Part 2 — Structuring User Interfaces with Components

Angular applications are built from components: small, focused building blocks that each own a part of the user interface and its behavior.

In this chapter you will learn:

- How an Angular 20 component is structured
- How the CLI generates components with the new naming scheme
- How to display and control data in templates (with the modern `@if`, `@for`, `@switch` syntax)
- How components talk to each other using inputs and outputs
- How to style components and manage CSS encapsulation
- How lifecycle hooks and change detection work at a high level
- Where older syntax (`*ngIf`, `@Input`, etc.) still appears and how to read it

---

Anatomy of an Angular Component (Angular 20 Style)

In Angular 20, when you generate a component, the CLI now uses **simpler file names**:

- `product-list.ts` (instead of `product-list.component.ts`)
- `product-list.html`
- `product-list.css` (Ninja Squad Blog)

The idea is to reduce redundancy: the file name already tells you what this unit is.

A minimal root component looks like this:

```
// src/app/app.ts
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  templateUrl: './app.html',
  styleUrl: './app.css',
  imports: [RouterOutlet],
  standalone: true
})
export class App {
  title = 'World';
}
```

Explanation:

- `selector: 'app-root'` – the tag you will use in `index.html`.
- `templateUrl` / `styleUrl` – point to the external HTML and CSS files.
- `imports: [RouterOutlet]` – because Angular 16+ uses **standalone components**, every component explicitly imports what it needs (other components, directives, pipes).
- `standalone: true` – tells Angular that this class stands on its own and is not declared in an NgModule.
- The class is named `App` (not `AppComponent`) to match the new naming style.

Legacy note: In older Angular versions, you would typically see:

- File: `app.component.ts`
- Class: `AppComponent`
- No `standalone: true` and no `imports` array (components were declared in NgModules).

Creating a Component with the CLI

To generate a feature component in Angular 20:

```
ng generate component product-list
```

With the new naming convention, this will create:

```
src/app/product-list/product-list.ts
src/app/product-list/product-list.html
src/app/product-list/product-list.css
src/app/product-list/product-list.spec.ts
```

The TypeScript file might look like this:

```
// src/app/product-list/product-list.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-product-list',
  templateUrl: './product-list.html',
  styleUrl: './product-list.css',
  standalone: true
})
export class ProductList {
}
```

Explanation:

- The class name is `ProductList` (not `ProductListComponent`), consistent with the updated Angular 20 style guide.(Ninja Squad Blog)
- This component does not import anything yet; we will add imports later when needed.
- `standalone: true` makes this component directly usable in other components via the `imports` array.

To **use** this component inside `App`, you import it and add it to the `imports` array:

```
// src/app/app.ts
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { ProductList } from './product-list/product-list';
```

```
@Component({
  selector: 'app-root',
  templateUrl: './app.html',
  styleUrl: './app.css',
  standalone: true,
  imports: [RouterOutlet, ProductList]
})
export class App {
  title = 'World';
}
```

And in `app.html`:

```
<!-- src/app/app.html -->
<div class="content">
  <app-product-list></app-product-list>
</div>
```

Explanation:

- Importing `ProductList` in `App` and putting it into `imports` makes Angular aware of the `app-product-list` selector in this template.
- The template then simply uses `<app-product-list>`, which Angular binds to the `ProductList` class.

Displaying Data in the Template

Component templates can render values from the class using **interpolation** or **property binding**.

Interpolation

```
<h1>Hello, {{ title }}</h1>
```

Explanation:

- `{{ title }}` is interpolation; Angular evaluates `title` in the component instance and inserts its string value into the DOM.

Property binding

```
<h1 [innerText]="title"></h1>
```

Explanation:

- `[innerText]="title"` binds the DOM property `innerText` of the `<h1>` to the `title` property of the component.
- The square brackets indicate **one-way binding** from the component to the DOM.

Modern Control Flow: @if, @for, @switch

Angular 17+ introduced a new control-flow syntax that is more readable and more efficient than the older directive-based approach.

Conditional rendering with @if

Example with a product list:

```
// product-list.ts
import { Component } from '@angular/core';

interface Product {
  id: number;
  title: string;
}

@Component({
  selector: 'app-product-list',
  templateUrl: './product-list.html',
  styleUrl: './product-list.css',
  standalone: true
})
export class ProductList {
  products: Product[] = [];
}
```

```
<!-- product-list.html -->
@if (products.length > 0) {
  <h1>Products ({{ products.length }})</h1>
} @else {
  <p>No products found!</p>
}
```

Explanation:

- `@if` decides whether the block of HTML should exist in the DOM at all.
- If `products.length > 0`, Angular adds the `<h1>` to the DOM; otherwise, it adds the `<p>`.

Legacy note (older Angular):

```
<h1 *ngIf="products.length > 0">Products ({{ products.length }})</h1>
<p *ngIf="products.length === 0">No products found!</p>
```

`*ngIf` is still supported, but the new `@if` syntax is the recommended style going forward.

Looping over data with @for

Let’s populate some mock products:

```
// product-list.ts
export class ProductList {
  products: Product[] = [
    { id: 1, title: 'Keyboard' },
    { id: 2, title: 'Microphone' },
    { id: 3, title: 'Web camera' },
    { id: 4, title: 'Tablet' }
  ];
}
```

Now, use `@for` in the template:

```
<!-- product-list.html -->
<ul class="pill-group">
  @for (product of products; track product.id) {
```

```
    <li class="pill">{{ product.title }}</li>
  } @empty {
    <p>No products found!</p>
  }
</ul>
```

Explanation:

- `@for (product of products; track product.id)` iterates over `products` and exposes each item as `product`.
- `track product.id` tells Angular to use the `id` field to keep DOM nodes stable when items change, improving performance.
- `@empty` defines what to show when `products` is an empty array.

Legacy note:

```
<li *ngFor="let product of products">{{ product.title }}</li>
```

`*ngFor` is the older syntax with similar behavior.

Switching templates with @switch

You can pick different content based on a value:

```
<!-- product-list.html -->
<ul class="pill-group">
  @for (product of products; track product.id) {
    <li class="pill">
      @switch (product.title) {
        @case ('Keyboard') { ? }
        @case ('Microphone') { ? }
        @default { ? }
      }
      {{ product.title }}
    </li>
  } @empty {
    <p>No products found!</p>
  }
</ul>
```

Explanation:

- `@switch (product.title)` compares the title for each product.
- `@case` defines what to render when the expression matches a specific value.
- `@default` is rendered when no case matches.

Legacy note:

```
<div [ngSwitch]="product.title">
  <span *ngSwitchCase="'Keyboard'">?</span>
  <span *ngSwitchCase="'Microphone'">?</span>
  <span *ngSwitchDefault>?</span>
</div>
```

Again, `[ngSwitch]` and `*ngSwitchCase` are the older equivalents.

Handling User Interaction (Event Binding)

To send information from the template back to the component, Angular uses **event bindings**.

Extend the `ProductList` class:

```
// product-list.ts
export class ProductList {
  products: Product[] = [
    { id: 1, title: 'Keyboard' },
    { id: 2, title: 'Microphone' },
    { id: 3, title: 'Web camera' },
    { id: 4, title: 'Tablet' }
  ];

  selectedProduct: Product | undefined;
}
```

Update the template:

```
<!-- product-list.html -->
<ul class="pill-group">
  @for (product of products; track product.id) {
    <li class="pill" (click)="selectedProduct = product">
      {{ product.title }}
    </li>
  } @empty {
    <p>No products found!</p>
  }
</ul>

@if (selectedProduct) {
  <p>You selected: <strong>{{ selectedProduct.title }}</strong></p>
}
```

Explanation:

- `(click)="selectedProduct = product"` listens for the browser's `click` event and executes the assignment in the component instance.
- `selectedProduct` becomes the currently clicked product, and the `@if` block below reacts by showing its title.

Styling Components and View Encapsulation

Angular lets you bind classes and styles dynamically.

Class binding

```
<li
  class="pill"
  [class.selected]="selectedProduct && selectedProduct.id ===
product.id"
>
  {{ product.title }}
</li>
```

Explanation:

- `[class.selected]="...condition..."` will add or remove the `selected` class based on whether the condition evaluates to `true` or `false`.

You can also bind an entire object:

```
// product-list.ts
isSelected(product: Product) {
  return this.selectedProduct?.id === product.id;
}
```

```
<li
  class="pill"
  [class.selected]="isSelected(product)"
>
  {{ product.title }}
</li>
```

Style binding

```
<p [style.color]="selectedProduct ? 'green' : 'inherit'">
  {{ selectedProduct ? 'Product chosen' : 'No product selected' }}
</p>
```

Explanation:

- `[style.color]` controls a single style property dynamically, based on component state.

View encapsulation

By default, Angular scopes CSS per component (Emulated mode), so styles from `product-list.css` will only affect that component's template.

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

@Component({
  selector: 'app-product-detail',
  templateUrl: './product-detail.html',
  styleUrl: './product-detail.css',
  standalone: true,
  encapsulation: ViewEncapsulation.Emulated // default
```

```
})
export class ProductDetail {
}
```

If you explicitly set:

```
encapsulation: ViewEncapsulation.None
```

then styles defined in `product-detail.css` can leak into other parts of the app. This can be useful for global styling, but must be used carefully.

Passing Data Between Components (Inputs and Outputs)

Real-world applications rarely keep all UI in a single component. Often, a parent component owns the data and passes a piece of it down to a child component.

Passing data down with input()

Create a detail component:

```
// src/app/product-detail/product-detail.ts
import { Component, input } from '@angular/core';
import type { Product } from '../product-list/product-list';

@Component({
  selector: 'app-product-detail',
  templateUrl: './product-detail.html',
  styleUrl: './product-detail.css',
  standalone: true
})
export class ProductDetail {
  product = input<Product>();
}
```

Template:

```
<!-- product-detail.html -->
@if (product()) {
  <p>
    You selected:
    <strong>{{ product()!.title }}</strong>
  </p>
}
```

Explanation:

- `product = input<Product>()` defines an input signal for this component.
- In the template, `product()` reads the current value of that input.
- The `@if` guard ensures we only render details when a product is actually provided.

Now, use `ProductDetail` in `ProductList`:

```
// product-list.ts
import { Component } from '@angular/core';
import { ProductDetail } from '../product-detail/product-detail';

@Component({
  selector: 'app-product-list',
  templateUrl: './product-list.html',
  styleUrl: './product-list.css',
  standalone: true,
  imports: [ProductDetail]
})
export class ProductList {
  products: Product[] = [ /* ... */ ];
  selectedProduct: Product | undefined;
}
```

```
<!-- product-list.html -->
<ul class="pill-group">
  @for (product of products; track product.id) {
    <li class="pill" (click)="selectedProduct = product">
      {{ product.title }}
    </li>
  } @empty {
    <p>No products found!</p>
  }
</ul>
```

```
<app-product-detail [product]="selectedProduct"></app-product-detail>
```

Explanation:

- `[product]="selectedProduct"` binds the parent's `selectedProduct` property into the child's `product` input.
- Angular takes care of updating the child when the parent selection changes.

Legacy note: Previously, you would see:

```
@Input() product!: Product;
```

instead of `product = input<Product>()`.

Sending events up with output()

Let the detail component notify the parent that the user wants to add the product to a cart.

In `ProductDetail`:

```
import { Component, input, output } from '@angular/core';
import type { Product } from '../product-list/product-list';

@Component({
  selector: 'app-product-detail',
  templateUrl: './product-detail.html',
  styleUrl: './product-detail.css',
  standalone: true
})
export class ProductDetail {
  product = input<Product>();
  added = output<Product>();

  addToCart() {
    if (this.product()) {
      this.added.emit(this.product()!);
    }
  }
}
```

Template:

```
<!-- product-detail.html -->
@if (product()) {
  <div>
    <p>
      You selected:
      <strong>{{ product()!.title }}</strong>
    </p>
    <button (click)="addToCart()">Add to cart</button>
  </div>
}
```

Explanation:

- `added = output<Product>()` declares an output event that can carry a `Product` payload.
- `this.added.emit(...)` triggers the event.

In the parent (`ProductList`):

```
// product-list.ts
onAdded(product: Product) {
  alert(`${product.title} added to the cart!`);
}
```

```
<!-- product-list.html -->
<app-product-detail
  [product]="selectedProduct"
  (added)="onAdded($event)"
></app-product-detail>
```

Explanation:

- `(added)="onAdded($event)"` listens to the child's `added` output.
- `$event` contains the product emitted by `addToCart()`.
- The parent can now update a cart, fire analytics, or display a message.

Legacy note: Older Angular projects use:

```
@Output() added = new EventEmitter<Product>();
```

instead of `added = output<Product>()`.

Template Reference Variables and viewChild

Sometimes you need direct access to a child component instance.

Template reference variable

```
<!-- product-list.html -->
<app-product-detail
  #detail
  [product]="selectedProduct"
  (added)="onAdded($event)"
></app-product-detail>

<p *ngIf="detail.product()">
  Detail says: {{ detail.product()!.title }}
</p>
```

Explanation:

- `#detail` creates a template reference to the `ProductDetail` instance.
- This reference exposes the public API of `ProductDetail` (here: `product()`).

Querying a child in TypeScript with viewChild

You can also get the child instance from the parent class:

```
// product-list.ts
import { Component, AfterViewInit, viewChild } from '@angular/core';
import { ProductDetail } from '../product-detail/product-detail';

@Component({
  selector: 'app-product-list',
```

```
  templateUrl: './product-list.html',
  styleUrl: './product-list.css',
  standalone: true,
  imports: [ProductDetail]
})
export class ProductList implements AfterViewInit {
  productDetail = viewChild(ProductDetail);

  ngAfterViewInit(): void {
    console.log('Detail product:', this.productDetail()?.product());
  }
}
```

Explanation:

- `viewChild(ProductDetail)` tells Angular to look for a `ProductDetail` in this component's view.
- `ngAfterViewInit` is the hook where the child is guaranteed to be created and accessible.

Legacy note: Previously:

```
@ViewChild(ProductDetail) productDetail!: ProductDetail;
```

Change Detection Strategy

Angular automatically refreshes views when data changes. By default, it runs change detection for the entire component tree on each relevant event.

You can optimize this with `ChangeDetectionStrategy.OnPush`:

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

@Component({
  selector: 'app-product-detail',
  templateUrl: './product-detail.html',
  styleUrl: './product-detail.css',
  standalone: true,
```

```
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class ProductDetail {
  // ...
}
```

Explanation:

- With `OnPush`, Angular will only re-check this component when:
  - An input reference changes
  - An event handler on this component runs
  - An observable bound in the template emits (via async pipe), etc.
- This significantly improves performance in large and complex UIs.

Lifecycle Hooks Overview

Lifecycle hooks allow you to run custom logic at specific moments in a component's life.

Common hooks:

- `ngOnInit` – runs after the component's inputs are first set.
- `ngOnDestroy` – runs right before Angular removes the component from the DOM.
- `ngOnChanges` – runs whenever an input binding changes.
- `ngAfterViewInit` – runs after the view and child views have been initialized.

Example:

```
import {
  Component,
  OnInit,
  OnDestroy,
  OnChanges,
  SimpleChanges,
```

```
  AfterViewInit
} from '@angular/core';

@Component({
  selector: 'app-product-detail',
  templateUrl: './product-detail.html',
  styleUrl: './product-detail.css',
  standalone: true
})
export class ProductDetail
  implements OnInit, OnDestroy, OnChanges, AfterViewInit {

  ngOnInit(): void {
    // Good place to fetch data or initialize values
    console.log('ProductDetail initialized');
  }

  ngOnChanges(changes: SimpleChanges): void {
    // React to input changes
    console.log('Changes:', changes);
  }

  ngAfterViewInit(): void {
    // Child components and view are ready
    console.log('View initialized');
  }

  ngOnDestroy(): void {
    // Cleanup: timers, subscriptions, listeners, etc.
    console.log('ProductDetail destroyed');
  }
}
```

Explanation:

- Each hook gives you a predictable place to put specific kinds of logic:
  - `ngOnInit` instead of doing heavy work in the constructor.
  - `ngOnDestroy` to release resources.
  - `ngOnChanges` to react to new input values.
  - `ngAfterViewInit` to work with child components or DOM elements that weren’t available earlier.

Summary

In this chapter you have seen how, in **Angular 20**:

- Components are generated with **simpler file names** like `product-list.ts`, `product-list.html`, `product-list.css`.
- Standalone components (`standalone: true`) and explicit `imports` have become the default way to structure an app.
- The modern control-flow syntax (`@if`, `@for`, `@switch`) replaces older structural directives in new code, while you still need to understand `*ngIf`, `*ngFor` and `ngSwitch` for legacy templates.
- Data flows into components via `input()` and out via `output()`, replacing `@Input` and `@Output` in new code.
- Class and style bindings, along with view encapsulation, give you fine control over component-level CSS.
- Template reference variables and `viewChild` let you reach deeper into the component tree when necessary.
- Change detection strategies and lifecycle hooks help you tune both performance and behavior.

Revision #4

Created 2025-11-19 16:19:48 UTC by Carsten

Updated 2025-12-17 19:58:11 UTC by Carsten