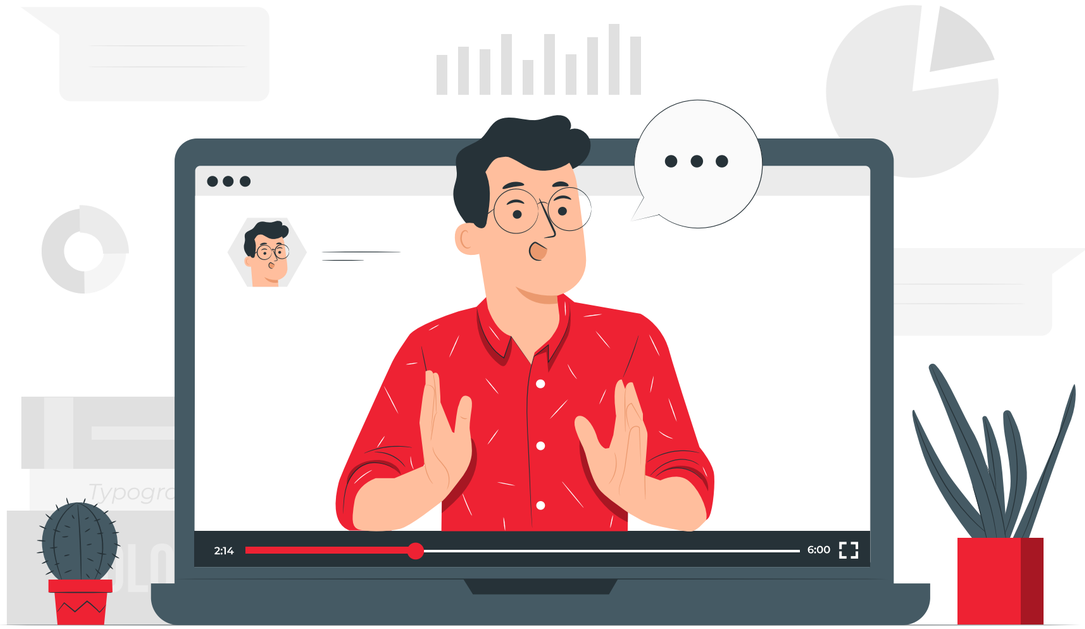
# Angular course

# Table of Contents

# Logistics

- Schedules
- Lunch & breaks
- Other questions?

# About me



# Getting started

# Table of Contents

# Client-server architecture

- Refers to a mode of **communication** between two computers
  - → the client sends a **request** to the server
  - → the server sends the **response** back to the client
- This communication generally uses the HTTP **protocol** (but other protocols exist…)
- Each HTTP **message** between the client and the server generally consists of two parts
  - → the **headers**, which contain contextual metadata
  - → the **body**, which contains the transmitted data

# Client-server architecture - Example

- **headers** of a document request, sent by the client (a web browser)

```
GET /home HTTP/1.1
Accept: text/html
Accept-Encoding: gzip
```

- **body** of the response, returned by the server (a web page)

```
<!doctype html>
<html lang="en">
  <head>
    <title>Home</title>
    <link href="styles.css" rel="stylesheet" />
  </head>
  <body>
    <app-root></app-root>
    <script src="main.js"></script>
  </body>
</html>
```

😉 *This is not a random example; it's actually the typical server return for an Angular application*

# Web browser technologies

- A web browser is software capable of displaying **web pages**
- Web pages are built around 3 main technologies
  - → **HTML**
  - → **CSS**
  - → **JavaScript**

# HTML - HyperText Markup Language

- HTML is a **HyperText Markup Language** used to **structure the content** of web pages

```
<h1>Google Chrome is a web browser</h1>

<p> <a href="https://www.google.com/chrome/">More info</a> </p>

<img src="https://www.google.com/chrome/static/images/chrome-logo-m100.svg" />
```

- Tag syntax
  - → **opening** `<tag>` and **closing** `</tag>` tags (such as **`h1`**, **`p`**, **`a`**, ...) with content in between
  - → **self-closing** `<tag />` tags (such as **`img`**, ...) with no content
  - → **attributes** `attribute-name="value"` (such as **`href`**, **`src`**, ...) applicable to opening and self-closing tags
  - → the **content** (between the opening and closing tags) may contain other *nested tags*

# CSS - Cascading Style Sheets 

- CSS is a **rules-based language** used to control the visual **formatting** of web pages

```
<button>Valider</button>

<style>
  button {
    padding: 15px;
    background-color: yellow;
  }
</style>
```

- Syntax of **rules**
  - → **selector** targeting one or more elements of the web page: `selector { ... }`
  - → **declarations** applying to this selector: `property: value;`
- A style sheet can be defined in a `<style>` tag, or in an external file
  - → `<link href="styles.css" rel="stylesheet" />`

# JavaScript JS

- JavaScript is a **scripting language** used to add **interactivity** to web pages

```
<button onclick="showAlert()">Valider</button>

<script>
  function showAlert() {
    window.alert('Button clicked!');
  }
</script>
```

- A script can be defined in a `<script>` tag, or in an external file

  → `<script src="main.js"></script>`

# HTML - CSS - JavaScript

- All 3 technologies are indeed present in the web page given above as an example
  - → **HTML:** all the tags in the document
  - → **CSS:** loaded by the `styles.css` file
  - → **JavaScript:** loaded by the `main.js` file

```
<!doctype html>
<html lang="en">
  <head>
    <title>Home</title>
    <link href="styles.css" rel="stylesheet" />
  </head>
  <body>
    <app-root></app-root>
    <script src="main.js"></script>
  </body>
</html>
```

😉 *Later, we'll explain the role of the `<app-root>` tag in relation to **Angular**...*

# Technologies outside the web browser

- Ultimately, an Angular application **runs** in a web browser
- The artefacts of such an application are therefore HTML, CSS and JavaScript files, which the browser knows how to interpret
- However, an Angular application is **built** using additional technologies (not understood by the browser), which improve the developer experience and the quality of the artefacts
- These technologies, used only during the development phase, are mainly
  - → **TypeScript**
  - → **Node.js**
  - → **NPM**
  - → **Vite**

# TypeScript 

- TypeScript is a **superset** of JavaScript, which improves and secures the production of JavaScript code
- Unlike JavaScript, TypeScript is a **typed programming language**

```
// JavaScript
let data;          // There is no constraint on the possible values
data = 1;          // ✅ Here it's a `number`
data = true;       // ✅ And here it's a `boolean`
```

```
// TypeScript
let data: number; // Only values of type `number` are allowed
data = 1;          // ✅ Here the assignment is valid
data = true;       // ❌ And here the assignment is invalid
```

# TypeScript 


- A TypeScript program must be **transpiled into JavaScript** before it can be executed in the web browser
- Transpilation simply involves **removing the typing** to make it a valid JavaScript program
- TypeScript is used in the **development phase** whereas JavaScript is used in the **execution phase**

# Node.js

- Node.js is a technology that allows JavaScript code to be executed **outside the browser**
- With Node.js, the **execution context** for JavaScript is your **operating system**
- Node.js can, for example, access your file system, find out the characteristics of your processor, etc...

```
# Running the following commands in your computer's Terminal...
node
process.arch # ...returns for example: `x64` (Intel 64-bit processor)
```

*In a web browser, on the other hand, JavaScript's execution context is the web page with which it interacts. JavaScript can, for example, know the user's preferred language, the size of the browser window, etc...*

```
# Running the following command in your browser's console...
window.innerWidth # ...returns for example: `1135` (window width in px)
```

# NPM (Node package Manager)

- NPM is the **package manager** for the Node.js JavaScript runtime environment
- Provides programs and libraries for the JavaScript ecosystem in the form of downloadable packages from a **registry**
- Example of installing a package and then using it

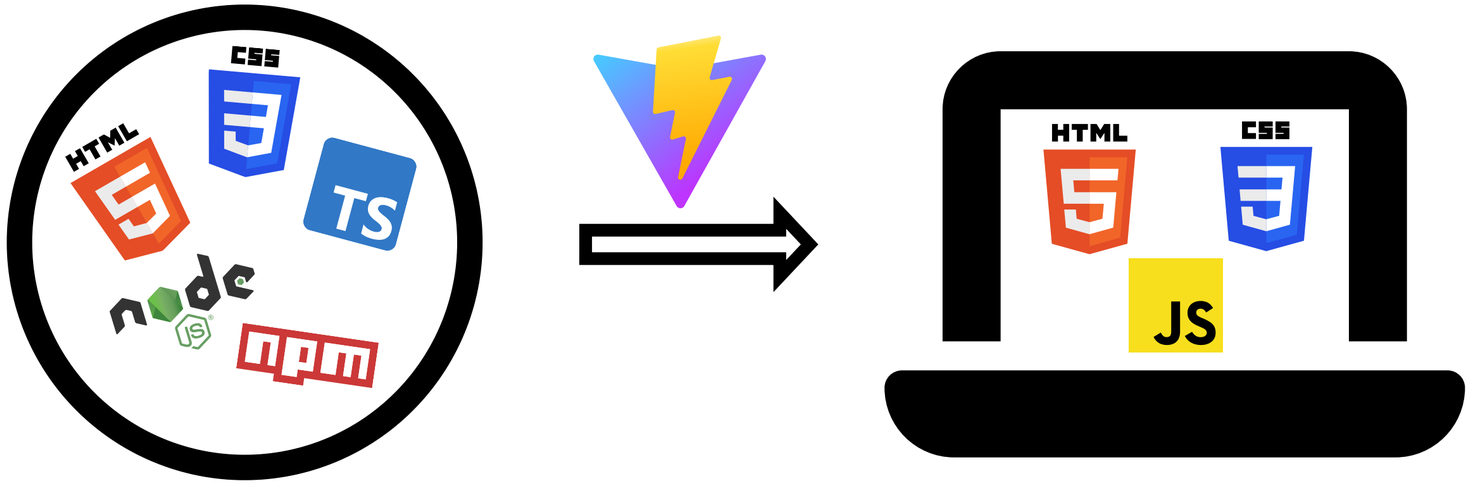
```
# Running the following command in a Terminal,
# will install the `@angular/cli` package globally on your computer
npm install --global @angular/cli

# Once the package has been installed, it globally provides the `ng` command,
# which for example, lets you generate an Angular application skeleton
ng new
```

😉 *Note: within a Node.js project folder, installing a package without the `--global` option, will install it locally for the project.*

# Vite

- Vite is a **build tool** for modern web applications
- Main features
  - → **development server** (dev server)
  - → **build artefacts command** (bundler)



# Angular 

- A web framework that enables developers to **create fast, reliable applications**
- Announced in 2014, it's a total rewrite of **AngularJS** (although some concepts remain)
- First release of **Angular 2** in September 2016
- Major release every 6 months
- Last major version 21 released in November 2025
- Maintained by a dedicated team at **Google**

# Angular - The big picture 1/2

- In the **development phase**, you write components in TypeScript
  - → Angular has a component-based architecture
  - → and use plain HTML templates

```
import { Component } from '@angular/core';
import { bootstrapApplication } from '@angular/platform-browser';

@Component({
  selector: 'app-root',              // --> CSS selector
  template: '<p>Hello world!</p>',   // --> HTML template
})
class App {}                         // --> JavaScript class (supercharged by its decorator's metadata)

bootstrapApplication(App);
```

*(for the moment, let's leave aside the code implementation details and focus on the big picture...)*

# Angular - The big picture 2/2

- In the **execution phase** (once the app has been built and is running in a web browser), Angular tries to **bootstrap** the application

  → it searches the web page for the tag corresponding to the component's **CSS selector**

  → it then renders the component's **HTML template** inside this tag

```
<app-root>
  <p>Hello world!</p>
</app-root>
```

😉 *You now know the role of the `<app-root>` tag in relation to **Angular**, which was present in the web page given above as an example*

# In-depth resources

- **HTML - CSS - JavaScript:** https://developer.mozilla.org
- **TypeScript:** https://www.typescriptlang.org
- **Node.js:** https://nodejs.org
- **NPM:** https://npmjs.com
- **Vite:** https://vitejs.dev
- **Angular:** https://angular.dev

# Getting started - Summary

**In this chapter on getting started, we have covered the following topics**

- Client-server architecture
- HTML
- CSS
- JavaScript
- TypeScript
- Node.js
- NPM
- Vite
- Angular

# Getting started - Questions

# Getting started - Lab 1

# Workspace

# Table of Contents

# Workspace

An Angular workspace is **structured** by the following elements

- `package.json`
- `tsconfig.json`
- `angular.json`
- `src/*`

# Workspace - package.json 1/5

The presence of a **`package.json`** file indicates that the directory is the root of a **Node.js** project

- **Dependencies** of the Angular framework are scoped under **`@angular/*`**
- Angular also depends on some **third-party libraries**

```
// --> @angular/* dependencies
{
  "dependencies": {
    "@angular/common": "...",
    "@angular/compiler": "...",
    "@angular/core": "...",
    "@angular/forms": "...",
    "@angular/platform-browser": "...",
    "@angular/router": "..."
  },
  "devDependencies": {
    "@angular/build": "...",
    "@angular/cli": "...",
    "@angular/compiler-cli": "...",
  }
}
```

```
// --> third-party libraries
{
  "dependencies": {
    "rxjs": "...",
    "tslib": "...",
  },
  "devDependencies": {
    "typescript": "...",
    "vitest": "..."
  }
}
```

# Workspace - package.json 2/5

Package versions generally follow the semver (semantic versioning) standard...

```
// Contents of the `package.json` file of the <packageName> package
{
  "name": "<packageName>",
  "version": "<major>.<minor>.<patch>"
}
```

- **`major`**: might introduce breaking changes
- **`minor`**: can add new features but in a retro-compatible way
- **`patch`**: bug fixes

# Workspace - package.json 3/5

...allowing a range of versions when **installing** or **updating** a package

- `1.2.3` will install the **exact** version
- `~1.2.3` will install any **patch** update such as
  - → `1.2.4`
  - → `1.2.5`
  - → `1.2.99`
- `^1.2.3` will install any **minor** update such as
  - → `1.2.3`
  - → `1.3.0`
  - → `1.99.0`

For a given dependency, the exact version installed is locked in the `package-lock.json` configuration file

# Workspace - package.json 4/5

- Dependencies are typically downloaded from the NPM registry and installed locally in `./node_modules` directory
- You can manage any dependency of your project using NPM commands

```
npm install <packageName>

npm update <packageName>

npm remove <packageName>
```

- However, to update the Angular framework dependencies, use the dedicated command

```
ng update
```

**In-depth resources:**

- Angular update guide
- Angular version compatibility (with Node.js, TypeScript, ...)

# Workspace - package.json 5/5

- The **`package.json`** file also contains scripts...

```
{
  "scripts": {
    "ng": "ng",
    "start": "ng serve",
    "build": "ng build",
    "watch": "ng build --watch --configuration development",
    "test": "ng test"
  }
}
```

- ...that can be run using the shell command

```
npm run <scriptName>

# Build your Angular app
npm run build
```

# Workspace - tsconfig.json

The presence of a `tsconfig.json` file indicates that the directory is the root of a **TypeScript** project

- Specifies the root files and the **compiler options** required to compile the project
- Supplies **Angular specific options** to the compiler

```
{
  "compilerOptions": {
    "strict": true,
    "experimentalDecorators": true,
    ...
  },
  "angularCompilerOptions": {
    "strictInputAccessModifiers": true,
    "strictTemplates": true,
    ...
  }
}
```

# Workspace - angular.json 1/2

The presence of an **`angular.json`** file indicates that the directory is the root of an **Angular** project

- Provides workspace-wide and project-specific **configuration** defaults
- These are used for build and development tools provided by the **Angular CLI**

```
{
  "projects": {
    "zenika-ng-website": {
      "sourceRoot": "src",
      "projectType": "application",
      "prefix": "app",
      "architect": {
        "build": {},
        "serve": {},
        "test": {},
      }
    }
  }
}
```

# Workspace - angular.json 2/2

- The build **`"options"`** in the architect section are frequently used

```
{
  "projects": {
    "zenika-ng-website": {
      "architect": {
        "build": {
          "options": {
              "index": "src/index.html", // This is optional as it is the default value.
              "browser": "src/main.ts",
              "tsConfig": "tsconfig.app.json",
              "assets": [{ "glob": "**/*", "input": "public" }],
              "styles": ["src/styles.css"]
          }
        }
      }
    }
  }
}
```

# Workspace - src/*

- `index.html`: final **document** of the Single Page Application (SPA)
- `main.ts`: **entry point** of the app (from which Vite builds the JavaScript bundle)
- `app/app.*`: **main component** of the app (the one used to bootstrap the app)
- `styles.css`: **global styles** of the app
- `public/*`: **resources** of the app (images, pdf, ...)

When running the **`ng build`** shell command all these files are compiled and combined to produce the final application bundle ready for production (mainly HTML, CSS and JavaScript files)

```
ng build
```

When the build is complete, the application bundle is in the **`dist/`** directory

# Angular CLI

- The Angular CLI is a command-line interface tool that you use to
  - → **initialize**
  - → **develop**
  - → **scaffold**
  - → **maintain** applications
- It is usually installed globally on your system

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

- Here are some of the commands available

```
ng new my-app-name
ng serve
ng test
ng build
```

# Angular CLI - Generate 1/3

The **generate** (or simply g) command is often used to quickly scaffold the different parts of an Angular application

```
# Generate components
ng generate component menu
ng g c product

# Generate services
ng generate service catalog-resource
ng g s basket-resource

# Generate pipes
ng generate pipe sort-array

# And many more...
```

You can easily get help for each type of CLI command

```
ng --help
ng generate --help
ng generate component --help
```

# Angular CLI - Generate 2/3

- From Angular v2 to v19, all files generated by the CLI were suffixed with their type (`*.component.ts`, `*.directive.ts`, `*.service.ts`, ...)
- Starting with **Angular v20**, this is no longer the case, as the Angular guide style has been simplified

Here's the code generated by the command `ng generate component menu` in the two different implementations

```ts
/* Angular 2, ..., 18, 19 */

// menu.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-menu',
  templateUrl: './menu.component.html',
  styleUrl: './menu.component.css',
})
export class MenuComponent {}
```

```ts
/* Angular 20, ... */

// menu.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-menu',
  templateUrl: './menu.html',
  styleUrl: './menu.css',
})
export class Menu {}
```

😉 *This course is based on the latest style guide recommendations*

# Angular CLI - Generate 3/3

- When updating from Angular 19 to 20, the following configuration is added to the **`angular.json`** file to preserve the previous behavior

```
{
  "schematics": {
    "@schematics/angular:component":    { "type": "component"  },
    "@schematics/angular:directive":    { "type": "directive"  },
    "@schematics/angular:service":      { "type": "service"    },
    "@schematics/angular:guard":        { "typeSeparator": "." },
    "@schematics/angular:interceptor": { "typeSeparator": "." },
    "@schematics/angular:module":       { "typeSeparator": "." },
    "@schematics/angular:pipe":         { "typeSeparator": "." },
    "@schematics/angular:resolver":     { "typeSeparator": "." }
  }
}
```

# Workspace - Summary

**In this chapter on workspace, we have covered the following topics**

- package.json
- tsconfig.json
- angular.json
- src/*
- Angular CLI

# Workspace - Questions

# Workspace - Lab 2

# TypeScript

*— prerequisites —*

# Table of Contents

# TypeScript - Types 1/3

💡 *This chapter covers the TypeScript prerequisites needed to confidently follow the rest of the training*

- Type examples: **`boolean`**, **`number`**

```
const alwaysTrue: boolean = true;

let age: number = 32;
age = 33;
age = 'John';              // ❌ Type 'string' is not assignable to type 'number'
```

- **Type inference:** used to provide type information when there is no explicit type annotation

```
const alwaysTrue = true;  // is still of type `boolean`

let age = 32;             // is still of type `number`
age = 33;
age = 'John';             // ❌ is still throwing the same Type error
```

😉 *Note that **`const`** and **`let`** are two different ways of defining variables*

# TypeScript - Types 2/3

- More type examples: `string`, `template string`, `array`, `object`

```
const name: string = 'John';

const hello: string = `Hello ${name}!`;

const nameList: string[] = ['John', 'Jane'];

const products: { title: string; price: number } = { title: 'Tee-shirt', price: 8.5 };
```

# TypeScript - Types 3/3

- Type **any** may be necessary in some cases, but **should be avoided** wherever possible...

```
let notSure: any = 4;
notSure = true;
```

- ...instead, use type **unknown** whenever possible

```
let x: unknown = 1;

if (typeof x === 'number') {
  x = x * 2;                    // <-- In this scope, TypeScript infers that `x` is of type `number`
}

if (typeof x === 'string') {
  x = x + '...';                // <-- In this scope, TypeScript infers that `x` is of type `string`
}
```

# TypeScript - Functions 1/3

- **Function declaration** (or statement)

```
function clickHandler() {
  console.log('Clicked!');
}

document.addEventListener('click', clickHandler);

clickHandler(); // ✅ The function has been declared and can therefore be referenced
```

- **Function expression**

```
document.addEventListener('click', function clickHandler() {  // <-- can be a "named" function...
  console.log('Clicked!');
});

clickHandler(); // ❌ Uncaught ReferenceError: `clickHandler` is not defined

document.addEventListener('click', function() {               // <-- ...or even an "anonymous" function
  console.log('Clicked!');
});
```

# TypeScript - Functions 2/3

- **Arrow function expression**

```
document.addEventListener('click', () => {                    // <-- is always "anonymous" function
  console.log('Clicked!');
});
```

⚠️ *Note that arrow functions do not treat the keyword* ***`this`*** *in the same way as functions defined with the keyword* ***`function`****, but this is beyond the scope of this course*

- **In-depth resource:** https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions

# TypeScript - Functions 3/3

- TypeScript allows typing for function arguments and return value
  - → Set **default** argument value with "="
  - → Define **optional** argument with "**?**"
  - → Use "**`return`**" keyword to return a value

```
function getFullName(lastName = 'Doe', firstName?: string): string {
  return firstName ? `${firstName} ${lastName}` : lastName;
}
```

- Arrow functions can be used without "**`return`**" keyword and still return a value

```
const sayHello = (name: string) => {
  return `Hello ${name}!`;
}

const sayHello = (name: string) => `Hello ${name}!`;    // <-- Same as above, but shorter!
```

# TypeScript - Destructuring syntax

Makes it possible to unpack values from **arrays**, or properties from **objects**, into distinct variables

- Destructuring array

```
const [a, b, ...rest] = [10, 20, 30, 40];

// a == 10
// b == 20
// rest == [30, 40]
```

- Destructuring object

```
const { a, b, ...rest } = { a: 10, b: 20, c: 30, d: 40 };

// a == 10
// b == 20
// rest == { c: 30, d: 40 }
```

# TypeScript - Spread syntax

- "Expands" an **array** or **object** into its elements
- In a way, spread syntax is the **opposite of rest syntax** (that we saw just above)

```
const sum = (a: number, b: number) => a + b;
sum(1, 2);                                  // <-- 3

const args = [1, 2];
sum(...args);                               // <-- 3
```

```
const arr = [1, 2, 3];
const arrCopy = [...arr];

console.log(arr !== arrCopy);               // <-- true
```

```
const obj = { a: 1, b: 2 };
const objCopy = { ...obj };

console.log(obj !== objCopy);               // <-- true
```

# TypeScript - Array instance methods

Arrays can be manipulated using methods such as the following

- Some methods are **destructive**...

```
[2, 0, 3, 1].sort();                                    // --> [0, 1, 2, 3]

[2, 0, 3, 1].sort((a, b) => b - a);                     // --> [3, 2, 1, 0]
```

- ...while others are **non-destructive**

```
[0, 1, 2, 3].map((value) => value * 10);                // --> [0, 10, 20, 30]

[0, 1, 2, 3].filter((value) => value % 2);              // --> [1, 3]

[0, 1, 2, 3].reduce((sum, value) => sum + value, 0);    // --> 6
```

# TypeScript - Adding item to an Array

There are two ways of adding an element to an array

- In a **mutable** way...

```
const items = [0, 1, 2, 3];

items.push(4);

console.log(items);                                    // --> [0, 1, 2, 3, 4]
```

- ...in an **immutable** way

```
const items = [0, 1, 2, 3];

const newItems = [...items, 4];

console.log(newItems);                                 // --> [0, 1, 2, 3, 4]
```

😉 *Treating a value as immutable means prohibiting any modification to it after its creation. Immutability is an important concept in functional programming and state management.*

# TypeScript - Classes 1/4

*Classes and interfaces are similar to those in Object Oriented Programming (OOP)*

- Classes are composed of one **constructor**, **properties** and **methods**
- Explicitly defining a constructor is optional
- Properties and methods are accessible with the `this` keyword

```
class Person {
  name = '';

  constructor() {} // this is optional

  sayHello() {
    console.log(`Hello, I'm ${this.name}!`);
  }
}

const person = new Person();
person.name = 'John';
person.sayHello(); // --> Hello, I'm John!
```

# TypeScript - Classes 2/4

- 3 scopes for encapsulation: **`public`**, **`protected`** and **`private`**
  - → **`public`** is the default scope
  - → private scope alternative: using standard JavaScript private field (with the hash prefix #)

```
class Demo {
  prop1 = 1;
  protected prop2 = true;
  private prop3 = 'Secret';

  #prop4 = 'Big secret'; // <-- standard JavaScript private field

  method1() {}
  protected method2() {}
  private method3() {}

  #method4() {} // <-- standard JavaScript private field
}
```

# TypeScript - Classes 3/4

- Possibility to have "**getter**" and "**setter**"

```
class Person {
  public firstName: string;
  public lastName: string;

  constructor(firstName: string, lastName: string) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  get fullName(): string {
    return `${this.firstName} ${this.lastName}`;
  }

  set fullName(value: string) {
    const [firstName, lastName] = value.split(' ');
    this.firstName = firstName;
    this.lastName = lastName;
  }
}

// ...
```

# TypeScript - Classes 4/4

```
// ...

const person = new Person('John', 'Doe');

// Calling the "getter" function
console.log(person.fullName);       // --> John Doe

// Calling the "setter" function
person.fullName = 'Jean Dupont';

console.log(person.firstName);      // --> Jean
console.log(person.lastName);       // --> Dupont
```

# TypeScript - Interfaces

- Can be used to define object shape

```
interface Person {
  name: string;
  age: number;
}

const person: Person = { name: 'John Doe', age: 33 };
```

- Can be used on classes with the **`implements`** keyword

```
interface Musician {
  play(): void;
}

class TrumpetPlayer implements Musician {
  play(): void {
    console.log('I play trumpet!');
  }
}
```

# TypeScript - Generics

- Similar to generics in *Java* or *C#*
- Generics need typing at instantiation

```
class Log<T> {
  log(value: T) {
    console.log(value);
  }
}

const logOfNumber = new Log<number>();
logOfNumber.log(5);
logOfNumber.log(6);


const logOfString = new Log<string>();
logOfString.log('Hello');
logOfString.log('world!');
```

# TypeScript - Decorators

- A Decorator is a special kind of declaration that can be attached to a class declaration, method, accessor, property, or parameter
- Decorators can be used to **observe**, **modify** or **replace** the value to which they are applied
- Decorators use the form `@expression`, where `expression` must evaluate to a **function** that will be **called at runtime** with information about the decorated declaration

```
function MyClassDecorator(/* ... */) { /* ... */ }

function MyMethodDecorator(/* ... */) { /* ... */ }

@MyClassDecorator
class Foo {

  @MyMethodDecorator
  bar() {}
}
```

# TypeScript - Summary

**In this chapter on TypeScript, we have covered the following topics**

- Types
- Functions
- Destructuring syntax
- Spread syntax
- Array instance methods
- Classes
- Interfaces
- Generics
- Decorators

# TypeScript - Questions

# Components

# Table of Contents

# Components - Definition 1/3

- Components are the **main building blocks** of Angular applications
- Each component represents a **part of a larger web page**
- Organizing an application into components helps **provide structure to your project**, clearly separating code into specific parts that are easy to maintain and grow over time

# Components - Definition 2/3

- Defined with the **`@Component`** class decorator, which provides the component's metadata
  - → must have a **`selector`** so that it can be inserted into any other component template
  - → must have a **`template`** (or **`templateUrl`**) that defines what is to be displayed

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

@Component({
  selector: 'app-hello',
  template: '<p>Hello world!</p>',
})
export class Hello {}
```

# Components - Definition 3/3

- You build an application by composing multiple components together
- A component that depends on other components must import them in order to use them in its template

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

@Component({
  selector: 'app-root',
  imports: [Hello],
  template: `
    <h1>My Awesome App</h1>
    <app-hello />
  `,
})
export class App {}
```

😉 *The **App** component is the only one to be **bootstrapped**. Its selector (**`app-root`**) is searched for in the web page. All other components are necessarily **imported** by the main component or its children.*

# Components - Template

- The template can be configured in two ways:
  - → using a **`template`** property: string literal (as shown above)
  - → using a **`templateUrl`** property: path to an HTML file (relative to the component)

```ts
// app.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.html',
})
export class App {}
```

```html
<!-- app.html -->

<h1>My Awesome App</h1>
```

# Components - Styles

The styles can be configured in two ways:

- using a **`styles`** property that contains the expected CSS rules

```
@Component ({
  styles: `h1 { font-weight: normal; }`
})
export class App {}
```

- using a **`styleUrl`** property that indicates a path to **`.css`** (or **`.scss`**) file

```
@Component ({
  styleUrl: './app.css'
})
export class App {}
```

```
/* app.css */

h1 { font-weight: normal; }
```

# Template syntax - Text interpolation

- Uses the syntax `{{ expression }}`
- The **`expression`** is converted into a **`string`** and displayed as such
- Angular defines a precise syntax for these expressions
  - → accepts basic JavaScript expressions
  - → more: https://angular.dev/guide/templates
- All **`public`** or **`protected`** component properties can be used in the template
- An expression used in the template must not change the component state

```
@Component ({
  selector: 'app-product-card',
  template: `<p>{{ product?.title }}</p>`
})
export class ProductCard {
  protected product?: Product;
}
```

# Template syntax - Property binding

- Generic syntax for setting the value of a **DOM property**
- Using the syntax `[propertyName]="expression"`

```
<button [disabled]="isUnchanged">Save</button>   <!-- HTML property -->

<app-checkout-form [formData]="data" />          <!-- Component input -->
```

- Angular provides a special syntax for applying dynamic `class` and `style` properties

```
<p [class.highlight]="isHighlight">Hello</p>

<button [style.color]="isHighlight ? 'orange': null">Save</button>
```

# Template syntax - Attribute binding

- Generic syntax for setting the value of an **HTML attribute**
- Using the syntax `[attr.attributeName]="expression"`
- Pay attention to the difference between "DOM properties" and "HTML attributes"!

Example: `role` is a valid HTML attribute of the `<div>` tag, but there's no such DOM property!

```
<div role="status">OK</div>
```

```
<div [attr.role]="expression">OK</div>
```

```
<div [role]="expression">NOT OK</div>

<!-- ❌ Can't bind to 'role' since it isn't a known property of 'div'. -->
```

# Template syntax - Event listeners

- Generic syntax for listening to an event of an HTML element
- Using the syntax `(eventName)="expression"`

```
<button (click)="handler()">Save</button>                    <!-- HTML event -->

<app-checkout-form (formSubmitted)="onFormSubmitted()" />    <!-- Component output -->
```

- Angular provides a special syntax for handling "pseudo" events

```
<input (keyup.enter)="onEnter()" />
```

# Template syntax - Event listeners | $event

- In this example, we listen to the **`input`** event of the **`<input />`** element

```
@Component ({
  selector: 'app-demo',
  template: `<input [value]="name" (input)="updateName($event.target)" />`,
})
export class Demo {
  name = 'John';

  updateName(eventTarget: HTMLInputElement) {
    this.name = eventTarget.value;
  }
}
```

- **`$event`** refers to the native browser DOM **`InputEvent`**
- We achieve a *two-way data binding* using both property binding and Event listeners

  → the **class** property **`name`** and the **template** input **`value`** will always be in sync

# Components - Mini lab

✋ *Given its length, this chapter has been divided into two parts*

💪 *Now it's time to put what you've learned into practice, with this mini lab*

---

- Create a **`Counter`** component
  - → it should display a **`count`** property starting at **`0`**
  - → the **`count`** can be incremented or decremented using HTML buttons
  - → the decrement button should be disabled when the **`count`** is equal to **`0`**
  - → the increment button should be disabled when the **`count`** is equal to **`5`**
- Import the created component into your application component's class
- Insert it into your application component's template

---

👇 *When you have finished, we will move on to the second part of the chapter...*

# Components - Input 1/4

- Use the **`input()`** function to declare a component class property as input
- It acts as a wrapper around the value
- To read the value contained in the input, you need to call it as a function

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

@Component ({
  selector: 'app-counter',
  template: `<p>{{ count() }}</p>`
})
export class Counter {
  count = input<number>(0);
}
```

- Inputs without a default value have an implicit **`undefined`** value

```
count = input<number>(); // is equivalent to `input<number | undefined>();`
```

# Components - Input 2/4

- The consumer of this component can optionally bind to the input in its template

```
import { Component } from '@angular/core';
import { Counter } from './counter/counter.ts';

@Component ({
  selector: 'app-root',
  imports: [Counter],
  template: `
    <app-counter />                                  <!-- rendering:  <p></p>  -->

    <app-counter [count]="parentCount" />            <!-- rendering:  <p>5</p> -->
  `
})
export class App {
  protected parentCount = 5;
}
```

# Components - Input 3/4

- Use the **`input.required()`** function to declare a component class property as required input

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

@Component ({
  selector: 'app-counter',
  template: `<p>{{ count() }}</p>`
})
export class Counter {
  count = input.required<number>();
}
```

# Components - Input 4/4

- The consumer of this component must bind to the required input in its template
- Angular will throw an error if the required input is missing

```
import { Component } from '@angular/core';
import { Counter } from './counter/counter.ts';

@Component ({
  selector: 'app-root',
  imports: [Counter],
  template: `
    <app-counter />     <!-- ❌ Required input 'count' from component Counter must be specified. -->

    <app-counter [count]="parentCount" />
  `
})
export class App {
  protected parentCount = 5;
}
```

# Components - Output 1/2

- Use the **`output()`** function to declare a component class property as output

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

@Component ({
  selector: 'app-counter',
  template: `<button (click)="increment()">{{ count }}</button>`
})
export class Counter {
  protected count = 0;

  countChange = output<number>();

  protected increment() {
    this.count += 1;
    this.countChange.emit(this.count);
  }
}
```

# Components - Output 2/2

- The consumer of this component can bind to the event in its template

```
import { Component } from '@angular/core';
import { Counter } from './counter/counter.ts';

@Component ({
  selector: 'app-root',
  imports: [Counter],
  template:
    `<app-counter (countChange)="updateCount($event)" />
     <p>Count: {{ parentCount }}</p>`
})
export class App {
  protected parentCount?: number;

  protected updateCount(count: number) {
    this.parentCount = count;
  }
}
```

- **Output events** are never propagated to the consumer's parent component, whereas **native DOM events** are (event bubbling)

# Components - Model input 1/4

- Use the **`model()`** function to declare a component class property as model input
- Unlike regular inputs, model inputs allow the component author to write values into the property

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

@Component ({
  selector: 'app-counter',
  template: `<button (click)="increment()">{{ count() }}</button>`
})
export class Counter {
  count = model<number>(0);

  protected increment() {
    this.count.update((count) => count + 1);
  }
}
```

# Components - Model input 2/4

- The consumer of this component can bind to both "property" and "event" in its template

```
import { Component } from '@angular/core';
import { Counter } from './counter/counter.ts';

@Component ({
  selector: 'app-root',
  imports: [Counter],
  template: `
    <app-counter [count]="parentCount" (countChange)="updateCount($event)" />
  `
})
export class App {
  protected parentCount = 5;

  protected updateCount(count: number) {
    this.parentCount = count;
  }
}
```

- The **`output`** name is based on the **`input`** name but with the suffix: **`"Change"`**

# Components - Model input 3/4

- Use the "Banana in a box" [🍌] syntax to easily achieve two-way data binding

```
import { Component } from '@angular/core';
import { Counter } from './counter/counter.ts';

@Component ({
  selector: 'app-root',
  imports: [Counter],
  template: `
    <app-counter [(count)]="parentCount" />
  `
})
export class App {
  protected parentCount = 5;

  protected updateCount(count: number) {
    this.parentCount = count;
  }
}
```

# Components - Model input 4/4

- Unlike **`inputs`** which are "readonly", **`models`** are "writable"

```
@Component({ ... })
export class Counter {
  count = model(0);
  constructor() {
    console.log(this.count());          // <-- output: 0

    this.count.set(1);
    console.log(this.count());          // <-- output: 1

    this.count.update((c) => c + 1);
    console.log(this.count());          // <-- output: 2
  }
}
```

- **`input`** and **`model`** are in fact "**signals**"
- **`.set()`** and **`.update()`** methods are part of the signals API
- Signals play a crucial role in the Angular reactivity model and whole chapter is devoted to them later in the course

# Components - Summary

**In this chapter on components, we have covered the following topics**

- **Class-Template interactions**
  - → @Component decorator
  - → Selector
  - → Template
  - → Styles
  - → Text interpolation
  - → Property binding
  - → Attribute binding
  - → Event listeners
- **Parent-Child communication**
  - → Input
  - → Output
  - → Model

---

😉 *To go further, discover: Content projection with `ng-content`*

# Components - Questions

# Components - Lab 3

# Testing

# Table of Contents

# Testing - Introduction

- Testing your Angular application helps you check that your application is working as you expect
  - → Unit tests are crucial for **catching bugs** early, ensuring **code quality**, and facilitating **safe refactoring**
- Testing requires two functional building blocks
  - → A **test runner** that identifies and runs the test files in a testing environment
  - → An **assertion library** that verifies the expected behavior
- Angular uses Vitest as test runner (which uses Chai for assertions with Jest expect-compatible APIs)
  - → Vitest runs your unit tests in a Node.js environment
  - → Using jsdom to emulate the DOM
- Test files are identified by the pattern: `*.spec.ts`
- Run your test suite with the shell command: `ng test`

# Testing - Vitest | Assertion library

- Organize your tests using **`describe`** and **`it`** functions
- Identify the "thing" being tested using **`expect`**
- Use assertions to verify the expected behavior: **`toBe`**, **`toBeTruthy`**, **`toContain`**, ...

```
describe('boolean variable', () => {
  let value: boolean;

  it('should be inverted when using "!" operator', () => {
    // Given
    value = true;

    // When
    value = !value;

    // Then
    expect(value).toBe(false);
  });
});
```

😉 *Follow the 3-step pattern in each test: "Given", "When", "Then"*

# Testing - Vitest | Assertion library - Hooks

- Use hooks to setup and teardown your tests

  → **`beforeEach`, `afterEach`, `beforeAll`, `afterAll`**

```
describe('boolean variable', () => {
  let value: boolean;

  beforeEach(() => {
    // Given
    value = true;
  });

  it('should be inverted when using "!" operator', () => {
    // When
    value = !value;

    // Then
    expect(value).not.toBe(true); // <-- notice the usage of `.not` to negate the assertion
  });
});
```

# Testing - Vitest | Assertion library - Mocks 1/2

- Create **mocks** using: `vi.fn()`

```
function dataHandler(callback: CallableFunction) {
  const data = Math.random();
  callback(data);
}

describe('dataHandler', () => {
  it('shoud call the callback function', () => {
    // Given
    const mock = vi.fn();

    // When
    dataHandler(mock);

    // Then
    expect(mock).toHaveBeenCalled();
  });
});
```

😉 *Use mocks to create a "fake" version of an internal or external dependency of your test*

# Testing - Vitest | Assertion library - Mocks 2/2

- Create **spies** using: `vi.spyOn()`

```
function dataHandler(callback: CallableFunction) {
  const data = Math.random();
  callback(data);
}

describe('dataHandler', () => {
  it('shoud call the callback function with a random value', () => {
    // Given
    const mathRandomSpy = vi.spyOn(Math, 'random').mockReturnValue(5);
    const mock = vi.fn();

    // When
    dataHandler(mock);

    // Then
    expect(mathRandomSpy).toHaveBeenCalled();
    expect(mock).toHaveBeenCalledWith(5);
  });
});
```

# Testing - Angular | TestBed

- Angular provides a powerful testing environment called **`TestBed`**

  → Testing configuration is reset for every test (executed in **`beforeEach`**)

```
import { TestBed } from '@angular/core/testing';

describe('MyFeature', () => {
  beforeEach(async () => {
    await TestBed.configureTestingModule({ /* Test setup */ }).compileComponents();
  });

  it('should work', () =>  { /* ... */ } });

  it('should work too', () =>  { /* ... */ } });
});
```

# Testing - Angular | TestBed - createComponent

- Components combine a **TypeScript class** and an **HTML template**
  - → You should test that they work together as intended
- `TestBed.createComponent` helps you create the component being tested
  - → The `fixture` gives you access to the **component instance** and its **template in the DOM**
  - → You must wait until the fixture is stable before verifying assertions

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { App } from './app';

await TestBed.configureTestingModule({ imports: [App] }).compileComponents();

let fixture = TestBed.createComponent(App);

let component = fixture.componentInstance;
let hostElement = fixture.nativeElement;

await fixture.whenStable();
```

# Testing - Component testing strategies

- Class testing
  - → **pros:** easy to setup, easy to write
  - → **cons:** does not make sure your component behaves the way it should
- DOM testing
  - → **pros:** make sure your component behaves exactly the way it should
  - → **cons:** harder to setup, harder to write

✅ Overall, **DOM testing** is **preferable** as it is **more robust**, but it requires more work

# Testing - Comp. with no dep. 1/4

- Let's test a simple counter **component with no dependencies**

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

@Component({
  selector: 'app-counter',
  template: '<button (click)="increment()">{{ count() }}</button>'
})
export class Counter {
  count = model(0);

  protected increment() {
    this.count.update((count) => count + 1);
  }
}
```

# Testing - Comp. with no dep. 2/4

- Test setup

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { Counter } from './counter';

describe('Counter', () => {
  let fixture: ComponentFixture<Counter>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({ imports: [Counter] }).compileComponents();

    fixture = TestBed.createComponent(Counter);

    await fixture.whenStable(); // <-- The template state needs to be initialized manually
  });
});
```

# Testing - Comp. with no dep. 3/4

- Actual tests

```ts
import { By } from '@angular/platform-browser';

it('should display 0', () => {
  // 👇 Getting element using `nativeElement`
  const button = (fixture.nativeElement as HTMLElement).querySelector('button');

  expect(button?.textContent).toContain(0);
});

it('should increment the count when clicking', async () => {
  // 👇 Getting element using `debugElement`
  const button = fixture.debugElement.query(By.css('button')).nativeElement;

  button?.click();
  expect(fixture.componentInstance.count()).toBe(1); // <-- Class testing (no need to wait)

  await fixture.whenStable();                // <-- Wait for the click event to be reflected in the DOM
  expect(button?.textContent).toContain(1); // <-- DOM testing
});
```

# Testing - Comp. with no dep. 4/4

- Actual tests (suite)

```
it('should call count "update" method when clicking', () => {
  const countUpdateSpy = vi.spyOn(fixture.componentInstance.count, 'update');

  const button = (fixture.nativeElement as HTMLElement).querySelector('button');
  button?.click();

  expect(countUpdateSpy).toHaveBeenCalled();
});
```

💪 *The rest of the chapter deals with **advanced techniques** for **testing components with dependencies***

# Testing - Comp. with dep.

- Let's test a more complex **component with dependencies**
- We're going to explore *two different approaches* to test this use case

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

@Component({
  selector: 'app-number-parity',
  imports: [Counter],
  template: `
    <app-counter [(count)]="count" />

    <span>{{ count % 2 ? 'is odd' : 'is even' }}</span>
  `,
})
export class NumberParity {
  count = 0;
}
```

# Testing - Comp. with dep. | First approach 1/2

- Test setup **with implicit dependency import**

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { By } from '@angular/platform-browser';
import { NumberParity } from './number-parity';

describe('NumberParity', () => {
  let component: NumberParity;
  let fixture: ComponentFixture<NumberParity>;

  beforeEach(async () => {
    // 👇 Note: the `Counter` is implicitly imported, as it appears in the `NumberParity` imports!
    await TestBed.configureTestingModule({ imports: [NumberParity] }).compileComponents();

    fixture = TestBed.createComponent(NumberParity);
    component = fixture.componentInstance;

    await fixture.whenStable();
  });
});
```

# Testing - Comp. with dep. | First approach 2/2

- Actual tests **accessing the dependency instance** (the child component instance)

```
it('should bind parent "count" to child component', () => {
  const counter: Counter = fixture.debugElement.query(By.directive(Counter)).componentInstance;

  // 👇 Accessing the child component instance properties
  const counterCount = counter.count();
  expect(counterCount).toBe(component.count);
});

it('should be "odd" when child component emits', async () => {
  const counter: Counter = fixture.debugElement.query(By.directive(Counter)).componentInstance;

  // 👇 Accessing the child component instance methods
  counter.count.set(1);
  await fixture.whenStable();

  const span = (fixture.nativeElement as HTMLElement).querySelector('span');
  expect(span?.textContent).toContain('odd');
});
```

# Testing - Comp. with dep. | Second approach 1/2

- Test setup **allowing unknown HTML elements**

```
import { CUSTOM_ELEMENTS_SCHEMA } from '@angular/core';
import { Counter } from '../counter/counter';
// The rest of the imports...

describe('NumberParity', () => {
  let component: NumberParity;
  let fixture: ComponentFixture<NumberParity>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({ imports: [NumberParity] })
      .overrideComponent(NumberParity, {
        remove: { imports: [Counter] },
        add: { schemas: [CUSTOM_ELEMENTS_SCHEMA] },
      })
      .compileComponents();

    // The rest of the setup...
  });
});
```

# Testing - Comp. with dep. | Second approach 2/2

- Actual tests using `debugElement.properties` and `debugElement.triggerEventHandler`

```
it('should bind parent "count" to child component', () => {
  const debugElement = fixture.debugElement.query(By.css('app-counter'));

  // 👇 Accessing Angular bindings on the child DOM element
  expect(debugElement.properties['count']).toBe(component.count);
});

it('should be "odd" when child component emits', async () => {
  const debugElement = fixture.debugElement.query(By.css('app-counter'));

  // 👇 Triggering Angular events on the child DOM element
  debugElement.triggerEventHandler('countChange', 1);
  await fixture.whenStable();

  const span = (fixture.nativeElement as HTMLElement).querySelector('span');
  expect(span?.textContent).toContain('odd');
});
```

# Testing - Summary

**In this chapter on testing, we have covered the following topics**

- Test runner
- Assertion library
- Mocks
- TestBed
- Component testing
- Testing strategies for component with dependencies

# Testing - Questions

# Testing - Lab 4

# Control flow

# Table of Contents

# Control flow

- Angular templates support control flow blocks that let you **conditionally show, hide, and repeat elements**
- The most common control flows are
  - → `@if`
  - → `@for`
  - → `@switch`

😉 *Another control flow is the `@defer` block, but it is beyond the scope of this course*

# Control flow - @if 1/2

- Conditionally display content with `@if {}`, `@else if {}` and `@else {}`

```
@if (todos === undefined) {

  <p>Please wait, your todo list is being loaded...</p>

} @else if (todos.length === 0) {

  <p>Your todo list is empty.</p>

} @else {

  <p>You have {{ todos.length }} todos in your list.</p>
}
```

*In this example, we assume that the property `todos: Todo[]` is defined on the component class*

# Control flow - @if 2/2

- The `@if` conditional supports saving the result of the conditional expression into a variable for reuse inside of the block

```
@if (todos === undefined) {

  <p>Please wait, your todo list is being loaded...</p>

} @else if (todos.length; as todosLength) {

  <p>You have {{ todosLength }} todos in your list.</p>
}
```

# Control flow - @for 1/3

- Repeat content with the `@for` block

```
<ul>
  @for (todo of todos; track todo.id) {

    <li>{{ todo.title }}</li>

  }
</ul>
```

- The `track` expression allows Angular to maintain a relationship between your data and the DOM nodes on the page
- This allows Angular to optimize performance by executing the minimum necessary DOM operations when the data changes

# Control flow - @for 2/3

- Inside `@for` blocks, several implicit variables are always available...

```
<ul>
  @for (todo of todos; track todo.id) {
    <li>{{ $index + 1 }}/{{ $count }} {{ todo.title }}</li>
  }
</ul>
```

- ...but can be aliased if needed, using `let` syntax

```
<ul>
  @for (todo of todos; track todo.id; let idx = $index, cnt = $count) {
    <li>{{ idx + 1 }}/{{ cnt }} {{ todo.title }}</li>
  }
</ul>
```

- Here's the list of the implicit variables which are self-explanatory

  → `$count`, `$index`, `$first`, `$last`, `$even`, `$odd`

# Control flow - @for 3/3

- Providing a fallback for `@for` blocks with the `@empty` block

```
<ul>
  @for (todo of todos; track todo.id; let index = $index, count = $count) {

    <li>{{ index + 1 }}/{{ count }} {{ todo.title }}</li>

  } @empty {

    <li>Your todo list is empty.</li>
  }
</ul>
```

# Control flow - @switch

- Conditionally display content with the `@switch` block

```
@switch (todos.length) {
  @case (0) {
    <p>Your todo list is empty.</p>
  }

  @case (1) {
    <p>You have one todo in your list.</p>
  }

  @default {
    <p>You have {{ todos.length }} todos in your list.</p>
  }
}
```

# Control flow - Summary

**In this chapter on control flow, we have covered the following topics**

- @if
- @for
- @switch

# Control flow - Questions

# Control flow - Lab 5

# Directives

# Table of Contents

# Directives

- Live in the **component template**
- Need a **host element** to be attached to
- Add **additional behavior** to host elements in your template
- Defined in a single place, it can be used in several components
- Angular offers several **built-in directives** to manage routing, forms, and what users see

# Directives

There are three types of directives:

- **Attribute directives**: change the appearance or behavior of DOM elements
- **Structural directives**: change the DOM layout by adding and removing DOM elements
- **Components**: yes! components are in fact directives that embed their own template

*Note:*

- Components have already been covered 🎉
- Structural directives are complex and beyond the scope of this course 🚀
- Therefore, this course focuses only on **attribute directives** ✅

😉 *In this chapter, we'll cover the definition and usage of **custom** attribute directives.*
*Later in the course, you'll discover some Angular **built-in** attribute directives such as `RouterLink` (Routing) and `NgModel` (Forms).*

# Attribute directive - Definition

- To create a directive, add the `@Directive` decorator on a class
- `ElementRef` gives you access to the host element
- `Renderer2` let you change the appearance or behavior of the host element

```
import { Directive, ElementRef, Renderer2, inject } from '@angular/core';

@Directive({ selector: '[appHighlight]' })
export class Highlight {
  constructor() {
    const elementRef = inject(ElementRef);
    const renderer = inject(Renderer2);

    renderer.listen(elementRef.nativeElement, 'mouseenter', () => {
      renderer.setStyle(elementRef.nativeElement, 'backgroundColor', 'yellow');
    });
    renderer.listen(elementRef.nativeElement, 'mouseleave', () => {
      renderer.setStyle(elementRef.nativeElement, 'backgroundColor', null);
    });
  }
}
```

# Attribute directive - Usage

- Import the directive **`class`** in your component
- Use the directive **`selector`** to attach it to DOM elements in the component template

```
import { Component } from '@angular/core';
import { Highlight } from './highlight.ts';

@Component({
  selector: 'app-root',
  imports: [Highlight],
  template: `<p appHighlight> Highlight me! </p>`,
})
export class App {}
```

- At runtime, if we open the Chrome inspector, we can verify that the style has been correctly applied to the paragraph

```
<p style="background-color: yellow"> Highlight me! </p>
```

# Attribute directive - Host metadata

- When possible, instead of the **`Renderer2`** (imperative programming), use the **`host`** metadata (declarative programming) to configure *host binding* and *event listener*

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

@Directive ({
  selector: '[appHighlight]',
  host: {
    '[style.backgroundColor]': 'currentColor',
    '(mouseenter)': 'onMouseEnter()',
    '(mouseleave)': 'onMouseLeave()',
  }
})
export class Highlight {
  currentColor?: string;

  onMouseEnter() { this.currentColor = 'yellow'; }

  onMouseLeave() { this.currentColor = undefined; }
}
```

😉 *Note that **`host`** property also applies to component metadata*

# Attribute directive - Input and Output 1/2

- Use **`input`** and **`output`** functions to make the directive configurable

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

@Directive ({
  selector: '[appHighlight]',
  host: { /* ...same bindings as previous slide... */ }
})
export class Highlight {
  currentColor?: string;
  highlightColor = input('yellow', { alias: 'appHighlight' });
  highlighted = output<boolean>();

  onMouseEnter() {
    this.currentColor = this.highlightColor();
    this.highlighted.emit(true);
  }
  onMouseLeave() {
    this.currentColor = undefined;
    this.highlighted.emit(false);
  }
}
```

# Attribute directive - Input and Output 2/2

- Use regular property binding and event listeners on the host element

```
import { Component } from '@angular/core';
import { Highlight } from './highlight.ts';

@Component({
  selector: 'app-root',
  imports: [Highlight],
  template: `
    <p [appHighlight]="highlightColor" (highlighted)="highlightedHandler($event)">
      Highlight me!
    </p>
  `,
})
export class App {
  highlightColor = 'green';

  highlightedHandler(highlighted: boolean) {
    console.log('Is highlighted?', highlighted);
  }
}
```

# Directives - Testing 1/2

- Create a `Wrapper` component for DOM testing purposes

```
import { Component } from '@angular/core';
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { By } from '@angular/platform-browser';
import { Highlight } from './highlight';

@Component({
  selector: 'app-wrapper',
  imports: [Highlight],
  template: '<div appHighlight="green">Highlight me!</div>',
})
class Wrapper {}

// ...
```

# Directives - Testing 2/2

- Use `By.directive` to retrieve the directive being tested

```
// ...

describe('Highlight', () => {
  let fixture: ComponentFixture<Wrapper>;
  let hostElement: HTMLElement;

  beforeEach(async () => {
    await TestBed.configureTestingModule({ imports: [Wrapper] }).compileComponents();

    fixture = TestBed.createComponent(Wrapper);
    await fixture.whenStable();

    hostElement: HTMLElement = fixture.debugElement.query(By.directive(Highlight)).nativeElement;
  });

  it('should work', async () => {
    expect(hostElement.style.backgroundColor).toBe('green');
  });
});
```

# Directives - Summary

**In this chapter on directives, we have covered the following topics**

- @Directive decorator
- Selector
- Renderer2
- Host metadata
- Input and Output
- Attribute directives

---

😉 *To go further, discover: Structural directives*

# Directives - Questions

# Directives - Lab 6

# Signals

# Table of Contents

# Signals - Definition

- 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**
- Signals may be either **writable** or **read-only**

😉 *Later, we'll talk about a process called **synchronization** to understand when and why you should **use signals rather than raw values** to manage the state of your application...*

# Signals - signal

- Use `signal` function to create a writable signal

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

const count = signal<number>(0);

console.log(count());                    // <-- output: 0

count.set(1);

console.log(count());                    // <-- output: 1

count.update((c) => c + 1);

console.log(count());                    // <-- output: 2
```

# Signals - computed

- Use the **`computed`** function to derive a signal from other signals
- Re-evaluated only when the signals on which they depend change
- Computed signals are read-only

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

const count = signal<number>(0);

const isEven = computed(() => count() % 2 === 0);

console.log(isEven());                // <-- output: true

count.set(1);

console.log(isEven());                // <-- output: false

count.update((c) => c + 1);

console.log(isEven());                // <-- output: true
```

# Signals - effect

- Use the **`effect`** function to run "side effect", whenever one or more signal values change
- Re-evaluated only when the signals on which they depend change
- Effect signals run at least once

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

const count = signal<number>(0);

effect(() => {
  console.log('The current count is: ', count()); // <-- Will output: 0, 1, 2
});

count.set(1);

count.update((c) => c + 1);
```

😉 *Another signal is the **`linkedSignal`**, but it is beyond the scope of this course*

# Signals - Usage in components

- When a signal changes, Angular will automatically re-render the templates that depend on it
- This process is highly efficient, whether the signal is modified in the component itself or in another part of the application

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

@Component({
  selector: 'app-counter-delay',
  template: `<button (click)="increment()">{{ count() }}</button>`,
})
export class CounterDelay {
  count = signal(0);

  increment() {
    // Angular will correctly synchronize the UI with the updated signal value,
    // even if the signal mutation occurs asynchronously!
    setTimeout(() => this.count.update((count) => count + 1), 1000);
  }
}
```

# Signals - Component input and model 1/2

- Note that **`input`** and **`model`** functions (mentioned in the chapter on components) **are in fact signals**
- Using signals for parent/child communication, makes this communication highly reactive and efficient

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

@Component({
  selector: 'app-counter',
  template: `<button (click)="increment()">{{ count() }}</button>`,
})
export class Counter {
  count = model(0);

  protected increment() {
    this.count.update((count) => count + 1);
  }
}

// ...
```

# Signals - Component input and model 2/2

```
// ...

@Component({
  selector: 'app-root',
  imports: [Counter],
  template: `
    <app-counter [(count)]="appCount" />

    <p>{{ appCount() }}</p>
  `,
})
export class App {
  protected appCount = signal(0);
}
```

# Signals - Testing 1/3

- Angular provides powerful tooling for testing signal-based components

Let's revisit the **`Counter`** component...

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

@Component({
  selector: 'app-counter',
  template: `<button (click)="increment()">{{ count() }}</button>`,
})
export class Counter {
  count = model(0);

  protected increment() {
    this.count.update((count) => count + 1);
  }
}
```

# Signals - Testing 2/3

- Use **`inputBinding`**, **`outputBinding`** and **`twoWayBinding`** functions in the test component **`bindings`** options

```
import { inputBinding, outputBinding, signal, twoWayBinding } from '@angular/core';
import { TestBed } from '@angular/core/testing';
import { Counter } from './counter';

describe('Counter', () => {
  beforeEach(async () => await TestBed.configureTestingModule({ imports: [Counter] }).compileComponents());

  it('should work', async () => {
    const count = signal(1);                          // <-- Define "input"
    const countChange = vi.fn();                      // <-- Define "output"

    const fixture = TestBed.createComponent(Counter, {
      bindings: [
        inputBinding('count', count),                 // <-- Bind "input"
        outputBinding('countChange', countChange),    // <-- Bind "output"
        // twoWayBinding('count', count)              // <-- Alternative (to handle both input and output)
      ],
    });

    // ...
```

# Signals - Testing 3/3

```
    // ...

    const component = fixture.componentInstance;

    expect(component.count()).toBe(1);

    count.set(2);                                    // <-- Interact with "input" bindings
    expect(component.count()).toBe(2);

    (fixture.nativeElement as HTMLElement).querySelector('button')?.click();
    await fixture.whenStable();
    expect(component.count()).toBe(3);
    expect(countChange).toHaveBeenCalledWith(3);     // <-- Interact with "output" bindings
  });
});
```

# Signals - Synchronization process

- The goal of synchronization is to keep the **UI** in sync with the **state** of the application
- **signals** play a crucial role in enabling Angular to know exactly when and which parts of the UI need to be synchronized
- As a rule of thumb, if the part of the **state to be rendered in your templates** only **changes through signals** then your UI should always be in sync with the state of your application

⚠️ *Note that with signals Angular has entered a new era. Previously, the synchronization process was achieved using a third-party library called **Zone.js**.*

- *This was a very complex process, formerly called **Change detection***
- *In a nutshell, Zone.js was responsible for telling Angular when to trigger its change detection process and update the UI to reflect the state of the application*
- *So today, Angular no longer relies on Zone.js, and that's why we've entered the era of **Zoneless applications***

# Signals - Summary

**In this chapter on signals, we have covered the following topics**

- Signal
- Computed signal
- Effect
- Usage in components
- Component input and model
- Synchronization process

# Signals - Questions

# Signals - Lab 7

# Dependency injection

# Table of Contents

# Dependency injection - In a nutshell

- A broad category encompassing any value or feature that an application needs

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

export class ApiService {                          // <-- 1. Defining
  getData() { return 'Hello World!'; }
}

export const appConfig: ApplicationConfig = {
  providers: [ApiService],                         // <-- 2. Providing
};

@Component({
  selector: 'app-root',
  template: '<h1>{{ data }}</h1>',
})
export class App {
  private apiService = inject(ApiService);         // <-- 3. Injecting

  data = this.apiService.getData();                // <-- 4. Consuming
}
```

# Dependency injection - Injectable decorator

- Use `@Injectable` decorator and `providedIn` metadata to **provide a service globally** right from its definition

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

@Injectable({
  providedIn: 'root', // <-- Instruct Angular to provide the service automatically (if used by the app)
})
export class ApiService {
  getData() { return 'Hello World!'; }
}

export const appConfig: ApplicationConfig = {
  providers: [],      // <-- Therefore, it is no longer necessary to provide it manually!
};
```

# Dependency injection - Injectors & Singleton 1/3

- Injectors are responsible for **providing dependencies** to any part of the application (components, services, ...)
- An application can have more than one injector:
  - → the `providers` array in the `ApplicationConfig` object is the main injector...
  - → ...but dependencies can be **provided at other levels** *(more details below)*
- The important point is that **within an injector every dependency is a singleton**

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

@Injectable({
  providedIn: 'root',                                  // <-- Providing globally
})
export class DataService {
  data = signal<string | undefined>(undefined);        // <-- Defining property
}

// ...
```

# Dependency injection - Injectors & Singleton 2/3

```
// ...

@Component({
  selector: 'app-data-setter',
  template: '<button (click)="setData()">Set data</button>',
})
export class DataSetter {
  private dataService = inject(DataService);          // <-- Injecting

  protected setData() {
    this.dataService.data.set('Hello World!');        // <-- Setting property in one place
  }
}

// ...
```

# Dependency injection - Injectors & Singleton 3/3

```
// ...

@Component({
  selector: 'app-data-getter',
  template: '<h1>{{ data() }}</h1>',                    // <-- output: Hello World!
})
export class DataGetter {
  private dataService = inject(DataService);           // <-- Injecting

  protected data = this.dataService.data;              // <-- Getting modified property in another place
}
```

😉 *In this example, **`DataSetter`** and **`DataGetter`** components share the same **`DataService`** instance*

- Services allow **data** to be **shared** between any parts of the application (components, services, ...)

# Dependency injection - Injectors hierarchy

- During a dependency injection
  - → the local injector tries to **find a compatible provider**
  - → if it can't find one, it forwards the request to its **parent injector**
  - → and so on up to the application's **main injector**
  - → if no provider can be found, Angular **throws an error**
- In a typical Angular application, **most services are provided globally** at the application configuration level
- However, it is sometimes useful to **delegate part of a component's logic to a dedicated service**, which is then **provided at the component level itself**

# Dependency injection - Component providers

- Use the `providers` array in the component decorator metadata to manually **provide a service locally**
  - → The service lifecycle (creation and destruction) follows the component lifecycle
  - → A service provided in a component can also be injected into its child components

```
@Component({
  selector: 'app-parent',
  providers: [ParentService],                  // <-- Service provided locally
  imports: [Child],
  template: '<app-child />',
})
export class Parent {
  parentService = inject(ParentService);       // <-- Get the service from the local injector
}

@Component({ selector: 'app-child', template: '...' })
export class Child {
  parentService = inject(ParentService);       // <-- Get the service from the parent injector
}
```

# Dependency injection - Injection context

- The dependency injection (DI) system relies internally on a **runtime context** where the current **injector is available**
- This means that injectors can only work when code is executed in such a context

```
@Component({
  selector: 'app-root',
  template: '...',
})
export class App {
  private dataService = inject(DataService);     // ✅ Field initialization

  constructor() {
    const dataService = inject(DataService);     // ✅ Class constructor
  }

  doSomething() {
    const dataService = inject(DataService);     // ❌ Class method
  }
}
```

# Dependency injection - Providers | ClassProvider

- So far we've provided services by adding them to the **`provider`** array

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

export const appConfig: ApplicationConfig = {
  providers: [ApiService],
};
```

- It is in fact a shorthand of the class provider, whose full syntax is

```
import { ApplicationConfig, ClassProvider } from '@angular/core';

export const appConfig: ApplicationConfig = {
  providers: [
    {
      provide: ApiService,
      useClass: ApiService,
    },
  ] satisfies ClassProvider,
};
```

# Dependency injection - Providers | ValueProvider

- Use **`InjectionToken`** and **`ValueProvider`** to provide primitive values (such as **`string`**, **`number`**, ...)

```ts
import {
  InjectionToken, ValueProvider, ApplicationConfig, Component, inject
} from '@angular/core';

const APP_TITLE = new InjectionToken<string>('app title');

const appTitleProvider: ValueProvider = { provide: APP_TITLE, useValue: 'My Awesome App' };

export const appConfig: ApplicationConfig = {
  providers: [appTitleProvider],
};

@Component({/* ... */})
export class App {
  appTitle = inject(APP_TITLE); // <-- 'My Awesome App'
}
```

In the next chapter on **`Pipes`**, you'll see how Angular uses **`InjectionTokens`**

😉 *Note that there's also a **`FactoryProvider`**, but it is beyond the scope of this course*

# Dependency injection - App Initializer

- Perform asynchronous tasks before the application is bootstrapped
- Supports dependency injection

```
import { ApplicationConfig, provideAppInitializer } from '@angular/core';
import { Observable } from 'rxjs';

export const appConfig: ApplicationConfig = {
  providers: [
    provideAppInitializer((): Observable<unknown> | Promise<unknown> | void => {
      // In this example, we restore the user's status before bootstrapping the application
      return inject(UserService).fetchUser();
    }),
  ],
};
```

😉 *Note that Angular executes the initializer function in an injection context*

😉 *Note that there is another initialization function:* ***`provideEnvironmentInitializer`***

# Dependency injection - Testing in isolation

- You can configure the providers in your **`TestBed`**
- Powerful mechanism that isolates the element you really want to test
- Use **`TestBed.inject`** to access the service instance in your test

In the following example, we test a component in isolation, replacing the service with a Mock:

```
import { TestBed } from '@angular/core/testing';

describe('App', () => {
  let apiService: ApiService;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [App],
      providers: [{ provide: ApiService, useClass: ApiServiceMock }],
    }).compileComponents();

    apiService = TestBed.inject(ApiService); // <-- Get the `ApiServiceMock`
  });
});
```

# Dependency injection - Summary

**In this chapter on dependency injection, we have covered the following topics**

- @Injectable decorator
- Injectors and Singleton pattern
- Injectors hierarchy
- App providers
- Component providers
- Injection context
- ClassProvider
- ValueProvider
- App Initializer

# Dependency injection - Questions

# Dependency injection - Lab 8

# Pipes

# Table of Contents

# Pipes - Definition

- Special operator in Angular template expressions
- Transform data declaratively in your template
- Transformation functions are declared once and then used across multiple templates
- Angular provides a lot of pipes for common use cases...

```
import {
  LowerCasePipe, UpperCasePipe, TitleCasePipe,
  CurrencyPipe, DecimalPipe, PercentPipe,
  DatePipe, JsonPipe, SlicePipe, KeyValuePipe,
} from '@angular/common';
```

- ... but you can also create custom pipes based on your business logic

# Pipes - Usage in template

- Are applied using the "|" symbol
- Can be chained
- Additional parameters can be passed using the ":" symbol

```
import { Component } from '@angular/core';
import { DatePipe, UpperCasePipe, CurrencyPipe } from '@angular/common';

@Component({
  selector: 'app-root',
  imports: [DatePipe, UpperCasePipe, CurrencyPipe],
  template: `
    <p>{{ myDate | date }}</p>                              <!-- 29 août 2023 -->
    <p>{{ myDate | date | uppercase }}</p>                  <!-- 29 AOÛT 2023 -->
    <p>{{ myPrice | currency : 'EUR' : 'symbol' }}</p>  <!-- 123,46 €     -->
  `,
})
export class App {
  myDate = new Date();
  myPrice = 123.456789;
}
```

# Pipes - Custom

- Can be generated using Angular CLI: `ng generate pipe <pipeName>`
- Use the `@Pipe` decorator on a class
- Class must implement the `PipeTransform` interface (i.e. the `transform` method)

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

@Pipe({ name: 'joinArray' })
export class JoinArrayPipe implements PipeTransform {
  transform(value: (string | number)[], separator = ' '): string {
    return value.join(separator);
  }
}
```

- Usage example:

```
<p>List: {{ ['apple', 'orange', 'banana'] | joinArray : ' / ' }}</p>

<!-- List: apple / orange / banana -->
```

# Pipes - Configuration

Some Angular pipes can be configured globally

Here's an example with the `CurrencyPipe`

- Depending on the locale:
  - → should display `$3.50` for United States (this is the default behavior)
  - → should display `3,50 $` for France
- You may also need to configure the default symbol to be `€` instead of `$`:
  - → should display `€3.50` for United States
  - → should display `3,50 €` for France

# Pipes - Configuration | CurrencyPipe

- Here's the configuration to display the currency in EUR for France (`3,50 €`)

```ts
// src/app/app.config.ts

import { ApplicationConfig, LOCALE_ID, DEFAULT_CURRENCY_CODE } from '@angular/core';

// Defines how to format currency, date, ... in french
import { registerLocaleData } from '@angular/common';
import localeFr from '@angular/common/locales/fr';
registerLocaleData(localeFr);

export const appConfig: ApplicationConfig = {
  providers: [
    { provide: LOCALE_ID, useValue: 'fr' },
    { provide: DEFAULT_CURRENCY_CODE, useValue: 'EUR' },
  ],
};
```

# Pipes - Usage in class

- Can be instantiated directly in TypeScript code (using **new** operator)
- Can also be injected like any provider...
  - → ...but must be provided in the **`providers`** array (Component or ApplicationConfig)
  - → the injected pipe will respect the global configuration, if any

```
import { Component, inject } from '@angular/core';
import { CurrencyPipe, UpperCasePipe } from '@angular/common';

@Component ({ selector: 'app-root', providers: [CurrencyPipe] })
class App {
  constructor() {
    console.log(new UpperCasePipe().transform('Hello World!'));    // <-- HELLO WORLD!

    console.log(inject(CurrencyPipe).transform(123.456789));       // <-- 123,46 €
  }
}
```

# Pipes - Pure

- Transformation function can be marked as "pure" if it has the following properties:
  - → the function return values are identical for identical arguments
  - → the function has no side effects
- When Angular re-evaluate a template, it will only re-evaluate the pipe if its input value **reference** has changed
- Pipes are pure by default

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

@Pipe({ name: 'fancy' /*, pure: true */ })
export class FancyPipe implements PipeTransform {
  transform(value: string): string {
    return `Fancy ${value}`;
  }
}
```

# Pipes - Impure 1/2

- Angular always re-evaluate "impure" pipe, even if its input value **reference** has not changed
  - → Should be used for input value such as `Array` or `Object` that may be mutated over time

Example: because Angular's `JsonPipe` is defined as `impure`, after clicking on the button, the mutated object will be properly displayed in the UI.

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

@Component({
  selector: 'app-root',
  imports: [JsonPipe],
  template:
    `<pre>{{ data | json }}</pre>

    <button (click)=" data.msg = 'Bye' ">Mutate</button>`,
})
export class App {
  data = { msg: 'Hello' };
}
```

# Pipes - Impure 2/2

- Let's look again at the custom pipe used as an example earlier

  → It should be defined as **`impure`** because its input is an **`Array`** that may be mutated

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

@Pipe({ name: 'joinArray', pure: false }) // <-- Should be impure!
export class JoinArrayPipe implements PipeTransform {
  transform(value: (string | number)[], separator = ' '): string {
    return value.join(separator);
  }
}

@Component({
  selector: 'app-root',
  template: `{{ appList | joinArray }}
    <button (click)=" appList.push('kiwi') ">Mutate</button>`, // <-- Mutation
})
export class App {
  appList = ['apple', 'orange', 'banana'];
}
```

# Pipes - Testing

- A Pipe is nothing but a function!
- Instantiate the pipe in a **`beforeEach`** hook
- Call the **`transform`** method to test all possible cases

```
import { JoinArrayPipe } from './pipes/join-array-pipe';

describe('JoinArrayPipe', () => {
  let pipe;

  beforeEach(() => {
    pipe = new JoinArrayPipe();
  });

  it('should works', () => {
    const output = pipe.transform(['apple', 'orange', 'banana'], ', ');

    expect(output).toEqual('apple, orange, banana');
  });
});
```

# Pipes - Summary

**In this chapter on pipes, we have covered the following topics**

- @Pipe decorator
- Built-in pipes
- Custom pipe
- Configuration
- Usage in template
- Usage in class
- Pure VS Impure

# Pipes - Questions

# Pipes - Lab 9


PHP
CSS
HTML
<HTML>
<HEAD>
<TITLE> Lorem ipsum </TITLE>
</HEAD>
<BODY>
<H1> Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam</H1>
<P> <IMG SRC= "Lorem ipsum"></P>

# RxJS

*— prerequisites —*

# Table of Contents

# RxJS

💡 *This chapter covers the **RxJS** prerequisites needed to confidently follow the next chapter on **HTTP client***

- RxJS refers to a **paradigm** called ReactiveX (http://reactivex.io/)
  - → an API for asynchronous programming with observable streams
  - → implemented in all major programming languages: *RxJava, Rx.NET, ...*
- Let's focus on the JavaScript implementation: *RxJS*

# RxJS - In a nutshell

- **Observables**
  - → represent a **stream** of data that can be **subscribed to**
  - → allowing multiple **values** to be **emitted** over time

# RxJS - Building blocks

- To understand RxJS, you need to learn the following concepts
  - → `Observable` and `Observer`
  - → `Subscription`
  - → `Operators`

# RxJS - Observable & Observer 1/2

```ts
import { Observable, Observer } from 'rxjs';

const data$ = new Observable<number>((subscriber) => {
  subscriber.next(1);                                  // <-- Emit next value
  subscriber.next(2);                                  // <-- Emit next value
  subscriber.complete();                               // <-- Mark as complete...
  // subscriber.error('Oops!');                        // <-- ...or mark as in error
});

const observer: Partial<Observer<number>> = {
  next: (data: number) => console.log(data),           // <-- Listen to "next" events
  complete: () => console.log('Done'),                 // <-- Listen to "complete" event
  error: (err: unknown) => console.error(err),         // <-- Listen to "error" event
};

data$.subscribe(observer);                             // output: 1, 2, Done
```

- The **`subscriber`** shapes the behavior of the observable
- The **`observer`** specifies which events you want to listen to
- Subscriber and observer method names match: **`next`**, **`complete`** (and also **`error`**)

# RxJS - Observable & Observer 2/2

- Use a function as observer to simply listen to **`next`** events

```
data$.subscribe({
  next: (data: number) => console.log(data),              // <-- Object property as "next" observer
});

data$.subscribe((data: number) => console.log(data));    // <-- Function as "next" observer
```

# RxJS - Subscription

- A Subscription is what starts an Observable, keeps it running, and gives you a way to stop it

```
import { Observable, Subscription } from 'rxjs';

const data$ = new Observable<number>((subscriber) => {
  let data = 0;

  const interval = setInterval(() => {
    subscriber.next(++data);            // <-- Emit next value every second ad infinitum
    console.log('tick');
  }, 1000);

  return () => clearInterval(interval); // <-- Return the resource cleanup function
});

const subscription: Subscription = data$.subscribe((data: number) => {
  console.log(data);
  if (data === 3) {
    subscription.unsubscribe();         // <-- Unsubscribe from data$ and execute
                                        //     the resource cleanup function
  }
}); // output: 1, tick, 2, tick, 3, tick
```

# RxJS - Operators 1/2

- Operators such as **`filter`** and **`map`** are functions that let you filter, transform, and control the Observable

```
import { Observable, filter, map } from 'rxjs';

const data$ = new Observable<number>((subscriber) => {
  subscriber.next(1);
  subscriber.next(2);
  subscriber.complete();
});

data$.pipe(/* no operator */).subscribe(console.log);                    // output: 1, 2

data$.pipe(filter((data) => data % 2 === 0)).subscribe(console.log);     // output: 2

data$.pipe(map((data) => data * 10)).subscribe(console.log);             // output: 10, 20

data$.pipe(
  filter((data) => data % 2 === 0),
  map((data) => data * 10)
).subscribe(console.log);                                                // output: 20
```

# RxJS - Operators 2/2

- The **tap** operator lets you "peek" into an Observable without changing the data (used for side effects only)

```
import { Observable, tap, map } from 'rxjs';

const data$ = new Observable<number>((subscriber) => {
  subscriber.next(1);
  subscriber.next(2);
  subscriber.complete();
});

let evenValuesCount = 0;                                    // <-- Defined out of the stream

data$.pipe(
  tap((data) => {
    if (data % 2 === 0) evenValuesCount += 1;               // <-- Handle side effect
    return 'ignored value';                                 // <-- Return value is ignored
  }),
  map((data) => data * 10)
).subscribe(console.log);                                   // output: 10, 20
```

😉 *Note: **`filter`**, **`map`**, and **`tap`** are **synchronous** operators.*
*There are also **asynchronous** operators, but this is beyond the scope of this course.*

# RxJS - Summary

- By convention, a variable representing an observable ends with the symbol `$`
- The **Observable** implementation is a function that uses the **Subscriber** methods to emit the stream events
  - → `.next()`, `.complete()` and `.error()`
- The `.subscribe()` method activates the observable to emit its data stream
  - → It accepts an **object** (`Partial<Observer>`) or a **function** as observer to listen to the stream events
  - → It returns a `Subscription` allowing the consumer to `.unsubscribe()` from the activated observable
- **Unsubscription** is necessary to **avoid memory leaks** when the consumer is no longer interested in the data
  - → Unless the observable is already in `"complete"` or `"error"` state
- The `Operators` allow to transform the emitted values and make the observables very powerful 🚀

# RxJS - Questions

# HTTP client

# Table of Contents

# HTTP client - Getting started 1/4

- Let's use the **`jsonplaceholder`** API to display todo items

```
const TODOS_URL = 'https://jsonplaceholder.typicode.com/todos';
```

- Here's the **`Todo`** item interface...

```
interface Todo {
  id: number;
  title: string;
  completed: boolean;
}
```

- ...and the first two items of the API response

```
[
  { "id": 1, "title": "delectus aut autem", "completed": false },
  { "id": 2, "title": "quis ut nam facilis et officia qui", "completed": false },
  ...
]
```

# HTTP client - Getting started 2/4

- Provide the **`HttpClient`** service using the **`provideHttpClient()`** helper function
- Optionally, configure the service to use the browser's native **`fetch`** API, by adding the **`withFetch()`** feature

```
import { ApplicationConfig } from '@angular/core';
import { provideHttpClient, withFetch } from '@angular/common/http';

export const appConfig: ApplicationConfig = {
  providers: [
    provideHttpClient(withFetch()),                    // <-- 1. Add provider
  ],
};
```

😉 *Note: providing the HTTP client is optional, unless you need to add features*

😉 *Note: **`withInterceptors(...)`** is another common feature, but this is beyond the scope of this course*

# HTTP client - Getting started 3/4

- Use the **`HttpClient`** service into components that needs to display the data

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

@Component({
  selector: 'app-todo-list',
  template: `<pre>{{ todos | json }}</pre>`,
  imports: [JsonPipe],
})
export class TodoList {
  private httpClient = inject(HttpClient);          // <-- 2. Inject service

  protected todos?: Todo[];

  constructor() {
    this.httpClient
      .get<Todo[]>(TODOS_URL)                       // <-- 3. Define shape of GET request
      .subscribe((todos) => (this.todos = todos));  // <-- 4. Execute request and store response
  }
}
```

# HTTP client - Getting started 4/4

- There are many **`HttpClient`** methods and they are highly configurable
- They describe the shape of requests as **RxJS Observables**

```
class HttpClient {
  // --- Generic method (for advanced use cases) ---
  request<R>(method: string, url: string, options?: HttpOptions): Observable<R>;

  // --- Shorthand methods (enough in most cases) ---
  get<R>(url: string, options?: HttpOptions): Observable<R>;
  post<R>(url: string, body: any, options?: HttpOptions): Observable<R>;

  // .put(), .patch(), .delete(), ...
}

interface HttpOptions {
  headers?: HttpHeaders;
  params?: HttpParams;
  responseType?: 'json';
  // ...
}
```

# HTTP client - Service and Component layers 1/2

- While `HttpClient` can be injected and used directly into **components**
- It is recommended to create reusable, injectable **services** which encapsulate data access logic
- **Data providers** *(service layer)* should only **expose the shape of requests** and let **data consumers** *(component layer)* **subscribe** to them

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

@Injectable({
  providedIn: 'root'
})
export class TodoService {
  private httpClient = inject(HttpClient);

  fetch() {
    return this.httpClient.get<Todo[]>(TODOS_URL); // <-- Do NOT "subscribe" in service layer
  }
}
```

# HTTP client - Service and Component layers 2/2

- Subscribe in components to react to each status of the request as needed (**loading**, **error** and **fetched**)

```
import { Component, inject, signal } from '@angular/core';
import { JsonPipe } from '@angular/common';
import { TodoService } from './todo-service';

@Component({
  selector: 'app-todo-list',
  template: `<pre>{{ todos() | json }}</pre>`,
  imports: [JsonPipe],
})
export class TodoList {
  private todoService = inject(TodoService);

  protected todos = signal<Todo[] | undefined>(undefined);

  constructor() {
    this.todoService.fetch().subscribe((todos) => this.todos.set(todos)); // <-- Do "subscribe" in
  }                                                                        //     component layer
}
```

# HTTP client - State management 1/3

- To share data between components, **store** the fetched data in the **service layer** by leveraging **RxJS operators**

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

@Injectable({
  providedIn: 'root'
})
export class TodoService {                                    // <-- Data source provider
  private httpClient = inject(HttpClient);

  private _todos = signal<Todo[] | undefined>(undefined);     // <-- Encapsulate data
  todos = this._todos.asReadonly();                           // <-- Expose data

  fetch(): Observable<Todo[]> {
    return this.httpClient
      .get<Todo[]>(TODOS_URL)
      .pipe(tap((todos) => this._todos.set(todos)));          // <-- RxJS operators to handle side effect
  }
}
```

# HTTP client - State management 2/3

- **Subscribe** in the **component layer**, consuming centralised data and **handling** potential **errors**

```
import { Component, inject, signal } from '@angular/core';
import { JsonPipe } from '@angular/common';
import { TodoService } from './todo-service';

@Component({
  selector: 'app-todo-list',
  templateUrl: 'todo-list.html',
  imports: [JsonPipe],
})
export class TodoList {                                    // <-- Data source consumer
  private todoService = inject(TodoService);

  todos = this.todoService.todos; // Data can be consumed here and in other components too...
  hasError = signal(false);

  constructor() {
    // ...while fetching data can be done in one strategic place
    this.todoService.fetch().subscribe({ error: () => this.hasError.set(true) });
  }
}
```

# HTTP client - State management 3/3

- Finally, in the component template, adapt the display according to the different statuses of the request (**loading**, **error** and **fetched**)

```
<!-- todo-list.html -->

@if (todos() === undefined) {

  <p>Initial loading...</p>

} @else if (hasError()) {

  <p>An error occurred...</p>

} @else {

  <pre>{{ todos() | json }}</pre>
}
```

# HTTP client - Testing 1/2

- Angular provides **`provideHttpClientTesting`** and **`HttpTestingController`** for mocking HTTP requests

```
import { provideHttpClient, withFetch } from '@angular/common/http';
import { provideHttpClientTesting, HttpTestingController } from '@angular/common/http/testing';
import { TestBed } from '@angular/core/testing';

describe('TodoService', () => {
  let service: TodoService;
  let httpTestingController: HttpTestingController;

  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [provideHttpClient(withFetch()), provideHttpClientTesting()],
    });

    service = TestBed.inject(TodoService);
    httpTestingController = TestBed.inject(HttpTestingController);
  });

  // ...
```

# HTTP client - Testing 2/2

- The Controller can be injected into tests and used for mocking and flushing requests

```
  // ...

  it('should fetch and store todos', () => {
    const responseMock: Todo[] = [{ id: 1 } as Todo, { id: 2 } as Todo];

    service.fetch().subscribe((todos) => {
      expect(todos).toEqual(responseMock);
      expect(service.todos()).toEqual(responseMock);
    });

    const req = httpTestingController.expectOne('https://jsonplaceholder.typicode.com/todos');
    expect(req.request.method).toEqual('GET');
    req.flush(responseMock);

    httpTestingController.verify(); // assert that there are no outstanding requests
  });
});
```

# HTTP client - Summary

**In this chapter on http client, we have covered the following topics**

- Provider
- Provider options
- Service
- Methods (RxJS Observables)
- Error handling
- State management

---

😉 *To go further, discover: Reactive data fetching with* `httpResource`

# HTTP client - Questions

# HTTP client - Lab 10

# Routing

# Table of Contents

# Routing

💡 In a single-page application (SPA), the browser only makes a request to a web server for the first page
After that, a client-side router takes over, controlling which content displays based on the URL

- The Angular router allows you to
  - → display different **views**
  - → at a defined **insertion point**
  - → depending on the **browser's URL**
- By default, the router is already provided in the `app.config.ts` file

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

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

# Routing - Routes

- Define the routes of your app by associating different **components** to different **paths** in the `app.routes.ts` file
  - → Define path **parameters** using the syntax `:paramName`
  - → Catch unknown paths using **wildcard** route (`**`)
    then redirect to a known path or display a dedicated "Not found" page

```ts
import { Routes } from '@angular/router';

export const routes: Routes = [
  { path: '', component: Home, pathMatch: 'full' },

  { path: 'contacts', component: ContactList },

  { path: 'contacts/:id', component: Contact },

  { path: '**', redirectTo: '/' },            // <-- Option 1. Redirect to home page
  // { path: '**', component: PageNotFound }, // <-- Option 2. Display "Not found" page
];
```

# Routing - RouterOutlet

- Define the insertion point using the `<router-outlet />` directive

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

@Component ({
  selector: 'app-root',
  imports: [RouterOutlet],
  template: `
    <header>My Awesome App</header>

    <main>
      <router-outlet />
    </main>

    <footer>Copyright - Zenika</footer>
  `
})
export class App {}
```

# Routing - RouterLink 1/3

- Navigate between views using the **`routerLink`** directive

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

@Component ({
  selector: 'app-nav',
  imports: [RouterLink],
  template:
   `<a routerLink="/"> Home </a>

    <a routerLink="/contacts"> Contact list </a>

    <a routerLink="/contacts/1"> Contact 1 </a>

    <a [routerLink]="['/contacts', id]"> Contact {{ id }} </a>`
})
export class Nav {
  id = 2;
}
```

😉 *Note: using **`href`** attribute instead of **`routerLink`** directive will trigger full-page reload*

# Routing - RouterLink 2/3

- Use `routerLinkActive` directive to specify one or more CSS classes to be added when the linked route is active

```
import { Component } from '@angular/core';
import { RouterLink, RouterLinkActive } from '@angular/router';

@Component ({
  selector: 'app-nav',
  imports: [RouterLink, RouterLinkActive],
  template: `
    <a routerLink="/" routerLinkActive="link-active"> Home </a>

    <a routerLink="/contacts" routerLinkActive="link-active"> Contact list </a>

    <a routerLink="/contacts/1" routerLinkActive="link-active"> Contact 1 </a>
  `,
  styles: `.link-active { color: blue }`,
})
export class Nav {}
```

# Routing - RouterLink 3/3

- Use `routerLinkActiveOptions` input to add the classes only when the URL matches the link exactly

```
import { Component } from '@angular/core';
import { RouterLink, RouterLinkActive } from '@angular/router';

@Component ({
  selector: 'app-nav',
  imports: [RouterLink, RouterLinkActive],
  template: `
    <a
      routerLink="/"
      [routerLinkActive]="['link-active']"
      [routerLinkActiveOptions]="{ exact: true }"
    >
      Home
    </a>
  `,
  styles: `.link-active { color: blue }`,
})
export class Nav {}
```

# Routing - Router

- Use the **`Router`** service to navigate programmatically on the component class side

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

@Component ({
  selector: 'app-root',
  template: '<button (click)="navigate()">Go to contact list</button>'
})
export class App {
  private router = inject(Router);

  protected navigate() {
    this.router.navigate(['/contacts']); // Same as <a [routerLink]="['/contacts']">Contacts</a>
  }
}
```

😉 *Whenever possible, prefer using the **`routerLink`** directive on the component template side*

# Routing - ActivatedRoute

- Use the **`ActivatedRoute`** service to observe route parameters

```ts
import { Component, inject } from '@angular/core';
import { ActivatedRoute, Params } from '@angular/router';
import { takeUntilDestroyed } from '@angular/core/rxjs-interop';

@Component ({
  template: 'Contact ID: {{ id }} (dynamic).'
})
export class Contact {
  private activatedRoute = inject(ActivatedRoute);

  id!: number;

  constructor() {
    this.activatedRoute.params.pipe(takeUntilDestroyed()).subscribe((params: Params) => {
      this.id = Number(params['id']); // note: route parameters are always of type `string`
    });
  }
}
```

😉 *Note that **`params`** is an RxJS Observable*

# Routing - ActivatedRoute | Snapshot

- Use the **`ActivatedRoute`** snapshot to retrieve route parameters once

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

@Component ({
  template: 'Contact ID: {{ id }} (static).'
})
export class Contact {
  private activatedRoute = inject(ActivatedRoute);

  id = Number(this.activatedRoute.snapshot.params['id']);
}
```

# Routing - With component input binding 1/2

*Using **`ActivatedRoute`** requires the understanding of observables*

- Add **`withComponentInputBinding()`** feature to the router configuration to enable binding information from the router state directly to the component's inputs

```
import { ApplicationConfig } from '@angular/core';
import { provideRouter, withComponentInputBinding } from '@angular/router';
import { routes } from './app.routes';

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

# Routing - With component input binding 2/2

- Define a **route parameter** named `id`

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

export const routes: Routes = [
  { path: 'contacts/:id', component: Contact }
];
```

- In the routed view, define a **component input** with the same name

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

@Component ({
  template: 'Contact ID: {{ id }} (dynamic).'
})
export class Contact {
  private activatedRoute = inject(ActivatedRoute);

  id = input.required<number>({ transform: numberAttribute });
}
```

# Routing - Route title

- Use the **`title`** property to define a unique title for each route, so that they can be identified in the browser history

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

export const routes: Routes = [
  {
    path: '',
    component: Home,
    title: 'Home',
  },
  {
    path: 'contacts',
    component: ContactList,
    title: 'Contacts',
  },
];
```

🎉 *At this point, we have covered the **basics** of Routing!*

👇 *The rest of the chapter is **optional** and covers **advanced concepts***

# Routing - Nested routes

- Use the **`children`** property to define nested views

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

export const routes: Routes = [
  {
    path: 'contacts/:id',
    component: Contact,
    children: [
      { path: 'view', component: ViewContact },
      { path: 'edit', component: EditContact },
    ],
  },
];
```

*For this example to work, the **`Contact`** component template must contain the nested **`<router-outlet />`** directive*

# Routing - Lazy Loading 1/3

- Configure your routes to lazy load modules using `loadComponent`

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

export const routes: Routes = [
  {
    path: 'contacts',

    // Use lazy-loaded JavaScript module...
    loadComponent: () => import('./contact-list/contact-list.ts').then(
      (module) => module.ContactList
    ),

    // ...instead of eagerly-loaded component
    /* component: ContactList, */
  },
];
```

# Routing - Lazy Loading 2/3

- Use `default` export to get rid of `.then((module) => ...)` part

```
@Component({
  selector: 'app-contact-list',
  template: `...`,
})
export class ContactList {}

export default ContactList;
```

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

export const routes: Routes = [
  {
    path: 'contacts',
    loadComponent: () => import('./contact-list/contact-list.ts'),
  },
];
```

# Routing - Lazy Loading 3/3

- Lazy load routes using `loadChildren`

```ts
// src/app/contacts/contacts.routes.ts
import { Routes } from '@angular/router';

export default [
  { path: '', component: ContactList },
  { path: ':id', component: Contact },
] satisfies Routes;
```

```ts
// src/app/app.routes.ts
import { Routes } from '@angular/router';

export const routes: Routes = [
  {
    path: 'contacts',
    loadChildren: () => import('./contacts/contacts.routes.ts'),
  },
];
```

# Routing - Guards

- Use route guards to **prevent users from navigating** to parts of an application **without authorization**
- Available route guards
  - → `canActivate`
  - → `canActivateChild`
  - → `canDeactivate`
  - → `canMatch`
  - → `resolve`

*In this course, we will focus on `canActivate` and `canMatch` guards*

# Routing - Guards | Can activate

- Define the guard by implementing the **`CanActivateFn`** interface

```
import { inject } from '@angular/core';
import { CanActivateFn, ActivatedRouteSnapshot } from '@angular/router';
import { ContactService } from './contact-service';

export const contactGuard: CanActivateFn = (snapshot: ActivatedRouteSnapshot) => {

  const id = snapshot.params['id']; // <-- Remember that the route path was: 'contacts/:id'

  return inject(ContactService).isAllowed(Number(id));
};
```

- Add the guard to the **`canActivate`** route configuration

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

export const routes: Routes = [
  { path: 'contacts/:id', component: Contact, canActivate: [contactGuard] }
];
```

# Routing - Guards | Can match

- Define the guard by implementing the **`CanMatchFn`** interface

```
import { inject } from '@angular/core';
import { CanMatchFn, Route, UrlSegment } from '@angular/router';
import { ContactService } from './contact-service';

export const contactGuard: CanMatchFn = (route: Route, segments: UrlSegment[]) => {

  const id = segments.at(1)?.path; // <-- Remember that the route path was: 'contacts/:id'

  return inject(ContactService).isAllowed(Number(id));
};
```

- Add the guard to the **`canMatch`** route configuration

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

export const routes: Routes = [
  { path: 'contacts/:id', component: Contact, canMatch: [contactGuard] },
  { path: 'contacts/:id', component: NoContactFallback }, // <-- activated when guard returns false
];
```

# Routing - Guards | Difference in behaviour

- `canActivate: [...]`
  - → If all guards return `true`, navigation continues
  - → If any guard returns `false`, navigation is **cancelled**
- `canMatch: [...]`
  - → If all guards return `true`, navigation continues
  - → If any guard returns `false`, navigation is **skipped** for matching and **next route configurations are processed** instead

# Routing - Guards | Redirect command

- The guard can eventually return a **`RedirectCommand`** to instruct the Router to redirect rather than continue processing the current path
- This is particularly useful when a navigation is cancelled by a **`canActivate`** guard

```
import { inject } from '@angular/core';
import { CanActivateFn, Router, RedirectCommand } from '@angular/router';

export const contactGuard: CanActivateFn = () => {
  const authService = inject(AuthService);
  const router = inject(Router);

  if (!inject(AuthService).isLoggedIn()) {
    const loginPath = router.parseUrl('/login');
    return new RedirectCommand(loginPath, { skipLocationChange: true });
  }

  return true;
};
```

# Routing - Summary

**In this chapter on routing, we have covered the following topics**

- **Basics**
  - Routes
  - RouterOutlet
  - RouterLink
  - Router
  - ActivatedRoute
  - withComponentInputBinding
  - Route title
- **Advanced concepts**
  - Nested routes
  - Lazy Loading
  - Guards

# Routing - Questions

# Routing - Lab 11

# Forms

# Table of Contents

# Forms - Modules 1/3

Angular provides two different ways to handle forms

- **Template-driven forms**
  - → the form is fully defined in the component *template*
  - → a TypeScript representation of the form is generated and managed by Angular
- **Reactive forms**
  - → the form is defined in the component *class*
  - → the form fields are then linked in the component template using property bindings
  - → you're responsible for ensuring the consistency of the form between the component and the template

# Forms - Modules 2/3

Any form can be created using either of the following techniques, but...

- **Template-driven forms**
  - → are recommended when form structure is not fixed over time
  - → example: fields are added/removed depending on a user's actions
- **Reactive forms**
  - → are recommended when you need to modify the form configuration programmatically over time
  - → example: changing a field validation requirement (from optional to required) depending on a user's actions

✅ The rest of this course focuses solely on Template-driven forms

# Forms - Modules 3/3

- Import the **`FormsModule`** in your components
- Use the available directives such as **`ngModel`**

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

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

- Under the hood, the **`ngModel`** directive tracks the **value**, user **interaction**, and **validation status** of the control element (such as an **`<input />`**) to which it is attached

# Forms - Getting started 1/2

- Angular reproduces the standard mechanisms of HTML forms...
- ...and supports common input types and their native validation attributes
- So, your template looks like something familiar!
- Here's a basic HTML form example with three fields:
  - → **`name`**, **`email`** (both required) and **`message`** (optional)

```
<form>

  <input name="name" placeholder="Your name" type="text" required />

  <input name="email" placeholder="Your email" type="email" required />

  <textarea name="message" placeholder="Leave us a message (optional)"></textarea>

  <button type="submit">Submit</button>
</form>
```

# Forms - Getting started 2/2

- In a component template, a **`<form>`** element defines an Angular form
  - → Angular automatically adds the **`ngForm`** directive to it
  - → so, don't add it manually!
- To register form fields such as **`<input />`**, you need to manually add the **`ngModel`** directive
  - → the **`name`** attribute is mandatory to register the field in the form

```
<form> <!-- Under the hood, it looks like: `<form ngForm>` -->

  <input ngModel name="name" placeholder="Your name" type="text" required />

  <input ngModel name="email" placeholder="Your email" type="email" required />

  <textarea ngModel name="message" placeholder="Leave us a message (optional)"></textarea>

  <button type="submit">Submit</button>
</form>
```

# Forms - Accessing ngForm & ngModel 1/2

- You can create template reference variables using the # symbol to access the underlying directives

```
<form #userForm="ngForm">

  <input #emailModel="ngModel" ngModel name="email" />

</form>
```

- Here, the template variable **`userForm`** holds the **`NgForm`** directive instance
- And the template variable **`emailModel`** holds the **`NgModel`** directive instance

These variables are very important and we will be using them throughout this chapter

😉 *But for now, let's look at where the names of the values `xyz="ngForm"` and `xyz="ngModel"` come from...*

# Forms - Accessing ngForm & ngModel 2/2

When creating a custom directive, you can define the **`exportAs`** metadata and use the defined value to access the directive instance in your template

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

@Directive({ selector: 'appHello', exportAs: 'helloExportedName' })
export class Hello {}

@Component({
  selector: 'app-root',
  imports: [Hello],
  template: '<div appHello #myDirective="helloExportedName" #myDiv></div>',
})
export class App {}
```

- Here, the template variable **`myDirective`** holds the **`Hello`** directive instance
- While the template variable **`myDiv`** simply holds the **`HTMLDivElement`** instance (default)

😉 *...so you've guessed that the **`NgModel`** directive metadata contains: **`{exportAs: 'ngModel'}`***

# Forms - NgModel 1/4

Let's take a closer look at the `NgModel` directive

- Works even outside a `<form>` element (**`name`** attribute is not mandatory in this case)
- Provides access to several **properties** reflecting the **state of the form field**

  → `untouched`/`touched`, `pristine`/`dirty`, `valid`/`invalid`

```
@Component({
  selector: 'app-root',
  imports: [FormsModule],
  template: `
    <input required ngModel #model="ngModel" />

    <p>The field is {{ model.valid ? 'valid' : 'invalid' }}.</p>
  `,
})
export class App {}
```

# Forms - NgModel 2/4

- Adds special **CSS classes** that reflect the state of the form field

  → `ng-untouched`/`ng-touched`, `ng-pristine`/`ng-dirty`, `ng-valid`/`ng-invalid`

```
@Component({
  selector: 'app-root',
  imports: [FormsModule],
  template: `
    <!-- 1. Initial state -->
    <input required ngModel class="ng-untouched ng-pristine ng-invalid" />

    <!-- 2. After the user has entered and left the input (without modification) -->
    <input required ngModel class="ng-touched ng-pristine ng-invalid" />

    <!-- 3. After the user has modified the input value -->
    <input required ngModel class="ng-touched ng-dirty ng-valid" />
  `,
  styles: [`.ng-valid{ color: green; }   .ng-touched.ng-invalid{ color: red; }`],
})
export class App {}
```

# Forms - NgModel 3/4

- You can also define **your own CSS classes** and bind them using the `NgModel` properties

```
@Component({
  selector: 'app-root',
  imports: [FormsModule],
  template: `
    <input
      required

      ngModel
      #model="ngModel"

      [class.is-valid]="model.valid"
      [class.is-invalid]="model.touched && model.invalid"
    />
  `,
  styles: [`.is-valid { color: green; }   .is-invalid { color: red; }`],
})
export class App {}
```

# Forms - NgModel 4/4

- Lets you achieve **two-way data binding** easily

```
@Component({
  selector: 'app-root',
  imports: [FormsModule],
  template: `
    <div>{{ data }}</div>

    <input [(ngModel)]="data" />

    <input [ngModel]="data" (ngModelChange)="data = $event" />

    <input #inputRef [value]="data" (input)="data = inputRef.value" />
  `,
})
export class App { data = ''; }
```

# Forms - NgForm 1/2

Now let's take a closer look at the `NgForm` directive

**Problem**

- By default, browsers perform natively form fields validation
- But Angular needs to take full control over this process
- Native mechanism will therefore conflict with Angular mechanism

**Solution**

- Angular disables native validation by adding `novalidate` attribute automatically

  → so, don't add it manually!

```
<form></form> <!-- will become `<form novalidate></form>` in the DOM -->
```

# Forms - NgForm 2/2

- Use the **`ngSubmit`** event to handle form submission
- Use the NgForm **`.value`** property to retrieve the entire form value as an object
- Use the NgForm **`.invalid`** (or **`.valid`**) property to determine the global form state

```
@Component({
  selector: 'app-root',
  imports: [FormsModule],
  template:
    `<form #userForm="ngForm" (ngSubmit)="submitForm(userForm.value)">

      <input ngModel name="name" required />
      <input ngModel name="email" type="email" required />
      <textarea ngModel name="message"></textarea>

      <button type="submit" [disabled]="userForm.invalid">Submit</button>
    </form>`,
})
export class App {
  submitForm(userFormValue: { name: string; email: string; message: string }) { /* ... */ }
}
```

# Forms - Validators 1/3

- A form field may have one or more validators
- As we said, Angular supports all HTML5 standard validators:
  - → `required`, `minlength`, `maxlength`, `min`, `max`, `type` and `pattern`
- But you can create custom validators too
  - → we'll come back to this later...

# Forms - Validators 2/3

- Use the **`.errors`** property on the **`NgModel`** directive to track the validation errors
- Here's an example with a form field that is *required* and must be a *valid email*

```
<input ngModel #emailModel="ngModel" required type="email" />

"{{ emailModel.errors | json }}"

<!--
  Depending on the field value, output might be:
    - "null"
    - "{ required: true }"
    - "{ email: true }"
-->
```

# Forms - Validators 3/3

- Use the `.hasError` method on the `NgModel` directive to check the presence of a particular error

```
<input ngModel #emailModel="ngModel" required type="email" />

@if (emailModel.hasError('required')) {

  <span style="color:red">
    The email is required.
  </span>

} @else if (emailModel.hasError('email')) {

  <span style="color:red">
    The given email is not valid.
  </span>

}
```

# Forms - Validators | Custom 1/2

- To create a custom validator, you need a **`Directive`** that implements the **`Validator`** interface

```
import { Directive, input } from '@angular/core';
import { AbstractControl, NG_VALIDATORS, ValidationErrors, Validator } from '@angular/forms';

@Directive({
  selector: '[appStartWith][ngModel]',
  providers: [{
    provide: NG_VALIDATORS, useExisting: StartWith, multi: true
  }],
})
export class StartWith implements Validator {
  startWith = input.required<string>({ alias: 'appStartWith' });

  validate(control: AbstractControl): ValidationErrors | null {
    if (typeof control.value !== 'string' || !control.value.startsWith(this.startWith())) {
      return { startWith: this.startWith() };   // <-- raise a validation error
    }
    return null;                                // <-- no error
  }
}
```

# Forms - Validators | Custom 2/2

- Here's an example of how to use this custom validator

```
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { StartWith } from './starts-with';

@Component({
  selector: 'app-root',
  imports: [FormsModule, StartWith],
  template: `
    <form>
      <input name="example" ngModel #model="ngModel" appStartWith="xyz" />

      @if (model.getError('startWith'); as expectedValue) {
        <span style="color: red">
          The value should start with: {{ expectedValue }}.
        </span>
      }
    </form>`,
})
export class App {}
```

# Forms - Summary

**In this chapter on forms, we have covered the following topics**

- Template-driven forms
- NgForm
- NgModel
- Template reference variables
- Built-in validators
- Custom validators

# Forms - Questions

# Forms - Lab 12


PHP
CSS
HTML
<HTML>
<HEAD>
<TITLE> Lorem ipsum </TITLE>
</HEAD>
<BODY>
<H1> Lorem ipsum dolor sit amet, consect adipiscing elit, sed diam</H1>
<P> <IMG SRC= "Lorem ipsum"></P>

# More on Components

*— appendix —*

# Table of Contents

# More on Components - View encapsulation 1/3

- By default, **component's styles are encapsulated** within the component's host element so that they **don't affect the rest of the application**

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

@Component ({
  selector: 'app-root',
  template: `<h1>Hello world</h1>`,
  styles: `h1 { color: red }`,
  encapsulation: ViewEncapsulation.Emulated, // <-- Default value
})
export class App {}
```

- At runtime, Angular adds **unique attributes** to achieve encapsulation

```
h1[_ngcontent-ng-529479] { color: blue }
```

```
<app-root _nghost-ng-529479>
  <h1 _ngcontent-ng-529479>Hello world</h1>
</app-root>
```

# More on Components - View encapsulation 2/3

- Use `:host {}` pseudo class to style the component's host element

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

@Component ({
  selector: 'app-root',
  template: `<h1>My Awesome App</h1>`,
  styles: `:host { display: block }`,
})
export class App {}
```

- At runtime, Angular transforms the pseudo class into **unique attributes**

```
[_nghost-ng-529479] { display: block }
```

```
<app-root _nghost-ng-529479>
  <h1 _ngcontent-ng-529479>Hello world</h1>
</app-root>
```

# More on Components - View encapsulation 3/3

- If needed, use **`ViewEncapsulation.None`** to disable component's encapsulation
- Then, all styles defined in the component are global and can therefore affect the entire page
  - → use with caution
  - → use fairly unique CSS selectors

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

@Component ({
  selector: 'app-root',
  template: `<h1 class="app-root__title">Hello world</h1>`,
  styles: `
    h1 { color: red }                 /* ❌ Looks dangerous, affects all <h1> tags in the page */

    .app-root__title { color: red }   /* ✅ Looks fine, uses a fairly unique CSS selector */
  `,
  encapsulation: ViewEncapsulation.None,
})
export class App {}
```

# More on Components - Content projection 1/3

- Allows to put HTML content inside the tag of an Angular component
- The `<ng-content />` element acts as a placeholder to mark where projected content should go

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

@Component ({ selector: 'app-root', template:
  `<app-card>
    <header>Title</header>
    <section>Content</section>
  </app-card>`
})
export class App {}
```

# More on Components - Content projection 2/3

- Ability to have multiple insertion points using the `select` property
- The select value must be a valid **CSS selector** targeting the HTML fragment to be used

```
@Component({ selector: 'app-card', template:
  `<article>
    <header> <ng-content select="[card-title]" /> </header>
    <section> <ng-content select="[card-content]"/> </section>
  </article>`
})
export class Card {}

@Component ({ selector: 'app-root', template:
  `<app-card>
    <span card-title>Title</span>
    <span card-content>Content</span>
  </app-card>`
})
export class App {}
```

# More on Components - Content projection 3/3

- Use `<ng-container>` to avoid adding unnecessary tags

```
@Component({ selector: 'app-card', template:
  `<article>
    <header> <ng-content select="[card-title]" /> </header>
    <section> <ng-content select="[card-content]"/> </section>
  </article>`
})
export class Card {}

@Component ({ selector: 'app-root', template:
  `<app-card>
    <ng-container card-title>Title</ng-container>
    <ng-container card-content>Content</ng-container>
  </app-card>`
})
export class App {}
```

# More on Components - Lifecycle 1/4

- It is possible to execute code using component lifecycle hooks
- More info: https://angular.dev/guide/components/lifecycle

```
import {
  Component, OnChanges, OnInit, AfterContentInit, AfterViewInit, OnDestroy, SimpleChanges
} from '@angular/core';

@Component ({/* ... */})
export class App implements
  OnChanges, OnInit, AfterContentInit, AfterViewInit, OnDestroy {

    constructor() {/* Perform tasks that do NOT depend on the component's inputs */}

    ngOnInit(): void {/* Perform tasks that depend on the component's inputs */}

    ngAfterContentInit(): void {/* ... */}

    ngAfterViewInit(): void {/* ... */}

    ngOnDestroy(): void {/* ... */}
  }
```

# More on Components - Lifecycle 2/4

- **`OnInit`** lifecycle hook is frequently used for initialization
- because you can safely read component **`inputs`** when this hook is triggered

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

@Component ({/* ... */})
export class Posts implements OnInit {
  userId = input.required<string>();

  protected posts?: Post[];

  ngOnInit() {
    // Doing this in the `constructor` will fail!
    // Because the property `userId` is `undefined` at the time the constructor is executed.
    this.fetchUserPosts(this.userId()).then((posts) => (this.posts = posts));
  }

  private fetchUserPosts(): Promise<Post[]> {/* ... */}
}
```

# More on Components - Lifecycle 3/4

- `OnDestroy` lifecycle hook is frequently used for cleaning up the component

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

@Component ({
  selector: 'app-interval',
  template: '<p>{{ data }}</p>'
})
export class Interval implements OnDestroy {
  protected data = 0;

  private interval = setInterval(() => this.data++, 1000);

  ngOnDestroy() {
    clearInterval(this.interval);
  }
}
```

# More on Components - Lifecycle 4/4

- `DestroyRef` allows you to achieve the same result as `ngOnDestroy`

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

@Component ({
  selector: 'app-interval',
  template: '<p>{{ data }}</p>'
})
export class Interval {
  protected data = 0;

  private interval = setInterval(() => this.data++, 1000);

  constructor() {
    inject(DestroyRef).onDestroy(() => clearInterval(this.interval));
  }
}
```

😉 *It is considered a more modern approach*

# More on Components - Queries 1/2

- It is possible to access template details from the class using **`viewChild`**
- Retrieved information is available as soon as **`AfterViewInit`** has been triggered

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

@Component({
  selector: 'app-hello', template: `<h1>Hello world!</h1>`
})
export class Hello {}

@Component({
  selector: 'app-root', template: `<app-hello />`
})
export class App implements OnInit, AfterViewInit {

  hello = viewChild(Hello);

  ngOnInit() { console.log(this.hello()); }              // <-- output: undefined
  ngAfterViewInit() { console.log(this.hello()); }       // <-- output: Hello
}
```

# More on Components - Queries 2/2

- `afterNextRender` allows you to achieve (almost) the same result as `AfterViewInit`
- Invoked the next time the application finishes rendering

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

@Component({
  selector: 'app-hello', template: `<h1>Hello world!</h1>`
})
export class Hello {}

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

  hello = viewChild(Hello);

  constructor() {
    afterNextRender(() => console.log(this.hello()));  // <-- output: Hello
  }
}
```

# More on Components - Questions

# <app-end />