Angular Learning

A beginner-friendly path from JavaScript to Angular

42 chapters

Contents

JAVASCRIPT

1 JavaScript, TypeScript, and Angular
2 Conditions and Loops
3 The DOM, jQuery, and Angular
4 Functions and Arrow Functions
5 Arrays and Array Methods
6 Strings: Joining and Templates
7 Objects
8 Classes
9 Modules: import and export
10 Promises and async / await
11 A Taste of REST and fetch

TYPESCRIPT

12 TypeScript Basics
13 Interfaces and Object Types
14 Typed Functions and Generics

ANGULAR

15 Getting Started with Angular
16 Your First App: Hello World
17 Files and Folder Structure
18 Signals: Angular's Reactivity
19 Components: The Basics
20 Templates and Data Binding
21 Inputs and Outputs
22 Two-way Binding
23 Control Flow in Templates
24 Component Lifecycle
25 Styles and Content Projection
26 Pipes
27 Directives
28 Template-driven Forms
29 Reactive Forms
30 Custom Form Controls (ControlValueAccessor)
31 Routing
32 Route Guards
33 REST APIs In Depth
34 CRUD with JSON Server

35 RxJS and Observables
36 HttpClient in a Component
37 Services and Dependency Injection
38 HttpClient with a Service
39 Sharing Data with a Service
40 Environments
41 HTTP Interceptors
42 Authentication Flow

CHAPTER 1

JavaScript, TypeScript, and Angular

Meet the three tools you'll use, and see how they team up.

1. What is JavaScript?

Think of a web page like a body. HTML is the bones that give it shape, CSS is the clothes that make it look nice, and JavaScript is the muscles that make it move. Anything that reacts when you click, type, or scroll is JavaScript doing its job.

HTML	CSS	JavaScript
Structure	Style	Behavior

More on javascript.info ↗

2. What is TypeScript?

TypeScript is just JavaScript with a helpful safety net. You tell it what kind of value a variable should hold — like a number or some text — and it warns you about mistakes before your app ever runs.

JavaScript

```
let age = 25;
```

TypeScript

```
let age: number = 25;
```

More on typescriptlang.org ↗

3. What is Angular?

Angular is a framework — basically a big box of ready-made tools for building web apps. Instead of building everything from scratch, it hands you components, data binding, routing, and forms. You write it all in TypeScript.

4. How they connect

Angular
The toolkit you build with

TypeScript
The language you write in

JavaScript
What actually runs in the browser

> *An easy way to remember it: Angular is the toolkit, TypeScript is the friendly language you write in, and JavaScript is what actually runs in the browser.*

5. Variables

A variable is just a labelled box where you keep a value so you can use it again later.

const
For values that never change (start here)

let
For values that might change

var
The old way — best to avoid it

let

```
let marks = 80;
marks = 90;
```

const

```
const appName = "Playground";
```

More on javascript.info ↗

6. Recap

- JavaScript makes a page actually do things.
- TypeScript adds a safety net and catches mistakes early.
- Angular uses TypeScript to build bigger apps in a tidy way.
- Reach for `const` first, and use `let` only when the value needs to change.

CHAPTER 2

Conditions and Loops

Teach your code to make choices and to repeat itself.

---

1. if / else

Sometimes you only want code to run in certain situations. `if` runs something when a condition is true, and `else` handles everything else.

if / else

```
let age = 18;
if (age ≥ 18) {
  // runs when true
} else {
  // runs otherwise
}
```

Example

```
<p id="out"></p>

<script>
  let age = 18;
  if (age ≥ 18) {
    document.getElementById("out").textContent = "Adult";
  } else {
    document.getElementById("out").textContent = "Minor";
  }
</script>
```

Output

Adult

More on javascript.info ↗

2. Comparisons and logic

To build a condition, you compare values and then combine the answers. These are the ones you'll reach for most often.

- `===` is equal, `≠=` is not equal
- `>` `<` `≥` `≤` compare numbers
- `&&` means and, `||` means or, `!` means not

More on javascript.info ↗

3. Loops

A loop lets you repeat the same code without copying it over and over. A `for` loop runs a set number of times.

for

```
for (let i = 1; i ≤ 5; i++) {
  // runs 5 times
}
```

Example

```
<ul id="list"></ul>

<script>
  for (let i = 1; i ≤ 5; i++) {
    const li = document.createElement("li");
    li.textContent = "Item " + i;
    document.getElementById("list").appendChild(li);
  }
</script>
```

Output

- Item 1
- Item 2
- Item 3
- Item 4
- Item 5

More on javascript.info ↗

4. Looping over a list

When you have a list, `for ... of` hands you one item at a time so you can work with each one.

Example

```
<p id="out">&lt;/p&gt;

<script>
  const fruits = ["apple", "mango", "kiwi"];
  let result = "";
  for (const fruit of fruits) {
    result += fruit + " ";
  }
  document.getElementById("out").textContent = result;
</script>
```

Output

apple mango kiwi

5. Recap

- `if` / `else` decides which code runs.
- Compare values with `===`, and join conditions with `&&` (and) / `||` (or).
- `for` repeats a set number of times.
- `for ... of` goes through a list one item at a time.

CHAPTER 3

The DOM, jQuery, and Angular

See how JavaScript actually changes the page — and how that idea grew into Angular.

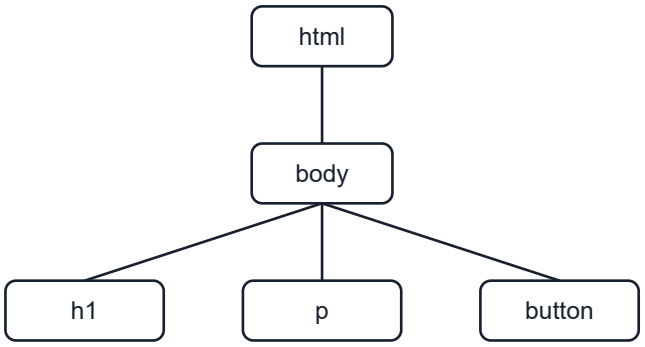
1. What is the DOM?

When the browser opens your HTML, it quietly turns the page into a tree of elements. That tree is called the **DOM**, and JavaScript can read it and change it while the page is open.

HTML
What you write

DOM tree
What the browser builds from it

JavaScript
Reads and changes it



The browser turns your HTML into a tree like this.

More on javascript.info ↗

2. Working with the DOM

Changing the page is usually two simple steps: first find the element you want, then change it.

Find

```
const h = document.getElementById("title");
```

Change

```
h.textContent = "Hello";
```

- `querySelector()` — grabs the first element that matches a CSS selector
- `addEventListener()` — runs your code when something happens, like a click
- `textContent` / `innerHTML` — change what's shown inside an element

Example

```
<h2 id="title">Hello</h2>
<button id="btn">Change text</button>

<script>
  document.getElementById("btn").addEventListener("click", function () {
    document.getElementById("title").textContent = "You changed me!";
  });
</script>
```

Output

Hello

Change text

More on javascript.info ↗

3. Why it matters

Changing the DOM is what makes a page feel alive instead of frozen. Try the counter below — click it a few times, then tweak the code and run it again.

User action → JavaScript runs → DOM changes → Page updates

Example

```
<button id="b">Clicked 0 times</button>

<script>
  let count = 0;
  document.getElementById("b").addEventListener("click", function () {
    count = count + 1;
    document.getElementById("b").textContent = "Clicked " + count + " times";
  });
</script>
```

Output

Clicked 0 times

Example

```
<input id="name" placeholder="Type your name">
<p id="out"></p>

<script>
  document.getElementById("name").addEventListener("input", function (e) {
    document.getElementById("out").textContent = "Hello, " + e.target.value;
  });
</script>
```

Output

Type your name

4. From JavaScript to Angular

Plain JavaScript gave you full control, but a lot of it was manual and repetitive. jQuery made the same work shorter and easier. As apps got bigger, people needed real structure — and that's where frameworks like Angular came in.

JavaScript → jQuery → Angular

> *Short version: JavaScript gave power, jQuery gave convenience, and Angular gave structure.*

5. Recap

- The DOM is the browser's live version of your page.
- JavaScript uses it to find elements and change them.
- jQuery made that easier, and Angular brought structure to whole apps.

CHAPTER 4

Functions and Arrow Functions

Wrap code so you can reuse it — and meet the shorter arrow style.

1. What is a function?

A function is a chunk of code with a name, so you can run it whenever you need it instead of writing it again. You can hand it **inputs** (called parameters), and it can hand back a **result** (the return value).

A function takes inputs and gives back a result.

Declaration

```
function add(a, b) {
  return a + b;
}

add(2, 3); // 5
```

Example

```
<p id="out"></p>

<script>
  function add(a, b) {
    return a + b;
  }
  document.getElementById("out").textContent = "add(2, 3) = " + add(2, 3);
</script>
```

Output

add(2, 3) = 5

Functions and parameters at a glance

More on javascript.info ↗

2. Arrow functions

Arrow functions are just a shorter way to write the same thing. When the body is a single expression, you can drop the `return` and the curly braces.

Normal

```
function add(a, b) {
  return a + b;
}
```

Arrow

```
const add = (a, b) => a + b;
```

Example

```
<p id="out">
</p>

<script>
  const add = (a, b) => a + b;
  const square = n => n * n;
  document.getElementById("out").textContent =
    "add(2,3)=" + add(2, 3) + ", square(5)=" + square(5);
</script>
```

Output

add(2,3)=5, square(5)=25

Arrow functions at a glance

More on javascript.info ↗

3. Recap

- A function bundles up code you can reuse anytime.
- Parameters are the inputs; `return` sends back the result.
- Arrow functions are a shorter style: `(a, b) => a + b`.
- If the body is a single expression, it returns on its own — no `return` needed.

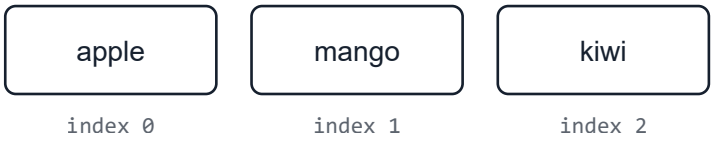
CHAPTER 5

Arrays and Array Methods

Keep many values in one list, then loop through them and reshape them.

1. What is an array?

An array is a single list that holds many values in order. You reach each value by its position number, called the **index** — and counting starts at 0, not 1.



Each item sits at a numbered position, starting at 0.

Array

```
const fruits = ["apple", "mango", "kiwi"];
fruits[0];       // "apple"
fruits.length;   // 3
```

Example

```
<p id="out"></p>

<script>
  const fruits = ["apple", "mango", "kiwi"];
  document.getElementById("out").textContent =
    "first=" + fruits[0] + ", count=" + fruits.length;
</script>
```

Output

first=apple, count=3

More on javascript.info ↗

2. Looping over an array

`forEach` runs a little function once for every item in the list. Arrow functions keep it nice and short.

Example

```
<ul id="list"></ul>

<script>
  const fruits = ["apple", "mango", "kiwi"];
  fruits.forEach(fruit => {
    const li = document.createElement("li");
    li.textContent = fruit;
    document.getElementById("list").appendChild(li);
  });
</script>
```

Output

- apple
- mango
- kiwi

3. map and filter with arrow functions

`map` takes each item and builds a brand-new list from the results. `filter` keeps only the items that pass a test you give it.

map

```
[1, 2, 3].map(n => n * 2);
// [2, 4, 6]
```

filter

```
[1, 2, 3, 4].filter(n => n % 2 === 0);
// [2, 4]
```

Example

```
<p id="out">&lt;/p&gt;

<script>
  const nums = [1, 2, 3, 4, 5];
  const doubled = nums.map(n => n * 2);
  const evens = nums.filter(n => n % 2 === 0);
  document.getElementById("out").textContent =
    "doubled: " + doubled.join(", ") + "  |  evens: " + evens.join(", ");
</script>
```

Output

doubled: 2, 4, 6, 8, 10 | evens: 2, 4

More on javascript.info ↗

4. Recap

- An array is an ordered list, and counting starts at 0.
- `forEach` runs your code for each item.
- `map` builds a new, changed list.
- `filter` keeps only the items that pass your test.
- Arrow functions make these read like one clean line.

CHAPTER 6

Strings: Joining and Templates

Build text from pieces using + and the cleaner backtick style.

1. Joining text with +

A string is just text. To build a bigger string from smaller pieces, you can glue them together with the + sign. This is called concatenation.

Joining

```
const name = "Asha";
"Hello " + name + "!"; // "Hello Asha!"
```

Example

```
<p id="out"></p>

<script>
  const name = "Asha";
  const age = 20;
  document.getElementById("out").textContent =
    "Hello " + name + ", age " + age;
</script>
```

Output

Hello Asha, age 20

More on javascript.info ↗

2. Template literals (backticks)

All those + signs get messy fast. Instead, you can wrap text in backticks `` and drop values straight in with ${ }. It reads much more naturally.

With +

```
"Hello " + name + ", age " + age
```

With ``

```
`Hello ${name}, age ${age}`
```

Example

```
<p id="out"></p>

<script>
  const name = "Asha";
  const age = 20;
  document.getElementById("out").textContent = `Hello ${name}, age ${age}`;
</script>
```

Output

Hello Asha, age 20

> *Angular's `{{ name }}` in templates is the same idea — drop a value right into your text.*

3. Recap

- A string is text, written in quotes.
- + joins strings together.
- Backticks with `${ }` are a cleaner way to mix text and values.
- This is the same idea as Angular's `{{ }}`.

CHAPTER 7

Objects

Group related values together under one name.

1. What is an object?

An object groups related values together as named pairs. Instead of three loose variables for a user, you keep them in one tidy bundle. Each name is called a **property**.

Object

```
const user = {
  name: "Asha",
  age: 20,
  isActive: true
};
```

Objects at a glance

More on javascript.info ↗

2. Reading and changing properties

Use a dot to read a property or to change it.

Read

```
user.name; // "Asha"
```

Change

```
user.age = 21;
```

Example

```
<p id="out"></p>

<script>
  const user = { name: "Asha", age: 20 };
  user.age = 21;
  document.getElementById("out").textContent =
    user.name + " is now " + user.age;
</script>
```

Output

Asha is now 21

> *An Angular component keeps its data as properties just like this — and your template reads them with `{{ user.name }}`.*

3. Recap

- An object bundles related values under one name.
- Each value is a property, written as `name: value`.
- Read or change a property with a dot: `user.age`.
- Angular components hold their data the same way.

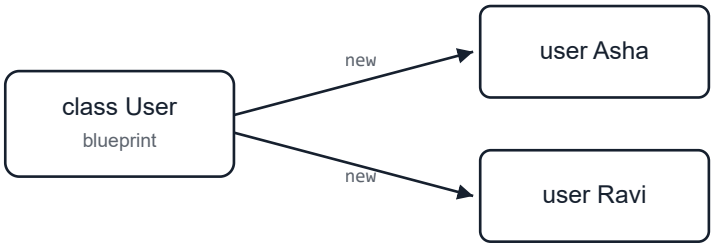
CHAPTER 8

Classes

A blueprint for making objects — and the shape of every Angular component.

1. What is a class?

A class is a blueprint for making objects. You describe what data it holds and what it can do once, then create as many objects from it as you like with `new`.



One class (blueprint) can create many objects.

Class

```
class User {
  constructor(name) {
    this.name = name;
  }
  greet() {
    return "Hi, " + this.name;
  }
}

const u = new User("Asha");
u.greet(); // "Hi, Asha"
```

More on javascript.info ↗

2. The parts of a class

constructor
Runs when you create the object; sets up its data

this
Refers to the object you're working with

methods
Functions the object can do, like greet()

Example

```
<p id="out"></p>

<script>
  class User {
    constructor(name) {
      this.name = name;
    }
    greet() {
      return "Hi, " + this.name;
    }
  }

  const u = new User("Asha");
  document.getElementById("out").textContent = u.greet();
</script>
```

Output

Hi, Asha

> *An Angular component is just a TypeScript class with a `@Component` label on top. Everything you learn here applies directly.*

3. Recap

- A class is a blueprint; `new` builds an object from it.
- The `constructor` sets up the object's data.
- `this` points to the current object.
- Methods are the things the object can do.
- Angular components are classes — so you already know the shape.

CHAPTER 9

Modules: import and export

Split code across files and share it between them.

---

1. Why modules?

Real apps are too big for one file. Modules let you split your code into separate files and share pieces between them. You **export** what a file offers, and **import** what you need elsewhere.

More on javascript.info ↗

2. Sharing code with export

Put `export` in front of anything you want other files to be able to use.

math.js

```
export function add(a, b) {
  return a + b;
}

export const PI = 3.14;
```

3. Pulling it in with import

In another file, `import` the names you need and use them.

app.js

```
import { add, PI } from "./math.js";

add(2, 3); // 5
PI;        // 3.14
```

> *Every Angular file starts this way. `import { Component } from '@angular/core';` is exactly this — pulling in code from another module.*

4. Recap

- Modules split your code across many files.
- `export` shares something from a file.
- `import` brings it into another file.
- Angular files import the tools they need at the top.

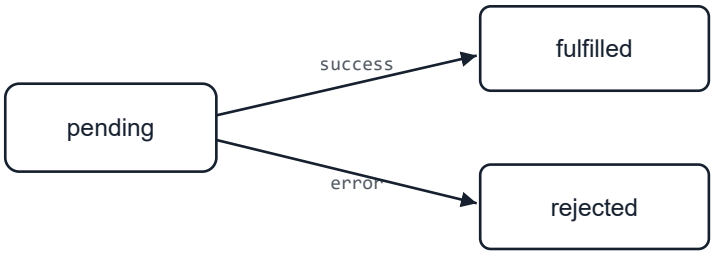
CHAPTER 10

Promises and async / await

How JavaScript handles things that take time, like loading data.

1. What is a Promise?

Some things take time — loading data, a timer, an image. JavaScript doesn't wait around for them; it keeps going and deals with the result later. A **Promise** is a placeholder for a value that isn't ready yet. It promises to give you the result once it's done — whether it worked or failed.



A Promise starts pending, then settles as fulfilled or rejected.

More on javascript.info ↗

2. Handling the result: .then and .catch

You can't use the value right away, so you hand the Promise a function to run once it's ready. `.then(...)` handles success, and `.catch(...)` handles errors. Press Run and wait one second.

Example

```
<p id="out">Waiting...</p>

<script>
  const wait = new Promise(function (resolve) {
    setTimeout(function () {
      resolve("Done after 1 second!");
    }, 1000);
  });

  wait.then(function (message) {
    document.getElementById("out").textContent = message;
  });
</script>
```

Output

Done after 1 second!

3. A cleaner way: async / await

Chaining lots of `.then(...)` calls gets hard to read. `async` / `await` lets you write code that waits, but still reads top to bottom like normal steps. `await` pauses until the Promise is ready.

.then

```
wait().then(msg => {
  show(msg);
});
```

async / await

```
const msg = await wait();
show(msg);
```

Example

```
<p id="out">Waiting...</p>

<script>
  function wait(ms) {
    return new Promise(function (resolve) {
      setTimeout(resolve, ms);
    });
  }

  async function run() {
    document.getElementById("out").textContent = "Loading...";
    await wait(800);
    document.getElementById("out").textContent = "Finished!";
  }

  run();
</script>
```

Output

Finished!

More on javascript.info ↗

4. Recap

- A Promise stands for a value that will arrive later.
- It starts *pending*, then becomes *fulfilled* (success) or *rejected* (error).
- `.then(...)` handles success; `.catch(...)` handles errors.
- `async` / `await` is a cleaner way to write the same thing.
- `fetch` (next chapter) returns a Promise.

CHAPTER 11

A Taste of REST and fetch

A first peek at loading data from the internet. We'll go deeper later.

1. What is a REST API?

Most apps don't keep all their data inside themselves — they ask a server for it. A REST API is just a set of web addresses you can call to get or send data. Your code makes a request, the server sends a response back.

REST and fetch at a glance

Your app asks; the server answers.

> *This is a quick taste only — we'll cover REST properly in a later chapter.*

2. Loading data with fetch

`fetch` is the built-in way to call an API. It returns the response a moment later, so we use `.then(...)` to handle the data once it arrives. Press Run to load a real item from a free practice API.

Example

```
<p id="out">Loading...</p>

<script>
  fetch("https://jsonplaceholder.typicode.com/todos/1")
    .then(response => response.json())
    .then(data => {
      document.getElementById("out").textContent = "Title: " + data.title;
    })
    .catch(() => {
      document.getElementById("out").textContent = "Could not load (check your connection).";
    });
</script>
```

Output

Title: delectus aut autem

More on javascript.info ↗

3. Recap

- A REST API is a set of web addresses for getting and sending data.
- Your app makes a request; the server sends a response.
- `fetch` calls an API, and `.then(...)` handles the data when it arrives.
- We'll dig into REST in depth in a later chapter.

CHAPTER 12

TypeScript Basics

Add types to your values so mistakes are caught before the app runs.

1. Why types?

Plain JavaScript lets you put anything in any variable, so a typo or wrong value only blows up when the app runs. TypeScript lets you say what kind of value belongs where, and warns you while you type — long before a user ever sees it.

Write TypeScript → Types checked → Compiles to JavaScript → Runs in browser

More on typescriptlang.org ↗

2. Type annotations

Add a type after a colon. These are the everyday ones you'll use constantly.

types

```
let age: number = 25;
let name: string = "Asha";
let isActive: boolean = true;
let scores: number[] = [80, 90, 75];
```

3. Type inference

You don't always have to write the type. If you give a value right away, TypeScript figures the type out for you — and still protects it.

inference

```
let city = "Delhi"; // inferred as string

city = 42; // Error: a number isn't a string
```

> *Let TypeScript infer when it's obvious; add a type yourself when it makes the code clearer.*

4. any and unknown

any turns the safety off for a value — avoid it. unknown is the safe version: you must check what it is before using it.

any vs unknown

```
let a: any = "hello";
a.foo.bar; // allowed, but risky

let u: unknown = "hello";
// u.length; // not allowed until you check the type
```

> *Reaching for any a lot usually means you're fighting the types. Prefer real types or unknown.*

5. Recap

- TypeScript checks types before the code runs.
- Annotate with a colon: `age: number`.
- Common types: `string`, `number`, `boolean`, `number[]`.
- Inference fills in obvious types for you.
- Avoid `any`; use real types or `unknown`.

CHAPTER 13

Interfaces and Object Types

Describe the shape of your objects so they stay consistent.

---

1. Describing an object's shape

Objects (Chapter 7) hold named values. An **interface** describes the shape an object must have — which properties it needs and what type each one is.

interface

```
interface User {
  name: string;
  age: number;
  isActive: boolean;
}
```

More on typescriptlang.org ↗

2. Using an interface

Once you have a shape, you can use it as a type. TypeScript then checks that every object matches it — no missing or misspelled properties.

using it

```
const user: User = {
  name: "Asha",
  age: 20,
  isActive: true
};

// Missing 'age' would be an error
```

3. Optional and readonly properties

A `?` marks a property as optional. `readonly` means it can be set once but not changed afterwards.

options

```
interface Product {
  readonly id: number;  // can't be changed
  name: string;
  discount?: number;    // optional
}
```

4. Type aliases and unions

A `type` alias gives a name to any type. A **union** (`|`) lets a value be one of several types — handy for fixed sets of options.

type

```
type ID = string | number;
type Status = "active" | "paused" | "closed";

let userId: ID = 101;
let state: Status = "active";
```

> *Rule of thumb: use `interface` for object shapes, and `type` for unions and aliases.*

5. Recap

- An `interface` describes an object's shape.
- Use it as a type so objects must match it.
- `?` makes a property optional; `readonly` locks it.
- `type` names any type; `|` builds a union of options.
- Interfaces for shapes, types for unions.

CHAPTER 14

Typed Functions and Generics

Give functions typed inputs and outputs, and reuse types with generics.

---

1. Typing functions

Functions (Chapter 4) get types too: one for each parameter, and one for what they return. Now TypeScript checks that you call them correctly.

typed function

```
function add(a: number, b: number): number {
  return a + b;
}

add(2, 3);      // ok
add(2, "3");    // Error: "3" is not a number
```

Typed functions at a glance

More on typescriptlang.org ↗

2. Functions with union types

Parameters can use unions too, so a function only accepts the values you allow.

union param

```
function setStatus(state: "on" | "off") {
  // ...
}

setStatus("on");    // ok
setStatus("maybe"); // Error
```

3. Generics: types as inputs

Sometimes a function should work with *any* type but still keep that type. A **generic** uses a placeholder (often T) that is filled in when the function is used.

generic

```
function first<T>(items: T[]): T {
  return items[0];
}

first<string>(["a", "b"]); // returns a string
first([1, 2, 3]);          // T inferred as number
```

> *You've already used generics: `signal<number>(0)` and `input<string>()` from the Angular chapters work exactly this way.*

4. Recap

- Type each parameter and the return value of a function.
- Union parameters limit which values are allowed.
- Generics keep a type while staying reusable: <T>.

- The type can be passed in or inferred from the arguments.
- Angular's `signal<T>` and `input<T>` are generics.

CHAPTER 15

Getting Started with Angular

Install the tools and create your first Angular app.

1. What you need first

Angular runs on **Node.js**, which also gives you **npm** — the tool that installs code packages. Install Node first, and npm comes along with it. Pick the LTS (long-term support) version.

Install Node.js → Install Angular CLI → ng new → ng serve

Download Node.js ↗

2. Install the Angular CLI

The **Angular CLI** is a command-line tool that creates apps, runs them, and builds them for you. Install it once, globally, with npm.

Terminal

```
npm install -g @angular/cli
```

Check it worked by asking for its version:

Terminal

```
ng version
```

More on angular.dev ↗

3. Create a new app

Use `ng new` and give your app a name. The CLI asks a couple of setup questions (like which stylesheet format), then builds the whole project folder for you.

Terminal

```
ng new my-app
```

> *This can take a minute — the CLI is downloading and wiring up everything your app needs.*

4. Run it in the browser

Move into the new folder and start the development server. It watches your files and reloads the page automatically whenever you save.

Terminal

```
cd my-app
ng serve
```

Now open `http://localhost:4200` in your browser. That's your app running.

More on angular.dev ↗

5. Recap

- Install **Node.js** (LTS) — it includes npm.
- Install the CLI once: `npm install -g @angular/cli`.
- Create an app: `ng new my-app`.
- Run it: `cd my-app` then `ng serve`.
- Open `http://localhost:4200` to see it.

CHAPTER 16

Your First App: Hello World

Edit the starter app and show your own message on the screen.

1. What ng new gave you

After `ng new` and `ng serve` (Chapter 15), you saw a starter page in the browser. That page is one component — the root component, `App`. Let's open it and make it say hello our way.

The root component is two files that work together: the class (the data and logic) and the template (the HTML it shows).

app.ts (the class)

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

@Component({
  selector: 'app-root',
  templateUrl: './app.html'
})
export class App {
  title = 'Hello, World!';
}
```

app.html (the template)

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

2. How it reaches the screen

The class has a property `title`. The template reads it with `{{ title }}` — the interpolation from Chapter 6. Angular renders the root component into the `<app-root>` tag inside `index.html`, and that's what you see.

title in class → {{ title }} in template → Rendered into <app-root> → Shown in browser

3. Try it live

Edit the code below and press Run. Change the `username` value, or the text in the template, and watch the output update.

component.ts

```
username = 'World';
```

component.html

```
<h1>Hello, {{ username }}!</h1>
<p>Welcome to your first Angular app.</p>
```

Output

Hello, World!

Welcome to your first Angular app.

> *Change `username = 'World'` to your own name and press Run — the heading updates, because the template just shows whatever the class holds.*

4. Recap

- The starter app is a single root component, `App`.
- A component is a class (data) plus a template (HTML).
- `{{ title }}` shows a class property on the page.
- Angular renders the root component into `<app-root>` in `index.html`.
- Change the class value and the screen follows.

CHAPTER 17

Files and Folder Structure

A tour of what the CLI creates and which files matter most.

1. The big picture

It looks like a lot of files at first, but you'll only touch a few of them day to day. Here's the shape of a fresh app.

my-app/

```
my-app/
├ src/
│  ├ app/
│  │  ├ app.ts          # root component (a class)
│  │  ├ app.html        # its template
│  │  ├ app.css         # its styles
│  │  ├ app.config.ts   # app-wide setup
│  │  └ app.routes.ts   # page routes
│  ├ main.ts            # starts the app
│  ├ index.html         # the single HTML page
│  └ styles.css         # global styles
├ public/               # images & static files
├ angular.json          # build settings
├ package.json          # dependencies & scripts
└ tsconfig.json         # TypeScript settings
```

> *Depending on the Angular version, the root files may be named `app.ts` or `app.component.ts`. The idea is the same either way.*

2. The files you'll actually use

src/app/
Your components live here — this is where you spend most of your time

main.ts
The starting point that boots the app (you rarely change it)

index.html
The one real HTML page; your whole app loads inside it

styles.css
Global styles for the entire app

app.config.ts
App-wide setup, like the router and HttpClient

package.json
Lists your dependencies and the npm scripts (like start and build)

More on angular.dev ↗

3. Inside a component

A component is usually a small group of files with the same name: the class, its template, and its styles. The class ties them together with a `@Component` label on top.

hello.ts

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

@Component({
  selector: 'app-hello',
  templateUrl: './hello.html',
  styleUrl: './hello.css'
})
export class Hello {
  name = 'World';
}
```

> *Look closely: it's just a class (Chapter 8) with a label, it imports from a module (Chapter 9), and the template shows data with {{ }} (Chapter 6). You already know the pieces.*

4. Recap

- Most of your work happens in `src/app/`.
- `main.ts` starts the app; `index.html` hosts it.
- `styles.css` is global; component styles stay with the component.
- `angular.json` and `package.json` configure the project.
- A component is a class + template + styles, joined by `@Component`.

CHAPTER 18

Signals: Angular's Reactivity

Values that know when they change — and update the screen for you.

1. What is a signal?

A **signal** is a value that knows when it changes. When you update a signal, anything that depends on it — other values, and the screen — updates automatically. You no longer have to manually find an element and change it like we did with the DOM; Angular tracks the connections for you.

Change the signal, and everything that depends on it updates by itself.

More on angular.dev ↗

2. Reading and changing a signal

Create a signal with `signal(startValue)`. To read it, you **call it like a function** with `()`. To change it, use `set` (a new value) or `update` (based on the old value).

signal

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

const count = signal(0);

count();            // read   → 0
count.set(5);       // set    → 5
count.update(n => n + 1); // update → 6
```

> *In a template you read it the same way: `{{ count() }}`. When `count` changes, that spot on the screen refreshes on its own.*

Try the reactive idea live — click the button and the number on screen updates by itself. (The mini-preview uses a plain property; in a real app this would be a `signal`, but the auto-updating behaviour is the same.)

component.ts

```
count = 0;
increase() {
  count = count + 1;
}
```

component.html

```
<button (click)="increase()">You clicked {{ count }} times</button>
```

Output

You clicked 0 times

3. Derived values with computed

A `computed` signal is built from other signals. You never set it directly — it recalculates itself whenever the signals it reads change.

computed

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

const count = signal(2);
const double = computed(() => count() * 2);

double();         // 4
count.set(10);
double();         // 20  (updated by itself)
```

4. Reacting with effect

An `effect` runs a piece of code whenever the signals it uses change. It's for side effects — things like saving to storage or logging — not for calculating values.

effect

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

effect(() => {
  console.log('Count is now', count());
});
// runs again automatically every time count changes
```

> *This very app uses signals — the light/dark theme and the current chapter are stored as signals, and the page reacts when they change.*

5. Recap

- A signal is a value that tracks its own changes.
- Read it by calling it: `count()`.
- Change it with `set` or `update`.
- `computed` builds a value from other signals and refreshes itself.
- `effect` runs code when the signals it uses change.
- This is how modern Angular keeps the screen in sync with your data.

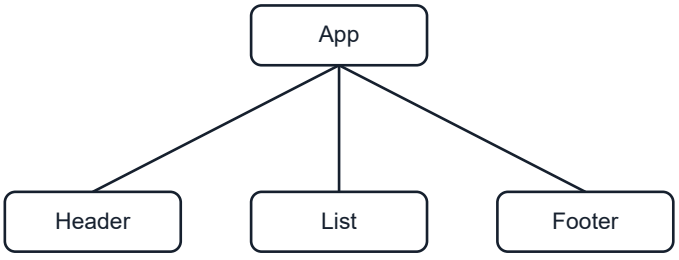
CHAPTER 19

Components: The Basics

The building block of every Angular app — a reusable piece of UI.

1. What is a component?

A component is one reusable piece of the screen — a header, a button, a card, a whole page. An Angular app is really just a tree of components nested inside each other.



An app is a tree of components.

More on angular.dev ↗

2. The parts of a component

A component is a class (Chapter 8) with a `@Component` label on top. The label tells Angular three things: the tag name to use it as, where its template is, and where its styles are.

hello.ts

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

@Component({
  selector: 'app-hello',     // the tag name
  templateUrl: './hello.html', // the view
  styleUrl: './hello.css'      // the styles
})
export class Hello {
  name = 'World';
}
```

selector
The custom tag you write to use the component

template
The HTML the component shows

styles
CSS that applies only to this component

class
The data and logic behind it

3. Using a component

Once a component has a selector, you use it like an HTML tag inside another component's template. First make it available by adding it to `imports`.

app.ts

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

@Component({
  selector: 'app-root',
  imports: [Hello],
  template: `<app-hello />`
})
export class App {}
```

> *Modern Angular components are **standalone** — each one lists its own imports, so there are no big NgModules to manage.*

Here's a component you can edit and run — the class holds the data, the template shows it:

component.ts

```
title = 'My First Component';
tagline = 'Reusable piece of UI';
```

component.html

```
<h2>{{ title }}</h2>
<p>{{ tagline }}</p>
```

Output

My First Component

Reusable piece of UI

4. Creating one with the CLI

You don't write all those files by hand. The CLI generates a component — class, template, and styles — and wires the naming for you.

Terminal

```
ng generate component hello
# short version:
ng g c hello
```

More on angular.dev ↗

5. Recap

- A component is a reusable piece of UI; an app is a tree of them.
- It's a class with a `@Component` label.
- The label sets the `selector`, template, and styles.
- Use a component by its selector tag, after adding it to `imports`.
- Generate one with `ng g c name`.

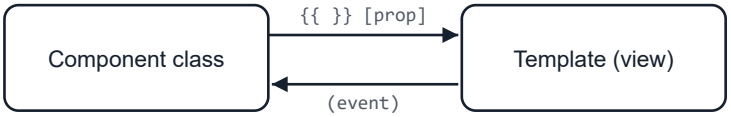
CHAPTER 20

Templates and Data Binding

Connect your component's data to what shows on screen.

1. Data flows both ways

Binding is the link between your component class and its template. Data can flow from the class into the view, and events can flow from the view back to the class.



Class to view with bindings; view to class with events.

More on angular.dev ↗

2. Interpolation: showing values

Use `{{ }}` to drop a value from the class straight into the text — just like the template literals from Chapter 6.

template

```
// class:  name = 'Asha';

<h1>Hello {{ name }}</h1>
```

Edit and run — the heading shows whatever the class holds, and updates when it changes:

component.ts

```
name = 'World';
change() {
  name = name === 'World' ? 'Angular' : 'World';
}
```

component.html

```
<h1>Hello, {{ name }}!</h1>
<button (click)="change()">Change name</button>
```

Output

Hello, World!

Change name

3. Property binding: setting properties

Square brackets `[ ]` set an element's property from a value in your class. Here the button's `disabled` property follows the `isBusy` value.

template

```
// class:  isBusy = true;

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

Edit and run — toggling the value flips the Save button between enabled and disabled:

component.ts

```
busy = false;
toggle() {
  busy = !busy;
}
```

component.html

```
<button (click)="toggle()">Toggle</button>
<button [disabled]="busy">Save</button>
<p>busy = {{ busy }}</p>
```

Output

Toggle Save

busy = false

4. Event binding: reacting to actions

Round brackets `( )` run a method in your class when an event happens, like a click.

template

```
// class:  save() { ... }

<button (click)="save()">Save</button>
```

A working counter you can edit and run — each click calls a method that updates the value, and the template re-shows it:

component.ts

```
count = 0;
add() {
  count = count + 1;
}
```

component.html

```
<button (click)="add()">Clicked {{ count }} times</button>
```

Output

Clicked 0 times

5. Two-way binding: forms

Sometimes you want both directions at once — show a value and update it as the user types. The `[( )]` syntax ("banana in a box") does both.

template

```
// class:  name = signal('');

<input [(ngModel)]="name" />
<p>You typed: {{ name() }}</p>
```

> *`ngModel` comes from `FormsModule`, so add it to the component's `imports`.*

Here's a working Angular example you can edit and run. The preview uses `[value]` + `(input)` — exactly what `[(ngModel)]` does under the hood — so you can watch the binding update live as you type:

component.ts

```
text = '';
```

component.html

```
<input id="text" [value]="text" (input)="text = $event.target.value">
<p>You typed: {{ text }}</p>
```

Output

You typed:

6. Template reference variables

Put `#name` on an element to grab a reference to it, then use that name elsewhere in the same template.

template

```
<input #box>
<button (click)="show(box.value)">Show</button>
```

7. Recap

- `{{ value }}` shows a value as text.
- `[prop]="value"` sets a property from the class.
- `(event)="method()"` runs code on an action.
- `[(ngModel)]="value"` binds both ways for inputs.
- `#ref` grabs an element to use elsewhere in the template.

CHAPTER 21

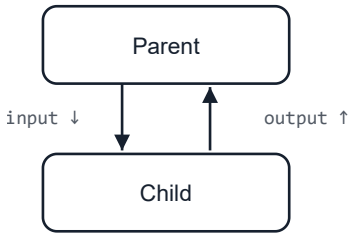
Inputs and Outputs

How a parent component passes data down and a child sends events back up.

---

1. Talking between components

Apps are trees of components, so neighbours need to share. The rule is one-directional and simple: a parent passes data **down** to a child through an **input**, and a child sends events **up** to its parent through an **output**. Data down, events up.



Data flows down through inputs; events flow up through outputs.

> *Keeping data flowing one way (down) and events flowing the other (up) is what makes big component trees predictable — you always know who owns the data.*

More on angular.dev ↗

2. Example 1: passing data down with input()

An `input()` is how a child receives a value. It returns a **signal**, so you read it by calling it with `()` — and the view updates automatically if the parent changes it. You can give it a default value.

greeting.ts (child)

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

@Component({
  selector: 'app-greeting',
  template: `<p>Hello, {{ name() }}!</p>`
})
export class Greeting {
  name = input('friend'); // default if the parent passes nothing
}
```

The parent sets it with property binding — the same `[ ]` from Chapter 20. A plain string needs no brackets; use brackets to bind to a value.

app.html (parent)

```
<app-greeting name="Asha" />        <!— static text —>
<app-greeting [name]="userName" />  <!— bound to a property —>
```

The parent's value lands in the child's input signal.

3. Input options: required, alias, transform

Inputs can do more than hold a default. These options come up constantly in real components.

input options

```
import { input, numberAttribute } from '@angular/core';

// Must be provided — Angular errors if the parent forgets it
userId = input.required<number>();

// Exposed to the parent under a different name: [label]="..."
title = input('', { alias: 'label' });

// Convert the incoming value (e.g. attribute string → number)
age = input(0, { transform: numberAttribute });
```

> *`input.required` is great for values a component can't work without — you get a clear error instead of a silent bug.*

4. Example 2: sending events up with output()

An `output()` lets a child notify its parent. The child calls `.emit(value)`; the value it sends becomes `$event` in the parent. Type the output so the payload is checked.

like-button.ts (child)

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

@Component({
  selector: 'app-like-button',
  template: `<button (click)="liked.emit(count)">👍 Like</button>`
})
export class LikeButton {
  count = 1;
  liked = output<number>(); // emits a number
}
```

app.ts (parent)

```
// class:
//   total = 0;
//   onLike(amount: number) { this.total += amount; }

<app-like-button (liked)="onLike($event)" />
<p>Likes: {{ total }}</p>
```

The child emits a value; it arrives as $event in the parent.

5. Example 3: input and output together

Most real components use both. Here a **stepper** receives its current count as an input and emits the new count whenever a button is pressed — the parent stays the owner of the value.

stepper.ts (child)

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

@Component({
  selector: 'app-stepper',
  template: `
    <button (click)="change(-1)">-</button>
    <span>{{ count() }}</span>
    <button (click)="change(1)">+</button>`
})
export class Stepper {
  count = input(0);
  countChange = output<number>();

  change(step: number) {
    this.countChange.emit(this.count() + step);
  }
}
```

app.ts (parent)

```
// class:  total = 0;

<app-stepper
  [count]="total"
  (countChange)="total = $event" />

<p>Total: {{ total }}</p>
```

> *Because the input is `count` and the output is `countChange`, the parent can shorten those two lines to a two-way binding: `[(count)]="total"` — that's the next chapter.*

6. Example 4: passing an object as input

Inputs aren't limited to strings — they carry any type: numbers, booleans, arrays, or whole objects. A card component can take a full `user` object.

user-card.ts (child)

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

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

@Component({
  selector: 'app-user-card',
  template: `
    <div class="card">
      <h3>{{ user().name }}</h3>
      <p>Age: {{ user().age }}</p>
    </div>`
})
export class UserCard {
  user = input.required<User>();
}
```

app.html (parent)

```
// class:  currentUser = { name: 'Asha', age: 20 };

<app-user-card [user]="currentUser" />
```

> *Brackets are required here — `[user]` binds to a real object. Without brackets the child would receive the literal string "currentUser".*

7. Example 5: a list item that emits its id

A very common pattern: render a list of child components, and let each one tell the parent when it's clicked or removed. The child emits its own id; the parent owns the list and removes it.

todo-item.ts (child)

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

@Component({
  selector: 'app-todo-item',
  template: `
    <li>
      {{ text() }}
      <button (click)="remove.emit(id())">×</button>
    </li>`
})
export class TodoItem {
  id = input.required<number>();
  text = input.required<string>();
  remove = output<number>();
}
```

app.ts (parent)

```
// class:
//   todos = [{ id: 1, text: 'Learn' }, { id: 2, text: 'Build' }];
//   removeTodo(id: number) {
//     this.todos = this.todos.filter(t => t.id !== id);
//   }

@for (todo of todos; track todo.id) {
  <app-todo-item
    [id]="todo.id"
    [text]="todo.text"
    (remove)="removeTodo($event)" />
}
```

> *The parent keeps ownership of the list; the child just announces "remove me" by id. That's data down, events up in action.*

8. Recap

- **Data down, events up**: `input()` receives, `output()` sends.
- An input is a signal — read it with `name()`; give it a default.
- Options: `input.required`, `alias`, and `transform`.
- Set an input from the parent with `[name]="value"`.
- A child fires an output with `thing.emit(payload)`; the parent reads `$event`.
- An input x plus output `xChange` can be used as two-way `[(x)]`.

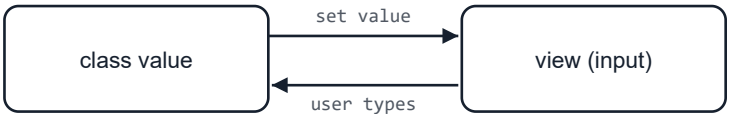
CHAPTER 22

Two-way Binding

Keep a value and the screen in sync, in both directions at once.

1. What is two-way binding?

Property binding sends data *into* the view; event binding sends events *out*. **Two-way binding** does both at once: the class updates the view, and the view updates the class — they stay in sync automatically.



The value and the view update each other.

The syntax is `[( )]` — square brackets (property in) plus parentheses (event out). People call it the **"banana in a box"**.

More on angular.dev ↗

2. It's just two bindings

Two-way binding is sugar. `[(value)]` expands to a property binding plus an event binding named `valueChange` — these two lines do exactly the same thing:

Two-way

```
<app-x [(value)]="count" />
```

The long form

```
<app-x
  [value]="count"
  (valueChange)="count = $event" />
```

> *Rule: `[(thing)]` works when something has both a `thing` input and a `thingChange` output.*

3. Example 1: ngModel on a form input

The most common two-way binding is `[(ngModel)]` on an `<input>` — type in the box and the class value updates live. Needs `FormsModule`.

real Angular

```
// class:  name = signal('');

<input [(ngModel)]="name">
<p>Hello, {{ name() }}!</p>
```

> *Run the live version in the next example — it uses `[value]` + `(input)`, which is exactly what `[(ngModel)]` does under the hood.*

4. Example 2: the manual equivalent

To see the two halves clearly, here's the same input written as a property binding plus an event binding — no `ngModel` at all.

component.ts

```
text = 'edit me';
```

component.html

```
<input [value]="text" (input)="text = $event.target.value">
<p>You typed: {{ text }}</p>
```

Output

edit me

You typed: edit me

5. Example 3: two-way on your own component

You can give your own component a two-way property with the `model()` function. It creates both the input and the matching `...Change` output for you, so a parent can use `[( )]` on it.

counter.ts (child)

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

@Component({
  selector: 'app-counter',
  template: `
    <button (click)="value.set(value() + 1)">
      Count: {{ value() }}
    </button>`
})
export class Counter {
  value = model(0); // a two-way bindable signal
}
```

app.ts (parent)

```
// class:  total = signal(0);

<app-counter [(value)]="total" />
<p>Parent sees: {{ total() }}</p>
```

> *Click the child's button and the parent's `total` updates too — because `model()` binds both ways.*

6. Example 4: a toggle switch with model()

A reusable on/off switch. The child owns the button; the parent reads and sets the value through two-way binding.

toggle.ts (child)

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

@Component({
  selector: 'app-toggle',
  template: `
    <button (click)="on.set(!on())">
      {{ on() ? 'ON' : 'OFF' }}
    </button>`
})
export class Toggle {
  on = model(false);
}
```

app.ts (parent)

```
// class:  enabled = signal(false);

<app-toggle [(on)]="enabled" />
<p>Notifications: {{ enabled() ? 'enabled' : 'disabled' }}</p>
```

7. Example 5: a star rating with model()

`model()` shines for custom input controls. This star rating both shows the current value and reports changes — all through one two-way binding.

rating.ts (child)

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

@Component({
  selector: 'app-rating',
  template: `
    @for (star of [1, 2, 3, 4, 5]; track star) {
      <button (click)="value.set(star)">
        {{ star ≤ value() ? '★' : '☆' }}
      </button>
    }`
})
export class Rating {
  value = model(0);
}
```

app.ts (parent)

```
// class:  score = signal(3);

<app-rating [(value)]="score" />
<p>You rated: {{ score() }} / 5</p>
```

> *`model.required<T>()` works too, when the parent must provide a starting value.*

8. Recap

- Two-way binding keeps a value and the view in sync in both directions.
- Syntax is `[(x)]` — "banana in a box".
- It's shorthand for `[x]` + `(xChange)`.
- `[(ngModel)]` is the everyday case on form inputs (needs `FormsModule`).
- Add two-way to your own component with `model()`.

CHAPTER 23

Control Flow in Templates

Show, hide, repeat, and switch parts of your template.

---

1. Showing things with @if

`@if` shows a block only when a condition is true, with an optional `@else`. It's the same idea as the if/else from Chapter 2, but inside the template.

template

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

Here it is as a live Angular example — press the button to flip the value and watch the block switch. Edit it and run:

component.ts

```
show = true;
toggle() {
  show = !show;
}
```

component.html

```
<button (click)="toggle()">Toggle</button>

@if (show) {
  <p>Now you see me</p>
} @else {
  <p>Now you don't</p>
}
```

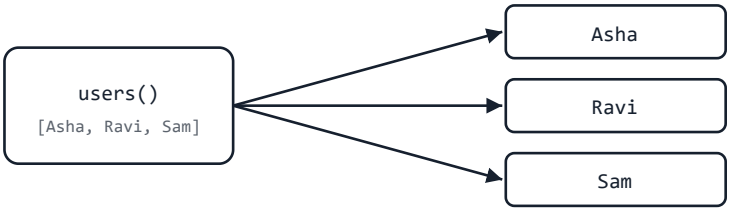
Output

Toggle

Now you see me

2. Repeating with @for

`@for` renders a block once for each item in a list. The `track` part helps Angular update the list efficiently — usually track a unique id.



One list item in, one rendered block out — for every item.

template

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

> *`@empty` is optional — it shows when the list has no items.*

A live Angular example — add items and watch `@for` render each one. Edit it and run:

component.ts

```
items = ['Asha', 'Ravi'];
add() {
  items = [...items, 'Item ' + (items.length + 1)];
}
clear() {
  items = [];
}
```

component.html

```
<button (click)="add()">Add</button>
<button (click)="clear()">Clear</button>

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

Output

3. Choosing with @switch

`@switch` picks one block out of several based on a value — handy when there are more than two options.

template

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

A live example — click a button to change the status and watch `@switch` pick the matching block. Edit it and run:

component.ts

```
status = 'loading';
set(value) {
  status = value;
}
```

component.html

```
<button (click)="set('loading')">Loading</button>
<button (click)="set('error')">Error</button>
<button (click)="set('done')">Done</button>

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

Output

Loading Error Done

Loading...

4. The modern way

You may see older code using `*ngIf` and `*ngFor` as attributes. The newer `@if` and `@for` blocks are built into the template, easier to read, and the recommended choice today.

More on angular.dev ↗

5. Recap

- `@if` / `@else` shows or hides a block.
- `@for (item of list; track item.id)` repeats a block.
- `@empty` covers the empty-list case.
- `@switch` / `@case` / `@default` picks one of many.
- These replace the older `*ngIf` / `*ngFor`.

CHAPTER 24

Component Lifecycle

The stages a component goes through, from created to removed.

1. What is the lifecycle?

Every component is born, lives on the screen for a while, and is eventually removed. Angular lets you run your own code at these moments by adding special methods called **lifecycle hooks**.

Angular calls hooks at each stage of a component's life.

More on angular.dev ↗

2. The hooks you'll use most

constructor
Runs first; use it for injecting services, not for heavy work

ngOnInit
Runs once after the component is set up; good for initial loading

ngOnChanges
Runs whenever an input value changes

ngOnDestroy
Runs just before the component is removed; clean up here

3. Using a hook

To use a hook, implement its interface and add the method. Here the component loads data when it starts and cleans up when it leaves.

timer.ts

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

@Component({ selector: 'app-timer', template: `...` })
export class Timer implements OnInit, OnDestroy {
  ngOnInit() {
    console.log('started');
  }
  ngOnDestroy() {
    console.log('cleaning up');
  }
}
```

> *With signals and `computed`, you often need fewer hooks — Angular reacts to changes for you. But `ngOnInit` and `ngOnDestroy` are still very common.*

> *Lifecycle hooks need a real component being created and destroyed, so try this one in your own Angular app — add a `console.log` in each hook and watch the order in the browser console.*

ADVANCED **The other lifecycle hooks**

Angular has a few more hooks for advanced cases. You'll rarely reach for these as a beginner, but here they are for completeness.

ngOnChanges
Runs whenever an input value changes (before ngOnInit the first time)

ngDoCheck
Runs on every change-detection pass; for your own custom change checks

ngAfterContentInit
Once, after projected content (ng-content) is ready

ngAfterContentChecked
After Angular checks the projected content

ngAfterViewInit
Once, after the component's view and child views are ready

ngAfterViewChecked
After Angular checks the component's view

> *The "Checked" hooks run very often, so keep them light. Most apps only ever need `ngOnInit` and `ngOnDestroy`.*

Full lifecycle reference on angular.dev ↗

5. Recap

- Lifecycle hooks run your code at key moments.
- `constructor` first, then `ngOnInit` once set up.
- `ngOnChanges` runs when inputs change.
- `ngOnDestroy` runs before removal — clean up there.
- Implement the matching interface to use a hook.

CHAPTER 25

Styles and Content Projection

Scoped styling, passing markup into a component, and reaching its elements.

1. Component styles are scoped

A component's styles only affect that component's own template. So a `.title` rule in one component won't accidentally restyle a `.title` somewhere else. Angular keeps each component's CSS to itself — this is called view encapsulation.

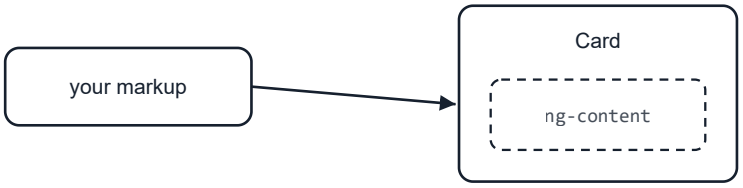
card.css

```
.title {
  font-size: 1.2rem;
}
/* applies only inside Card's template */
```

More on angular.dev ↗

2. Content projection with ng-content

Sometimes a component is a wrapper — like a card — and you want to drop your own content inside it. `<ng-content>` marks the slot where that content lands.



Markup you write between the tags drops into the ng-content slot.

card.ts (the wrapper)

```
@Component({
  selector: 'app-card',
  template: `
    <div class="card">
      <ng-content />
    </div>`
})
export class Card {}
```

using it

```
<app-card>
  <h2>Hello</h2>
  <p>This goes into the slot.</p>
</app-card>
```

More on angular.dev ↗

3. Reaching elements with viewChild

Now and then you need a real element from your template — to focus an input, for example. Mark it with a reference (Chapter 20) and grab it with `viewChild`.

search.ts

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

@Component({
  selector: 'app-search',
  template: `
    <input #box>
    <button (click)="focus()">Focus</button>`
})
export class Search {
  box = viewChild<ElementRef>('box');
  focus() {
    this.box()?.nativeElement.focus();
  }
}
```

> *Reach for this only when you truly need the element. Most of the time, bindings and signals already do what you want.*

4. Recap

- Component styles are scoped to that component only.
- `<ng-content>` is a slot for markup passed in from outside.
- Whatever you put between the component's tags lands in that slot.
- `viewChild` grabs an element from the template when you need it.
- Prefer bindings and signals first; use queries sparingly.

CHAPTER 26

Pipes

Transform how values look in the template without changing the data.

---

1. What is a pipe?

A pipe formats a value for display using the `|` symbol in a template. It only changes what's shown on screen — the underlying data stays exactly the same.

template

```
<p>{{ price | currency }}</p>
<!— data: price = 1500  →  shows $1,500.00 —>
```

value → | pipe → formatted output

Some pipes take **arguments** after a colon: `{{ value | pipeName:arg1:arg2 }}`.

More on angular.dev ↗

2. Text pipes

These change the casing of text.

uppercase / lowercase / titlecase

```
{{ 'hello' | uppercase }}        <!— HELLO —>
{{ 'HELLO' | lowercase }}        <!— hello —>
{{ 'hello world' | titlecase }}  <!— Hello World —>
```

Try it — type below and watch the pipes format your text live:

component.ts

```
text = 'hello world';
```

component.html

```
<input id="t" [value]="text" (input)="text = $event.target.value">
<p>upper: {{ text | uppercase }}</p>
<p>title: {{ text | titlecase }}</p>
```

Output

hello world

upper: HELLO WORLD

title: Hello World

3. Number pipes

Format numbers as plain decimals, currency, or percentages. The `number` pipe's argument is `'minIntegerDigits.minFractionDigits-maxFractionDigits'`.

number / currency / percent

```
{{ 3.14159 | number:'1.0-2' }}   <!— 3.14 —>
{{ 1500 | currency:'USD' }}      <!— $1,500.00 —>
{{ 0.25 | percent }}             <!— 25% —>
```

Change the amount and see the currency and number pipes reformat it:

component.ts

```
price = 1500;
```

component.html

```
<input id="p" type="number" [value]="price" (input)="price = +$event.target.value">
<p>{{ price | currency:'USD' }}</p>
<p>{{ price | number:'1.0-2' }}</p>
```

Output

1500

$1,500.00

1,500

4. The date pipe

`date` formats a date value. Pass a format string to control how it looks.

date

```
{{ today | date }}              <!— Jun 12, 2026 —>
{{ today | date:'shortTime' }}  <!— 9:30 AM —>
{{ today | date:'dd/MM/yyyy' }} <!— 12/06/2026 —>
{{ today | date:'fullDate' }}   <!— Friday, June 12, 2026 —>
```

Edit and run — these show today's date in different formats:

component.ts

```
today = new Date();
```

component.html

```
<p>default: {{ today | date }}</p>
<p>time: {{ today | date:'shortTime' }}</p>
<p>full: {{ today | date:'fullDate' }}</p>
```

Output

default: 17/06/2026

time: 11:25

full: Wednesday, 17 June 2026

5. Data pipes: json, slice, keyvalue

`json` is handy for debugging — it prints any value as JSON. `slice` takes part of a list or string. `keyvalue` turns an object into a list you can loop over.

json (debug)

```
<pre>{{ user | json }}</pre>
<!— { "name": "Asha", "age": 20 } —>
```

slice (first 3)

```
@for (item of items | slice:0:3; track item) {
  <li>{{ item }}</li>
}
```

keyvalue

```
@for (entry of user | keyvalue; track entry.key) {
  <li>{{ entry.key }}: {{ entry.value }}</li>
}
```

Edit and run — see `json` and `slice` in action:

component.ts

```
user = { name: 'Asha', age: 20 };
items = ['apple', 'mango', 'kiwi', 'pear', 'plum'];
```

component.html

```
<pre>{{ user | json }}</pre>

<p>first 3: {{ items | slice:0:3 }}</p>
<ul>
  @for (item of items | slice:0:3; track item) {
    <li>{{ item }}</li>
  }
</ul>
```

Output

```
{
  "name": "Asha",
  "age": 20
}
```

first 3: apple,mango,kiwi

- apple
- mango
- kiwi

6. The async pipe

`async` unwraps a value that arrives later — a Promise or an Observable — and keeps the screen updated when it changes. It also cleans up the subscription for you.

async

```
// class:  user$ = this.http.get('/api/user');

@if (user$ | async; as user) {
  <p>{{ user.name }}</p>
}
```

> *The `async` pipe pairs perfectly with HttpClient and signals — no manual subscribe/unsubscribe needed.*

7. Chaining pipes

You can pipe a value through several pipes in a row — they run left to right.

chained

```
{{ today | date:'fullDate' | uppercase }}
<!— FRIDAY, JUNE 12, 2026 —>
```

8. Writing your own pipe

When no built-in pipe fits, write your own. Let the CLI scaffold it for you:

Terminal

```
ng generate pipe exclaim
# short version:
ng g p exclaim
```

A pipe is a class with a `transform` method, marked with `@Pipe`.

exclaim.pipe.ts

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({ name: 'exclaim' })
export class ExclaimPipe implements PipeTransform {
  transform(value: string): string {
    return value + '!';
  }
}

// use it:  {{ 'hi' | exclaim }}  →  hi!
```

Pipes are **pure** by default: they only re-run when their input changes, which keeps them fast. An *impure* pipe (like `async`) re-runs more often — use them sparingly.

Custom pipes on angular.dev ↗

9. Recap

- A pipe formats a value for display with `|` — the data is unchanged.
- Text: `uppercase`, `lowercase`, `titlecase`.
- Numbers: `number`, `currency`, `percent`.
- Also `date`, `json`, `slice`, `keyvalue`, and `async`.
- Arguments come after a colon; chain pipes left to right.
- Write your own with `@Pipe` and a `transform` method.

CHAPTER 27

Directives

Add behavior or change appearance and layout of elements.

---

1. What is a directive?

A directive is an instruction you attach to an element to give it extra behavior. Angular has three kinds.

Components
A directive with its own template — the ones you've been building

Attribute directives
Change the look or behavior of an element (e.g. ngClass, ngStyle)

Structural directives
Add or remove elements, changing the layout (e.g. @if, @for)

More on angular.dev ↗

2. ngClass — toggle CSS classes

`ngClass` adds or removes classes based on conditions. Give it an object of `class: condition` pairs.

ngClass

```
// class:  isActive = true;  hasError = false;

<div [ngClass]="{ active: isActive, error: hasError }">
  Status
</div>
<!— gets class="active" —>
```

Click to toggle the class and watch the style change:

component.ts

```
active = false;
toggle() {
  active = !active;
}
```

component.html

```
<style>
  .pill { padding: 6px 12px; border-radius: 999px; background: #e5e7eb; }
  .active { background: tomato; color: white; }
</style>

<button (click)="toggle()">Toggle</button>
<span class="pill" [ngClass]="{ active: active }">Status</span>
```

Output

Toggle Status

3. ngStyle — set inline styles

`ngStyle` sets one or more inline styles from your data.

ngStyle

```
// class:  color = 'tomato';  size = 18;

<p [ngStyle]="{ color: color, 'font-size.px': size }">
  Styled text
</p>
```

Toggle to change the inline styles live:

component.ts

```
big = false;
toggle() {
  big = !big;
}
```

component.html

```
<button (click)="toggle()">Toggle size</button>
<p [ngStyle]="{ color: big ? 'tomato' : 'gray', 'font-size.px': big ? 28 : 16 }">
  Resize me
</p>
```

Output

Toggle size

Resize me

> *For a single class or style, the plain bindings `[class.active]="isActive"` and `[style.color]="color"` are often simpler.*

4. ngModel — two-way binding

`ngModel` is an attribute directive that binds a form control both ways (Chapter 19). It comes from `FormsModule`.

ngModel

```
<input [(ngModel)]="name" />
<p>Hello {{ name }}</p>
```

5. Structural directives

Structural directives add or remove elements. Modern Angular uses the built-in `@if`, `@for`, and `@switch` blocks (Chapter 23) for this.

modern control flow

```
@if (isLoggedIn) {
  <p>Welcome!</p>
}

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

Older code uses the `*`-prefixed forms, which you'll still see around:

older form

```
<p *ngIf="isLoggedIn">Welcome!</p>
<li *ngFor="let item of items">{{ item }}</li>
```

> *The * is shorthand that wraps the element in a hidden template. Prefer the new @ blocks in fresh code.*

6. Writing your own directive

Scaffold a directive with the CLI:

Terminal

```
ng generate directive highlight
# short version:
ng g d highlight
```

A custom attribute directive is a class marked with `@Directive`. It can react to the host element — here it sets a yellow background.

highlight.directive.ts

```
import { Directive, ElementRef } from '@angular/core';

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  constructor(el: ElementRef) {
    el.nativeElement.style.background = 'yellow';
  }
}

// use it:  <p appHighlight>Highlighted</p>
```

Use `host` listeners and bindings to react to events like hover or clicks.

Attribute directives on angular.dev ↗

7. Recap

- Directives add behavior to elements; components are directives with a template.
- Attribute directives change look/behavior: `ngClass`, `ngStyle`, `ngModel`.
- Structural directives change layout: use `@if` / `@for` / `@switch`.
- Older code uses `*ngIf` / `*ngFor` — same idea, older syntax.
- Write your own with `@Directive` and a CSS-attribute selector.

CHAPTER 28

Template-driven Forms

Build forms right in the template with ngModel — great for simple forms.

1. Two ways to build forms

Angular gives you two styles of forms. They do the same job — collect and validate user input — but put the logic in different places.

Template-driven
Form logic lives in the template with ngModel. Simple and quick — this chapter.

Reactive
Form model lives in the TypeScript class. More powerful for big or dynamic forms — next chapter.

> *Reach for template-driven forms for small things like a login or contact form. Reach for reactive forms when the form is large, dynamic, or needs serious validation and testing.*

More on angular.dev ↗

2. Setup

Template-driven forms come from `FormsModule`. Add it to the component's `imports`.

component

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

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

3. Binding inputs with ngModel

Each input uses `[(ngModel)]` for two-way binding and a `name` attribute so the form can track it. Wrap the inputs in a `<form>` and handle `(ngSubmit)`.

signup.html

```
<form (ngSubmit)="submit()">
  <input name="name" [(ngModel)]="model.name" placeholder="Name">
  <input name="email" [(ngModel)]="model.email" placeholder="Email">
  <button type="submit">Sign up</button>
</form>
```

component

```
submit() {
  console.log(this.model); // { name: '...', email: '...' }
}
```

4. Validation

Add normal HTML validators like `required` and `minlength`. Give the input a template reference (`#name="ngModel"`) to read its state and show messages.

signup.html

```
<input
  name="name"
  [(ngModel)]="model.name"
  required
  minlength="3"
  #name="ngModel">

@if (name.invalid && name.touched) {
  <small>Name must be at least 3 characters.</small>
}
```

> *Useful states on a control: `valid` / `invalid`, `touched` (the user visited it), and `dirty` (the value changed).*

5. Using the whole form's state

Give the form a reference with `#form="ngForm"` to check if the entire form is valid — handy for disabling the submit button.

signup.html

```
<form #form="ngForm" (ngSubmit)="submit()">
  <input name="name" [(ngModel)]="model.name" required>
  <button type="submit" [disabled]="form.invalid">
    Sign up
  </button>
</form>
```

ADVANCED Nested groups

Group related fields with `ngModelGroup`. The grouped controls become a nested object inside the form value.

template

```
<form #f="ngForm">
  <input name="name" ngModel>

  <div ngModelGroup="address">
    <input name="street" ngModel>
    <input name="city" ngModel>
  </div>
</form>
```

f.value

```
{
  name: '...',
  address: { street: '...', city: '...' }
}
```

ADVANCED Splitting a form across components

To move part of a form into a child component, let the child reuse the parent's `NgForm` through `viewProviders`. Then the child's `ngModel` controls register with the parent form automatically.

address.ts (child)

```
import { Component } from '@angular/core';
import { ControlContainer, NgForm, FormsModule } from '@angular/forms';

@Component({
  selector: 'app-address',
  imports: [FormsModule],
  viewProviders: [
    { provide: ControlContainer, useExisting: NgForm }
  ],
  template: `
    <input name="street" [(ngModel)]="model.street">
    <input name="city" [(ngModel)]="model.city">`
})
export class Address {
  model = { street: '', city: '' };
}
```

> *Without that `viewProviders` line, the child's inputs wouldn't join the parent form.*

ADVANCED A list of fields (form array)

Template-driven forms have no `FormArray`. For a dynamic list, loop over an array in your model and give each input a unique `name`.

component

```
phones = [''];

addPhone() { this.phones.push(''); }
removePhone(i: number) { this.phones.splice(i, 1); }
```

template

```
@for (phone of phones; track i; let i = $index) {
  <input [name]="'phone' + i" [(ngModel)]="phones[i]">
  <button type="button" (click)="removePhone(i)">×</button>
}
<button type="button" (click)="addPhone()">Add phone</button>
```

> *This works but is manual. For dynamic lists, reactive forms' `FormArray` is cleaner — see the Reactive Forms chapter.*

9. Recap

- Template-driven forms keep the logic in the template; import `FormsModule`.
- Bind inputs with `[(ngModel)]` and give each a `name`.
- Handle submit with `(ngSubmit)`.
- Validate with `required`, `minlength`, etc.; read state via `#ref="ngModel"`.
- Use `#form="ngForm"` for the whole form's validity.
- Best for small, simple forms.

CHAPTER 29

Reactive Forms

Define the form model in TypeScript — powerful for bigger, dynamic forms.

---

1. Form model in the class

Reactive forms flip things around: instead of the template owning the form, you build the form model in TypeScript. That makes the form easy to read, test, and change at runtime — ideal for larger or dynamic forms.

imports

```
import { ReactiveFormsModule } from '@angular/forms';

@Component({
  imports: [ReactiveFormsModule],
  /* ... */
})
```

More on angular.dev ↗

2. FormGroup and FormControl

A `FormControl` is a single field. A `FormGroup` bundles controls into one form. You create them in the class, then connect them in the template.

signup.ts

```
import { Component } from '@angular/core';
import { FormGroup, FormControl } from '@angular/forms';

export class Signup {
  form = new FormGroup({
    name: new FormControl(''),
    email: new FormControl('')
  });

  submit() {
    console.log(this.form.value); // { name: '...', email: '...' }
  }
}
```

signup.html

```
<form [formGroup]="form" (ngSubmit)="submit()">
  <input formControlName="name" placeholder="Name">
  <input formControlName="email" placeholder="Email">
  <button type="submit">Sign up</button>
</form>
```

3. Validation with Validators

Pass validators when you create each control. They live in the class, not the template.

signup.ts

```
import { FormGroup, FormControl, Validators } from '@angular/forms';

form = new FormGroup({
  name: new FormControl('', [Validators.required, Validators.minLength(3)]),
  email: new FormControl('', [Validators.required, Validators.email])
});
```

Read a control's state to show messages and control the submit button:

signup.html

```
<input formControlName="name">
@if (form.controls.name.invalid && form.controls.name.touched) {
  <small>Name needs at least 3 characters.</small>
}

<button [disabled]="form.invalid">Sign up</button>
```

4. FormBuilder shorthand

Writing `new FormControl` everywhere gets repetitive. `FormBuilder` (injected with `inject`) builds the same form with less code.

signup.ts

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

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

  form = this.fb.group({
    name: ['', [Validators.required, Validators.minLength(3)]],
    email: ['', [Validators.required, Validators.email]]
  });
}
```

5. Reacting to changes

Because the model lives in code, you can watch any value change as the user types — useful for live search, dependent fields, or autosave.

signup.ts

```
this.form.controls.name.valueChanges.subscribe(value => {
  console.log('name is now', value);
});
```

ADVANCED

Nested form groups

A `FormGroup` can hold another `FormGroup` — perfect for grouping related fields like an address.

signup.ts

```
form = this.fb.group({
  name: [''],
  address: this.fb.group({
    street: [''],
    city: ['']
  })
});
```

signup.html

```
<form [formGroup]="form">
  <input formControlName="name">

  <div formGroupName="address">
    <input formControlName="street">
    <input formControlName="city">
  </div>
</form>
```

> *The value nests too: `form.value.address.city`.*

ADVANCED Splitting a form across components

Pass a nested `FormGroup` down to a child component as an input. The child binds it with `[formGroup]` — the parent still owns the whole form.

parent.html

```
<!— form.controls.address is a FormGroup —>
<app-address [group]="form.controls.address" />
```

address.ts (child)

```
import { Component, input } from '@angular/core';
import { FormGroup, ReactiveFormsModule } from '@angular/forms';

@Component({
  selector: 'app-address',
  imports: [ReactiveFormsModule],
  template: `
    <div [formGroup]="group()">
      <input formControlName="street">
      <input formControlName="city">
    </div>`
})
export class Address {
  group = input.required<FormGroup>();
}
```

ADVANCED Dynamic lists with FormArray

A `FormArray` holds a changing number of controls — like a list of phone numbers the user can add to or remove.

phones.ts

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

private fb = inject(FormBuilder);

form = this.fb.group({
  phones: this.fb.array([ this.fb.control('') ])
});

get phones() {
  return this.form.get('phones') as FormArray;
}
addPhone() { this.phones.push(this.fb.control('')); }
removePhone(i: number) { this.phones.removeAt(i); }
```

phones.html

```
<div formArrayName="phones">
  @for (ctrl of phones.controls; track i; let i = $index) {
    <input [formControlName]="i">
    <button type="button" (click)="removePhone(i)">×</button>
  }
</div>
<button type="button" (click)="addPhone()">Add phone</button>
```

9. Recap

- Reactive forms define the model in the class; import `ReactiveFormsModule`.
- `FormControl` is one field; `FormGroup` bundles them.
- Connect with `[formGroup]` and `formControlName`.
- Add `Validators` when creating controls; read `.invalid`, `.touched`.
- `FormBuilder` (`fb.group`) is a shorter way to build the form.
- `valueChanges` lets you react as values change.
- Best for large, dynamic, or heavily-validated forms.

CHAPTER 30

Custom Form Controls (ControlValueAccessor)

Make your own component work with ngModel and formControlName.

1. Why a custom control?

Native inputs work with forms out of the box — you can write `[(ngModel)]` or `formControlName` on an `<input>`. But a component you build (a star rating, a toggle, a colour picker) doesn't... unless you teach it how. That bridge is the **ControlValueAccessor** (CVA).

Form / ngModel → ControlValueAccessor → Your component

> *If you only need two-way binding, `model()` (Chapter 22) is simpler. Reach for a CVA when you want your control to plug into **forms** — `formControlName`, validation, `touched/disabled` state.*

More on angular.dev ↗

2. The four methods

A CVA implements four methods that connect your component to the form.

writeValue(value)
Form → component: show a value the form gives you

registerOnChange(fn)
Component → form: call fn when your value changes

registerOnTouched(fn)
Component → form: call fn when the user interacts (touched)

setDisabledState(isDisabled)
Form → component: react to being disabled (optional)

3. Example: a star-rating control

Register the component as a value accessor with the `NG_VALUE_ACCESSOR` token, then implement the four methods. After this, the rating works like any built-in input.

rating.ts

```
import { Component, forwardRef } from '@angular/core';
import { ControlValueAccessor, NG_VALUE_ACCESSOR } from '@angular/forms';

@Component({
  selector: 'app-rating',
  template: `
    @for (star of [1, 2, 3, 4, 5]; track star) {
      <button type="button" [disabled]="disabled" (click)="set(star)">
        {{ star ≤ value ? '★' : '☆' }}
      </button>
    }`,
  providers: [{
    provide: NG_VALUE_ACCESSOR,
    useExisting: forwardRef(() => Rating),
    multi: true
  }]
})
export class Rating implements ControlValueAccessor {
  value = 0;
  disabled = false;
  private onChange = (_: number) => {};
  private onTouched = () => {};

  set(star: number) {
    if (this.disabled) return;
    this.value = star;
    this.onChange(star);   // tell the form
    this.onTouched();
  }

  writeValue(value: number) { this.value = value ?? 0; }
  registerOnChange(fn: (v: number) => void) { this.onChange = fn; }
  registerOnTouched(fn: () => void) { this.onTouched = fn; }
  setDisabledState(isDisabled: boolean) { this.disabled = isDisabled; }
}
```

4. Using it like a real input

Now your component drops into either kind of form — no special wiring.

Template-driven

```
<app-rating [(ngModel)]="score" name="score" />
```

Reactive

```
<app-rating formControlName="score" />
```

> *Because it's a real form control, `Validators.required`, `touched`, and a disabled form state all just work.*

5. When to use it

Anywhere you build a custom input that should live inside a form:

- Star ratings, like/heart toggles, thumbs up/down.
- On/off switches and segmented toggles.
- Colour pickers and sliders.
- Tag / chip inputs (value is an array of strings).
- Currency, phone, or masked text inputs.
- Star/emoji pickers, star sliders, star meters — any custom widget that holds a value.

6. Recap

- A **ControlValueAccessor** lets your component act like a native form input.
- Register it with `NG_VALUE_ACCESSOR` (`multi: true`, `forwardRef`).
- `writeValue` takes a value in; `registerOnChange` sends changes out.
- `registerOnTouched` reports interaction; `setDisabledState` handles disabling.
- Then it works with both `[(ngModel)]` and `formControlName`.
- For plain two-way (no forms), prefer `model()` instead.

CHAPTER 31

Routing

Show different pages for different URLs, with path and query parameters.

---

1. What is routing?

A single-page app doesn't reload the whole page when you move around. Instead, the **router** watches the URL and swaps which component is shown — so each "page" has its own address you can bookmark, share, and use Back/Forward with.

URL changes → Router matches a route → Component shown in <router-outlet>

> *This very app uses the router — each chapter is a route like `/chapter/chapter-01`.*

More on angular.dev ↗

2. Setting up routes

You define a list of routes — each maps a `path` to a `component` — and register them with `provideRouter`.

app.routes.ts

```
import { Routes } from '@angular/router';
import { Home } from './home';
import { About } from './about';

export const routes: Routes = [
  { path: '', component: Home },
  { path: 'about', component: About }
];
```

app.config.ts

```
import { provideRouter } from '@angular/router';
import { routes } from './app.routes';

export const appConfig = {
  providers: [provideRouter(routes)]
};
```

Then place a `<router-outlet>` where the matched component should appear:

app.html

```
<nav>...</nav>
<router-outlet />
```

3. Linking between pages

Use `routerLink` instead of `href` so navigation happens without a full reload. `routerLinkActive` adds a class to the active link.

template

```
<a routerLink="/">Home</a>
<a routerLink="/about" routerLinkActive="active">About</a>
```

> *Add `RouterLink` (and `RouterLinkActive`) to the component's `imports`.*

4. Path (route) parameters

A **path parameter** is part of the URL that changes — like an id. Mark it with `:` in the route.

route

```
{ path: 'product/:id', component: ProductPage }
// matches /product/42, /product/7, ...
```

Read it from `ActivatedRoute`. Use the `paramMap` observable so it updates when the id changes:

product-page.ts

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

export class ProductPage {
  private route = inject(ActivatedRoute);
  id = '';

  constructor() {
    this.route.paramMap.subscribe(params => {
      this.id = params.get('id') ?? '';
    });
  }
}
```

> *With `provideRouter(routes, withComponentInputBinding())` you can skip ActivatedRoute and just declare `id = input<string>()` — the router feeds the `:id` straight into the input.*

5. Query strings

Query parameters are the `?key=value` bits after the path — great for things like search terms, filters, or page numbers. They don't need to be declared in the route.

linking with query params

```
<a [routerLink]="['/search']" [queryParams]="{ q: 'angular', page: 2 }">
  Search
</a>
<!— goes to /search?q=angular&page=2 —>
```

reading them

```
private route = inject(ActivatedRoute);

constructor() {
  this.route.queryParamMap.subscribe(params => {
    const term = params.get('q');     // 'angular'
    const page = params.get('page');  // '2'
  });
}
```

6. Navigating from code

Sometimes you navigate after an action (a login, a save). Inject `Router` and call `navigate`.

navigate()

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

private router = inject(Router);

openProduct(id: number) {
  this.router.navigate(['/product', id], {
    queryParams: { ref: 'home' }
  });
}
```

7. Redirects and the 404 route

Send the empty path to a default page, and catch anything unknown with a wildcard `**` route. Order matters — the wildcard goes last.

routes

```
export const routes: Routes = [
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', component: Home },
  { path: 'product/:id', component: ProductPage },
  { path: '**', component: NotFound }
];
```

8. Recap

- Routes map a `path` to a `component`; register with `provideRouter`.
- `<router-outlet>` is where the matched component shows.
- Navigate with `routerLink`, or `Router.navigate()` from code.
- Path params (`:id`) are read from `ActivatedRoute.paramMap`.
- Query params (`?q=...`) are read from `queryParamMap`.

- `''` redirects to a default; `**` catches unknown URLs.

CHAPTER 32

Route Guards

Protect routes with guards — like blocking pages behind a login.

1. What is a route guard?

A **guard** is a small function the router runs before (or after) navigating. It decides whether the user is allowed to go there — for example, blocking a dashboard unless the user is logged in.

User opens a route → Guard runs → Allow ✓ or redirect ✗

More on angular.dev ↗

2. Generating a guard with the CLI

Don't write the file by hand — let the CLI scaffold it. `ng generate guard` creates the guard file (and its test).

Terminal

```
ng generate guard auth
# short version:
ng g guard auth
```

The CLI then asks which kind of guard you want — pick one (or more) with the spacebar:

prompt

```
? Which type of guard would you like to create?
  ◯ CanActivate
  ◯ CanActivateChild
  ◯ CanDeactivate
  ◯ CanMatch
```

Choosing **CanActivate** generates a ready-to-fill functional guard:

auth.guard.ts (generated)

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

export const authGuard: CanActivateFn = (route, state) => {
  return true;
};
```

> *It also creates `auth.guard.spec.ts`. Now you just fill in the logic — that's the next step.*

3. An auth guard with canActivate

A modern guard is just a function of type `CanActivateFn`. Inside it you can `inject` services. Return `true` to allow, or a `UrlTree` (from `router.parseUrl`) to redirect.

auth.guard.ts

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

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

  if (auth.isLoggedIn()) {
    return true;
  }
  // not logged in → send to the login page
  return router.parseUrl('/login');
};
```

4. Attaching the guard to a route

List the guard in the route's `canActivate` array. You can attach several guards.

routes

```
import { authGuard } from './auth.guard';

export const routes: Routes = [
  { path: 'login', component: Login },
  {
    path: 'dashboard',
    component: Dashboard,
    canActivate: [authGuard]
  }
];
```

> *Now visiting `/dashboard` while logged out bounces the user to `/login`.*

5. Other guards

canActivate
Can the user enter this route? (auth, roles)

canActivateChild
Same check, applied to a route's children

canDeactivate
Can the user leave? (warn about unsaved changes)

canDeactivate example

```
export const unsavedGuard: CanDeactivateFn<EditPage> = (page) => {
  return page.hasUnsavedChanges()
    ? confirm('Leave without saving?')
    : true;
};
```

6. Common use cases

Guards show up all over real apps. A few you'll meet often:

Authentication
Require a logged-in user before showing a page; otherwise redirect to /login.

Roles & permissions
Let only admins reach /admin; send others to a 'not allowed' page.

Unsaved changes
canDeactivate: warn before leaving a half-filled form.

Onboarding / setup
Force new users through a setup step before the main app.

Subscription / paywall
Block premium routes unless the plan is active.

Feature flags
Hide a route that isn't enabled yet for this user.

role guard example

```
export const adminGuard: CanActivateFn = () => {
  const auth = inject(AuthService);
  const router = inject(Router);
  return auth.isAdmin() ? true : router.parseUrl('/not-allowed');
};
```

> *A guard can also return a `Promise` or `Observable` of `boolean/UrlTree` — handy when the check needs a server call (e.g. verifying a token).*

7. Recap

- A guard is a function the router runs to allow or block navigation.
- `CanActivateFn` returns `true`, `false`, or a `UrlTree` to redirect.
- `inject()` services inside the guard (auth, router).
- Attach with `canActivate: [authGuard]` on the route.
- `canActivateChild` guards child routes; `canDeactivate` guards leaving.

CHAPTER 33

REST APIs In Depth

Resources, HTTP methods, CRUD, and status codes — the full picture.

---

1. A quick recap

Back in the taste chapter we saw that a REST API is a set of web addresses your app calls to get and send data. Now let's go deeper: how those addresses are organised, which HTTP method to use, and what the server's reply means.

Your app → HTTP request → Server + database → HTTP response

Revisit Chapter 11: A Taste of REST and fetch ↗HTTP methods on MDN ↗

2. Resources and endpoints

REST is built around **resources** — things like users, products, or posts. Each resource has a collection address and an item address.

Collection

```
/users          all users
/products       all products
```

Single item

```
/users/1        the user with id 1
/products/42    the product with id 42
```

> *Use plural nouns for collections (`/users`), and put the id in the path for one item (`/users/1`) — not verbs like `/getUser`.*

3. HTTP methods = CRUD

The four basic data operations — Create, Read, Update, Delete (CRUD) — map directly onto four HTTP methods.

GET
Read — fetch a collection or one item (never changes data)

POST
Create — add a new item to a collection

PUT
Update — replace an existing item with new data

DELETE
Delete — remove an item

CRUD on /users

```
GET    /users        → list users      (Read)
GET    /users/1      → one user        (Read)
POST   /users        → create a user    (Create)
PUT    /users/1      → update user 1    (Update)
DELETE /users/1      → remove user 1    (Delete)
```

4. GET — read data

`GET` asks for data and never changes anything. It has no request body — everything it needs is in the URL. The server replies with `200 OK` and the data.

request → response

```
GET /users/1

HTTP/1.1 200 OK
{
  "id": 1,
  "name": "Asha",
  "age": 20
}
```

> *GET is **safe** (changes nothing) and **idempotent** (asking twice gives the same answer), so results can be cached.*

5. POST — create data

`POST` sends new data in the request body to a *collection*. The server creates the item, assigns it an `id`, and replies with `201 Created` and the new item.

request → response

```
POST /users
Content-Type: application/json
{
  "name": "Meera",
  "age": 22
}

HTTP/1.1 201 Created
{ "id": 3, "name": "Meera", "age": 22 }
```

> *POST is **not idempotent**: send it twice and you create two items. You POST to the collection (`/users`), not to an id.*

6. PUT — update data

`PUT` replaces an existing item at its id with the full object you send. The server applies it and replies with `200 OK` and the updated item.

request → response

```
PUT /users/1
Content-Type: application/json
{
  "name": "Asha",
  "age": 21
}

HTTP/1.1 200 OK
{ "id": 1, "name": "Asha", "age": 21 }
```

> *PUT is **idempotent**: sending the same update again leaves the item in the same state. Send the whole object — fields you leave out can be wiped.*

7. DELETE — remove data

`DELETE` removes the item at its id. It usually needs no body, and the server often replies with `204 No Content` (success, nothing to return).

request → response

```
DELETE /users/1

HTTP/1.1 204 No Content
```

> *DELETE is **idempotent**: once the item is gone, deleting it again still ends with it gone (often a 404 the second time).*

8. What a request carries

A request has a **method**, a **URL**, optional **headers** (extra info like the data format or an auth token), and — for POST and PUT — a **body** with the data.

POST /users

```
POST /users HTTP/1.1
Content-Type: application/json
Authorization: Bearer <token>

{
  "name": "Asha",
  "age": 20
}
```

The server replies with a **status code** and usually a body (often JSON).

Response

```
HTTP/1.1 201 Created
Content-Type: application/json

{
  "id": 3,
  "name": "Asha",
  "age": 20
}
```

9. Status codes

The status code tells you what happened. The first digit is the family.

2xx — Success
200 OK, 201 Created, 204 No Content (e.g. after DELETE)

4xx — Your request was wrong
400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found

5xx — The server failed
500 Internal Server Error, 503 Service Unavailable

> *Rule of thumb: 2xx = it worked, 4xx = fix your request, 5xx = the server has a problem.*

10. Safe and idempotent

Two handy ideas: a **safe** method doesn't change data (GET). An **idempotent** method gives the same result no matter how many times you repeat it.

- `GET` — safe and idempotent.
- `PUT` and `DELETE` — idempotent (repeating gives the same end state).
- `POST` — not idempotent (calling it twice creates two items).

11. Recap

- REST is organised around resources: `/users` and `/users/1`.
- CRUD maps to methods: GET, POST, PUT, DELETE.
- Requests carry a method, URL, headers, and (sometimes) a body.
- Status codes: 2xx success, 4xx your fault, 5xx server's fault.
- GET is safe; PUT/DELETE are idempotent; POST is not.

CHAPTER 34

CRUD with JSON Server

Spin up a fake REST API and do the four CRUD operations against it.

---

1. What is JSON Server?

To practice talking to an API you need a server. **JSON Server** turns a plain JSON file into a full REST API in seconds — no backend code needed. It's perfect for learning and prototypes.

Terminal

```
npm install -g json-server
```

Create a `db.json` file with some starting data. Each top-level key becomes a resource (a collection).

db.json

```
{
  "users": [
    { "id": 1, "name": "Asha", "age": 20 },
    { "id": 2, "name": "Ravi", "age": 25 }
  ]
}
```

Terminal

```
json-server --watch db.json
# API now running at http://localhost:3000
```

> *You instantly get the endpoints `/users` and `/users/:id` — and JSON Server saves every change back into `db.json`.*

JSON Server on GitHub ↗

2. What is CRUD?

CRUD stands for the four things you do with data: **C**reate, **R**ead, **U**pdate, and **D**elete. Every app — a todo list, a shop, a social feed — is mostly CRUD underneath.

Over a REST API (Chapter 33), each CRUD operation is a different HTTP method.

Create
POST — add a new item

Read
GET — fetch items

Update
PUT — change an existing item

Delete
DELETE — remove an item

Revisit Chapter 33: REST APIs In Depth ↗

3. Read — GET

Reading has two shapes: get the whole collection, or get one item by its id. These are plain `GET` requests — you can even open the URLs in your browser.

get all

```
GET http://localhost:3000/users

[
  { "id": 1, "name": "Asha", "age": 20 },
  { "id": 2, "name": "Ravi", "age": 25 }
]
```

get one

```
GET http://localhost:3000/users/1

{ "id": 1, "name": "Asha", "age": 20 }
```

On the command line you can try it with `curl`:

Terminal

```
curl http://localhost:3000/users
curl http://localhost:3000/users/1
```

4. Create — POST

Send a new item in the request body to the collection. JSON Server gives it a new `id` and saves it, then returns the created item.

Terminal

```
curl -X POST http://localhost:3000/users \
  -H "Content-Type: application/json" \
  -d '{ "name": "Meera", "age": 22 }'
```

response

```
{ "id": 3, "name": "Meera", "age": 22 }
```

5. Update — PUT

Point at one item's id and send the full updated object. JSON Server replaces it and returns the new version.

Terminal

```
curl -X PUT http://localhost:3000/users/1 \
  -H "Content-Type: application/json" \
  -d '{ "name": "Asha", "age": 21 }'
```

response

```
{ "id": 1, "name": "Asha", "age": 21 }
```

6. Delete — DELETE

Point at the item's id with the `DELETE` method. JSON Server removes it and returns an empty object.

Terminal

```
curl -X DELETE http://localhost:3000/users/1
```

> *Check `db.json` after each call — you'll see the file actually change as you create, update, and delete.*

7. Pagination

A real collection can hold thousands of rows — you never load them all at once. **Pagination** fetches one page at a time: ask for a page number and a page size, and the server returns just that slice.

Page 1 (1–10) → Page 2 (11–20) → Page 3 (21–30)

JSON Server paginates with the `_page` and `_limit` query parameters.

requests

```
GET /users?_page=1&_limit=10   # items 1-10
GET /users?_page=2&_limit=10   # items 11-20
GET /users?_page=3&_limit=10   # items 21-30
```

To build page buttons you also need the **total count**. JSON Server sends it in the `X-Total-Count` response header — see it with `curl -i`:

Terminal

```
curl -i "http://localhost:3000/users?_page=1&_limit=10"

HTTP/1.1 200 OK
X-Total-Count: 57
...
[ /* 10 users */ ]
```

From the total and the page size you compute how many pages there are:

total pages

```
const total = 57;         // from X-Total-Count
const limit = 10;
const pages = Math.ceil(total / limit);  // 6
```

> *Version note: JSON Server v1 uses `_per_page` instead of `_limit` and returns a wrapper object — `{ first, prev, next, last, pages, data }` — with the page in `data`. Older v0 uses `_limit` + the `X-Total-Count` header shown above.*

8. Reading the count in Angular

`HttpClient` hides headers by default. Pass `{ observe: 'response' }` to get the full response, then read `X-Total-Count` from `res.headers`.

users.ts

```
import { HttpClient, HttpParams } from '@angular/common/http';

getPage(page: number, limit = 10) {
  const params = new HttpParams()
    .set('_page', page)
    .set('_limit', limit);

  return this.http.get<User[]>('http://localhost:3000/users', {
    params,
    observe: 'response'   // give me headers too
  });
}

// usage:
this.getPage(2).subscribe(res => {
  const users = res.body ?? [];
  const total = Number(res.headers.get('X-Total-Count'));
  const pages = Math.ceil(total / 10);
});
```

9. Sorting and filtering

Pagination usually pairs with sorting and filtering — all just query parameters in JSON Server.

examples

```
GET /users?_sort=age&_order=desc   # sort by age, high → low
GET /users?city=Delhi              # only users in Delhi
GET /users?age_gte=18&age_lte=30   # age between 18 and 30
GET /users?q=asha                  # full-text search
```

> *Combine them freely: `/users?city=Delhi&_sort=age&_page=1&_limit=10` — filter, sort, then page.*

10. Recap

- **JSON Server** turns a `db.json` into a real REST API for practice.
- **CRUD** = Create, Read, Update, Delete.
- Create → `POST /users`, Read → `GET /users` or `/users/1`.
- Update → `PUT /users/1`, Delete → `DELETE /users/1`.
- JSON Server saves changes straight into `db.json`.

CHAPTER 35

RxJS and Observables

Streams of values over time — the foundation behind HttpClient and forms.

1. What is an Observable?

You've been using Observables already — `HttpClient` returns them and you call `.subscribe()`. An **Observable** is a stream of values that arrive over time: it might emit one value (an HTTP response), many (every click or keystroke), or none. RxJS is the library of tools for working with these streams.

An observable emits values over time until it completes.

Promise

```
// one value, runs immediately
fetch('/api').then(r => ...)
```

Observable

```
// many values, runs on subscribe
http.get('/api').subscribe(r => ...)
```

> *Observables are **lazy**: nothing happens until you `subscribe`. A Promise runs as soon as it's created.*

More on rxjs.dev ↗

2. Basic: creating and subscribing

Make simple observables with `of` (a fixed set of values) or `from` (an array or promise). Subscribe to receive each value.

create + subscribe

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

of(1, 2, 3).subscribe(n => console.log(n));    // 1, 2, 3
from(['a', 'b']).subscribe(x => console.log(x)); // a, b
```

Long-lived streams keep emitting, so you must stop listening to avoid leaks. In components, `takeUntilDestroyed()` cleans up automatically.

auto cleanup

```
import { interval } from 'rxjs';
import { takeUntilDestroyed } from '@angular/core/rxjs-interop';

interval(1000)
  .pipe(takeUntilDestroyed())
  .subscribe(n => console.log(n)); // stops when the component is destroyed
```

3. Medium: operators with pipe

Operators transform a stream. You chain them inside `.pipe(...)` — like array `map`/`filter`, but for values arriving over time.

map / filter / tap

```
import { of, map, filter, tap } from 'rxjs';

of(1, 2, 3, 4).pipe(
  filter(n => n % 2 === 0),        // keep evens
  tap(n => console.log('saw', n)), // side-effect
  map(n => n * 10)                 // transform
).subscribe(n => console.log(n));  // 20, 40
```

> *Each operator returns a new observable; the original stream is untouched.*

4. Medium: the async pipe

In templates, the `async` pipe subscribes for you and unsubscribes when the component is destroyed — no manual `subscribe` needed.

component + template

```
// class:
users$ = this.http.get<User[]>('/api/users');

// template:
@if (users$ | async; as users) {
  @for (u of users; track u.id) {
    <li>{{ u.name }}</li>
  }
}
```

5. Advanced: switchMap (dependent calls)

When one stream should trigger another — like a search box firing an HTTP request per keystroke — use `switchMap`. It cancels the previous inner request when a new value arrives, exactly what a type-ahead needs.

type-ahead search

```
import { debounceTime, distinctUntilChanged, switchMap } from 'rxjs';

this.searchControl.valueChanges.pipe(
  debounceTime(300),       // wait for a pause in typing
  distinctUntilChanged(),  // ignore if unchanged
  switchMap(term =>        // swap to a new request, cancel the old
    this.http.get(`/api/search?q=${term}`)
  )
).subscribe(results => this.results = results);
```

> *Relatives: `mergeMap` (run all), `concatMap` (one after another), `exhaustMap` (ignore new while busy). `switchMap` is the go-to for search and navigation.*

6. Advanced: combining streams

Run several requests together with `forkJoin` (wait for all, like `Promise.all`), or react whenever any of several streams changes with `combineLatest`.

forkJoin (parallel)

```
import { forkJoin } from 'rxjs';

forkJoin({
  user: this.http.get('/api/user/1'),
  posts: this.http.get('/api/user/1/posts')
}).subscribe(({ user, posts }) => {
  // both have arrived
});
```

7. Advanced: Subjects

A **Subject** is both an observable and an emitter — you can push values into it. A **BehaviorSubject** also remembers the latest value and hands it to new subscribers, which makes it handy for simple shared state in a service.

BehaviorSubject

```
import { BehaviorSubject } from 'rxjs';

private count = new BehaviorSubject(0);
readonly count$ = this.count.asObservable();

increase() {
  this.count.next(this.count.value + 1);
}
```

> *Today, **signals** are often simpler for state; use observables for streams and async events. Bridge them with `toSignal(obs$)` and `toObservable(sig)`.*

8. Recap

- An Observable is a lazy stream of values over time; subscribe to receive them.
- Create with `of` / `from`; clean up with `takeUntilDestroyed()`.
- Transform with operators inside `.pipe()`: `map`, `filter`, `tap`.
- The `async` pipe subscribes and cleans up for you in templates.
- `switchMap` chains dependent calls; `forkJoin` runs calls in parallel.
- `Subject` / `BehaviorSubject` let you push values; signals are often simpler for state.

CHAPTER 36

HttpClient in a Component

Call your JSON Server API straight from a component with HttpClient.

---

1. Turn on HttpClient

In the last chapter we hit the API with `curl`. Now let's do it from Angular. Angular talks to APIs through `HttpClient` — switch it on once in your app config.

app.config.ts

```
import { provideHttpClient } from '@angular/common/http';

export const appConfig = {
  providers: [provideHttpClient()]
};
```

Then `inject` it wherever you need it. We'll keep everything in one component here — no separate service.

users.ts

```
import { Component, inject } from '@angular/core';
import { HttpClient } from '@angular/common/http';

export class Users {
  private http = inject(HttpClient);
  private url = 'http://localhost:3000/users';
}
```

HttpClient on angular.dev ↗

2. Read — GET the list

Call `http.get` with the URL and `subscribe` to receive the data. `get<User[]>` tells TypeScript the response is an array. Store it in a signal and load it when the component starts.

users.ts

```
import { Component, inject, signal, OnInit } from '@angular/core';
import { HttpClient } from '@angular/common/http';

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

@Component({ /* ... */ })
export class Users implements OnInit {
  private http = inject(HttpClient);
  private url = 'http://localhost:3000/users';
  users = signal<User[]>([]);

  ngOnInit() {
    this.http.get<User[]>(this.url)
      .subscribe(data => this.users.set(data));
  }
}
```

users.html

```
<ul>
  @for (user of users(); track user.id) {
    <li>{{ user.name }} ({{ user.age }})</li>
  }
</ul>
```

> *The request doesn't run until you `subscribe` — that's how Observables work.*

3. Create — POST

Send a new object in the body. When it succeeds, reload the list so the screen shows it.

users.ts

```
addUser() {
  const newUser = { name: 'Meera', age: 22 };
  this.http.post<User>(this.url, newUser).subscribe(() => {
    this.http.get<User[]>(this.url)
      .subscribe(data => this.users.set(data));
  });
}
```

4. Update — PUT

Point at the item's id and send the full updated object.

users.ts

```
saveUser(user: User) {
  const updated = { ...user, age: 30 };
  this.http.put<User>(`${this.url}/${user.id}`, updated)
    .subscribe(result => console.log('updated', result));
}
```

5. Delete — DELETE

Delete by id, then drop it from the signal so the list updates instantly.

users.ts

```
deleteUser(id: number) {
  this.http.delete(`${this.url}/${id}`).subscribe(() => {
    this.users.update(list => list.filter(u => u.id !== id));
  });
}
```

users.html

```
@for (user of users(); track user.id) {
  <li>
    {{ user.name }}
    <button (click)="deleteUser(user.id!)">Delete</button>
  </li>
}
```

6. Handling errors

Network calls can fail. Pass an object to `subscribe` with `next` and `error` handlers.

users.ts

```
this.http.get<User[]>(this.url).subscribe({
  next: data => this.users.set(data),
  error: err => console.error('Request failed:', err.status)
});
```

7. Recap

- Enable Angular HTTP with `provideHttpClient()`.
- `inject(HttpClient)` in the component and call `get/post/put/delete`.
- Requests are Observables — they run when you `subscribe`.
- Type the response: `get<User[]>`.
- Handle failures with an `error` handler in `subscribe`.

- This works for a small screen; for bigger apps you'd move the calls into a service.

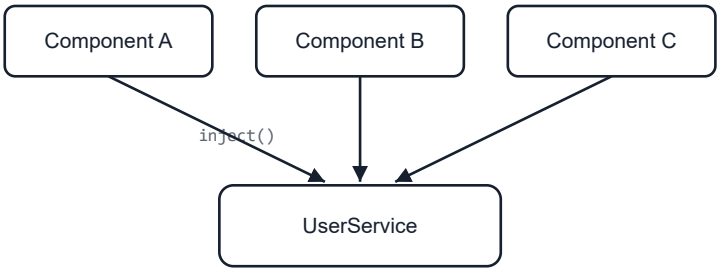
CHAPTER 37

Services and Dependency Injection

Share logic and data across components with injectable services.

1. What is a service?

A **service** is a plain class for logic and data that isn't tied to one screen — fetching data, holding shared state, calculations, logging. Components stay focused on the view; services do the rest, and many components can share one.



Many components share one service.

More on angular.dev ↗

2. Why services? What they fix

Without services, logic ends up copy-pasted into components, and data gets passed through layers of inputs and outputs just to reach where it's needed. Services solve that.

Reuse
Write logic once, use it in any component — no duplication

Thin components
Components handle the view; services handle data and rules

Single source of truth
One place owns the data, so screens stay in sync

Easy to test & swap
Test a service on its own; replace it without touching components

3. Generating a service

Let the CLI create it. A service is just a class marked `@Injectable`.

Terminal

```
ng generate service user
# short version:
ng g s user
```

user.service.ts

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

@Injectable({ providedIn: 'root' })
export class UserService {
  private users = ['Asha', 'Ravi'];

  getUsers() {
    return this.users;
  }
  addUser(name: string) {
    this.users.push(name);
  }
}
```

> *`providedIn: 'root'` makes one shared instance available everywhere — a **singleton** for the whole app.*

4. Dependency Injection (DI)

You don't create a service with `new`. You **ask** for it and Angular hands you the shared instance — that's **dependency injection**. Use the `inject()` function.

user-list.ts

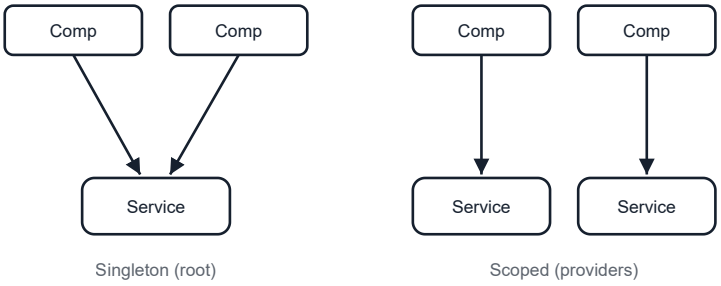
```
import { Component, inject } from '@angular/core';
import { UserService } from './user.service';

@Component({ selector: 'app-user-list', template: '...' })
export class UserList {
  private users = inject(UserService);

  list = this.users.getUsers();
}
```

You ask the injector; it provides the singleton.

> *You can also inject through the constructor: `constructor(private users: UserService) {}`. Both do the same thing; `inject()` is the modern style.*

5. Singleton vs scoped services

Where you *provide* a service decides how many instances exist. `providedIn: 'root'` creates one **singleton** shared by the whole app. Listing a service in a component's `providers` array instead creates a fresh, **scoped** instance for each copy of that component (and its children).



Singleton: one shared instance. Scoped: one per component.

singleton — shared app-wide

```
@Injectable({ providedIn: 'root' })
export class CounterService {
  count = signal(0);
}
// Every component that injects it sees the SAME count.
```

scoped — one per component

```
@Injectable() // note: no providedIn
export class CounterService {
  count = signal(0);
}

@Component({
  selector: 'app-widget',
  providers: [CounterService], // fresh instance per <app-widget>
  template: '...'
})
export class Widget {
  counter = inject(CounterService);
}
// Two <app-widget> elements each get their OWN count.
```

Use a singleton
Global, shared data: auth, cart, app config, caches

Use a scoped service
Per-component state that shouldn't leak: a wizard/form's state, a local store

> *A scoped service is created with its component and destroyed with it — handy for state that should reset each time the component appears.*

6. Recap

- A service is a class for reusable logic/data, marked `@Injectable`.
- Generate one with `ng g s name`.
- `providedIn: 'root'` = one shared singleton for the whole app.
- Get it with `inject(MyService)` (or the constructor).
- `providedIn: 'root'` = singleton; a component's `providers` array = a scoped instance per component.
- Services keep components thin and avoid duplicated code.

CHAPTER 38

HttpClient with a Service

Move your API calls into a service that components can share.

1. Why put HttpClient in a service?

In Chapter 36 we called the API straight from a component. That's fine for one screen, but the URL and request logic get copied around as the app grows. Move them into a **service**: one place owns the API, and every component just asks it for data.

Component → Service → HttpClient → Server

More on angular.dev ↗

2. The data service

Inject `HttpClient` once, keep the base URL in one spot, and expose a method per operation. Each returns an Observable — the request runs when something subscribes.

user.service.ts

```
import { inject, Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

export interface User { id?: number; name: string; age: number; }

@Injectable({ providedIn: 'root' })
export class UserService {
  private http = inject(HttpClient);
  private url = 'http://localhost:3000/users';

  getAll()            { return this.http.get<User[]>(this.url); }
  getOne(id: number)  { return this.http.get<User>(`${this.url}/${id}`); }
  create(user: User)  { return this.http.post<User>(this.url, user); }
  update(u: User)     { return this.http.put<User>(`${this.url}/${u.id}`, u); }
  remove(id: number)  { return this.http.delete(`${this.url}/${id}`); }
}
```

3. Example 1 & 2: read all and read one

The component injects the service and subscribes. It holds no URL and no HTTP code.

list.ts

```
private users = inject(UserService);
list = signal<User[]>([]);

ngOnInit() {
  this.users.getAll().subscribe(data => this.list.set(data));
}

open(id: number) {
  this.users.getOne(id).subscribe(user => console.log(user));
}
```

4. Example 3, 4 & 5: create, update, delete

Create and update send an object; delete takes an id. After a change, refresh the list (or update the signal locally).

list.ts

```
add() {
  this.users.create({ name: 'Meera', age: 22 })
    .subscribe(() => this.reload());
}

save(user: User) {
  this.users.update({ ...user, age: 30 })
    .subscribe(updated => console.log('saved', updated));
}

remove(id: number) {
  this.users.remove(id).subscribe(() => {
    this.list.update(l => l.filter(u => u.id !== id));
  });
}

private reload() {
  this.users.getAll().subscribe(data => this.list.set(data));
}
```

5. Loading and error state

Because the service returns an Observable, the component decides how to handle waiting and failure — usually with a couple of signals.

list.ts

```
loading = signal(false);
error = signal<string | null>(null);

load() {
  this.loading.set(true);
  this.users.getAll().subscribe({
    next: data => { this.list.set(data); this.loading.set(false); },
    error: () => { this.error.set('Could not load'); this.loading.set(false); }
  });
}
```

> *Tip: the async pipe can subscribe for you — `@if (users.getAll() | async; as list) { ... }` — though caching the Observable in a field is better than calling it in the template.*

6. Recap

- Put `HttpClient` and the API URL in a service, not the component.
- Expose one method per operation, each returning an Observable.
- Components `inject` the service and subscribe — they stay thin.

- The same service is reused by every component that needs the data.
- Handle loading/errors in the component (or via the `async` pipe).

CHAPTER 39

Sharing Data with a Service

Let far-apart components communicate through a shared service.

1. The problem: prop drilling

Inputs and outputs are perfect between a parent and its direct child. But when data must travel through many levels — or between two components that aren't related at all — passing it down through every layer ("prop drilling") gets painful and brittle.

Threading data through every level vs. one shared service.

More on angular.dev ↗

2. A shared state service

A `providedIn: 'root'` service is a single shared instance. Put the state in it as **signals**, and any component — however deep or unrelated — can read and update the same data.

cart.service.ts

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

@Injectable({ providedIn: 'root' })
export class CartService {
  readonly items = signal<string[]>([]);
  readonly count = computed(() => this.items().length);

  add(item: string) {
    this.items.update(list => [...list, item]);
  }
  clear() {
    this.items.set([]);
  }
}
```

3. Two unrelated components, one source

A product list (deep in one part of the tree) adds items; a cart badge (in the header) shows the count. They never talk to each other — they both just use the service.

product.ts

```
private cart = inject(CartService);

buy(name: string) {
  this.cart.add(name);
}
```

cart-badge.ts

```
private cart = inject(CartService);

// template:  🛒 {{ cart.count() }}
count = this.cart.count;
```

> *Because both injected the same singleton, adding in one place updates the badge instantly — no inputs, outputs, or shared parent needed.*

4. Sending events across components

For one-off events (a toast message, a "refresh" ping) rather than stored state, expose a `Subject` from the service. Components subscribe to react.

notify.service.ts

```
import { Injectable } from '@angular/core';
import { Subject } from 'rxjs';

@Injectable({ providedIn: 'root' })
export class NotifyService {
  private messages = new Subject<string>();
  readonly messages$ = this.messages.asObservable();

  show(text: string) {
    this.messages.next(text);
  }
}
```

anywhere

```
// sender:
this.notify.show('Saved!');

// listener (e.g. a toast component):
this.notify.messages$.subscribe(text => this.toast = text);
```

5. Recap

- Use inputs/outputs for direct parent ↔ child; use a service for far-apart or unrelated components.
- A `providedIn: 'root'` service is one shared instance for everyone.
- Hold shared state as **signals** (with `computed` for derived values).
- Any component injects the service and reads/updates the same data.
- For events instead of state, expose a `Subject` and subscribe.

CHAPTER 40

Environments

Keep different settings — like API URLs — for development and production.

1. Why environments?

Your app needs different settings depending on where it runs. In development the API is `http://localhost:3000`; in production it's your real server. Hard-coding URLs means editing code before every deploy. **Environment files** hold these settings separately, and Angular swaps them in automatically at build time.

ng serve → uses dev settings → ng build → uses prod settings

More on angular.dev ↗

2. Creating the files

The CLI scaffolds the environment files and wires them into the build for you.

Terminal

```
ng generate environments
```

It creates two files. The base one is used for production; the `.development` one overrides it during development.

src/environments/environment.ts

```
export const environment = {
  production: true,
  apiUrl: 'https://api.myapp.com'
};
```

src/environments/environment.development.ts

```
export const environment = {
  production: false,
  apiUrl: 'http://localhost:3000'
};
```

3. How the swap works

`ng generate environments` adds a **file replacement** to `angular.json`. During a development build, Angular replaces `environment.ts` with `environment.development.ts` — so your imports stay the same, but the values change.

angular.json (development config)

```
"development": {
  "fileReplacements": [
    {
      "replace": "src/environments/environment.ts",
      "with": "src/environments/environment.development.ts"
    }
  ]
}
```

> *`ng serve` uses the development configuration (so you get the dev values); `ng build` defaults to production (the base file).*

4. Using it in your code

Always import from `environment.ts` (no suffix). Angular substitutes the right file for the build. A service is the perfect place to use it — instead of a hard-coded URL.

user.service.ts

```
import { inject, Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { environment } from '../environments/environment';

@Injectable({ providedIn: 'root' })
export class UserService {
  private http = inject(HttpClient);
  private url = `${environment.apiUrl}/users`;

  getAll() {
    return this.http.get(this.url);
  }
}
```

> *Environment files are bundled into the browser, so never put secrets (API keys, passwords) in them — anyone can read the shipped code. Keep secrets on the server.*

5. Recap

- Environments hold per-build settings like the API URL.
- Create them with `ng generate environments`.
- `environment.ts` = production; `environment.development.ts` overrides it in dev.
- `angular.json` file replacement swaps them at build time.
- Import from `environment.ts` everywhere — Angular picks the right one.
- Never store secrets — env values ship to the browser.

CHAPTER 41

HTTP Interceptors

Run code on every request and response — auth tokens, errors, logging.

1. What is an interceptor?

An **interceptor** is a function that sits between your app and the server. Every HTTP request passes through it on the way out, and every response on the way back — so it's the perfect place to do something for *all* calls at once: attach an auth token, log requests, show a loading bar, or handle errors globally.

Your request → Interceptor → Server → Interceptor → Your code

> *Instead of adding the same header or try/catch in every service method, you write it once in an interceptor.*

More on angular.dev ↗

2. Writing and registering one

A modern interceptor is a function of type `HttpInterceptorFn`. It receives the request and a `next` function to pass it along. Register interceptors with `withInterceptors`.

logging.interceptor.ts

```
import { HttpInterceptorFn } from '@angular/common/http';

export const loggingInterceptor: HttpInterceptorFn = (req, next) => {
  console.log('→', req.method, req.url);
  return next(req); // pass it on
};
```

app.config.ts

```
import { provideHttpClient, withInterceptors } from '@angular/common/http';
import { loggingInterceptor } from './logging.interceptor';

export const appConfig = {
  providers: [
    provideHttpClient(withInterceptors([loggingInterceptor]))
  ]
};
```

3. Attaching an auth token

Requests are **immutable**, so you `clone` the request and add a header. Here we add a Bearer token from a service to every outgoing call.

auth.interceptor.ts

```
import { HttpInterceptorFn } from '@angular/common/http';
import { inject } from '@angular/core';
import { AuthService } from './auth.service';

export const authInterceptor: HttpInterceptorFn = (req, next) => {
  const token = inject(AuthService).token();
  if (!token) {
    return next(req);
  }
  const authReq = req.clone({
    setHeaders: { Authorization: `Bearer ${token}` }
  });
  return next(authReq);
};
```

> *You can `inject()` services inside an interceptor, just like in a guard.*

4. Handling errors globally

Use the RxJS `catchError` operator on the response stream to deal with failures in one place — for example, logging out and redirecting on a `401 Unauthorized`.

error.interceptor.ts

```
import { HttpInterceptorFn } from '@angular/common/http';
import { inject } from '@angular/core';
import { Router } from '@angular/router';
import { catchError, throwError } from 'rxjs';

export const errorInterceptor: HttpInterceptorFn = (req, next) => {
  const router = inject(Router);
  return next(req).pipe(
    catchError(err => {
      if (err.status === 401) {
        router.navigate(['/login']);
      }
      return throwError(() => err);
    })
  );
};
```

> *Interceptors run in the order you list them: `withInterceptors([authInterceptor, errorInterceptor])`.*

5. Recap

- An interceptor runs on every HTTP request and response.
- It's an `HttpInterceptorFn` that calls `next(req)`.
- Register with `provideHttpClient(withInterceptors([...]))`.
- Requests are immutable — `req.clone({ setHeaders })` to add headers.
- Use `catchError` for global error handling (e.g. 401 → login).
- Great companion to services and route guards for auth.

CHAPTER 42

Authentication Flow

Put it all together: login, store a token, attach it, and guard routes.

---

1. How the pieces fit

Real auth combines four things you've already learned: a **service** holds the token, the user **logs in**, an **interceptor** attaches the token to every request, and a **guard** blocks protected routes. This chapter wires them into one flow.

Login → Store token → Guard allows route → Interceptor adds token → API call

More on angular.dev ↗

2. 1. The AuthService

The service owns everything auth-related: calling the login API, storing the token (in `localStorage` so it survives refreshes), exposing it as a signal, and logging out.

auth.service.ts

```
import { inject, Injectable, signal } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { tap } from 'rxjs';

@Injectable({ providedIn: 'root' })
export class AuthService {
  private http = inject(HttpClient);
  readonly token = signal<string | null>(localStorage.getItem('token'));

  isLoggedIn() {
    return this.token() !== null;
  }

  login(email: string, password: string) {
    return this.http
      .post<{ token: string }>('/api/login', { email, password })
      .pipe(tap(res => {
        localStorage.setItem('token', res.token);
        this.token.set(res.token);
      }));
  }

  logout() {
    localStorage.removeItem('token');
    this.token.set(null);
  }
}
```

3. 2. The login page

A small form calls `login()`. On success it sends the user to where they were headed (the `returnUrl`), or to the dashboard.

login.ts

```
private auth = inject(AuthService);
private router = inject(Router);
private route = inject(ActivatedRoute);

submit(email: string, password: string) {
  this.auth.login(email, password).subscribe({
    next: () => {
      const returnUrl =
        this.route.snapshot.queryParamMap.get('returnUrl') ?? '/dashboard';
      this.router.navigateByUrl(returnUrl);
    },
    error: () => alert('Login failed')
  });
}
```

4. 3. The guard (with returnUrl)

The guard (Chapter 32) blocks protected routes when logged out — and remembers where the user wanted to go by passing a `returnUrl` to the login page.

auth.guard.ts

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

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

  if (auth.isLoggedIn()) {
    return true;
  }
  return router.createUrlTree(['/login'], {
    queryParams: { returnUrl: state.url }
  });
};
```

5. 4. The token interceptor

The interceptor (Chapter 41) attaches the stored token to every request, and logs the user out on a `401`.

auth.interceptor.ts

```
import { HttpInterceptorFn } from '@angular/common/http';
import { inject } from '@angular/core';
import { Router } from '@angular/router';
import { catchError, throwError } from 'rxjs';
import { AuthService } from './auth.service';

export const authInterceptor: HttpInterceptorFn = (req, next) => {
  const auth = inject(AuthService);
  const router = inject(Router);
  const token = auth.token();

  const sent = token
    ? req.clone({ setHeaders: { Authorization: `Bearer ${token}` } })
    : req;

  return next(sent).pipe(
    catchError(err => {
      if (err.status === 401) {
        auth.logout();
        router.navigate(['/login']);
      }
      return throwError(() => err);
    })
  );
};
```

6. 5. Wiring it up

Register the interceptor, and protect routes with the guard.

app.config.ts

```
providers: [
  provideHttpClient(withInterceptors([authInterceptor])),
  provideRouter(routes)
]
```

app.routes.ts

```
export const routes: Routes = [
  { path: 'login', component: Login },
  { path: 'dashboard', component: Dashboard, canActivate: [authGuard] }
];
```

> *Now: visiting /dashboard logged out → guard sends you to /login?returnUrl=/dashboard → after login you're sent back, and every API call carries your token.*

7. Recap

- **AuthService** logs in, stores the token (localStorage + signal), and logs out.
- The **login page** redirects to `returnUrl` on success.
- The **guard** blocks protected routes and passes a `returnUrl`.
- The **interceptor** attaches the token and handles 401s.
- Together they form a complete, real-world auth flow.