Angular

Version 22 · verified 2026-07-27

A gradual, compact walk through Angular's signal-first model — components, signals, control flow, DI, and zoneless.

Contents

Mental model

Angular's reactivity now runs on signals, not Zone.js: a signal is a value plus automatic dependency tracking, and Angular re-renders only the components that actually read a changed signal. Every new project is standalone and zoneless by default — there's no NgModule to register a component in, and nothing patches native async APIs to trigger change detection anymore. Because of that, mutating a plain object or array in place notifies nobody; only writing through a signal's `set()` / `update()` does.

Components are standalone by default

Property	Meaning
`selector`	tag name used in templates
`template` / `templateUrl`	inline or external HTML
`styleUrl`	external CSS file
`imports`	components/pipes it uses

Every component is standalone since Angular v19 — there's no `standalone: true` flag to set, and no NgModule required. A component lists what it uses directly in its own `imports` array, so dependencies are visible at the point of use instead of buried in a module file.

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

@Component({
  selector: 'user-badge',
  imports: [DatePipe],
  template: `
    <span>{{ joined | date }}</span>
  `,
})
export class UserBadge {
  joined = new Date();
}
```

> **Note:** Requires Angular v19+. Earlier versions need `standalone: true` written explicitly on every component.

Signals: state, computed & effect

API	Purpose
`signal(v)`	writable reactive value
`computed(fn)`	derived, memoized value
`effect(fn)`	side effect on change
`.set(v)` / `.update(fn)`	write to a signal

Read a signal by calling it — `count()`, not `count` — inside a template or a reactive context like `computed` or `effect`, and Angular tracks it as a dependency automatically. `computed()` values are read-only and re-run lazily, only when read again after a dependency changed.

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

const count = signal(0);
const doubled = computed(() => count() * 2);

effect(() => console.log(`x2 = ${doubled()}`));

count.set(5);
// logs: x2 = 10
```

> **Gotcha:** mutating an object or array held in a signal in place (`list().push(x)`) notifies nobody — call `.set()` / `.update()` with a new reference instead.

Inputs, outputs & two-way binding

Function	Direction
`input(default)`	parent to child

Function	Direction
`input.required<T>()`	parent to child, required
`output<T>()`	child to parent (event)
`model(default)`	two-way, `[(x)]` binding

`input()` and `output()` replace the `@Input` / `@Output` decorators with plain signal-returning functions: inputs are read-only signals, outputs expose `.emit()`. `model()` is for state a child writes back to its parent — binding `[(value)]="volume"` passes the parent's signal itself, not a snapshot of its value.

```
import {
  Component, model, input, output,
} from '@angular/core';

@Component({ selector: 'app-slider' })
class Slider {
  value = model(0);
  step = input(1);
```

```
  changed = output<number>();

  bump() {
    this.value.update(v => v + this.step());
    this.changed.emit(this.value());
  }
}
```

> **Note:** the `@Input()` / `@Output()` decorators still work. `input()` / `output()` / `model()` are the recommended default for new code, not a breaking replacement.

Control flow syntax

Block	Use
`@if` / `@else if` / `@else`	conditional
`@for (x of xs; track x.id)`	loop, track required
`@empty`	fallback for an empty `@for`

Block	Use
`@switch` / `@case` / `@default`	multi-branch

The built-in `@if` / `@for` / `@switch` blocks replace `*ngIf` / `*ngFor` / `*ngSwitch` and need no `CommonModule` import. `@for` requires a `track` expression — Angular has no way to diff a list efficiently without one, so omitting it is a compile error, not just a lint warning.

```
@for (item of items; track item.id) {
  <li>{{ item.name }}</li>
} @empty {
  <li>No items yet.</li>
}
```

> **Gotcha:** `@switch` uses `===` and has no fallthrough — stacking `@case` labels with no body between them (as in a JS `switch`) is the only way to share one branch.

Dependency injection with inject()

Where	Can call `inject()`?
field initializer	yes, recommended
constructor body	yes
route guard / factory fn	yes, in injection context
a method, after construction	no — throws `NG0203`

`inject()` reads a dependency from the current injection context and replaces constructor-parameter injection. It works in field initializers, constructors, and any plain function Angular calls while setting up injection, such as a route guard function.

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

@Injectable({ providedIn: 'root' })
class AuthService {
  isLoggedIn() { return true; }
}

@Component({ selector: 'app-root' })
class AppRoot {
  private auth = inject(AuthService);
}
```

> **Gotcha:** calling `inject()` after an `await`, inside a `setTimeout`, or in a lifecycle method throws `NG0203` — capture the dependency in a field first, use it later.

Async data with resource() & httpResource()

Signal	Meaning
`.value()`	resolved data, or `undefined`
`.isLoading()`	request currently in flight
`.error()`	error, or `undefined`
`.status()`	`'loading'`, `'resolved'`, …

`resource()` ties an async `loader` to a reactive `params` computation — Angular reruns the loader whenever a signal read inside `params` changes, exposing the result as signals instead of a `Promise`. `httpResource()` is the same shape specialized for `HttpClient`, interceptors included, so most components never call `.subscribe()`.

```
import {signal} from '@angular/core';
import {
  httpResource,
} from '@angular/common/http';
```

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

const userId = signal(1);
const user = httpResource<User>(
  () => `/api/users/${userId()}`,
);

// user.value(), user.isLoading(), user.error()
```

> **Note:** `resource()` and `httpResource()` are stable public APIs in Angular 22 — safe for production, not behind a developer-preview flag.

Zoneless change detection

Fact	Detail
Default since	Angular v21
Drives updates	signals, events, `AsyncPipe`
No longer patches	native async APIs (Zone.js)

Fact	Detail
Manual escape hatch	`ChangeDetectorRef.markForCheck()`

New projects no longer install Zone.js. Angular only re-renders when a signal read in a template changes, an event handler fires, or something explicitly calls `markForCheck()` — plain field mutation, the old Zone.js-patched behavior, silently stops updating the view.

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

class Legacy {
  private cdr = inject(ChangeDetectorRef);
  data?: string;

  load() {
    thirdPartyApi.fetch((result) => {
      this.data = result;
```

```
      this.cdr.markForCheck();
    });
  }
}
```

> **Warning:** a callback from a non-Angular library (a timer, a third-party SDK) that sets a plain field never repaints on its own — call `markForCheck()` yourself.

Further reading

- Angular — Signals guide
- Angular — Components
- Angular — Template control flow
- Angular — Dependency injection
- Angular — Zoneless