Angular 20: What's New and How to Upgrade.

Author: Mahmoud Alfaiyumi

Date: 2/7/2025

Table of Contents

Introduction

Angular 20 is a major milestone that modernizes the framework's reactivity, rendering, and developer ergonomics. It stabilizes Signals and Zoneless change detection, brings **incremental hydration** for faster SSR, introduces JavaScript-like **control-flow syntax** (@if, @for, @switch), an improved createComponent() API, richer template expressions (**, in, template literals), and a revamped **style guide** with simplified file naming. Notably, **host bindings** now support full type-checking. According to benchmarks, early adopters see **30–40% faster initial renders and ~50% fewer unnecessary re-renders** vs Angular 19. Below we explain each feature in depth with code examples, and then cover detailed migration steps and common pitfalls when upgrading from Angular 19 to 20.

1. Signal-Based Reactivity and Zoneless Change Detection

Angular 20 fully embraces Signals – a fine-grained reactivity model – and introduces Zoneless change detection. Signals (stable in Angular 20) let you wrap state in signal(), derive computed values with computed(), and run side-effects with effect(). Unlike the old Zone.js model, change detection can be triggered manually or automatically from signals, eliminating Zone.js overhead. This makes apps smaller and faster: Angular 20 "removes the need for Zone.js, making apps faster and smaller… reducing unnecessary re-renders, which can improve rendering speed by up to 30%". In Zoneless mode, debugging is simpler (cleaner stack traces) and change-detection control is explicit.

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

@Component({ selector: 'count-display', template: `
  <p>Clicks: {{ clicks() }}</p>
  <p>Double: {{ doubleClicks() }}</p>
  <button (click)="increment()">Click me</button>
`})
export class CountDisplay {
  // A reactive signal holding click count
  clicks = signal(0);

  // A computed signal derived from clicks
  doubleClicks = computed(() => this.clicks() * 2);

  increment() {
    this.clicks.update(c => c + 1);  // update triggers UI refresh
  }

  constructor() {
    // An effect that logs whenever clicks changes
    effect(() => console.log(`Clicked ${this.clicks()}`));
  }
}
```

In this example, reading clicks() (note calling the signal as a function) triggers change detection. Because signals re-run computations only when needed, rendering is efficient. You can also convert Observables to signals via toSignal() and vice versa for interop.

To enable Zoneless change detection in a new app, use the CLI flag or providers. For example:

```
 1. import { bootstrapApplication } from '@angular/platform-browser';
 2. import { provideZonelessChangeDetection, provideBrowserGlobalErrorListeners }
from '@angular/core';
 3. import { AppComponent } from './app.component';
 4.
 5. bootstrapApplication(AppComponent, {
 6.   providers: [
 7.     provideZonelessChangeDetection(),
 8.     provideBrowserGlobalErrorListeners(),
 9.   ]
10. });
```

Don't forget to remove the zone.js polyfill from angular.json. With Zoneless enabled, Angular no longer intercepts all async events; instead, you manually call methods like detectChanges() if needed. Many built-in APIs now work seamlessly with signals. Over time, this "future-ready" architecture will make apps more maintainable.

2. Incremental Hydration (SSR)

Server-Side Rendering (SSR) in Angular 20 gains **Incremental Hydration** as a stable feature. Instead of hydrating the entire page on the client at once (which can be slow), you can mark parts of the template as deferrable. Angular will download and hydrate a component *only when needed* (e.g. on user interaction or when it enters the viewport). This reduces initial JavaScript and speeds up Time-To-Interactive. For example:

```
1. import { provideClientHydration, withIncrementalHydration } from
'@angular/platform-browser';
2. bootstrapApplication(AppComponent, {
3.   providers: [
4.     // Enable incremental hydration globally
5.     provideClientHydration(withIncrementalHydration()),
6.   ]
7. });
```

Then in templates you use the @defer block with a trigger:

```
1. @defer (hydrate on viewport) {
2.   <!-- This <user-profile> loads and hydrates only when visible -->
3.   <user-profile [userId]="userId"></user-profile>
4. }
```

3. Control-Flow Directives (@if, @for, @switch)

Angular 20 introduces modern control-flow syntax that replaces *ngIf, *ngFor, and *ngSwitch with JavaScript-like blocks. For example, instead of:

```
<ng-container *ngIf="items.length > 0; else noItems">
  <ul>
    <li *ngFor="let item of items; trackBy: trackFn; let idx = index">
      Item {{ idx+1 }}: {{ item.name }}
    </li>
  </ul>
</ng-container>
<ng-template #noItems><p>No items</p></ng-template>
```

you can write:

```
@if (items.length > 0) {
  <ul>
    @for (item of items; track item.id; let idx = $index) {
      <li>Item {{ idx+1 }}: {{ item.name }}</li>
    }
  </ul>
} @else {
  <p>No items</p>
}
```

The @if block supports @else if and @else. The @for block automatically provides $index and $count loop variables. These new directives eliminate the boilerplate of extra <ng-template> tags and align template syntax with native JavaScript. They also enable compile-time type checks and better IDE assistance. For example, using in expressions is now supported:

```
@if ('admin' in userPermissions) {
  <button (click)="goToAdmin()">Admin Panel</button>
}
```

Under the hood, @if/@for map to structural directives with compile-time checks. Note that as of Angular 20, the legacy *ngIf/*ngFor/*ngSwitch are **deprecated** (planned for removal by v22). The CLI will offer a migration for you. A common pitfall is forgetting to import needed directives: Angular now warns if you use e.g. *ngTemplateOutlet without importing CommonModule. Also, if you use @for with a track function, you **must call** the function in the expression (e.g. track user => user.id) or the compiler will error, saving you from subtle bugs

4. Dynamic Component Creation (createComponent)

Angular's dynamic component API has been overhauled. The new createComponent() function is a declarative, type-safe way to instantiate components at runtime without ViewContainerRef or factories. It automatically handles DI, change detection, and content projection. Importantly, you can now bind inputs/outputs and apply directives at creation time using helper functions.

Example: creating a dialog component with input/output bindings (using Signals):

```
 1. import { createComponent, signal, inputBinding, outputBinding, twoWayBinding }
from '@angular/core';
 2. import { MyDialog } from './my-dialog.component';
 3.
 4. const canClose = signal(false);
 5. const title = signal('My dialog title');
 6.
 7. // Dynamically create MyDialog component
 8. const dialogRef = createComponent(MyDialog, {
 9.   bindings: [
10.     // Bind a Signal to the `canClose` input
11.     inputBinding('canClose', canClose),
12.     // Listen to the `onClose` output event
13.     outputBinding<DialogResult>('onClose', result => console.log('Closed with',
result')),
14.     // Two-way bind the `title` property to the signal
15.     twoWayBinding('title', title),
16.   ],
17.   directives: [
18.     // Apply other directives to the component
19.     FocusTrap,
20.     { type: HasColor, bindings: [ inputBinding('color', () => 'red') ] }
21.   ]
22. });
```

This example shows how createComponent() can attach MyDialog anywhere (outside any template), bind its inputs to signals or functions, and even attach additional directives (FocusTrap, HasColor). These improvements make dynamic creation much cleaner than the old ViewContainerRef.createComponent approach. Teams migrating should replace any manual factory logic with createComponent() and the new inputBinding, outputBinding, and twoWayBinding helpers. A common pitfall is forgetting to manage the component's lifecycle; note that createComponent() returns a component reference you can later destroy.

5. Enhanced Template Expressions

Angular 20 expands what you can do inside template interpolation and bindings. Supported JavaScript syntax now includes:

- **Exponentiation (**)**: e.g. `{{ count ** 2 }}` lets you square `count` inline.
- **`in` operator**: e.g. `@if ('name' in person) { ... }` checks if a property exists.
- **Untagged template literals**: e.g. `<div [class]="grid-cols-${colWidth}"></div>` combines strings without `ngClass`.

```
<input [(ngModel)]="value" type="number">
<p>Square: {{ value ** 2 }}</p>

@defer (click) {
  <div [class]="'btn btn-' + (isActive ? 'primary' : 'secondary')">Click me</div>
}

@if ('name' in item) {
  <span>Name: {{ item.name }}</span>
}
```

These features avoid boilerplate methods for simple logic. They also come with new compile-time diagnostics: for example, mixing `??` with `&&` now requires parentheses (`a ?? (b && c)`) or the compiler will error. As you migrate, watch out for these stricter rules: the compiler will alert you to missing structural imports or un-invoked track functions.

6. Simplified File Naming and Style Guide

Angular 20's updated style guide emphasizes **intentional naming**. By default, generated files no longer include type suffixes (`.component.ts`, `.service.ts`, etc.). For example, `ng generate component user` now creates `user.ts`, `user.html`, user.css, with a class `User` (not `UserComponent`). The CLI still puts a suffix on selectors (e.g. `<app-user>`). You can opt back in to old naming via schematic options in `angular.json`:

```
// angular.json (partial)
"schematics": {
  "@schematics/angular:component": { "type": "component" },
  "@schematics/angular:service":   { "type": "service" },
  "@schematics/angular:pipe":      { "typeSeparator": "." },
  "@schematics/angular:guard":     { "typeSeparator": "." },
  "@schematics/angular:module":    { "typeSeparator": "." }
}
```

This sample restores the old .component.ts suffix behavior. The new default is meant to **reduce boilerplate** and encourage more meaningful names. Teams migrating should decide on a naming convention: either adopt the new minimalist names (and run the CLI migrations to update class names) or configure schematics as above. Other style updates include preferring [class.foo]/[style.bar] bindings over ngClass/ngStyle, and using protected/readonly for template inputs

7. Type-Checked Host Bindings

In Angular 20 the compiler finally type-checks host bindings/listeners. If you use the host field in @Component/@Directive or the @HostBinding()/@HostListener() decorators, enabling a new tsconfig option will catch errors. For example:

```
// tsconfig.json
"angularCompilerOptions": {
  "typeCheckHostBindings": true
}
```

With this on, binding expressions are fully validated: the **left side** must be a valid property of the host element, and the **right side** must exist in the class. For example:

```
@Directive({ selector: 'label', host: { '[value]': 'value()' } })
export class FormLabel {
  // error: 'value' is not a property of <label>
}

@Directive({ selector: 'button', host: { '(click)': 'onClick' } })
export class MyButton { /* error: onClick not found */ }
```

These errors help catch typos and invalid bindings at compile time. The new language service also offers IntelliSense for host bindings. Make sure typeCheckHostBindings is enabled (new CLI projects include it by default) to benefit from these checks.

8. Migration Tips: Upgrading from Angular 19 to 20

Upgrading to Angular 20 should be done via the Angular CLI (ng update), but here are key manual steps and pitfalls to watch:

- **Remove Zone.js**: If you used Angular's dev preview of Zoneless, update to the new provider names. Remove the zone.js polyfill from angular.json, and add provideZonelessChangeDetection() and provideBrowserGlobalErrorListeners() to your bootstrap providers. If you used ng new with --experimental-zoneless, note the flag is now --zoneless. Also ensure tests opt out of zone (calling fixture.autoDetectChanges(false) now errors in zoneless tests).

- **Migrate control-flow syntax**: Run the `ng update` migration to replace `*ngIf`, `*ngFor`, `*ngSwitch` with `@if`/`@for`/`@switch` blocks. The CLI will prompt you. If you keep the old syntax, they will work but are deprecated. Ensure any custom templates import the right directives (Angular will error if a structural directive is used without an import). A common pitfall: forgetting to invoke the track function in `@for` (you must call e.g. `track user => user.id`, not just `track user.id`).
- **Adopt new APIs**: Replace deprecated reactivity methods: e.g. replace `TestBed.flushEffects()` with `TestBed.tick()`, and remove unused signal options (`forceRoot`, `rejectErrors`). Consider refactoring `Observable` or `BehaviorSubject` state to Signals using `signal()` or `toSignal()`. For dynamic creation, switch from `ComponentFactoryResolver`/`setInput` to `createComponent()` with `inputBinding`/`outputBinding`.
- **Update templates for new expressions**: If you mix nullish coalescing (`??`) and boolean operators, add parentheses (`a ?? (b && c)`) as Angular 20 now enforces this. Use `**` instead of custom math in templates for power, and use backticks for string interpolation inside bindings.
- **Enable strict checks**: Turn on the new diagnostics (or at least address their warnings). For example, if you get errors about missing structural directive imports, either import the needed `NgIf`, etc., or suppress the check via `extendedDiagnostics` in `tsconfig`.
- **Handle deprecations**: Angular 20 deprecates `TestBed.get()` (it will auto-migrate to `inject()`), and has removed `InjectFlags`. Migrations in `ng update` will handle most of these changes. Also note that `DOCUMENT` token moved from `@angular/common` to `@angular/core`.
- **CLI and tooling changes**: Upgrade to Node.js 20 and TypeScript 5.8, as Angular 20 requires them. After update, review `angular.json`: Angular now uses the new `@angular/build` instead of Webpack (`@angular-devkit/build-angular`), greatly reducing node_modules size. Verify your `tsconfig` uses `module: "preserve"` (to reflect modern bundlers) and the new “solution-style” project references. If your browserslist falls outside the new baseline (last 30 months), the CLI will warn.
- **Style/naming migrations**: Decide on adopting the new file naming. You can run the migration that updates `angular.json` schematics (as shown above) to restore old suffixes if desired. Otherwise, expect the CLI to generate succinct file names and classes. Old file names continue to work, but may look inconsistent with new code.
- **Watch for runtime changes**: Some subtle behavior changes include: error handling is stricter (exceptions in async tasks now surface to the global handler by default) and Angular DevTools will color-code OnPush components. Test suites may fail where they didn’t before (e.g. errors thrown in event handlers), so you may need to configure `rethrowApplicationErrors: false` in testing or fix the underlying issue.

Common pitfalls include forgetting to remove Zone.js (which can cause unexpected change detection behavior) and missing imports for new control-flow directives. Always run the migration schematics and fix any compilation errors thrown by the new diagnostics.

9. Performance Considerations

Angular 20's improvements translate to tangible gains. For example, apps often see **30–40% faster initial render times** and **~50% fewer unnecessary re-renders** compared to Angular 19. Removing Zone.js cuts out significant overhead, and incremental hydration can greatly speed up Time-To-Interactive in SSR apps. Additionally, the dev profiling enhancements (Angular-specific events in Chrome DevTools) help teams spot bottlenecks more easily. In performance-critical paths (e.g. large lists or complex components), switching to Signals and `@if`/`@for` can reduce wasted checks: with `@for`'s explicit tracking you avoid stale loops, and Signals ensure change detection only runs when the underlying state changes.

To maximize gains, measure your app before and after the upgrade. Use the new Angular DevTools (v21+ for Angular 20) to see the *Angular track* in Chrome's Performance panel, and enable `ng.enableProfiling()` in DevTools for detailed insights.

10. Strategic Recommendations

For teams migrating to Angular 20:

1. **Incrementally adopt new features.** Start new components/pages with Signals and Zoneless by default. Leave legacy `NgZone` code for later, and gradually refactor.
2. **Run `ng update` early.** The CLI handles many renames and deprecations. Review its migration summaries and address any remaining TODOs.
3. **Enable stricter TypeScript options.** Turning on `strictTemplates` and `typeCheckHostBindings` will catch issues early under Angular 20. Fix them before they hit runtime.
4. **Refactor templates consciously.** Convert structural directives at a comfortable pace. Use feature flags or separate branches if needed. Remember to import `CommonModule` or `BrowserModule` where control-flow directives are used.
5. **Train the team on Signals.** While similar to RxJS, Signals have unique patterns (e.g. always calling signals, avoiding side effects in getters). Update documentation and pair on writing reactive code with `signal` and `effect`.
6. **Leverage improved tooling.** Use updated Angular DevTools and Vitest (now supported) for faster testing. The new build system simplifies configuration – take advantage of the reduced build times and smaller dependencies.

By planning the migration, updating Node/TS versions first, then adjusting configuration, and finally refactoring code, you can smooth the transition. Embrace Angular 20's modern syntax and reactive model in stages. Over time, you'll benefit from a leaner, faster application and a more enjoyable developer experience.