@angular/core';
import { Subscription } from 'rxjs';

export class ClockComponent implements OnInit, OnDestroy {
  private sub?: Subscription;

  ngOnInit() {
    this.sub = interval(1000).subscribe(() => console.log('tick'));
  }

  ngOnDestroy() {
    this.sub?.unsubscribe();  // stop the subscription when the component is removed
  }
}
```

`ngOnDestroy` runs just before Angular removes the component — the place for cleanup, exactly mirroring the cleanup function returned from a React `useEffect`. This is where the manual subscriptions flagged in

Chapter 11's warning get unsubscribed, timers cleared, and listeners removed. The same rule from React applies: anything that "starts" something ongoing needs a matching "stop" here, or it leaks.

THE ASYNC PIPE AND TAKEUNTILDESTROYED AVOID MANUAL NGONDESTROY

Manually managing a `Subscription` and unsubscribing in `ngOnDestroy` is verbose. Two modern alternatives largely remove the need: the `async` pipe (Chapter 11) cleans up its own subscription automatically, and the `takeUntilDestroyed()` operator ties an Observable's lifetime to the component's automatically. Prefer those where possible; reach for an explicit `ngOnDestroy` for non-RxJS cleanup (a manual timer, a third-party library handle).

NGONCHANGES ONLY FIRES FOR @INPUT CHANGES — AND WATCH OBJECT MUTATION

`ngOnChanges` reacts to changes in `@Input` properties specifically, not to internal state changes. And like React's dependency comparison, it detects a changed input by reference for objects/arrays — mutating an object passed as an input in place (rather than passing a new one) won't be seen as a change, the same immutability concern from React's Fundamentals Chapter 3. Pass a new object/array to trigger `ngOnChanges` reliably.

ANGULAR HOOK	REACT USEEFFECT EQUIVALENT
ngOnInit	`useEffect(() => {...}, [])` (mount)
ngOnChanges	`useEffect(() => {...}, [someProp])`
ngOnDestroy	The cleanup function returned from `useEffect`

Coding Challenges

CHALLENGE 1

Build a component implementing OnInit that logs a message and fetches/sets some initial data in ngOnInit, confirming (via console) that it runs once after the component is created.

View solution

CHALLENGE 2

Build a child component with an @Input value, implementing OnChanges to log the previousValue and currentValue each time the input changes. Drive it from a parent with a button that changes the input.

View solution

CHALLENGE 3

Build a component that starts an RxJS interval subscription in ngOnInit (logging a tick each second) and properly unsubscribes in ngOnDestroy. Toggle the component's presence with @if in a parent to confirm the ticking stops when it's removed.

View solution

CHAPTER 12 QUICK REFERENCE

- **ngOnInit** — runs once after creation; the place for initial setup/data fetching (mount equivalent)
- Use `ngOnInit`, not the constructor, for init logic — inputs aren't ready in the constructor
- **ngOnChanges(changes)** — runs when an `@Input` changes; `SimpleChanges` gives previous/current values
- **ngOnDestroy** — runs before removal; the place for cleanup (unsubscribe, clear timers)
- Implement the matching interface (`OnInit` / `OnChanges` / `OnDestroy`) for type safety
- The `async` pipe and `takeUntilDestroyed()` reduce the need for manual `ngOnDestroy` cleanup
- `ngOnChanges` detects object/array input changes by reference — pass new ones, don't mutate
- **Next chapter:** standalone components and signals — Angular's modern direction

CHAPTER 13 OF 14

Standalone Components and Signals

CHAPTER 13

Standalone Components and Signals

Where Angular is heading — and a reactive primitive that feels a lot like useState

This whole course has quietly used the modern direction already — every component has been `standalone: true`, and Chapters 10–11 used `provideRouter` / `provideHttpClient` rather than the older module setup. This chapter names those choices explicitly, then introduces **signals** — Angular's newer reactivity primitive, the closest thing in Angular to React's `useState`.

Standalone Components, Recapped

Historically, Angular grouped components into **NgModules** — separate files declaring which components, directives, and pipes belonged together and what they could use. **Standalone components** (now the default) drop that layer: each component declares its own `imports` directly, as seen in every example so far. The result is less boilerplate, no `app.module.ts` to maintain, and an import list that lives right next to the component using it.

YOU WILL STILL ENCOUNTER NGMODULE CODE

Plenty of existing Angular apps and tutorials predate standalone components and are built around `@NgModule` with a `declarations` array and a root `AppModule`. It's fully supported and not going away soon, but new projects should use standalone (the `ng new` default). Recognize the module-based structure when reading older code; this course deliberately teaches only the standalone approach.

Signals — A Reactive Value

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

@Component({
  selector: 'app-counter',
  standalone: true,
  template: `
    <p>Count: {{ count() }}</p>
    <button (click)="increment()">+1</button>
  `,
})
export class CounterComponent {
  count = signal(0);

  increment() {
    this.count.update((n) => n + 1);
  }
}
```

`signal(0)` creates a reactive value holding `0`. Reading it is a *function call* — `count()`, both in the template and in code. Writing is done with `.set(value)` (a new value) or `.update(fn)` (based on the current one) — directly parallel to React's `setCount(5)` and `setCount(n => n + 1)`. When a signal changes, Angular knows precisely which parts of the template depend on it and updates only those, more surgically than the older change-detection mechanism.

SIGNAL	REACT USESTATE EQUIVALENT
`count = signal(0)`	`const [count, setCount] = useState(0)`
`count()`	`count` (read)
`count.set(5)`	`setCount(5)`
`count.update(n => n + 1)`	`setCount(n => n + 1)`

computed — Derived Signals

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

price = signal(100);
quantity = signal(2);

total = computed(() => this.price() * this.quantity());
// total() is 200, and recalculates automatically whenever price or quantity changes
```

`computed` derives a new signal from others — its value automatically recalculates whenever any signal it reads changes, and (importantly) is cached until then, so it only recomputes when genuinely needed. This is

the signals version of React's `useMemo` (Intermediate Chapter 7), but the dependency tracking is automatic: there's no dependency array to maintain — Angular knows `total` depends on `price` and `quantity` simply because it read them.

effect — Reacting to Signal Changes

```
import { effect } from '@angular/core';

constructor() {
  effect(() => {
    console.log('count is now', this.count());
  });
}
```

`effect` runs a side effect whenever any signal it reads changes — automatically tracked, the same way `computed` works. It's the rough equivalent of React's `useEffect` with the relevant value in its dependency array, again with no manual dependency list. Effects are for genuine side effects (logging, syncing to `localStorage`); deriving a value should use `computed` instead.

Why Signals Matter

Signals are Angular's strategic direction for reactivity — gradually offering a simpler, more explicit, and more performant alternative to the change-detection-plus-RxJS model that came before. They don't replace RxJS (Observables are still the right tool for streams of events and HTTP, Chapter 11), but for component-local state, signals are increasingly the recommended default. Inputs can even be signals now (`input()` instead of `@Input()`), and the `async` pipe has a signal counterpart in `toSignal()` — the ecosystem is steadily building around them.

> **COMING FROM REACT, SIGNALS WILL FEEL IMMEDIATELY FAMILIAR**
>
> Of everything in Angular, signals map most cleanly onto React knowledge: `signal` / `set` / `update` is `useState`, `computed` is `useMemo`, `effect` is `useEffect` — all with automatic dependency tracking instead of manual arrays. If the rest of Angular felt like a different paradigm, this is the chapter where it converges back toward familiar ground.

ANGULAR SIGNALS	REACT HOOKS
`signal()`	`useState`
`computed()`	`useMemo` (auto-tracked)
`effect()`	`useEffect` (auto-tracked)

Coding Challenges

CHALLENGE 1

Build a counter using a signal, with +1/-1/reset buttons calling set/update, reading the value with count() in the template.

View solution

CHALLENGE 2

Build a component with price and quantity signals and a computed total signal, plus inputs/buttons to change price and quantity, confirming total updates automatically with no manual recalculation.

View solution

CHALLENGE 3

Build a component with a signal whose value is persisted to localStorage via an effect (saving on every change) and read back as the initial value, so it survives a page refresh — the signals version of the React useLocalStorage hook.

View solution

CHAPTER 13 QUICK REFERENCE

- **Standalone components** (the default) replace NgModules; each declares its own `imports`
- **signal(value)** — a reactive value; read with `x()`, write with `.set()` / `.update()` (= `useState`)
- **computed(() => ...)** — a derived signal, auto-recalculated and cached (= `useMemo`, no dep array)
- **effect(() => ...)** — runs a side effect when read signals change (= `useEffect`, auto-tracked)
- Signals are Angular's strategic direction for reactivity — preferred for component-local state
- Signals complement, don't replace, RxJS — Observables still own event streams and HTTP
- **Next chapter:** testing — Jasmine, Karma, and TestBed (the final chapter)

CHAPTER 14 OF 14

Testing — Jasmine, Karma, and TestBed

CHAPTER 14

Testing — Jasmine, Karma, and TestBed

The final chapter — verifying components automatically, using the spec files Angular has generated all along

Every `ng generate component` has quietly created a `.spec.ts` file alongside each component (Chapter 1) — those are tests, and this chapter finally puts them to use. Angular's testing stack uses **Jasmine** (the test framework: `describe` / `it` / `expect`) run by **Karma** (the test runner, executing tests in a real browser), with **TestBed** as Angular's own utility for creating components in a test environment.

THE CONCEPTS CARRY STRAIGHT OVER FROM REACT TESTING

React Advanced Chapter 4 covered testing with Vitest + React Testing Library. The vocabulary differs — Jasmine's `it()` for React's `test()`, `jasmine.createSpy()` for `vi.fn()` — but the goals are identical: render a component, simulate interaction, assert on what the user would see. If that chapter made sense, this one will feel familiar despite the different tool names.

A Test for a Service — The Simplest Case

```
// counter.service.spec.ts
import { TestBed } from '@angular/core/testing';
import { CounterService } from './counter.service';

describe('CounterService', () => {
  let service: CounterService;

  beforeEach(() => {
    TestBed.configureTestingModule({});
    service = TestBed.inject(CounterService);
  });

  it('increments the count', () => {
    service.increment();
    expect(service.getCount()).toBe(1);
  });
});
```

`describe` groups related tests; `it` defines one; `beforeEach` runs before each test for fresh setup; `expect(...).toBe(...)` asserts. `TestBed.inject(CounterService)` retrieves the service through the same DI system the real app uses (Chapter 6), rather than constructing it manually — so the test exercises it exactly as it'd be used in practice.

Testing a Component with TestBed

```
// counter.component.spec.ts
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { CounterComponent } from './counter.component';

describe('CounterComponent', () => {
  let fixture: ComponentFixture<CounterComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [CounterComponent],  // standalone components go in imports
    }).compileComponents();

    fixture = TestBed.createComponent(CounterComponent);
    fixture.detectChanges();  // run initial change detection / render
  });

  it('increments when the +1 button is clicked', () => {
    const button = fixture.nativeElement.querySelector('button');
    button.click();
    fixture.detectChanges();  // re-render after the state change

    const text = fixture.nativeElement.textContent;
    expect(text).toContain('Count: 1');
  });
});
```

`TestBed.createComponent` returns a **fixture** — a handle to the live component plus its rendered DOM (`fixture.nativeElement`). The key Angular-specific detail is `fixture.detectChanges()`: unlike React Testing Library, which re-renders automatically, an Angular test must explicitly trigger change detection to render the initial view and to update it after a state change. Forgetting it is the most common reason an Angular test sees stale, un-updated DOM.

Mocking a Dependency

```
const fakeUserService = {
  getUsers: () => of([{ id: 1, name: 'Test User' }]),  // returns a fixed Observable
};

TestBed.configureTestingModule({
  imports: [UserListComponent],
  providers: [
    { provide: UserService, useValue: fakeUserService },  // swap the real service for the fake
  ],
});
```

Dependency injection makes testing easier: a component asks for `UserService`, and the test simply provides a fake one instead via `{ provide: UserService, useValue: fakeUserService }`. The component is none the wiser — it gets whatever DI supplies. This means a component can be tested in isolation, with a fake service returning fixed data, without ever hitting a real API — the same goal as React's mock functions (Advanced Chapter 4), achieved through DI rather than passing mocks as props.

> **DETECTCHANGES() IS THE GOTCHA TO REMEMBER**
>
> The single most common Angular-testing mistake is asserting on the DOM without calling `fixture.detectChanges()` first — after creating the component (to render the initial view) and after any action that changes state (to re-render). React Testing Library handles this automatically, so it's an easy habit to miss when coming from that background. If a test sees an empty or unchanged DOM, a missing `detectChanges()` is the first thing to check.

ANGULAR (JASMINE/TESTBED)	REACT (VITEST/RTL)
`describe / it / expect`	`describe` / `test` / `expect`
`TestBed.createComponent`	`render(...)`
`fixture.nativeElement.querySelector`	`screen.getByRole` / `getByText`
`fixture.detectChanges()`	(automatic)
`jasmine.createSpy()`	`vi.fn()`
`{ provide, useValue }` mock	Mock passed as a prop

Running the Tests

```
ng test   # launches Karma, runs every .spec.ts, watches for changes
```

`ng test` runs the whole suite, opening a browser Karma controls to execute the tests in, and re-running automatically as files change — the equivalent of a test runner's watch mode. The default starter project

already passes its one generated `AppComponent` test, so `ng test` works from the very first `ng new`.

Coding Challenges

CHALLENGE 1

Write a spec for a CounterService (with increment/decrement/reset) using TestBed.inject, asserting the count behaves correctly after each operation.

View solution

CHALLENGE 2

Write a component spec using TestBed.createComponent that clicks the +1 button and asserts the rendered text updates — remembering detectChanges() after creation and after the click.

View solution

CHALLENGE 3

Write a spec for a component that depends on a UserService, providing a fake service (via { provide, useValue }) returning fixed data, and assert the component renders that fake data without any real HTTP call.

View solution

CHAPTER 14 QUICK REFERENCE

- **Jasmine** — the test framework (`describe` / `it` / `expect` / `beforeEach`); **Karma** — the runner
- **TestBed.inject(Service)** — gets a service through DI for testing
- **TestBed.createComponent** → a **fixture** (live component + `nativeElement` DOM)
- **fixture.detectChanges()** — manually trigger render; required after creation and state changes
- Mock a dependency with **{ provide: Service, useValue: fake }** — DI makes isolation easy
- **ng test** — runs the suite in watch mode via Karma
- Same goals as React Testing Library; different tool names (`it` = `test`, `createSpy` = `vi.fn`)

🎉 That completes the Angular course — all 14 chapters, from the CLI and TypeScript basics through services, RxJS, signals, and testing.