

# What is New in Angular v17?

New Homepage: <https://angular.dev/>

## Sources & Online Blogs:

- [Minko Gechev - Medium](#)
- [Piotr Wiórek - Angular Love](#)
- [Bhadresh Panchal - RadixWeb.com](#)

## Test it online:

- See [WebContainers](#) in this context.
- [StackBlitz](#)

## Table of Contents

- [TypeScript 5.2+ Support \(Currently 5.3.3\)](#)
- [New Syntax for Control Flow in Templates](#)
- [Automatic Migration to Build-in Control Flow](#)
- [SSR \(Server Side Rendering\) Improvements](#)
- [New lifecycle hooks](#)
- [Vite and esbuild the default for new projects](#)
- [Dependency injection debugging in DevTools](#)
- [Material 3 Web Components Support](#)
- [View transitions support](#)
- [Defer loading of the animations module](#)
- [Input value transforms](#)
- [Community schematics](#)
- [Other Less Important Highlights](#)
  - [Signal](#)
    - [Effects](#)
    - [Signal-based components](#)
  - [Standalone Components](#)

## TypeScript 5.2+ Support (Currently 5.3.3)

Current Version at the time of writing is 5.3.3

# New Syntax for Control Flow in Templates

```
<div @if="role === 'admin'">
You're logged in as an admin.
</div>
```

```
<div @else if="role === 'user'">
You're logged in as a user.
</div>
```

```
<div @else>
You're not logged in.
</div>
```

```
<div @defer="condition">
When the condition is true, your content will be loaded lazily.
</div>
```

```
@defer {
  <comment-list />
}
```

```
@defer (on viewport) {
  <comment-list />
} @placeholder {
  <!-- A placeholder content to show until the comments load -->
  
}
```

```
@defer (on viewport) {
  <comment-list/>
} @loading {
  Loading...
} @error {
  Loading failed :(
} @placeholder {
  
}
```

```
<div @transition="animation">
```

This content will be animated when it is transitioned to or from.

</div>

<!-- Switch Old -->

```
<div [ngSwitch]="accessLevel">
  <admin-dashboard *ngSwitchCase="admin"/>
  <moderator-dashboard *ngSwitchCase="moderator"/>
  <user-dashboard *ngSwitchDefault/>
</div>
```

<!-- Switch New -->

```
@switch (accessLevel) {
  @case ('admin') { <admin-dashboard/> }
  @case ('moderator') { <moderator-dashboard/> }
  @default { <user-dashboard/> }
}
```

<!-- Built-in Loop -->

```
@for (user of users; track user.id) {
  {{ user.name }}
} @empty {
  Empty list of users
}
```

### Deferrable views offer a few more triggers:

- on idle — lazily load the block when the browser is not doing any heavy lifting
- on immediate — start lazily loading automatically, without blocking the browser
- on timer() — delay loading with a timer
- on viewport and on viewport() — viewport also allows to specify a reference for an anchor element. When the anchor element is visible, Angular will lazily load the component and render it
- on interaction and on interaction() — enables you to initiate lazy loading when the user interacts with a particular element
- on hover and on hover() — triggers lazy loading when the user hovers an element
- when — enables you to specify your own condition via a boolean expression

Deferrable views also provide the ability to prefetch the dependencies ahead of rendering them.

Adding prefetching is as simple as adding a `prefetch` statement to the `defer` block and supports all

the same triggers.

```
@defer (on viewport; prefetch on idle) {  
  <comment-list />  
}
```

## Automatic Migration to Build-in Control Flow

If you want to migrate your code to the new control flow syntax automatically, there is a schematic in the `@angular/core` package:

```
ng g @angular/core:control-flow
```

## SSR (Server Side Rendering) Improvements

New CLI will ask you at the time of the project creation if you want to enable SSR. If you want to enable it in a later stage, you can use the following command:

```
ng add @angular/ssr
```

## New lifecycle hooks

To improve the performance of Angular's SSR and SSG, in the long-term we'd like to move away from DOM emulation and direct DOM manipulations. At the same time, throughout most applications' lifecycle they need to interact with elements to instantiate third-party libraries, measure element size, etc.

To enable this, we developed a set of new lifecycle hooks:

- `afterRender` — register a callback to be invoked each time the application finishes rendering
- `afterNextRender` — register a callback to be invoked the next time the application finishes rendering

Only the browser will invoke these hooks, which enables you to plug custom DOM logic safely directly inside your components. For example, if you'd like to instantiate a charting library you can use `afterNextRender` :

```

@Component({
  selector: 'my-chart-cmp',
  template: `<div #chart>{{ ... }}</div>`,
})
export class MyChartCmp {
  @ViewChild('chart') chartRef: ElementRef;
  chart: MyChart|null;

  constructor() {
    afterNextRender(() => {
      this.chart = new MyChart(this.chartRef.nativeElement);
    }, {phase: AfterRenderPhase.Write});
  }
}

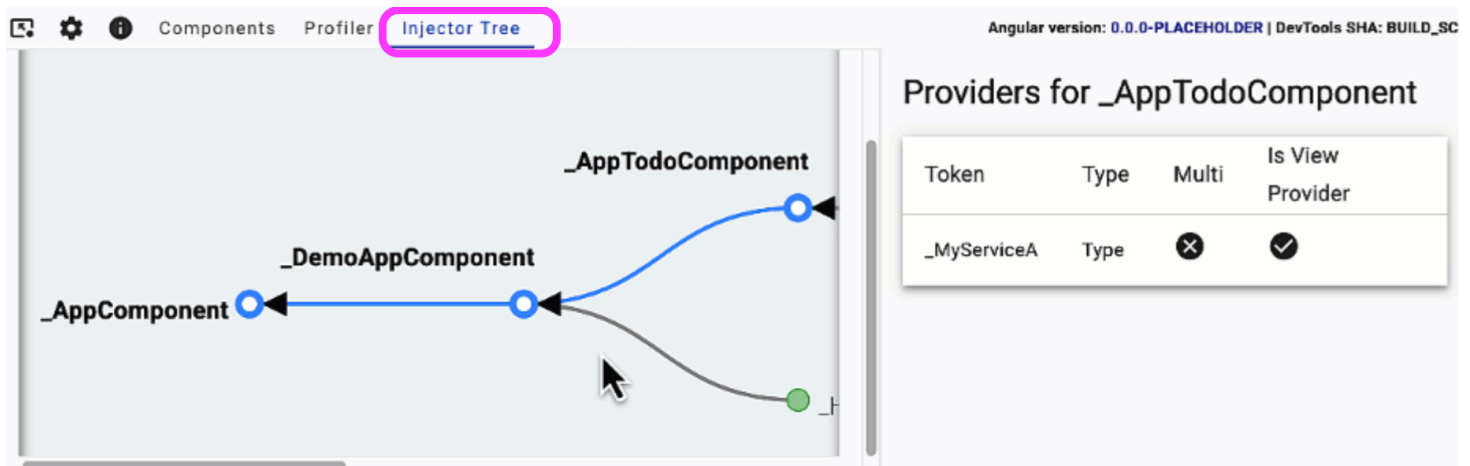
```

Each hook supports a phase value (e.g. read, write) which Angular will use to schedule callbacks to reduce layout thrash and improve performance.

## Vite and esbuild the default for new projects

Angular CLI 17 will use Vite and esbuild by default for new projects. This will improve the development experience and the build performance.

## Dependency injection debugging in DevTools



The screenshot shows the Angular DevTools interface. The 'Injector Tree' tab is selected, displaying a dependency graph. The graph shows a hierarchy of injectors: `_AppComponent` is the root, which provides `_DemoAppComponent`. `_DemoAppComponent` provides `_AppTodoComponent`. A mouse cursor is pointing at the connection between `_DemoAppComponent` and `_AppTodoComponent`.

To the right, the 'Providers for \_AppTodoComponent' panel is open, showing a table of providers:

Token	Type	Multi	Is View Provider
<code>_MyServiceA</code>	Type	✗	✓

## Material 3 Web Components Support

**Important Notice:** *Not to be confused with Angular Material*

[See Details online.](#)

```
npm install @material/mwc-button
```

```
<script type="module">
  import '@material/mwc-button/mwc-button.js';
</script>
<mwc-button raised label="Click Me"></mwc-button>
```

## View transitions support

[See View Transition API → Chrome Developer](#)

```
bootstrapApplication(App, {
  providers: [
    provideRouter(routes, withViewTransitions()),
  ]
});
```

```
<router-outlet @transition="animation">
  <route-a *route="let route; when: route.data.type === 'a'"></route-a>
  <route-b *route="let route; when: route.data.type === 'b'"></route-b>
</router-outlet>
```

```
route-a {
  transition: opacity 0.5s ease-in-out;
  opacity: 0;
}
route-a[active] {
  opacity: 1;
}
route-b {
  transition: opacity 0.5s ease-in-out;
  opacity: 0;
}
route-b[active] {
  opacity: 1;
}
```

`withViewTransitions` accepts an optional configuration object with a property `onViewTransitionCreated`, which is a callback that provides you some extra control:

- Decide if you'd like to skip particular animations
- Add classes to the document to customize the animation and remove these classes when the animation completes.

```
bootstrapApplication(App, {
  providers: [
    provideRouter(routes, withViewTransitions({
      onViewTransitionCreated: (transition) => {
        if (transition.fromRoute.data.type === 'a' &&
            transition.toRoute.data.type === 'b') {
          transition.skip();
        }
      }
    })),
  ],
});
```

## Defer loading of the animations module

Matthieu Riegler proposed and implemented a feature that allows you to lazily load the animation module via an async provider function:

```
import { provideAnimationsAsync } from '@angular/platform-browser/animations-async';

bootstrapApplication(RootCmp, {
  providers: [provideAnimationsAsync()]
});
```

## Input value transforms

A common pattern is having a component which receives a boolean input. This, however, sets constraints on how you can pass a value to such a component. For example if we have the following definition of an Expander component:

```

@Component({
  standalone: true,
  selector: 'my-expander',
  template: `...`
})
export class Expander {
  @Input() expanded: boolean = false;
}

```

...and we try to use it as:

You'll get an error that “string is not assignable to boolean”. Input value transforms allow you to fix this by configuring the input decorator:

```

@Component({
  standalone: true,
  selector: 'my-expander',
  template: `...`
})
export class Expander {
  @Input({ transform: booleanAttribute }) expanded: boolean = false;
}

```

You can find the original feature requests on GitHub — [Boolean properties as HTML binary attributes](#) and [Boolean properties as HTML binary attributes](#).

## Community schematics

To support the development of community schematics we shipped a couple of utility methods as part of `@schematics/angular/utility`. Now you can import an expression directly into the root of an Angular app and add a provider to the root of an Angular app, plus the already existing feature of adding dependency to `package.json`.

You can learn more in the [schematics guide in the documentation](#).



# Other Less Important Highlights

## Signal

I wish they hadn't integrated it at all, we have to observe if it proves itself.

```
interface Signal<T> {  
  (): T;  
  [SIGNAL]: unknown;  
}
```

```
interface WritableSignal<T> extends Signal<T> {  
  set(value: T): void;  
  update(updateFn: (value: T) => T): void;  
  mutate(mutatorFn: (value: T) => void): void;  
  asReadonly(): Signal<T>;  
}
```

```
function signal<T>(  
  initialValue: T,  
  options?: { equal?: (a: T, b: T) => boolean }  
): WritableSignal<T>
```

```
function computed<T>(  
  computation: () => T,  
  options?: {equal?: (a: T, b: T) => boolean}  
): Signal<T>;
```

```
function effect(  
  effectFn: (onCleanup: (fn: () => void) => void) => void,  
  options?: CreateEffectOptions  
): EffectRef;
```

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

```
@Injectable()
```

```
export class UserService {  
  private users = signal<Array<User>>([]);  
  users = this.users.asReadonly();  
  
  addUser(newUser: User): void {  
    this.users.mutate(users => users.push(newUser));  
  }  
  
  removeUser(id: string): void {  
    this.users.update(users => users.filter(user => user.id !== id));  
  }  
  
  getUser(id: string): User | null {  
    return this.users().find(user => user.id === id) ?? null;  
  }  
  
  resetUsers(): void {  
    this.users.set([]);  
  }  
}
```

```
const user = signal<User>({ id: 'someId', name: 'George Michael', age: 55 });  
const isAdult = computed(() => user().age >= 18));  
const color = computed(() => isAdult() ? 'green' : 'red');  
user.set({ id: 'someOtherId', name: 'Michael Jackson', age: 55 });
```

## Effects

```
const effectRef = effect(onCleanup => {
  const subscription = someObservable.subscribe(value => {
    // do something with the value
  });

  onCleanup(() => {
    // unsubscribe from the observable
    subscription.unsubscribe();
  });
});

function effect(
  effectFn: (onCleanup: (fn: () => void) => void) => void,
  options?: CreateEffectOptions
): EffectRef;

effect((onCleanup) => {
  const countValue = this.count();
  let secsFromChange = 0;
  const logInterval = setInterval(() => {
    console.log(
      `${countValue} had its value unchanged for ${++secsFromChange} seconds`
    );
  }, 1000);
  onCleanup(() => {
    console.log('Clearing and re-scheduling effect');
    clearInterval(logInterval);
  });
});
```

The timing of the effect is not strictly defined and depends on the strategy adopted by Angular. However, we can be sure of some principled rules when working with them. These are as follows:

- The effect will be triggered at least once.
- The effect will be triggered after at least one of the signals on which it depends (reads its value) changes.
- The effect will be called a minimum number of times. That means that if several signals on which the effect depends change their value at one time, the code will be executed only once.

Since the effect responds to a change in the signal on which it depends, it always remains active and ready to respond to changes. By default, its cancellation is done automatically. If the `manualCleanup` option is set, the effect will remain active after the component or directive is destroyed. To cancel it manually, we can use the `EffectRef` instance.

```
const effectRef = effect(() => {...}, { manualCleanup: true });
effectRef.destroy();
```

## Signal-based components

```
@Component({
  signals: true, // <--- HERE: ENABLES SIGNALS
  selector: 'counter',
  template: `
    <p>Counter: {{ count() }}</p>
    <button (click)="increment()">+1</button>
    <button (click)="decrement()">-1</button>`
})
export class CounterComponent {
  count = signal(0);
  increment(): void {
    this.count.update(value => value + 1);
  }
  decrement(): void {
    this.count.update(value => value - 1);
  }
}
```

**Readonly signals can be used as inputs:**

```

@Component({
  signals: true,
  selector: 'user-card',
  template: `
    <div class="user-card" [NgClass]="{ user-card-disabled: disabled() }">
      <p>Name: {{ name() }}</p>
      <p>Role: {{ role() }}</p>
    </div>`
})
export class UserCardComponent {
  name = input<string>(); // Signal<string | undefined>
  role = input('customer'); //Signal<string>
  disabled = input<boolean>(false, { alias: 'deactivated' }); //Signal<boolean>
}

```

## Writable signals:

```

@Component({
  signals: true,
  selector: 'counter'
  template: `
    <p>Counter: {{ count() }}</p>
    <button (click)="increment()">+1</button>`
})
export class CounterComponent {
  count = model(0); //WritableSignal<number>

  increment(): void {
    this.count.update(value => value + 1);
  }
}

```

```

@Component({
  signals: true,
  selector: 'widget'
  template: `
    <counter [(count)]="value" />`
})
export class WidgetComponent {
  value = signal(10);
}

```

## Event Emitter:

```

@Component({
  signals: true,
  selector: 'user-card',
  template: `
    <p>{{ user().name }}</p>
    <button (click)="edit()">Edit</button>
    <button (click)="remove()">Remove</button>`
})
export class UserCardComponent {
  user = input<User>();
  edit = output<User>(); //EventEmitter<User>
  remove = output<string>({ alias: 'delete' }) //EventEmitter<string>

  edit(): void {
    this.edit.emit(this.user());
  }

  remove(): void {
    this.remove.emit(this.user().id);
  }
}

```

## Converting to Signal:

@ViewChild, @ViewChildren, @ContentChild, @ContentChildren that are used for querying element(s) from the template convert to the corresponding functions that return Signal:

```

@Component({
  signals: true,
  selector: 'form-field',
  template: `
    <field-icon [icon]="someIcon" />
    <field-icon [icon]=someAnotherIcon" />
    <input #inputRef />`
})
export class FormFieldComponent {
  icons = viewChildren(FieldIconComponent); //Signal<FieldIconComponent[]>
  input = viewChild<ElementRef>('inputRef'); //Signal<ElementRef>

  errorHandler(): void {
    this.input().nativeElement.focus();
  }
}

```

## New change detection:

New change detection mechanism, changes also affect the area of lifecycle hooks:

```
function afterNextRender(fn: () => void): void;
```

The function is executed after the next change detection cycle is completed. This is useful whenever you want to read or write from the DOM manually.

```
function afterRender(fn: () => void): { destroy(): void }
```

The function is executed after every DOM update during rendering.

```
function afterRenderEffect(fn: () => void): { destroy(): void };
```

This is a special kind of effect that, when triggered (by a change in the signal from which it reads the value), executes at the same time as `afterRender`.

## From the set of previous hooks, the nature of two is left:

- `ngOnInit` is replaced by `afterInit`, and
- `ngOnDestroy` is replaced by `beforeDestroy`.

The timing of their execution is the same as their predecessors. That is, after the component is created and all inputs are set, and before the component is destroyed.

**The other hooks make no sense in the new change detection system and their operation can be replaced by using signals:**

- `ngOnChanges` – used to respond to input changes. Since now the input itself is a signal you can use `computed` to create a new signal based on it or register the corresponding operations inside the *effect*.
- `ngDoCheck` – reacts to each change detection cycle. Its operation can be transferred to *effect*.
- `ngAfterViewInit` – allows you to perform actions after the template is rendered. This place is now taken by `afterNextRender`.
- `ngAfterContentInit`, `ngAfterViewChecked`, `ngAfterContentChecked` – used to observe the results of queries from the view. Since queries are also signal-based and therefore reactive by default, signals can be used directly.

## Integration with RxJS:

```
export function toSignal<T, U extends T|null|undefined>(
  source: Observable<T>,
  options: { initialValue: U, requireSync?: false }): Signal<T|U>;
export function toSignal<T>(
  source: Observable<T>,
  options: { requireSync: true }): Signal<T>;
```

```
const myObservable = toObservable(mySignal);
myObservable.subscribe(console.log);
```

```
mySignal.set(12);
mySignal.set(36);
mySignal.set(23);
//Output: 23
```

## Standalone Components

**Old-school Angular modules disguised as components.**

Disadvantage: All imports must be repeated in the component.

Advantage: All components are lazy loaded.



Neither advantage nor disadvantage: Signal requires standalone components in some situations ([See Local Change Detection](#))

```
content_copy
@Component({
  standalone: true,
  selector: 'photo-gallery',
  imports: [ImageGridComponent],
  template: `
    ... <image-grid [images]="imageList"></image-grid>
  `,
})
export class PhotoGalleryComponent {
  // component logic
}
```

>> [More information to Standalone Components here online.](#)