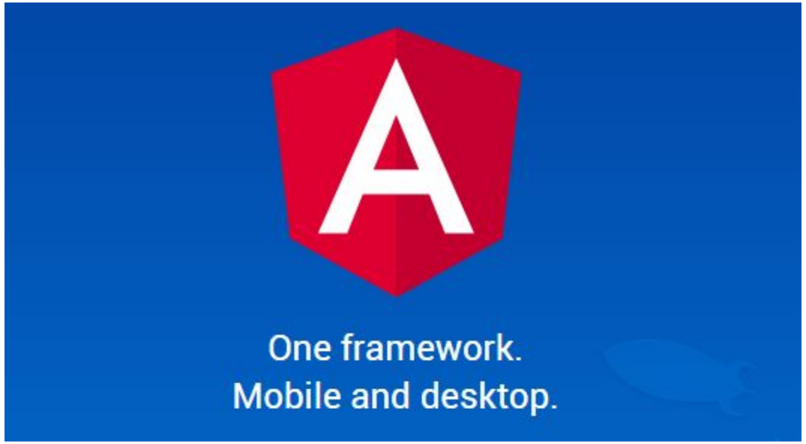


Damiano Amici

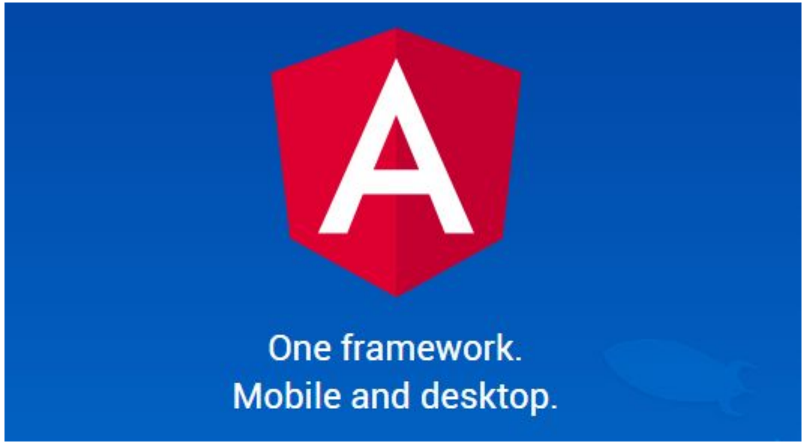
FullStack Developer

@ bit Time Professionals

bit Time professionals
consulenza | formazione | sviluppo

Blind Setup

- Install *GIT* from https://git-scm.com/download/win
- Install *Node.js* from https://nodejs.org
- Go to the command line and run
  - `npm install -g @angular/cli@latest`
- Install *VSCode* from https://code.visualstudio.com (recommended)



ANGULAR INTRODUCTION

bit Time professionals
consulenza | formazione | sviluppo

Overview

- Angular is a framework developed by Google for building large, scalable web applications.
- Uses TypeScript, a component-based architecture.
- Provides a complete ecosystem: routing, forms, HTTP, dependency injection, build tools.

Create the demo project

- **ng new demo**
- **cd demo**
- **ng serve**
  - Navigate to http://localhost:4200
- VSCode "Open folder" **C:\<YOUR_DIR>\demo**
- Let's study the project structure...

How we build angular applications

- We create components (HTML + TS + CSS) organized into modules.
- Use the Angular CLI to generate code, manage builds, run dev servers, and bundle for production.
- Architect the app using services, DI, routing, and observables for state and data handling.

What Are Angular Components?

Components are the fundamental building blocks of Angular applications.

Each component combines HTML template, TypeScript logic, and CSS styling.

They represent an independent and reusable section of the user interface.

Component Structure

- @Component decorator: defines metadata like selector, template, style.
- TypeScript class: contains logic and properties.
- HTML template: defines the visual structure.
- CSS styles: define the appearance.

Creating a Component

CLI: ng generate component component-name.

Generates: .ts, .html, .css/.scss, .spec.ts files.

Automatically added to a module (non-standalone).

Demo: Creating an Angular Component

First Component

Standalone Components

Introduced in Angular 14 to simplify architecture.

Do not require an NgModule.

Imported directly by other components or through the router.

Benefits of Standalone Components

Reduced complexity by removing the need for NgModules.

Better tree-shaking and improved performance.

Cleaner and more modular import structure.

Displaying Data

When it comes to outputting dynamic content in Angular applications, you get two main ways of binding your template or parts of your template to values generated by the component class.

- String Interpolation {{ value }}
- Property Binding

Property Binding

Is a key Angular feature that allows you to bind element properties to dynamic values.

For example, <img [src]="someSrc"> binds the src property of the underlying HTMLImageElement DOM object to the value stored in someSrc.

Create Your First Angular Component

- Create a new Angular component called UserCard.
- Add a property name and display it in the template.
- Add a property avatar or icon and display the image.

Introducing click event

- In the next exercises we'll need some user interaction using buttons.
- (click) is Angular’s event binding for listening to click events.
- It connects a DOM event (user clicks) to a method in the component.

Click Event Syntax

```
<button (click)="onButtonClick()">Click me</button>
```

When the button is clicked, Angular calls onButtonClick().

Exercise 0

Create a counter

Change Detection

Angular automatically detects and finds out when data changes and it then simply takes a look at the template of the component and verifies whether that template, now that the data changed, maybe produces a different DOM snapshot.

And if that's the case, it goes ahead and updates the UI so that changes are reflected there.

Change Detection

Angular does all of that automatically, and it does all of that automatically with help of a part of the Angular framework that's called zone.js.

Signals

A signal is a wrapper around a value that notifies interested consumers when that value changes. Signals can contain any value, from primitives to complex data structures.

You read a signal's value by calling its getter function, which allows Angular to track where the signal is used.

Why Signal Was Introduced

This allows Angular to update the UI in a more fine-grained way where it doesn't have to check everything for every possible event that could occur anywhere in the application.

Signal Properties

- holds a value
- notifies dependents when the value changes
- automatically updates the UI

computed()

computed() creates a read-only signal whose value depends on other signals.

Automatically updates when any dependency changes.

No manual subscriptions

Perfect for derived, combined, or formatted values

effect()

effect() runs side effects when signals change.

Useful for:

- logging
- localStorage sync
- calling services / APIs
- interacting with the outside world

Rules of effect()

- It should be used for side effects only
- Not for computing values → use computed() instead
- Runs immediately once when created
- Reruns only when a dependency signal changes
- Automatically cleaned up when the component is destroyed — no need for unsubscribe

When not to use effect()

Avoid using effects for propagation of state changes. This can result in ExpressionChangedAfterItHasBeenChecked errors, infinite circular updates, or unnecessary change detection cycles.

Instead, use computed signals to model state that depends on other state.

effect() clean up

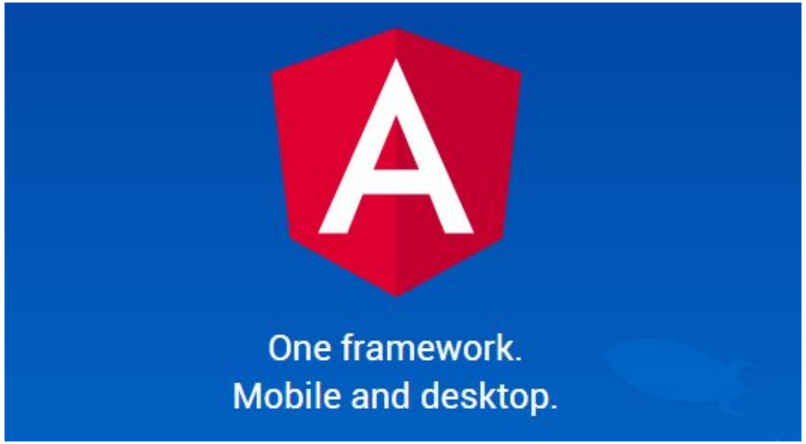
Effects might start long-running operations, which you should cancel if the effect is destroyed or runs again before the first operation finished.

When you create an effect, your function can optionally accept an onCleanup function as its first parameter.

This onCleanup function lets you register a callback that is invoked before the next run of the effect begins, or when the effect is destroyed.

Demo

Counter signal



Angular

COMMUNICATION BETWEEN COMPONENTS

@Input Properties

- A parent component can pass parameters to child component using child's input parameters
- Angular automatically updates data-bound properties.

```
@Component({
  selector: 'bank-account'
})
class BankAccount {
  @Input() bankName: string;
  @Input('account-id') id: string;
  normalizedBankName: string; //not updated
}
```

```
Here's how parent Component can use
these Input properties
<bank-account bank-name="RBC"
account-id="4747"></bank-account>
```

Signal Input

- instead of using Input decorator we will use input function from @angular/core
- makes the property become a signal inside our component.

```
@Component({
  selector: 'bank-account'
})
class BankAccount {
   bankName = input<string>();

}
```

```
Here's how parent Component can use
these Input properties
<bank-account bank-name="RBC"
account-id="4747"></bank-account>
```

Why Signal Input

- As a result, Angular can efficiently update only the UI parts or code values that depend on those signal-based sources.
- This leads to cleaner code, fewer bugs, and more predictable reactivity.

What is @Output?

- @Output() is a decorator used to emit events from a child component to a parent component.
- Enables child → parent communication.
- works together with an EventEmitter.
- Today this is replaced by the output function.

Signal Output Equivalent

output() is a new function that makes syntax looks cleaner when using the new input() function to get parameters, it is exactly the same thing as

@Output event = new EventEmitter();

Why Do We Need It?

- Parent components may need to react to actions happening inside a child.
- Useful for:
    - Button clicks inside child components
    - Form submissions
    - Passing selected data upward

bit Time professionals
consulenza | formazione | sviluppo

DEMO

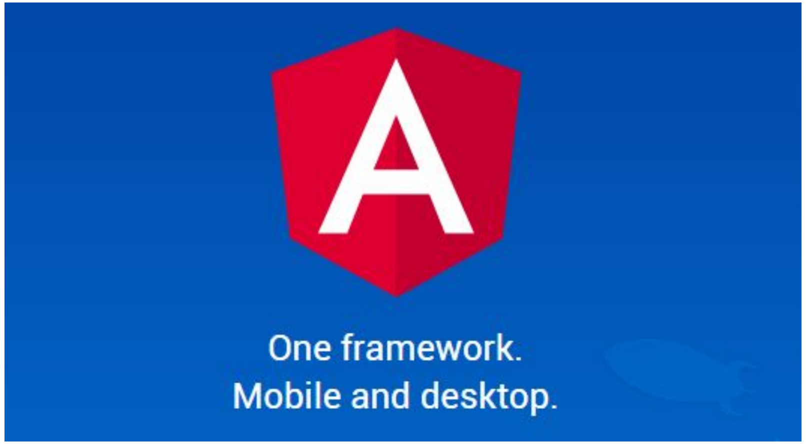
- userv2-select

Exercise 1

- Create a component ProductCardComponent that receive via @Input:
  - name: string
  - price: number
- In the component add a button “Add to cart”.
- When the user click the button, the component should emit an event @Output() add = new EventEmitter<void>().
- The ParentComponent should:
  - visualize a list of ProductCardComponent
  - listen the add event and show a log in console like "X added to cart".

Built-In control flow syntax

Angular



Angular

BUILT-IN CONTROL FLOW SYNTAX

bit Time professionals
consulenza | formazione | sviluppo

@for

@for is Angular's control flow loop, used to iterate over a collection and render a block of template for each item.

```
<ul>
  @for(let person of people; track $index) {
    <li>
      {{ person.first_name }}
    </li>
  }
</ul>
```

@for variables

@for provides built-in contextual variables:

$index – current index

$count – total number of items

$first – true for first element

$last – true for last element

$even – true on even index

$odd – true on odd index

@for samples

```
//phones: string[] = ['3496548798','3283216487','3336549875'];
@for(phone of phones; track $index) {
  <p>{{phone}}</p>
}
//people is declared as an array of object
<ul>
   @for(person of people; track $index) {
     <li>
       {{ person.first_name }}
     </li>
   }
</ul>
```

bit Time professionals
consulenza | formazione | sviluppo

DEMO

- for sample

Exercise 2

- Define a class Phone **value: string** and **type: string**
- Declare an array of phones (value and type) and display it in the template
  - Use the following standard types: "home","work","partner"

@if

- Sometimes an app needs to display a view or a portion of a view only under specific circumstances.
- The Angular **@if** syntax inserts or removes an element based on a truthy/falsy condition.

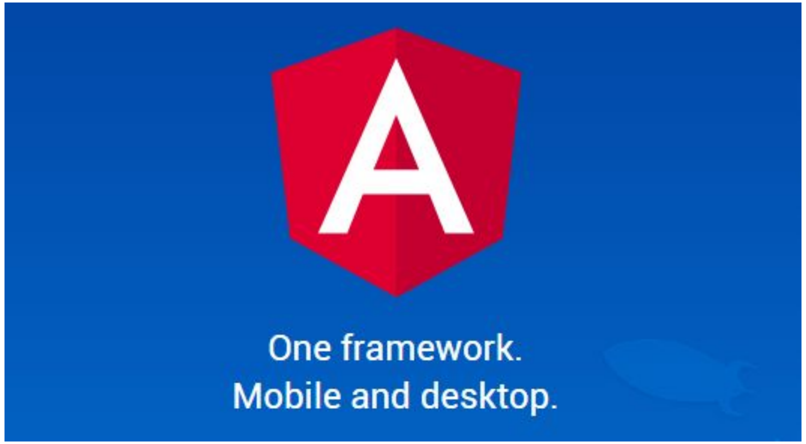
```
@if(people.length > 3) {
  <p>There are many heroes!</p>
}
```

DEMO

- If Sample

Exercise 3

- Define an array of 5 phones as defined in the previous exercise
- Write "(Warning! This is the home phone!!)" near the "home" phone number



Angular

DYNAMIC CSS STYLING

bit Time professionals
consulenza | formazione | sviluppo

Angular Template Property Syntax

First lets look at how we would change a <div> color property in pure JavaScript.

```
let myDiv = document.getElementById('my-div');
myDiv.style.color = 'orange';    // updating the div via its
                                 // properties
```

Now lets look at the primitives the new Angular 2 syntax gives us out of the box. Using the [property] syntax we can easily access any element or component properties.

```
<div [style.color]="'orange'">style using property syntax, this
text is orange</div>
```

- We can directly access the style property of our div element. This is different than an attribute.
- Properties are the properties defined on our DOM object just like the one we updated in plain JavaScript.

Property Syntax

We can also use the Angular property syntax to add CSS classes to elements.

```
<div [className]="'blue'">CSS class using
         property syntax, this text is blue</div>
```

Dynamic CSS Styling

- You can set a given **DOM element CSS properties** from Angular expressions.
- The simplest way to use this directive is by doing [style.<cssproperty>]="value"

```
<div [style.background-color]="'yellow'">
  Uses fixed yellow background
</div>
```

Dynamic CSS Styling

Another way to set fixed values is by using the NgStyle attribute and using key value pairs for each property you want to set, like this:

```
<div [ngStyle]="{color: 'white', 'background-color': 'blue'}">
  Uses fixed white text on blue background
</div>
```

But the real power of the NgStyle directive comes with using dynamic values

DEMO

- Dynamic Styling

ngClass

ngClass allows you to dynamically set and change the CSS classes for a given DOM element.

It works in the same way as in AngularJS

```
<div [ngClass]="['bold-text', 'green']">array of classes</div>
<div [ngClass]="'italic-text blue'">string of classes</div>
<div [ngClass]="{'small-text': true, 'red': true}">object of classes</div>
```

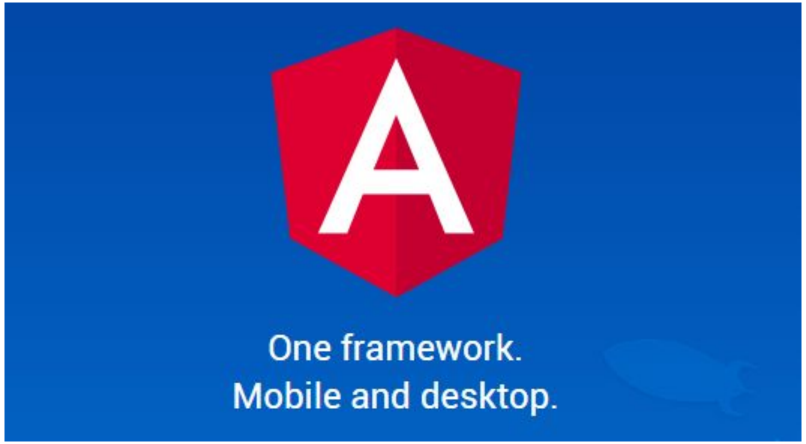
Exercise 4

Implement an Angular component that represents:

- a progress bar
- a status badge (es: “OK”, “Attention”, “Critical”)

The component must have:

- A numeric property progress (0–100).
- Two buttons:
  - “+10%” → increment progress by 10 (max 100)
  - “-10%” → decrement progress by 10 (min 0)
- Progress bar:
  - Dynamic width based on progress
  - use [style.width.%]="progress"
  - change color based on progress:
    - 0–30 → rosso (danger)
    - 31–70 → arancione (warning)
    - 71–100 → verde (success)
- Un badge di testo che mostra:
  - progress between 0 and 30 → “CRITICAL”
  - progress between 31 and 70 → “ATTENTION”
  - progress between 71 and 100 → “OK”



Angular

PIPES

Pipes

- A pipe takes in data as input and transforms it to a desired output.
- Here's how to transform a component's **birthday** property into a human-friendly date:

```
import {Component} from '@angular/core'
@Component({
  selector: 'hero-birthday',
  template: `<p>The hero's birthday is {{ birthday | date }}</p>`
})
export class HeroBirthday {
  public birthday = new Date(1988,3,15); // April 15, 1988
}
```

Built-in pipes

- Angular comes with a stock set of pipes such as `DatePipe`, `UpperCasePipe`, `LowerCasePipe`, `CurrencyPipe`, and `PercentPipe`. They are all immediately available for use in any template.

COMMON_PIPES

- AsyncPipe
- KeyValuePipe
- UpperCasePipe
- LowerCasePipe
- TitleCasePipe
- JsonPipe
- SlicePipe
- DecimalPipe
- PercentPipe
- CurrencyPipe
- DatePipe
- ReplacePipe
- I18nPluralPipe
- I18nSelectPipe

Parameterizing a Pipe

- A pipe may accept any number of optional parameters to fine-tune its output.
- We add parameters to a pipe by following the pipe name with a colon (:) and then the parameter value (e.g., currency:'EUR'). If our pipe accepts multiple parameters, we separate the values with colons (e.g. slice:1:5)

Parameterizing a Pipe

- We'll modify our **birthday** template to give the date pipe a format parameter. After formatting the hero's April 15th birthday should display as 15/04/1988.

```
<p>Hero's birthday is {{ birthday | date:"dd/MM/yyyy" }} </p>
```

- The parameter value can be any valid template expression such as a string literal or a component property.
- We can control the format through a binding the same way we control the **birthday** value through a binding.

Pipe parameters can be binding expressions

```
template: `
  <p>The hero's birthday is {{ birthday | date:format }}</p>
  <button (click)="toggleFormat()">Toggle Format</button>
`

export class HeroBirthday {
  birthday = new Date(1988,3,15); // April 15, 1988
  toggle = true; // start with true == shortDate
  get format() { return this.toggle ? 'shortDate' : 'fullDate'}
  toggleFormat() { this.toggle = !this.toggle; }
}
```

Chaining pipes

- We can chain pipes together in potentially useful combinations. In the following example, we chain the birthday to the DatePipe and on to the UpperCasePipe so we can display the birthday in uppercase.
- The following birthday displays as `APR 15, 1988`

```
<p>
  The chained hero's birthday is
  {{ birthday | date | uppercase}}
</p>
```

DEMO

Common Pipes

Writing Custom Pipes

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

@Pipe({  name: 'SexDecode'})
export class SexDecodePipe implements PipeTransform {
  transform(value: any, args?: any): any {
    switch (value) {
      case 'M': return 'Male';
      case 'F': return 'Female';
    }
    return '??';
  }
}
```

DEMO

Custom Pipe

bit Time professionals
consulenza | formazione | sviluppo

Exercise 5

Create a component called UserCardComponent that displays basic user information using Angular built-in pipes and one custom pipe.

Define a user object in the component's TypeScript file with:

- name: string
- birthDate: Date
- balance: number
- status: 'active' | 'inactive'

Display user information in the template using built-in pipes:

- Name → format using the titlecase pipe
- (e.g., "john doe" → "John Doe")
- Birth date → format using the date pipe
- Format: "longDate"
- Balance → format using the currency pipe
- Currency: "EUR"

Create a custom pipe named userStatus that transforms:

- "active" → "🟢 Active user"
- "inactive" → "🔴 Inactive user"

Add a button to toggle the user's status

Inputs

- In Angular, <input> elements are commonly used to capture user text.
- Angular provides multiple ways to bind data to an input field.
- Understanding the differences between binding styles is essential for predictable UI behavior.

One-Way Binding: From Component to Input

- Data flows component → input.
- The user can type, but changes won't update the component automatically.
- Good when you want the input to display data without tracking user edits.

```
<input [value]="username">
```

One-Way Binding: From Input to Component

- Data flows input → component.
- Used to manually capture user changes.
- More explicit but requires writing event handlers.

```
<input (input)="username = $event.target.value">
```

Two-Way Data Binding

- Combines property binding + event binding.
- Data flows both ways:
- UI updates when the model changes, and the model updates when the user types.
- Requires importing FormsModule.

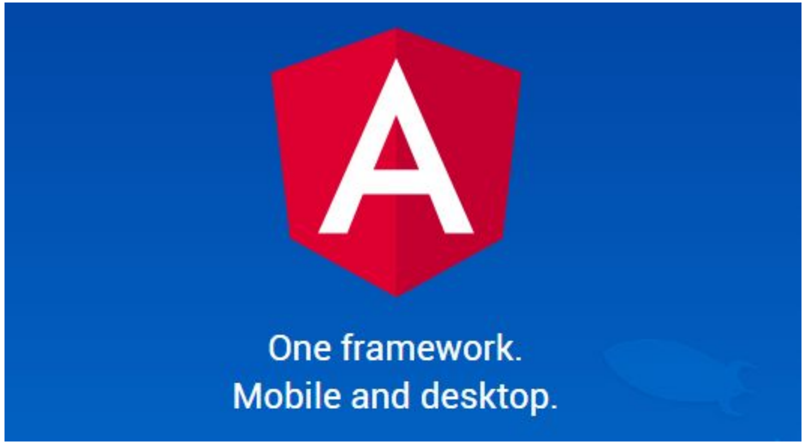
```
<input [(ngModel)]="username">
```

Exercise 6

Create a customer’s profile page.

The form should:

- Display the current customer data
- Allow the user to update name and phone number (two way data bindings)
- Validate that the email is valid (use input event)
- Update the customer email when "Set Email" is clicked.



Angular

FORMS

bit Time professionals
consulenza | formazione | sviluppo

Forms are not simple as you may think...

Forms can end up being really complex.

- Form inputs are meant to modify data, both on the page and the server
- Changes often need to be reflected elsewhere on the page
- Users have a lot of leeway in what they enter, so you need to validate values
- The UI needs to clearly state expectations and errors, if any
- Dependent fields can have complex logic
- We want to be able to test our forms, without relying on DOM selectors

Angular's two form building approaches.

Template-driven: simple, template-first.

Reactive forms: structured, model-driven.

Template-Driven Forms: What They Are

Form logic lives mainly in the template.

Uses ngModel.

Ideal for simple, small forms.

Less boilerplate but less control.

Using Template-Driven Forms

Import FormsModule.

Use ngModel to sign an input as trackable by the form.

Add #form=”ngForm” to reference form state in template.

Angular creates form model automatically.

DEMO

Login Form Sample

Reactive Forms: What They Are

- Form logic lives in the component class.
- Uses FormGroup, FormControl, FormBuilder.
- Ideal for large, dynamic forms.
- More predictable and testable.

Using Reactive Forms

- Import ReactiveFormsModule.
- Create FormGroup and FormControls.
- Bind with formGroup and formControlName.
- Full control over validation and state.

DEMO

Reactive Login Form Sample

Import forms related modules

```
import {
  FormsModule, //For template driven forms
  ReactiveFormsModule //For Reactive forms
} from '@angular/forms';
```

FormControls

- A **FormControl** represents a single input field – it is the smallest unit of an Angular form.
- FormControls encapsulate the field’s value, and states such as being valid, dirty (changed), or has errors
- To build up forms we create FormControls (and groups of FormControls) and then attach metadata and logic to them.

FormControl Sample

```
// create a new FormControl with the value "Daniele"
let nameControl = new FormControl("Daniele");
let name = nameControl.value; // -> Daniele

// now we can query this control for certain values:
nameControl.errors // -> StringMap of errors
nameControl.dirty  // -> false
nameControl.valid  // -> true
// etc
```

bit Time professionals
consulenza | formazione | sviluppo

FormGroup

- **FormGroup** is a wrapper class around a collection of FormControls.
  – FormGroup and FormControl have a common ancestor (AbstractControl) so that we can check the status or value of personInfo just as easily as a single FormControl (Composite).

```
let personInfo = new FormGroup({
 firstName: new FormControl("Daniele"),
 lastName: new FormControl("Teti"),
 zip: new FormControl("00100")
});
personInfo.value; // {"firstName":"Daniele", "lastName":"Teti",
                             "zip":"00100"}
```

The simplest form (One Way Binding)

```
<form #f="ngForm" (ngSubmit)="onSubmit(f.value)">
 <div>
  <label for="skuInput">SKU</label>
  <input type="text" id="skuInput" name="sku" ngModel>
 </div>
 <button type="submit">Submit</button>
</form>
```

The simplest form (One Way Binding)

```
<form #f="ngForm" (ngSubmit)="onSubmit(f.value)">
 <div>
  <label for="skuInput">SKU</label>
  <input type="text" id="skuInput" name="sku" ngModel>
 </div>
 <button type="submit">Submit</button>
</form>
```

Each form is automagically linked to NgForm directive and defined a FormGroup.
#f is a reference to the generated FormGroup, so f.value contains all the FormControl of the form

Using FormBuilder

- Building our **FormControls** and **FormGroups** implicitly using **ngForm** and **ngControl** is convenient, but doesn’t give us a lot of customization options.
- A more flexible and common way to configure forms is to use a **FormBuilder**.
- You can think of it as a “factory” object (it is a **Builder** design pattern)

Reactive Forms with FormBuilder

For this component we're going to be using the **formGroup** and **formControl** directives which means we need to import the appropriate classes

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

Validators and CSS

To better understand what's going on in validators, they add specific CSS classes to the input controls based on control state

State	Class if true	Class if false
Control has been visited	ng-touched	ng-untouched
Control's value has changed	ng-dirty	ng-pristine
Control's value is valid	ng-valid	ng-invalid

Custom Validators

Every now and then developers come across situations where existing validators are not suitable for their needs, if you are facing this situation your best option is to create a custom validator.

Template Driven: UpperCase Custom Validator

In this example the attribute **uppercase** is the custom validator, for this input to be valid it must contain uppercase text.

```
<input type="text" uppercase [(ngModel)]="field1" name="field1">
```

bit Time professionals
consulenza | formazione | sviluppo

Template Driven: UpperCase Custom Validator

```
import { Directive } from '@angular/core';
import { Validator, AbstractControl, NG_VALIDATORS } from '@angular/forms';
@Directive({
  selector: '[uppercase]',
  providers: [{ provide: NG_VALIDATORS, useExisting:
      OnlyUpperCaseValidatorDirective, multi: true }]})
export class OnlyUpperCaseValidatorDirective implements Validator {
  constructor() { }
  validate(c: AbstractControl) {
    if (!c.value) { return null; }
    if (c.value !== c.value.toUpperCase()) {
      return { 'UpperCase': c.value };
    }
    return null;
  }
}
```

Template Driven: UpperCase Custom Validator

- Every validator has a **validate()** method, Angular will call it every time the application needs to validate the field
- Note that the **validate()** method receives an **AbstractControl** as a parameter, which is the form control itself, this parameter will give us access to all of the field attributes, including the value.

OR

Reactive: UpperCase Custom Validator

```
const mustBeUppercase = (control: AbstractControl) => {
  const value = control.value as string;
  if (value !== value?.toUpperCase()) {
    return { mustBeUppercase: true };
  }
  return null;
};
```

Usage:

```
name: new FormControl('', {
      validators: [Validators.required, mustBeUppercase],
    }),
```

DEMO

- Custom Validated Form Sample

Reactive: Working with nested forms

- Nested forms allow you to structure complex data models by grouping related controls inside FormGroup objects.
- A nested FormGroup is simply a FormGroup inside another FormGroup.
- Each level of nesting mirrors the structure of the final data object.

DEMO

- Nested Form Sample
- Nested Form Advanced Sample

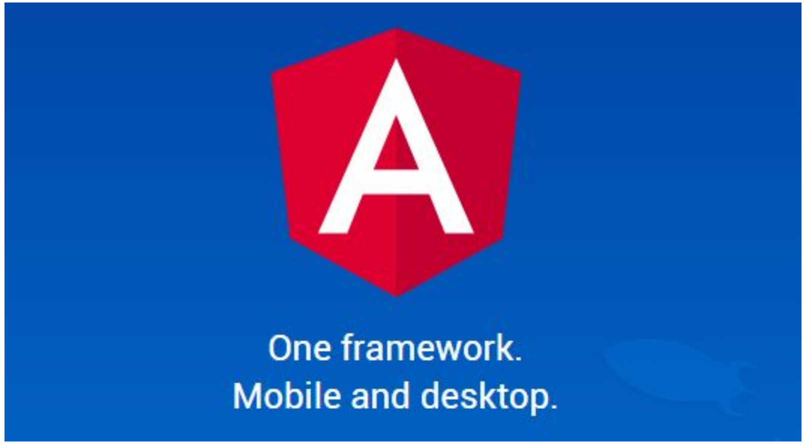
Exercise 7

Create a sign up page.

The form should contains:

- email
- password (at least 6 char with a special character and a number)
- confirm password (equal to password)
- first name (required)
- last name
- a section for address information like:
    - street
    - number
    - postal code
    - city
- Checkbox like “I agree to the terms and conditions” (required)
- A reset button
- A sign up button

In the address section, every input field is required. If the section is invalid, apply a red border to the entire section as well as to the specific invalid inputs.



Angular

COMPONENT LIFECYCLE

Component Lifecycle

- A Component has a lifecycle managed by Angular itself. Angular creates it, renders it, creates and renders its children, checks it when its data-bound properties change, and destroys it before removing it from the DOM.

- Angular offers Lifecycle hooks that give us visibility into these key moments and the ability to act when they occur.

The Lifecycle Hooks

- Directive and component instances have a lifecycle as Angular creates, updates, and destroys them.

- Developers can tap into key moments in that lifecycle by implementing one or more of the "Lifecycle Hook" methods in the component.

Directives and Components

Hook	Purpose
ngOnInit	Initialize the directive/component after Angular initializes the data-bound input properties.
ngOnChanges	Respond after Angular sets a data-bound input property. The method receives achanges object of current and previous values.
ngDoCheck	Detect and act upon changes that Angular can't or won't detect on its own. Called every change detection run.
ngOnDestroy	Cleanup just before Angular destroys the directive/component. Unsubscribe observables and detach event handlers to avoid memory leaks.

Components Only

Hook	Purpose
ngAfterContentInit	After Angular projects external content into its view.
ngAfterContentChecked	After Angular checks the bindings of the external content that it projected into its view.
ngAfterViewInit	After Angular creates the component's view(s).
ngAfterViewChecked	After Angular checks the bindings of the component's view(s).

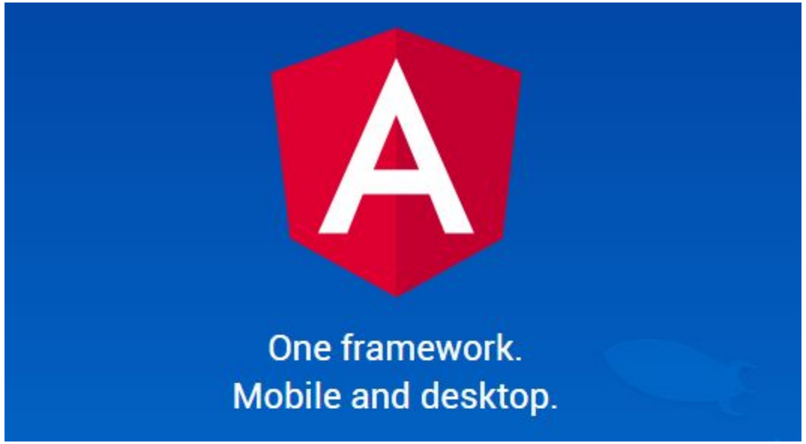
LifeCycle methods calling order

- **ngOnChanges** – called when an input or output binding value changes
- **ngOnInit** – after the first ngOnChanges
- **ngDoCheck** – developer's custom change detection
- **ngAfterContentInit** – after component content initialized
- **ngAfterContentChecked** – after every check of component content
- **ngAfterViewInit** – after component's view(s) are initialized
- **ngAfterViewChecked** – after every check of a component's view(s)
- **ngOnDestroy** – just before the directive is destroyed.

DEMO

Life Cycle Sample

bit Time professionals
consulenza | formazione | sviluppo



Angular

DIRECTIVES

What are Directives

Directives are **“enhancements”** for elements (built-in or components).

Example: `<input name=”title” ngModel />`

They can change the configuration (properties, attributes), styling or behaviour of elements.

Unlike Components, Directives have no template!

In other words: Components are Directives with a template.

Directives: One-Way Data Binding

- Data flows from the parent component → directive.
- The directive does not modify the parent's value.
- Useful for:
    - Configuring behaviors
    - Applying styling based on data
    - Reacting to external changes

Directives: Two-Way Data Binding

- Data flows in both directions:
    - Parent → child (input)
    - Child → parent (output)
- Allows a component or directive to update the parent's state.
- Implemented with the pattern:

```
@Input() value;
@Output() valueChange = new EventEmitter();
```

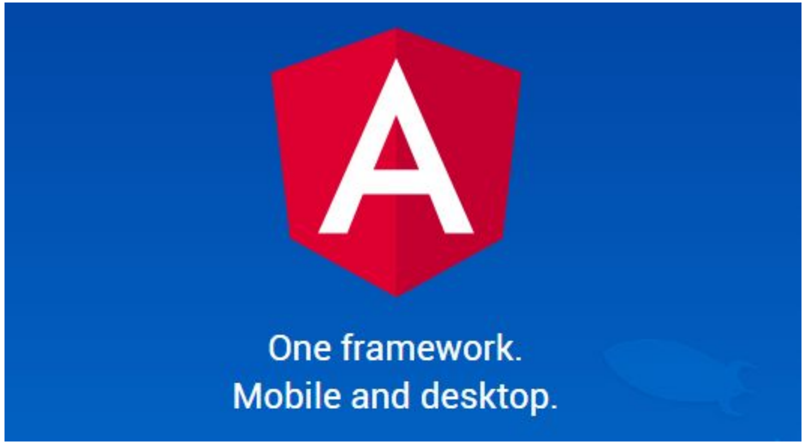
DEMO

Directives Sample

Exercise 8

Create an attribute directive called appHighlightEmpty that highlights an input field in red when it is empty.

- The directive must be applicable to <input> elements.
- If the input is empty → apply a red border.
- If the input contains text → restore the normal border.
- It must react live as the user types.



Angular

SERVICES

bit Time professionals
consulenza | formazione | sviluppo

Services

- Multiple components will need access to data and we don't want to copy and paste the same code over and over.
- Instead, we'll create a single reusable data service and learn to inject it in the components that need it.
- Refactoring data access to a separate service keeps the component lean and focused on supporting the view.

Service example

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

@Injectable()
export class SimpleService {
  constructor() { }
  public sum(a: number, b: number) {
    return a + b;
  }
}
```

What is a Service

A service is an **Injectable** class.

How to provide a service

Get it in the constructor, like this:

```
constructor(serviceName: ServiceClass) {}
```

This type serves as the so-called “Injection Token” which is used by angular to identify the “thing” (e.g. the service) it should create & inject.

How to provide a service

Or inject in the component like this:

```
serviceName = inject(ServiceClass)
```

How to provide a service

You don’t create service instances yourself – instead, you **request them from angular**

Element Injector

In a component you can specify an array of providers that this component can use.

The service is injectable only in the component and in his child component.

Instance one service for component instance.

Provide in root

ProvidedIn: “root” registers an injectable “thing”(e.g. a service) with the **“Application Root Environment Injector”**.

Therefore, ALL components, directives, services etc. can request the value (e.g. an instance of this service).

You can also specify this in **bootstrapApplication**, but it leads to a not optimized build because angular can not discard the “thing” if it is not used in the code.

DEMO

Tasks

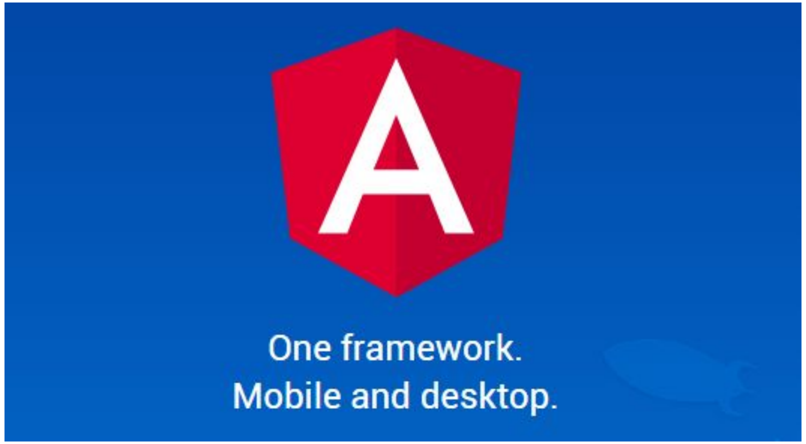
Exercise 9

Create an Angular service (ContactBookService) that manages all contact data and state for the application. The service must:

- Store a list of contacts (id, name, phone, email).
- Provide methods to add a new contact and select a contact for details.
- Maintain a text filter and expose a filteredContacts computed value.

Act as the single source of truth: all components must read and update data only through the service.
Implement the following components:

- ContactList — displays filteredContacts and allows selecting a contact.
- ContactDetail — shows details of the selected contact.
- ContactForm — adds new contacts via the service.



Angular

HTTP Client

bit Time professionals
consulenza | formazione | sviluppo

Getting and Saving Data with HTTP

Most front-end applications need to communicate with a server over the HTTP protocol, to download or upload data and access other back-end services. Angular provides a client HTTP API for Angular applications, the HttpClient service class in @angular/common/http.

Setting Up HttpClient

Before you can use HttpClient in your app, you must configure it using dependency injection.
With: `provideHttpClient()` in app configuration

Injecting HttpClient

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

export class UsersService {
  constructor(private http: HttpClient) {}
}
```

- Injected like any Angular service

HttpClient: Requests

```
getUsers() {  return this.http.get<User[]>('https://api.example.com/users'); }
createUser() {  return this.http.post<User>('https://api.example.com/users'); }
updateUser() {  return this.http.put<User>('https://api.example.com/users/${id}'); }
deleteUser() {  return this.http.delete<void>('https://api.example.com/users/${id}'); }
```

DEMO

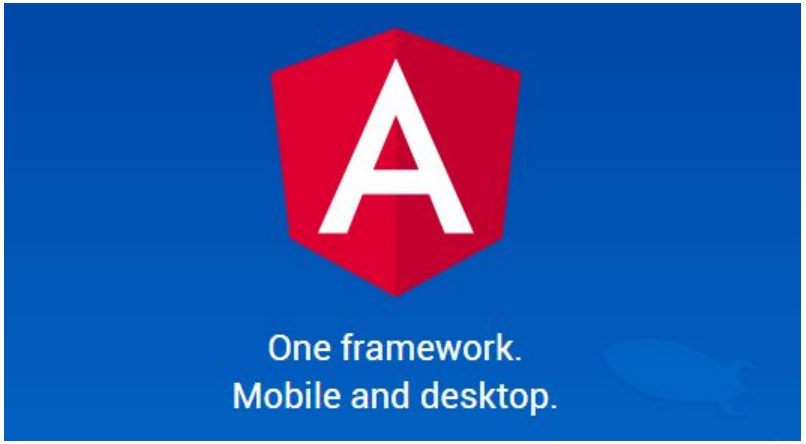
Customers List

Exercise 10

Make a customer list with a search input for filtering.

Separate service and component.

use GET /customers?name=fts.${value}



Angular

RxJS

Angular is Reactive by Design

Angular’s architecture is built around async events, data streams, and change detection.
Observables are the standard way to:

- handle asynchronous operations
- listen to continuous data updates
- react to events over time

RxJS = Reactive Extensions for JavaScript → a toolbox for reactive programming

What Problem Do Observables Solve?

Applications are full of asynchronous behavior:

- HTTP requests
- User interactions
- WebSocket streams
- Timers
- Form value changes
- Route and query param updates

Observables allow you to treat all of these as a single, unified abstraction: a stream of values over time.

Observable vs Promise

Promise	**Observable**
Emits one value	Can emit many values
Not cancellable	Cancellable
Eager (runs immediately)	Lazy (runs when subscribed)
Simpler API	Rich API (map, filter, debounce...)

Where Angular Uses Observables

Angular uses Observables internally for many features:

- HttpClient.get() returns an Observable
- FormControl.valueChanges returns an Observable
- EventEmitter uses Observables under the hood
- AsyncPipe is designed to work with Observables

Observables allow you to treat all of these as a single, unified abstraction: a stream of values over time.

Core Benefits of RxJS

RxJS gives us powerful tools:

- map → transform data
- filter → ignore unwanted values
- debounceTime → avoid spamming the server
- takeUntil → unsubscribe automatically

Helps build cleaner, safer, and more declarative async code.

Rules

Every time .subscribe() is called manually, you are responsible of unsubscription.

This is intended for valueChanges, HTTP polling, WebSocket, EventEmitter and every other Observable.

Observable Lifecycle

Observable flow:

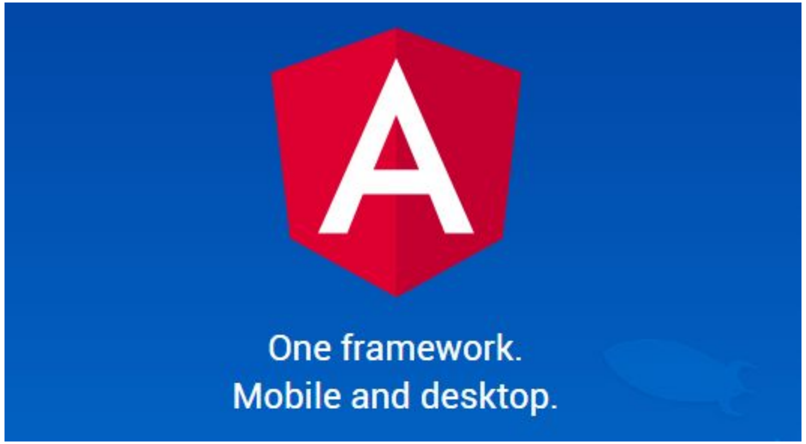
- Create a stream
- Subscribe to listen
- Transform with operators
- Handle next / error / complete
- Unsubscribe (important!)

Angular simplifies this with:

- async pipe
- takeUntilDestroyed()
- takeUntil()

DEMO

Observable Sample



Angular

Interceptors

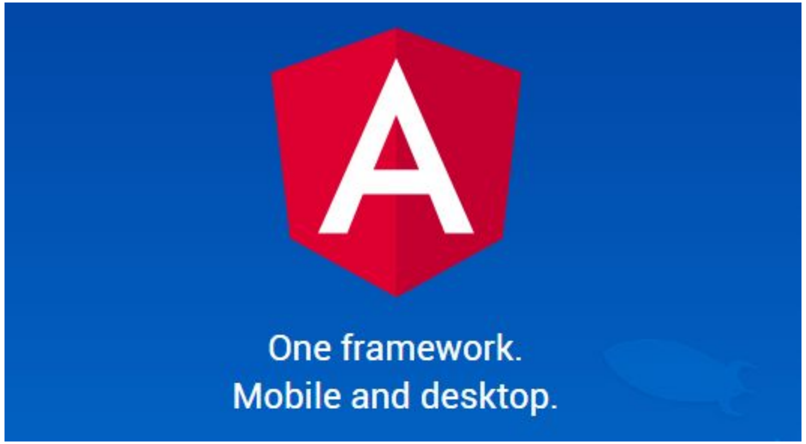
bit Time professionals
consulenza | formazione | sviluppo

What Are Http Interceptors?

Special functions that modify requests/responses globally, Common uses:

- Add auth tokens
- Log requests
- Retry failed calls
- Transform responses

Interceptors are run in the injection context of the injector which registered them, and can use Angular's inject API to retrieve dependencies.



ROUTER

Angular Router

- The Angular Router can interpret a browser URL and enables navigation from one view to the next as users perform application tasks. It can pass optional parameters along to the supporting view component.
- You can navigate imperatively when the user clicks a button, selects from a drop box, or in response to some other stimulus from any source.
- And the router logs activity in the browser's history journal so the back and forward buttons work as well.

Angular Router: How to configure

- Routing is now configured with provideRouter(routes)
- No more RouterModule imports in NgModules
- Defining Routes

Example route configuration:

```
export const routes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutComponent },
];
```

Navigate to another route in the template

```
// { path: 'articles', component: ArticlesListComponent }

<a href="#" [routerLink]="['/articles']">
  Articles
</a>

// { path: 'articles/:id', component: ArticleEditComponent}
<a href="#" [routerLink]="['/articles', 1]">
  Article #1
</a>
```

Navigate to another route by code

```
...
constructor(private router: Router) { }

backToList () {
    this.router.navigate(['/articles']);
}
...
```

bit Time professionals
consulenza | formazione | sviluppo

Angular Router: How to configure

- Routing is now configured with provideRouter(routes)
- No more RouterModule imports in NgModules
- Defining Routes

Example route configuration:

```
export const routes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutComponent },
];
```

Reading router parameters

```
//if route is defined as follows...
//{ path: 'articles/:id', component: ArticleEditComponent },

constructor(private route: ActivatedRoute) {}

ngOnInit() {
  const id = this.route.snapshot.paramMap.get('id')
  // or
  this.route.paramMap.subscribe(params => {
  this.id = params.get('id');
});

}
```

Defining Child Routes

- When some routes may only be accessible and viewed within other routes it may be appropriate to create them as child routes.

Defining Child Routes: Example

- The product details page may have a tabbed navigation section that shows the product overview by default. When the user clicks the "Technical Specs" tab the section shows the specs instead.
- If the user clicks on the product with ID 3, we want to show the product details page with the overview:

```
localhost:4200/product-details/3/overview
```

- When the user clicks "Technical Specs":

```
localhost:4200/product-details/3/specs
```

- Overview and specs are child routes of `product-details/:id`
- They are only reachable within product details.

Child Routes Example

```
export const routes: Routes = [
  { path: '', redirectTo: 'product-list', pathMatch: 'full' },
  { path: 'product-list', component: ProductList },
  { path: 'product-details/:id', component: ProductDetails,
    children: [
      { path: '', redirectTo: 'overview', pathMatch: 'full' },
      { path: 'overview', component: Overview },
      { path: 'specs', component: Specs }
    ]
  }
];
```

DEMO

- "routerSample"
  - Functionality1
  - Functionality2