ANGULAR

SUMMARY

- Introduction
- Reminders
- Getting started with Angular
- Components
- Unit testing
- Directives
- Services
- Pipes
- Http
- Router
- Forms

LOGISTICS

- Schedules
- Lunch & breaks
- Other questions ?

INTRODUCTION

SUMMARY

- *Introduction*
- Reminders
- Getting started with Angular
- Components
- Unit testing
- Directives
- Services
- Pipes
- Http
- Router
- Forms

ANGULAR - HISTORY

- Framework created by Google and announced in 2014
- Total rewrite of AngularJS, although some concepts remain
- First release of Angular 2 in September 2016
- Last major version 16 released in May 2023
- Component oriented framework
- Documentation: http://angular.io/

ANGULAR - VERSIONS 1/2

- Major release every 6 months

Version	Date	Description
2.0.0	2016/09	Final version
4.0.0	2017/03	New template compilation engine, Modularization of the animation system, Universal project, TypeScript 2.1+
5.0.0	2017/11	Improvement of the build (AOT), HttpClient, TypeScript 2.3
6.0.0	2018/05	CLI Integration, Angular Element, New experimental Ivy renderer
7.0.0	2018/10	CLI Prompts, Virtual Scroll, Drag and Drop, Angular Element
8.0.0	2019/05	Differential Loading, Dynamic Import, Builders API, Ivy, Bazel
9.0.0	2020/02	Ivy by default, ProvidedIn scope

ANGULAR - VERSIONS 2/2

Version	Date	Description
10.0.0	2020/06	Optional Stricter Settings, New Default Browser Configuration, TypeScript 3.9
11.0.0	2020/11	TypeScript 4.0, Remove deprecated support for IE 9, 10, and IE mobile
12.0.0	2021/05	Stylish improvements, nullish coalescing, Webpack 5 support, TypeScript 4.2
13.0.0	2021/11	Ivy only remove Old View Engine, Cli Cache, RxJS v7, TypeScript 4.4
14.0.0	2022/06	Strictly Typed Reactive Forms, Standalone Components with Optional NgModule
15.0.0	2022/12	Standalone API stable, Directive Composition API, Image Directive, Functional Router Guards
16.0.0	2023/05	Angular Signals, Bind router Information to Component Inputs, Required Component Inputs, Non-Destructive Hydration

ANGULAR - PROS

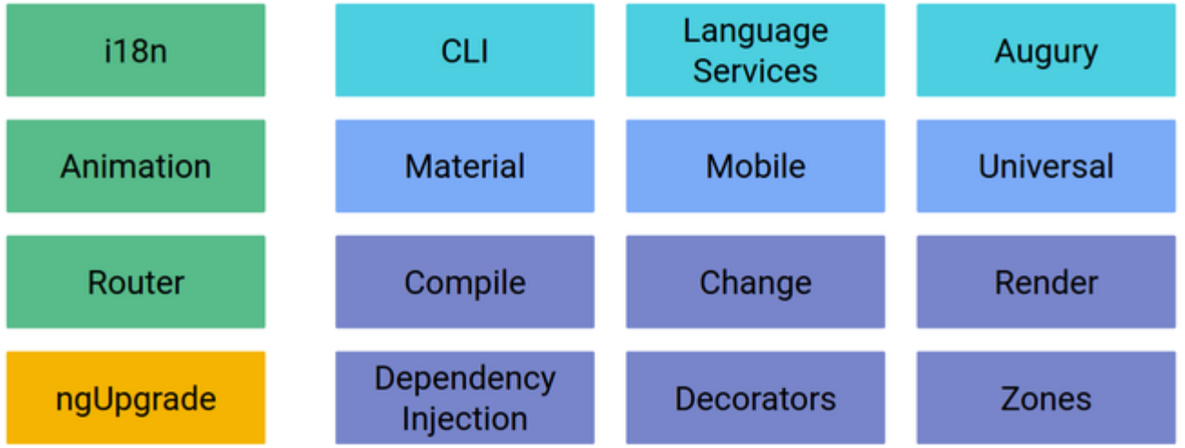
- 👍 Maintained by Google
- 👍 Component-based architecture
- 👍 Use plain HTML templates
- 👍 Two-way data binding
- 👍 Efficient testing
- 👍 Easy upgrade to new versions
- 👍 Powerful CLI

ANGULAR - CONS

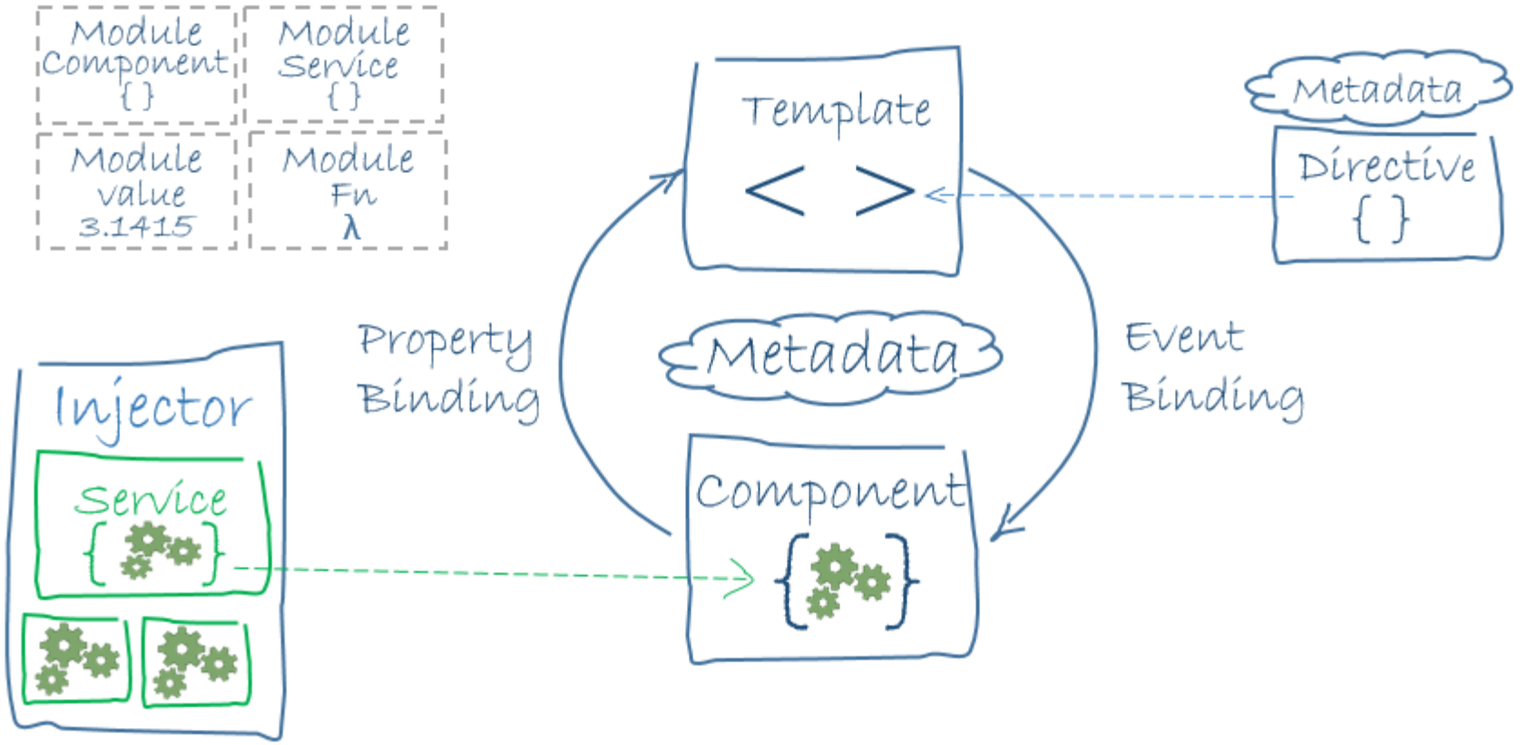
- 👎 Steep learning curve
- 👎 Requires new concepts to learn (`Zone`, `Observable`, ...)

ANGULAR - FRAMEWORK

- Angular, unlike Vue or React, is a complete Framework



ANGULAR - ARCHITECTURE 1/2



ANGULAR - ARCHITECTURE 2/2

- Main part
  - `Component`: TypeScript class that describes the component behavior
  - `Template`: HTML code that describes the end-user view
  - `Metadata`: Links the template and the component
- Other parts
  - `Directive`: Additional behavior that can be used in the component's template (`ngFor`, `ngIf`, ...)
  - `Pipe`: Transform strings, currency amounts, dates, and other data for display
  - `Service`: Business code implemented in classes that can be injected into the components, directives, other services, ...
  - `Injector`: Angular dependency injection system
  - `Module`: Grouping a set of features

ANGULAR - COMPONENT EXAMPLE

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

// Usage: <app-likes [numberOfLikes]="3" [like]="false" (likeChange)="likeChanged($event)" />
@Component({
  selector: 'app-likes',
  template: `
    <button (click)="toggleLike()">
      {{ numberOfLikes + (like ? 1 : 0) }}
      <i class="icon" [class.liked]="like"> ♥ </i>
    </button>
  `,
})
export class LikesComponent {
  @Input() public numberOfLikes = 0;
  @Input() public like = false;
  @Output() public likeChange = new EventEmitter<boolean>();

  protected toggleLike() {
    this.like = !this.like;
    this.likeChange.emit(this.like);
  }
}
```

VANILLA JS - COMPONENT EXAMPLE

- Open the file `Exercises/resources/likes-component-vanilla-js/index.html` in Chrome
- Understand the basics of:
  - DOM creation
  - DOM manipulation
  - Event handling
- Don't be afraid, it's not that hard! 😱
- Appreciate the abstraction layer provided by the Angular framework 😍

REMINDERS

SUMMARY

- Introduction
- *Reminders*
- Getting started with Angular
- Components
- Unit testing
- Directives
- Services
- Pipes
- Http
- Router
- Forms

HTML

- Basic HTML document is made of:
  - Tags
  - Attributes

```html
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Hello world!</title>
  </head>

  <body>
    <h1>Hello world!</h1> <!-- Tag with content -->

    <img src="/path/to/image.jpg" alt="Awesome image" /> <!-- Self closing tag -->
  </body>
</html>
```

CSS

- Basic CSS file is made of:
  - Selectors
  - Rules (any number of property/value pairs)
- The following HTML fragment (`*.html`)...

```
<my-tag class="my-class" my-attr>My content</my-tag>
```

- ...can be styled using the following (`*.css`)

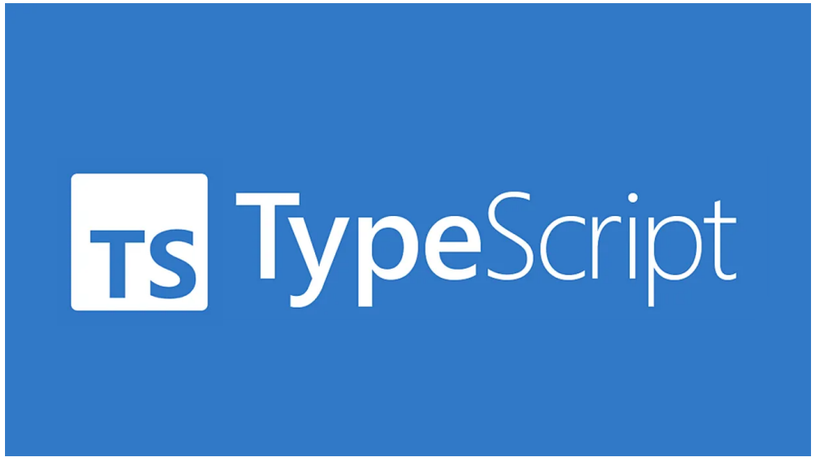
```
my-tag { border: 1px solid green; } /* Tag selector */

.my-class { background-color: lightgreen; } /* Class selector */

[my-attr] { color: yellow; } /* Attribute selector */
```

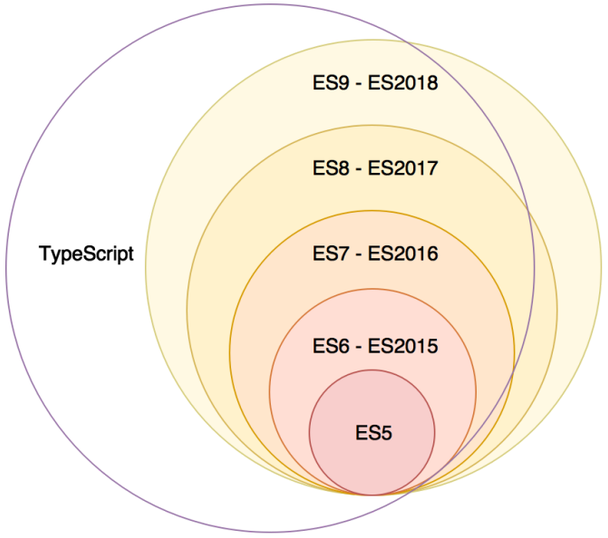
TYPESCRIPT

- Language created by Anders Hejlsberg in 2012
- Open-source project maintained by Microsoft
- Influenced by Java and C#



TYPESCRIPT

- TypeScript is a superset of JavaScript
- Compiles to JavaScript
- Supports all versions of JavaScript
- Almost every JavaScript program is a TypeScript program



TYPESCRIPT - FEATURES

- Types
- Interfaces
- Generics
- Decorators
- Definitions files
- ...

TYPESCRIPT - BASIC TYPES

Two ways to define variables: `const` and `let` (don't use `var`)

```
const alwaysTrue: boolean = true;
let age: number = 32;
age = 33;
```

There are several places where type inference is 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;
```

Some other types:

```
const name: string = 'Carl';
const names1: string[] = ['Carl', 'Laurent'];
const names2: Array<string> = ['Carl', 'Laurent'];
const notSure: any = 4; // <-- should be avoided
```

TYPESCRIPT - FUNCTIONS

- As in JavaScript: named, anonymous and arrow functions
- TypeScript allows typing for arguments and return value

```
function sayHello(message: string): void {}

const sayHello = function(message: string): void {};

const sayHello = (message: string): void => {};
```

- Define default value parameter with =
- Define optional parameter with ?
- Use `return` keyword to return a value

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

TYPESCRIPT - CLASSES

- Classes and Interfaces are similar to those in Object Oriented Programming
- Classes are composed of one constructor, properties and methods
- Explicitly defining a constructor is optional
- Properties and methods are accessible with `this` operator

```
class Person {
  name = '';

  constructor() {} // this is optional

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

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

TYPESCRIPT - CLASSES

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

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

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

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

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

TYPESCRIPT - CLASSES

- TypeScript provides a shortcut to link constructor arguments to class properties:

```
class Person {
  constructor(public firstName: string) {}
}
```

Which is equivalent to:

```
class Person {
  public firstName: string;

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

TYPESCRIPT - CLASSES

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

```
class Person {
  constructor(public firstName: string, public lastName: string) {}

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

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

const person = new Person('John', 'Doe');
console.log(person.fullName); // --> John Doe

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!');
  }
}
```

- Compiler throw an error while the interface contract is not respected
- Have no impact on generated JavaScript

TYPESCRIPT - GENERICS

- Similar to generics in Java or C#
- Generic functions/variables/classes/interfaces need typing at instantiation

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

const numericLog = new Log<number>();

numericLog.log(5); // Correct

numericLog.log('hello'); // Incorrect
```

NPM - NODE PACKAGE MANAGER

- Included in Node.js
- The main way to share modules in JavaScript
- The most popular package manager of all time!

NPM - COMMANDS

- Set up a folder as an npm package by creating a `package.json` file

```
npm init
```

- Download a module and install it in `./node_modules` directory

```
npm install <packageName>
```

- Install a module globally on your system (mostly used to install CLI tools)

```
npm install -g <packageName>
```

- Update/delete a package

```
npm update <packageName>
npm remove <packageName>
```

NPM - PACKAGE.JSON

- npm generate a `package.json` file which describes the project:
  - `name`: package name
  - `version`: package version
  - `scripts`: commands that can be run from the command line
  - Dependencies: `dependencies`, `devDependencies`, `peerDependencies`
  - ...

```
{
  "name": "<packageName",
  "version": "1.2.3",
  "scripts": {},
  "dependencies": {},
  "devDependencies": {}
}
```

NPM - PACKAGE.JSON | SCRIPTS

- Scripts are defined in the `"scripts"` section of the file:

```
{
  "scripts": {
    "start": "<shellCommand>",
    "test": "<shellCommand>",
    "my-awesome-script": "<shellCommand>",
  }
}
```

- And can be run with the command `npm run <scriptName>`:

```
npm run start              # alias: `npm start`
npm run test               # alias: `npm test`
npm run my-awesome-script
```

NPM - PACKAGE.JSON | DEPENDENCIES

- `dependencies`:
  - Required to run your project
- `devDependencies`:
  - Required to develop your project
  - Installed with the option: `npm install --save-dev <packageName>`
- `peerDependencies`:
  - Needed for some modules to work but not installed with `npm install`
  - Typically used for libraries

NPM - PACKAGE.JSON | VERSIONING

`package.json` versions must follow the semver (semantic versioning) standard

```
{
  "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

Example:

```
{
  "name": "my-awesome-package",
  "version": "1.2.3"
}
```

NPM - PACKAGE.JSON | VERSIONING

Allowing a range of versions when installing a package

- `1.2.3`: Install the exact version
- `~1.2.3`: Install any patch like `1.2.4`, `1.2.5`, ..., `1.2.9`
- `^1.2.3`: Install any minor version like `1.2.3`, `1.3.0`, ..., `1.9.0`
- `1.2.x`: `x` acts as a wildcard (equivalent to `~1.2.0`)

There are many other ways to set the version range using `<`, `>`, `>=`, `min-max` ...

GETTING STARTED WITH ANGULAR

SUMMARY

- Introduction
- Reminders
- *Getting started with Angular*
- Components
- Unit testing
- Directives
- Services
- Pipes
- Http
- Router
- Forms

HANDS ON!

For this section, let's start with Lab 1, then return to the slides.

You are about to:

- Setting up your environment
- Creating and running your Angular application
- Taking control of your application

Lab 1

FILE STRUCTURE

Back to slides, let's see how the application folder is structured:

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

This section will give you the big picture of how Angular works!

FILE STRUCTURE - PACKAGE.JSON

The presence of the `package.json` file makes this folder an NPM package powered by the Node.js runtime.

- Scripts can be run using the shell command `npm run <scriptName>`

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

FILE STRUCTURE - PACKAGE.JSON

- Dependencies of the Angular framework are scoped under `@angular/*`

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

FILE STRUCTURE - PACKAGE.JSON

- Angular also depends on some third-party libraries

```
{
  "dependencies": {
    "rxjs": "...",
    "tslib": "...",
    "zone.js": "..."
  },
  "devDependencies": {
    "@types/jasmine": "...",
    "jasmine-core": "...",

    "karma": "...",
    "karma-chrome-launcher": "...",
    "karma-coverage": "...",
    "karma-jasmine": "...",
    "karma-jasmine-html-reporter": "...",

    "typescript": "..."
  }
}
```

FILE STRUCTURE - TSCONFIG.JSON

- TypeScript is a primary language for Angular application development
- Browsers can't execute TypeScript directly
- Typescript must be "transpiled" into JavaScript using the `tsc` compiler
- The compiler requires some configuration described in the `tsconfig.json` file

```
{
  "compilerOptions": {
    "baseUrl": "./",
    "outDir": "./dist/out-tsc",
    ...
  },
  "angularCompilerOptions": {
    "strictInputAccessModifiers": true,
    "strictTemplates": true,
    ...
  }
}
```

FILE STRUCTURE - ANGULAR.JSON

- 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": {
      "root": "",
      "sourceRoot": "src",
      "projectType": "application",
      "prefix": "app",
      "schematics": {},
      "architect": {
        "build": {},
        "serve": {},
        "test": {},
        ...
      }
    }
    ...
  }
}
```

FILE STRUCTURE - ANGULAR.JSON

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

```
{
  "projects": {
    "zenika-ng-website": {
      "architect": {
        "build": {
          "options": {
             "outputPath": "dist/zenika-ng-website",
              "index": "src/index.html",
              "main": "src/main.ts",
              "polyfills": ["zone.js"],
              "tsConfig": "tsconfig.app.json",
              "assets": ["src/favicon.ico", "src/assets"],
              "styles": ["src/styles.css"],
              "scripts": []
          }
        }
      }
    }
  }
}
```

FILE STRUCTURE - SRC/APP/*

- `index.html`: final document of the Single Page Application (SPA)
- `main.ts`: entry point of the app
- `app/app.module.ts`: main module of the app
- `app/app.component.*`: main component of the app (the one used to bootstrap the app)
- `styles.css`: global styles of the app
- `assets/*`: 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
```

Under the hood, the command uses a bundler called `Webpack`.

WEBPACK

- Static module bundler
- Supports the different module systems (CommonJS, AMD, ES2015, ...)
- Available on NPM: `npm install -g webpack`
- Build a graph of all the dependencies of your application
- Uses a configuration file: `webpack.config.js`
  - `Entry`: indicates which module webpack should use to begin building out its internal dependency graph
  - `Output`: tells webpack where to emit the bundles it creates
  - `Loaders`: allow webpack to process any type of file like '.css', '.ts' (out of the box, only '.js' and '.json' are supported)
  - `Plugins`: perform tasks on multiple files at once like bundle optimization (whereas loaders operate at file level)

WEBPACK - CONFIGURATION EXAMPLE

```
// webpack.config.js
module.exports = {
  entry: './src/index.ts',

  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist'),
  },

  resolve: {
    extensions: ['.tsx', '.ts', '.js'],
  },

  module: {
    rules: [{ test: /\.tsx?$/, use: 'ts-loader', exclude: /node_modules/ }],
  },

  plugins: [new HtmlWebpackPlugin({ template: './src/index.html' })],
};
```

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

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
ng g s basket

# 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
```

COMPONENTS

SUMMARY

- Introduction
- Reminders
- Getting started with Angular
- *Components*
- Unit testing
- Directives
- Services
- Pipes
- Http
- Router
- Forms

COMPONENTS

- Defined with the `@Component` decorator on a class
  - 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 HelloComponent {}

@Component({
  selector: 'app-root',
  template: `
    <h1>My Awesome App</h1>
    <app-hello></app-hello> <!-- '<app-hello />' also works from Angular v15.1.0 -->
  `,
})
export class AppComponent {}
```

COMPONENT - TEMPLATE

- The template can be configured in two ways:
  - using a `template` property: string literal (as we saw in the previous slide)
  - using a `templateUrl` property: path to an HTML file (relative to the component)

```
// app.component.ts

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

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

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

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

COMPONENT - STYLES

The styles can be configured in two ways:

- using a `styles` property: array of string literal

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

- using a `styleUrls` property: array of path to CSS files

```
@Component ({
  styleUrls: ['./app.component.css']
})
export class AppComponent {}
```

```
/* app.component.css */
h1 { font-weight: normal; }
```

COMPONENT - STYLES

- You can style the component host element using the syntax `:host {}`

```
@Component ({
  selector: 'app-hello',
  template: 'Hello world!',
  styles: [`
    :host {
      display: block;
      background-color: yellow;
    }
  `]
})
export class HelloComponent {}
```

- Styles will be correctly applied in HTML rendering:

```
<app-hello styles="display: block; background-color: yellow;" />
```

TEMPLATE SYNTAX - STRING INTERPOLATION

- Uses the syntax `{{ expression }}`
- The `expression` is converted into a `string` and displayed as such
- Angular defines a precise syntax for these expressions
  - https://angular.io/guide/template-syntax#template-expressions:
  - can be almost any JavaScript expression - with some exceptions
- All `public` or `protected` component properties can be used in the template
- An expression used in template must not change the component state

```
@Component ({
  selector: 'app-product',
  template: `<p><img src="{{ product?.photo }}" /> {{ product?.title }}</p>`
})
export class ProductComponent {
  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-hero-form [hero]="currentHero" /> <!-- property of a component -->

<p [class.highlight]="isHighlight">Hello</p> <!-- special case -->

<button [style.color]="isHighlight? 'orange': 'black'">Save</button> <!-- special case -->
```

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"!
- The most common cases: `colspan`, `rowspan`, `aria-*`, ... for example

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

```
<td colspan="3">OK</td>
```

```
<td [attr.colspan]="expression">OK</td>
```

```
<td [colspan]="expression">NOT OK</td>

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

TEMPLATE SYNTAX - EVENT BINDING

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

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

<app-hero-form (deleted)="onHeroDeleted()" /> <!-- event of a component -->

<input (keyup.enter)="onEnter()" /> <!-- special case: pseudo events -->
```

TEMPLATE SYNTAX - EVENT BINDING

- Angular provides access to the event via the variable `$event` (can be used in expression)
- Native events are propagated to parent elements (event bubbling)
  - To stop the propagation, return `false` in the expression that processes the event
- Angular component events never propagate to parent elements!

Example of using `$event`:

```
<input [value]="firstName"
       (input)="firstName = $event.target.value" />
```

- `$event` refers to the native browser DOM `InputEvent`
- We achieve a two-way data binding using both property and event bindings
  - The class property `firstName` and the input `value` in the template will always be in sync

COMPONENT - INPUT

- `@Input()` decorator on a property of the component class
- The property name will be what you will use in the template

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

@Component ({
  selector: 'app-hello',
  template: `Hi {{ name }}!`
})
export class HelloComponent {
  @Input({ required: true }) name!: string;
}

@Component ({
  selector: 'app-root',
  template: `<app-hello [name]="userName" />` // <-- Hello John!
})
export class AppComponent {
  protected userName = 'John';
}
```

COMPONENT - OUTPUT

- `@Output()` decorator on a property (of type `EventEmitter`) of the component class
- The property name will be what you will use in the template

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

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

  @Output() increment = new EventEmitter<number>();

  onClick() { this.count += 1; this.increment.emit(this.count); }
}

@Component ({ selector: 'app-root', template: `<app-counter (increment)="log($event)" />` })
export class AppComponent {
  protected log(count: number) { console.log('Count:', count); }
}
```

COMPONENT - TWO-WAY DATA BINDING

- Convention: when `@Output` is named like `@Input` but with the suffix: `"Change"`
- Use the "Banana in a box" [🍌] syntax
- Synchronize properties between parent and child components

```
@Component ({ selector: 'app-counter', template:
  `<button (click)="countChange.emit(count = count + 1)">{{ count }}</button>`
})
export class CounterComponent {
  @Input()  count!: number;

  @Output() countChange = new EventEmitter<number>(); // <-- "[Output]" == "[Input]Change"
}

@Component ({ selector: 'app-root', template:
  `<app-counter [(count)]="appCount" />
   <button (click)="appCount = appCount + 1">{{ appCount }}</button>`
})
export class AppComponent {
  appCount = 0; // <-- `appCount` and `count` are always in sync!
}
```

COMPONENT - PROJECTION

- Allows to put HTML content inside the tag of an Angular component
- The `<ng-content />` directive allows reinserting the content in the component template

```
@Component({ selector: 'app-card', template:
  `<article>
    <ng-content />
  </article>`
})
export class CardComponent {}

@Component ({ selector: 'app-root', template:
  `<app-card>
    <header>Title</header>
    <section>Content</section>
  </app-card>`
})
export class AppComponent {}
```

COMPONENT - PROJECTION

- 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 CardComponent {}

@Component ({ selector: 'app-root', template:
  `<app-card>
    <span card-title>Title</span>
    <span card-content>Content</span>
  </app-card>`
})
export class AppComponent {}
```

COMPONENT - PROJECTION

- 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 CardComponent {}

@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 AppComponent {}
```

COMPONENT - LIFECYCLE

- It is possible to execute code using component lifecycle hooks
- More infos: https://angular.io/guide/lifecycle-hooks

```
import {
  Component, OnChanges, OnInit, AfterContentInit, AfterViewInit, OnDestroy, SimpleChanges
} from '@angular/core';

@Component ({/* ... */})
export class AppComponent implements
  OnChanges, OnInit, AfterContentInit, AfterViewInit, OnDestroy {

    ngOnChanges(changes: SimpleChanges): void {/* ... */}

    ngOnInit(): void {/* ... */}

    ngAfterContentInit(): void {/* ... */}

    ngAfterViewInit(): void {/* ... */}

    ngOnDestroy(): void {/* ... */}
  }
```

COMPONENT - LIFECYCLE | ONINIT

- `OnInit` lifecycle hook is frequently used for initialization
- Because you can safely read component `@Input`s when this hook is triggered

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

@Component ({/* ... */})
export class PostsComponent implements OnInit {
  @Intput({ required: true }) public userId!: string;

  protected posts?: Post[];

  ngOnInit(): void {
    // Doing this is 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[]> {/* ... */}
}
```

COMPONENT - LIFECYCLE | ONDESTROY

- `OnDestroy` lifecycle hook is frequently used for cleaning component

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

@Component ({
  selector: 'app-interval',
  template: '<p>{{ data }}</p>'
})
export class IntervalComponent implements OnDestroy {
  protected data = 0;

  private interval = setInterval(() => this.data++, 1000);

  ngOnDestroy(): void {
    clearInterval(this.interval);
  }
}
```

COMPONENT - VIEWCHILD

- It is possible to access template details from the class using `@ViewChild` decorator
- Retrieved informations are 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 HelloComponent {}

@Component({
  selector: 'app-root', template: `<app-hello />`
})
export class AppComponent implements OnInit, AfterViewInit {

  @ViewChild(HelloComponent) helloComponent?: HelloComponent;

  ngOnInit(): void { console.log(this.helloComponent); }        // <-- output: undefined

  ngAfterViewInit(): void { console.log(this.helloComponent); } // <-- output: HelloComponent
}
```

COMPONENT - DECLARATIONS

- Components must be declared in a NgModule (defined in detail later in the training)
- All components declared in a NgModule can see each other

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

@NgModule ({
  declarations: [AppComponent, CatalogComponent, ProductComponent]
})
export class AppModule {}
```

For example:

- `CatalogComponent` can be used in `AppComponent` template
- `ProductComponent` can be used in `CatalogComponent` template
- ...

COMPONENT - DECLARATIONS

- Components can be exported by a NgModule

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

@NgModule ({
  declarations: [CatalogComponent, ProductComponent],
  exports: [CatalogComponent, ProductComponent]
})
export class SharedModule {}

@NgModule ({
  imports: [SharedModule],
  declarations: [AppComponent]
})
export class AppModule {}
```

For example:

- `CatalogComponent` can be used in `AppComponent` template
- ...

Lab 2

UNIT TESTING

SUMMARY

- Introduction
- Reminders
- Getting started with Angular
- Components
- *Unit testing*
- Directives
- Services
- Pipes
- Http
- Router
- Forms

TESTING - INTRODUCTION

For testing an application in general, you need 2 functionalities:

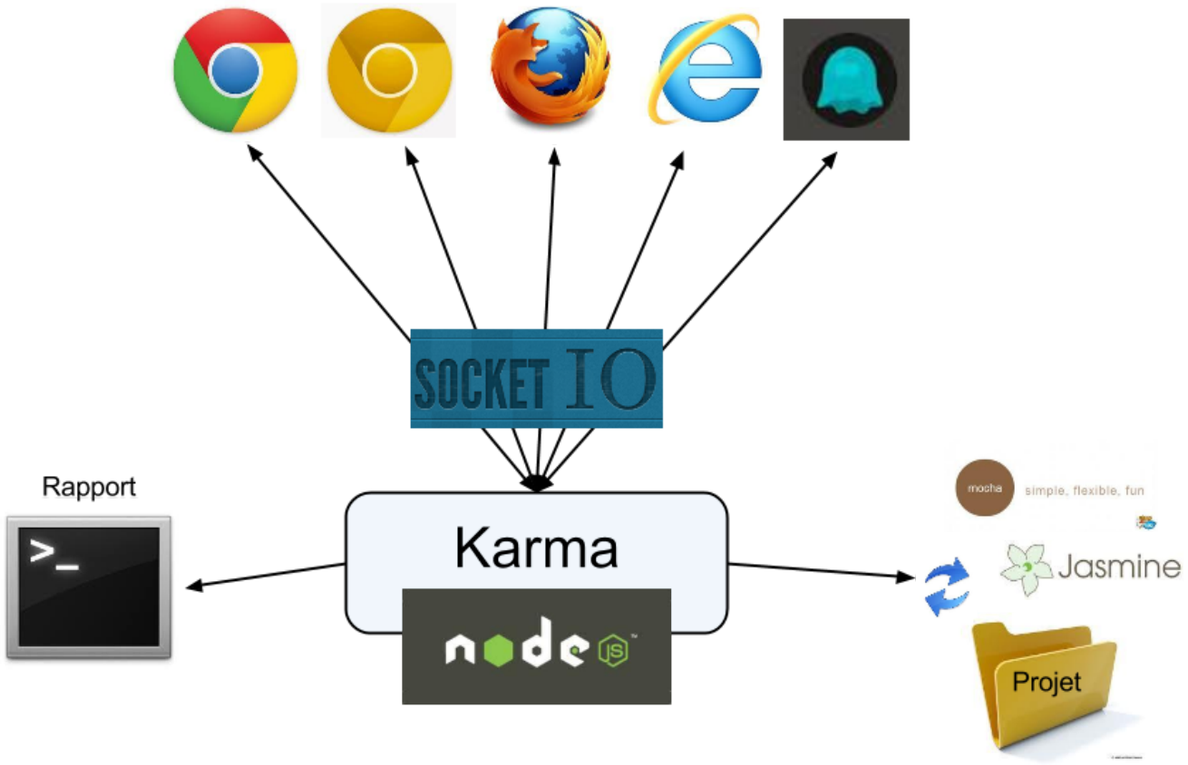
- A test runner that identifies and runs the files containing the tests
- An assertion library that verifies the expected behavior

Out of the box, Angular uses `Karma` as test runner and `Jasmine` as assertion library.

By default, test files are identified by the pattern: `*.spec.ts`.

TESTING - KARMA

- Karma is a tool that automates the execution of tests



TESTING - JASMINE

- Organize your tests using `describe` and `it` functions
- Follow the 3 steps pattern in each test: `Given`, `When`, `Then`
- Identify the thing being tested using `expect`
- Use matchers 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); // equivalent to `expect(value).toBeFalse();`
  });
});
```

TESTING - JASMINE | HOOKS

- Use hooks to setup and teardown your tests using:
  - `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.toBeTrue(); // <-- notice the usage of `.not`
  });
});
```

TESTING - JASMINE | SPIES

- Use a spy to watch how a method is been used during the test
- Create a spy: `jasmine.createSpy` or `spyOn`
- Spy matchers: `toHaveBeenCalled`, `toHaveBeenCalledWith`, `and.returnValue`, ...

```
// Given
class Counter {
  count = 0;

  increment() { this.count += 1; this.log('increment'); }

  log(message: string) { console.log('Counter:', message); }
}
const count = new Counter();
const logSpy = spyOn(count, 'log'); // <-- Spying on the `log` method

// When
count.increment();

// Then
expect(logSpy).toHaveBeenCalledWith('increment');
```

TESTING - ANGULAR ENVIRONMENT

- Angular provides a powerful testing environment called `TestBed`
- Angular testing configuration is reset for every test (executed in `beforeEach`)

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

describe('my feature', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({ /* Test setup */ });
  });

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

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

TESTING - COMPONENTS

- Components combine an HTML template and a TypeScript class
- You should test that they work together as intended
- `TestBed` helps you create the component's host element in the browser DOM
- The `fixture` gives you access to the component instance and its host element
- In the tests you must `detectChanges` manually verifying that the DOM state is correct

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

TestBed.configureTestingModule({ declarations: [AppComponent] });

let fixture = TestBed.createComponent(AppComponent);

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

fixture.detectChanges();
```

TESTING - COMPONENTS | STRATEGIES

Class testing:

- Pros: Easy to setup, Easy to write, Most usual way to write unit tests
- Cons: Does not make sure your component behave the way it should

DOM testing:

- Pros: Make sure your component behave exactly the way it should
- Cons: Harder to setup, Harder to write

✅ Overall, DOM testing is more robust, but require more work to setup.

TESTING - EXAMPLE 1

- A simple counter component

```
import { Component, EventEmitter, Input, Output } from '@angular/core';

@Component({
  selector: 'app-counter',
  template: '<button (click)="increment()">{{ count }}</button>'
})
export class CounterComponent {
  @Input() count = 0;
  @Output() countChange = new EventEmitter<number>();

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

TESTING - EXAMPLE 1

- Test setup

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

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

  beforeEach(() => {
    TestBed.configureTestingModule({ declarations: [CounterComponent] });

    fixture = TestBed.createComponent(CounterComponent);

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

TESTING - EXAMPLE 1

- Actual Tests (1/2)

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

it('should display 0', () => {
  // Getting element using `debugElement`
  const button = fixture.debugElement.query(By.css('button')).nativeElement;

  expect((button as HTMLButtonElement).textContent).toContain(0);
});

it('should increment the count when clicking', () => {
  // Getting element using `nativeElement`
  const button = (fixture.nativeElement as HTMLElement).querySelector('button');

  button?.click(); // <-- The class state get automatically updated
  expect(fixture.componentInstance.count).toBe(1); // <-- Class testing

  fixture.detectChanges(); // <-- The template state update needs to be triggered manually
  expect(button?.textContent).toContain(1); // <-- DOM testing
});
```

TESTING - EXAMPLE 1

- Actual Tests (2/2)

```
it('should emit output with the current count when clicking', () => {
  const emitSpy = spyOn(fixture.componentInstance.countChange, 'emit');

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

  expect(emitSpy).toHaveBeenCalledWith(1);
});
```

TESTING - EXAMPLE 2

- Component with dependency
- We're going to explore two different approaches to test this use case

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

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

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

TESTING - EXAMPLE 2 | FIRST APPROACH

- (1/2) Test setup with explicit dependency declaration

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

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

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [NumberParityComponent, CounterComponent] // <-- Dependency declared!
    });
    fixture = TestBed.createComponent(NumberParityComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });
});
```

TESTING - EXAMPLE 2 | FIRST APPROACH

- (2/2) Actual Tests accessing the dependency (the child component instance)

```
it('should bind count to the child component', () => {
  const counterComponent: CounterComponent =
    fixture.debugElement.query(By.directive(CounterComponent)).componentInstance;

  // Accessing the child component properties
  expect(counterComponent.count).toBe(component.count);
});

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

  // Accessing the child component methods
  counterComponent.countChange.emit(1);
  fixture.detectChanges();

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

TESTING - EXAMPLE 2 | SECOND APPROACH

- (1/2) Test setup allowing unknown HTML elements

```
import { CUSTOM_ELEMENTS_SCHEMA } from '@angular/core';
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { By } from '@angular/platform-browser';
import { CounterComponent } from '../counter/counter.component';
import { NumberParityComponent } from './number-parity.component';

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

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [NumberParityComponent], // <-- `CounterComponent` not declared...
      schemas: [CUSTOM_ELEMENTS_SCHEMA], // <-- ...but unknown HTML elements are allowed
    });
    fixture = TestBed.createComponent(NumberParityComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });
});
```

TESTING - EXAMPLE 2 | SECOND APPROACH

- (2/2) Actual Tests using:
  - `debugElement.properties` and `debugElement.triggerEventHandler`

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

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

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

  // Triggering events on the child element
  debugElement.triggerEventHandler('countChange', 1);
  fixture.detectChanges();

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

Lab 3

DIRECTIVES

SUMMARY

- Introduction
- Reminders
- Getting started with Angular
- Components
- Unit testing
- *Directives*
- Services
- Pipes
- Http
- Router
- Forms

DIRECTIVES

- Live in the component template
- Adds additional behavior to elements in your template
- Angular offers several built-in directives to manage forms, lists, styles, and what users see

There are 3 types of directives:

- `Attribute directive`: change the appearance or behavior of DOM elements
- `Structural directive`: change the DOM layout by adding and removing DOM elements
- `Components`: yes! components are directives enhanced with a template

✅ We've already covered components, so this section covers attribute and structural directives.

ATTRIBUTE DIRECTIVE

- To create a custom 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 } from '@angular/core';

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  constructor(elementRef: ElementRef, renderer: Renderer2) {

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

ATTRIBUTE DIRECTIVE

- Use the directive `selector` to attach it to DOM elements in the component template

```
<p appHighlight>Highlight me!</p>
```

- 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>
```

DIRECTIVES - HOST ELEMENT

- When possible, instead of the `Renderer2`:
  - use `@HostBinding` decorator to change the appearance of the host element
  - use `@HostListener` decorator to change the behavior of the host element

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

@Directive ({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  @HostBinding('style.backgroundColor') bgColor?: string;

  @HostListener('mouseenter') onMouseEnter() { this.bgColor = 'yellow'; }

  @HostListener('mouseleave') onMouseLeave() { this.bgColor = 'white'; }
}
```

DIRECTIVES - HOST ELEMENT

- You can achieve the same result using the `host` property of the `@Directive` decorator...

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

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

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

  onMouseLeave() { this.bgColor = 'white'; }
}
```

- ...but prefer the `HostBinding` and `HostListener` techniques

DIRECTIVES - INPUT AND OUTOUT 1/2

- Use `@Input` and `@Outout` decorators to make the directive configurable

```
import { Directive, Input, HostListener, HostBinding, Output } from '@angular/core';

@Directive ({ selector: '[appHighlight]' })
export class HighlightDirective {
  @Input() defaultBgColor = 'white';
  @Input() appHighlight = 'yellow';

  @Output() highlighted = new EventEmitter<boolean>();

  @HostBinding('style.backgroundColor') bgColor = this.defaultBgColor;

  @HostListener('mouseenter') onMouseEnter() {
    this.bgColor = this.appHighlight;
    this.highlighted.emit(true);
  }
  @HostListener('mouseleave') onMouseLeave() {
    this.bgColor = this.defaultBgColor;
    this.highlighted.emit(false);
  }
}
```

DIRECTIVES - INPUT AND OUTOUT 2/2

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

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

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

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

STRUCTURAL DIRECTIVE

Change the DOM layout by adding or removing DOM elements.

Let's take an example with the Angular built-in `NgIf` directive, which conditionally adds or removes an element.

- Shorthand using the `*` symbol (also called micro-syntax)
- The `<h1>` tag is only displayed when the condition `product.stock > 0` is `true`

```
<h1 *ngIf="product.stock > 0">{{ product.title }}</h1>
```

- Under the hood Angular creates an `<ng-template>` wrapper element.
  At runtime, Angular does not render `<ng-template>` but only the `<h1>`

```
<ng-template [ngIf]="product.stock">
  <h1>{{ product.title }}</h1>
</ng-template>
```

- A structural directive is therefore an attribute directive whose host element is a template

STRUCTURAL DIRECTIVE

- Let's create a custom structural directive which does the opposite of `NgIf`

```
import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';

@Directive({ selector: '[appUnless]' })
export class UnlessDirective {
  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainerRef: ViewContainerRef,
  ) {}

  @Input() set appUnless(condition: boolean) {
    const hasView = this.viewContainerRef.get(0) !== null;

    if (!condition && !hasView) {
      this.viewContainerRef.createEmbeddedView(this.templateRef);
    }
    else if (condition && hasView) {
      this.viewContainerRef.clear();
    }
  }
}
```

STRUCTURAL DIRECTIVE - COMBINATION

- Multiple structural directives can NOT be combined on the same host element
- For this use-case, use `<ng-container>`

```
<ng-container *ngFor="let product of products">
  <h1 *ngIf="product.stock > 0">{{ product.title }}</h1>
</ng-container>
```

- In fact, the example above is equivalent to

```
<h1 *ngFor="let product of products">
  <ng-container *ngIf="product.stock > 0">{{ product.title }}</ng-container>
</h1>
```

- At runtime `<ng-container>` does not get rendered (like `<ng-template>`)

```
<h1>Product 1</h1>
<h1>Product 2</h1>
```

DIRECTIVES - DECLARATION

- Like components, directives must be declared in `NgModule`

```
// --- app.module.ts ---

import { NgModule } from '@angular/core';
import { HighlightDirective } from './highlight/highlight.directive';

@NgModule ({
  declarations: [AppComponent, HighlightDirective]
})
export class AppModule {}

// --- app.component.ts ---

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

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

BUILT-IN ATTR. DIRECTIVES - NGSTYLE

- Adds CSS properties
- Takes an object with CSS properties as keys
- Use only for cases where pure CSS is not enough

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

@Component ({
  selector: 'app-font-size-selector',
  template: `
   <h1 [ngStyle]="{ 'font-size': currentSize + 'px' }">Example<h1>

   Change size: <input type="number" [value]="currentSize" (input)="changeSize($event)">
  `
})
export class FontSizeSelectorComponent {
  currentSize = 20;

  changeSize(event: Event) {
    this.currentSize = Number((event.target as HTMLInputElement).value);
  }
}
```

BUILT-IN ATTR. DIRECTIVES - NGCLASS

- The `ngClass` directive adds or removes CSS classes
- Can be used in addition to the standard class attribute
- Three syntaxes coexist:
  - `[ngClass]=" 'class2 class2' "`
  - `[ngClass]=" ['class1', 'class2'] "`
  - `[ngClass]=" { 'class1': hasClass1, 'class2': hasClass2 } "`
- The last syntax is the most commonly used

BUILT-IN ATTR. DIRECTIVES - NGCLASS

- Example of using the `ngClass` directive

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

@Component ({
  selector: 'app-toggle-highlight',
  template: `
    <div [ngClass]="{ 'highlight': isHighlighted }">
      {{ isHighlighted ? 'On' : 'Off' }}
    </div>

    <button (click)="isHighlighted = !isHighlighted">Toggle</button>
  `,
  styles: [`
    .highlight { background-color: yellow }
  `]
})
export class ToggleHighlightComponent {
  isHighlighted = false;
}
```

BUILT-IN STRUCT. DIRECTIVES - NGIF

- Adds or removes an HTML element based on a condition

```
<div *ngIf="condition">Lorem Ipsum</div>
```

- Ability to define an `else` clause
- Create a "template reference" using the # symbol

```
<div *ngIf="condition; else noContent">Lorem Ipsum</div>

<ng-template #noContent>No content available...</ng-template>
```

- Note that when you hide an element using CSS, the element remains part of the DOM

```
<div style="display: none">Lorem Ipsum</ div>
```

BUILT-IN STRUCT. DIRECTIVES - NGFOR (HARD WAY)

- Can duplicate a template for each item in a collection
- Use `<ng-template>` to define the content to duplicate
- Use `ngForOf` attribute (which is an `@Input` of the `NgFor` directive) to define the collection
- Use `let-myVar="value"` syntax to define variables inside the template for each iteration
  - use one of the values provided by Angular: `index`, `first`, `last`, `even` and `odd`

```
<ul>
  <ng-template ngFor [ngForOf]="products" let-product let-idx="index">
    <li>{{ idx + 1 }}: {{ product.title }}</li>
  </ng-template>
</ul>
```

- In fact, there's another value: `$implicit` which is optional

```
<ng-template ngFor [ngForOf]="products" let-product="$implicit">
  <p>{{ product.title }}</p> </ng-template>
```

BUILT-IN STRUCT. DIRECTIVES - NGFOR

- Use the micro-syntax `*ngFor` (like we did for `*ngIf`)
- Note that the `*ngFor` is directly placed on the element to duplicate
- Use the `trackBy` function to improve directive performance on large datasets

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

@Component ({
  selector: 'app-root',
  template: `
    <ul>
      <li *ngFor="let product of products; let idx = index; trackBy: trackByProductId">
        {{ idx + 1 }}: {{ product.title }}
      </li>
    </ul>`
})
export class AppComponent {
  products: Product[] = [/* ... */];

  trackByProductId(index: number, product: Product) { return product.id; }
}
```

BUILT-IN STRUCT. DIRECTIVES - NGSWITCH

- Adds or removes HTML elements based on a condition
- Is made of both "attribute" and "structural" directives
- Three directives available:
  - `[ngSwitch]`: container element for the different cases
  - `*ngSwitchCase`: element to display depending on a condition
  - `*ngSwitchDefault`: element to display as default value

```
The value is:
<ng-container [ngSwitch]="value">
  <strong *ngSwitchCase="0"> zero </strong>
  <strong *ngSwitchCase="1"> one </strong>
  <strong *ngSwitchCase="2"> two </strong>
  <strong *ngSwitchDefault> greater that two </strong>
</ng-container>
```

DIRECTIVES - TESTING

- Create a wrapper component for DOM testing purposes

```
import { Component } from '@angular/core';
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { HighlightDirective } from './highlight.directive';

@Component({
  selector: 'app-wrapper',
  template: '<div appHighlight>Highlight</div>',
})
class WrapperComponent {}

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

  beforeEach(() => {
    TestBed.configureTestingModule({ declarations: [WrapperComponent, HighlightDirective] });
    fixture = TestBed.createComponent(WrapperComponent);
    hostElement = fixture.nativeElement.querySelector('[appHighlight]') as HTMLElement;
  });
});
```

Lab 4

SERVICES

SUMMARY

- Introduction
- Reminders
- Getting started with Angular
- Components
- Unit testing
- Directives
- *Services*
- Pipes
- Http
- Router
- Forms

SERVICES - IN A NUTSHELL

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

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

export class ApiService {                          // <-- 1. Defining service
  fetchMsg() { return { data: 'Hello World!' }; }
}

@NgModule({
  providers: [ApiService],                         // <-- 2. Providing service
  declarations: [AppComponent],
})
export class AppModule {}

@Component({
  selector: 'app-root',
  template: '<h1>{{ msg.data }}</h1>',
})
export class AppComponent {
  constructor(private apiService: ApiService) {}  // <-- 3. Injecting service
  msg = this.apiService.fetchMsg();                // <-- 4. Consuming service
}
```

SERVICES - INJECTABLE

- If your service has dependencies, add `@Injectable` decorator to benefit from the dependency injection
- `@Component` decorator is implicitly `Injectable`

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

@Injectable()
export class ApiService {
  // Service dependencies requires `@Injectable` decorator
  constructor(private httpClient: HttpClient) {}

  fetchMsg() {
    return this.httpClient.get('/api/msg');
  }
}

@NgModule({
  providers: [ApiService],
  declarations: [AppComponent],
})
export class AppModule {}
```

SERVICES - INJECTABLE | PROVIDEDIN

- Use `providedIn` injectable metadata to provide a service globally right from its definition

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

@Injectable({
  providedIn: 'root' // <-- Service is automatically provided at the app root level
})
export class ApiService {
  constructor(private httpClient: HttpClient) {}

  fetchMsg() {
    return this.httpClient.get('/api/msg');
  }
}

@NgModule({
  providers: [], // <-- It is no longer necessary to provide it manually!
  declarations: [AppComponent],
})
export class AppModule {}
```

SERVICES - COMPONENT PROVIDERS

- You can use the `providers` property in the `@Component` decorator metadata
- Services defined in a component are also injectable in its child components
- The service lifecycle (created and destroyed) follows the component lifecycle

```
@Component ({
  selector: 'app-parent',
  template: '<app-child />',
  providers: [ApiService]
})
export class ParentComponent {
  constructor(public apiService: ApiService) {}
}

@Component ({ selector: 'app-child', template: '...' })
export class ChildComponent {
  // Get the service from the `ParentComponent` injector
  constructor(public apiService: ApiService) {}
}
```

SERVICES - INJECTORS

- Responsible for providing dependencies to components, services, ...
- An application can have more than one injector...
- ...but within one injector every dependency is a singleton

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

@Injectable({ providedIn: 'root' })
export class DataService { data?: string; }

@Component({ selector: 'app-setter', template: '...', })
export class SetterComponent {
  constructor(dataService: DataService) { dataService.data = 'Hello World!'; }
}

@Component({ selector: 'app-getter', template: '<h1>{{ data }}</h1>' })
export class GetterComponent {
  get data() { return this.dataService.data; } // <-- 'Hello World!'
  constructor(private dataService: DataService) {}
}
```

SERVICES - INJECTORS HIERARCHY

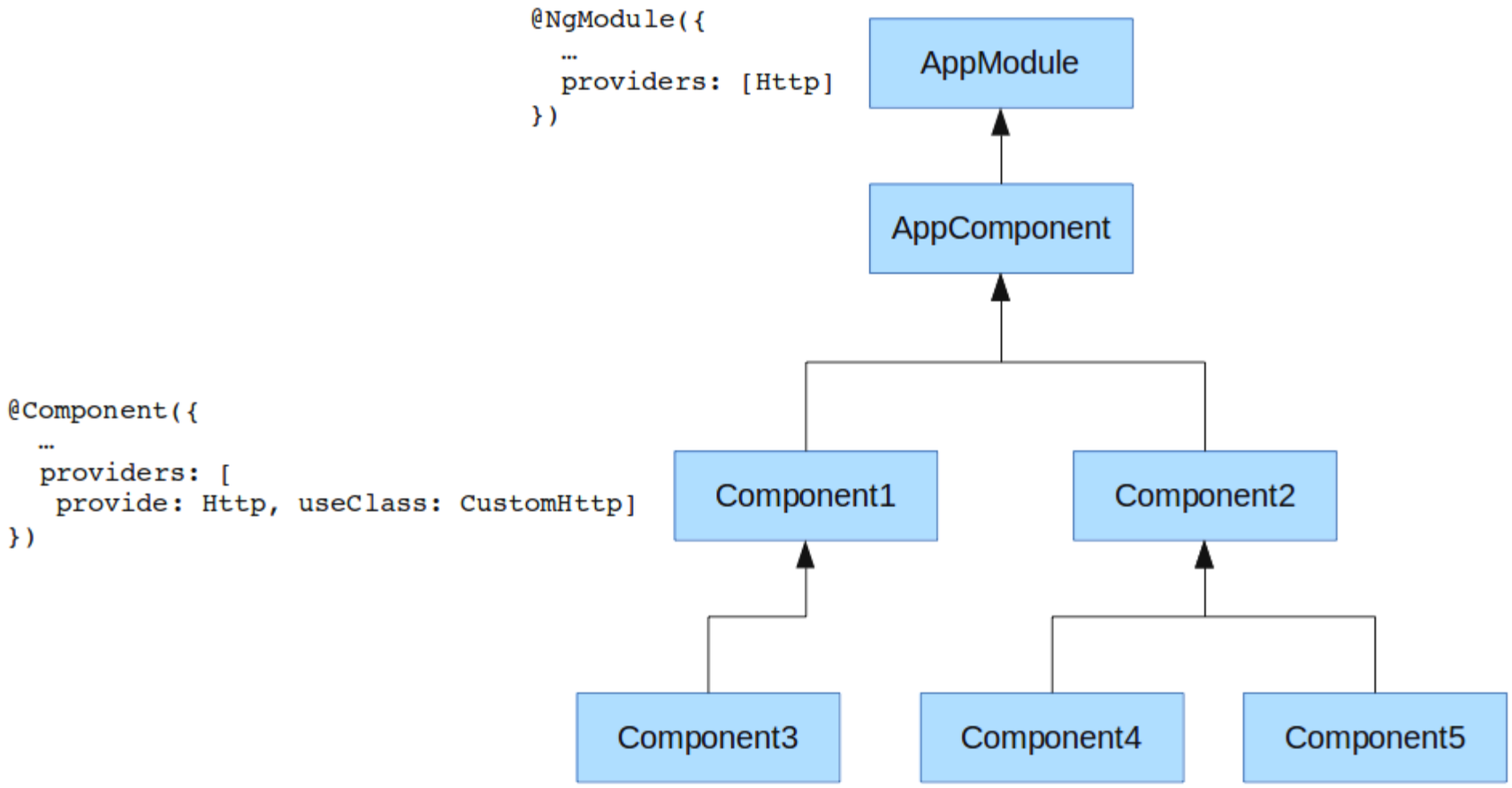
During a dependency injection:

- The local injector tries to find a compatible provider
- If it can't find one, it forwards the request to its parent
- And so on up to the application's main injector
- If no provider can be found, Angular throws an error

This mechanism is very powerful, but can be complex:

- Services can be overwritten locally
- But can also inadvertently hide the "expected" service

SERVICES - INJECTORS HIERARCHY



SERVICES - PROVIDERS | CLASSPROVIDER

A provider describes for the injector how to get an instance of a dependency

- The most common case is the class provider

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

const apiProvider: ClassProvider = {
  provide: ApiService,
  useClass: ApiStubService
};

@NgModule ({
  providers: [apiProvider]
})
export class AppModule {}

@Component({ /* ... */ })
export class AppComponent {
  constructor(apiService: ApiService) {
    console.log(apiService); // <-- will print `ApiStubService` to the console!
  }
}
```

SERVICES - PROVIDERS | CLASSPROVIDER

- Shorthand when `provide` and `useClass` properties point to the same value

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

@NgModule ({
  providers: [{ provide: ApiService, useClass: ApiService }]
})
export class AppModule {}
```

is equivalent to:

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

@NgModule ({
  providers: [ApiService]
})
export class AppModule {}
```

SERVICES - PROVIDERS | FACTORYPROVIDER

- Use factory provider when the provider needs information that is only available at runtime

```
import { NgModule, FactoryProvider } from '@angular/core';

function apiFactory(userService: UserService): FactoryProvider {
  // In this example, `isAdmin` is only available at runtime
  // because it depends on the logged-in user!
  return new ApiService(userService.isAdmin);
}

const apiProvider: FactoryProvider = {
  provide: ApiService,
  useFactory: apiFactory,
  deps: [UserService]
};

@NgModule ({
  providers: [apiProvider]
})
export class AppModule {}
```

SERVICES - PROVIDERS | VALUEPROVIDER

- Primitive values (like `string`, `number`, ...) can't be provided!?

```
@Component({ /* ... */ })
export class AppComponent {
  constructor(private appTitle: string) {} // ❌ This type is not supported as injection token
}
```

- `ValueProvider` and `InjectionToken` to the rescue

```
import { Component, Inject, InjectionToken, NgModule, ValueProvider } from '@angular/core';

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

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

@NgModule ({ providers: [appTitleProvider] }) export class AppModule {}

@Component({ /* ... */ })
export class AppComponent {
  constructor(@Inject(APP_TITLE) private appTitle: string) {}
}
```

SERVICES - TESTING

- You can configure the providers in your `TestBed`
- Don't hesitate to "stub" the real services
- 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 stub:

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

describe('AppComponent', () => {
  let apiService: ApiService;
  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [AppComponent],
      providers: [{ provide: ApiService, useClass: ApiStubService }],
    });
    apiService = TestBed.inject(ApiService); // <-- Get the `ApiStubService`
  });
});
```

Lab 5

PIPES

SUMMARY

- Introduction
- Reminders
- Getting started with Angular
- Components
- Unit testing
- Directives
- Services
- *Pipes*
- Http
- Router
- Forms

PIPES

- Transform strings, currency amounts, dates, and other data for display
- Simple functions to use in template expressions
- Accept an input value and return a transformed value
- You can write your own custom pipe
- Angular provides a good number of pipes for common use-cases (`@angular/common`)
  - `LowerCasePipe`, `UpperCasePipe`, `TitleCasePipe`
  - `CurrencyPipe`, `DecimalPipe`, `PercentPipe`
  - `DatePipe`, `JsonPipe`, `SlicePipe`, `KeyValuePipe`
  - `AsyncPipe`

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';

@Component({
  selector: 'app-root',
  template: `
    <p>{{ date | date }}</p> <!-- 29 août 2023 -->

    <p>{{ date | date | uppercase }}</p> <!-- 29 AOÛT 2023 -->

    <p>{{ price | currency : 'EUR' : 'symbol' }}</p> <!-- 123,46 € -->
  `,
})
export class AppComponent {
  date = new Date();
  price = 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: Array<string | number>, separator = ' '): unknown {
    return value.join(separator);
  }
}
```

- Usage example:

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

<!-- List: apple, orange, banana. -->
```

PIPES - CUSTOM | DECLARATION

- Declared like Components and Directives in the Module's `declarations` array
- Can be used in all components of the module in which they are declared

```
import { Component, NgModule } from '@angular/core';
import { JoinArrayPipe } from './pipes/join-array.pipe';

@NgModule ({
  declarations: [AppComponent, JoinArrayPipe]
})
export class AppModule {}

@Component({
  selector: 'app-root',
  template: `{{ appList | joinArray }}`,
})
export class AppComponent {
  appList = ['apple', 'orange', 'banana'];
}
```

PIPES - CONFIGURATION

Some Angular pipes can be configured to suit your needs.

Here's an example with the `CurrencyPipe`

Depending on the locale:

- should display `$3.50` for United States
- 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 €)

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

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

@NgModule({
  providers: [
    { provide: LOCALE_ID, useValue: 'fr' },
    { provide: DEFAULT_CURRENCY_CODE, useValue: 'EUR' },
  ],
})
export class AppModule {}
```

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 NgModule)
  - The injected pipe will respect the configuration, if any

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

@Component ({ selector: 'app-root', providers: [CurrencyPipe] })
class AppComponent {
  constructor(currencyPipe: CurrencyPipe) {

    const upperCasePipe = new UpperCasePipe();
    console.log(upperCasePipe.transform('Hello World!')); // <-- HELLO WORLD!

    console.log(currencyPipe.transform(123.456789)); // <-- 123.46 €
  }
}
```

PIPES - PURE

- Refers to the concept of pure function
- Angular will only re-evaluate the pipe if the input value *reference* has changed
- Optimizes the performance of the change detection mechanism
- 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

- Angular always re-evaluate the pipe even if the input value *reference* has not changed
- Suitable when the input value is an `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';

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

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

PIPES - IMPURE

- Let's take another look at the custom pipe shown above
- 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: Array<string | number>, separator = ' '): unknown {
    return value.join(separator);
  }
}

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

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

Lab 6

HTTP

SUMMARY

- Introduction
- Reminders
- Getting started with Angular
- Components
- Unit testing
- Directives
- Services
- Pipes
- *Http*
- Router
- Forms

HTTP - GETTING STARTED

Suppose our application needs to display todo items from the `jsonplaceholder` API

- Here's the shape of a `Todo` item...

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

- ...and the API response for `id` 1

```
{ "id": 1, "title": "delectus aut autem", "completed": false }
```

HTTP - GETTING STARTED

- To add Http capabilities to Angular, we need first to import the `HttpClientModule`...

```
import { NgModule } from '@angular/core';
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  imports: [HttpClientModule]
})
export class AppModule {}
```

HTTP - GETTING STARTED

- ...and then use the `HttpClient` service in our components

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

// ...
export class TodoComponent implements OnInit {
  todo?: Todo;

  constructor(private httpClient: HttpClient) {} // <-- Inject service

  ngOnInit(): void {
    this.httpClient
      // Define shape of GET request
      .get<Todo>('https://jsonplaceholder.typicode.com/todos/1')
      .subscribe(todo => this.todo = todo); // <-- Execute request and store response
  }

  addTodo(): void {
    this.httpClient
      // Define shape of POST request with JSON body
      .post('https://jsonplaceholder.typicode.com/todos', { title: 'test', completed: false })
      .subscribe() // <-- Execute request
  }
}
```

HTTP - GETTING STARTED

- Let's take a look at the interface of the `httpClient.get` method:

```
export declare class HttpClient {
  get<T>(url: string, options?: { /* ... */ }): Observable<T>;
}
```

What is an `Observable` ?

RXJS - INTRODUCTION

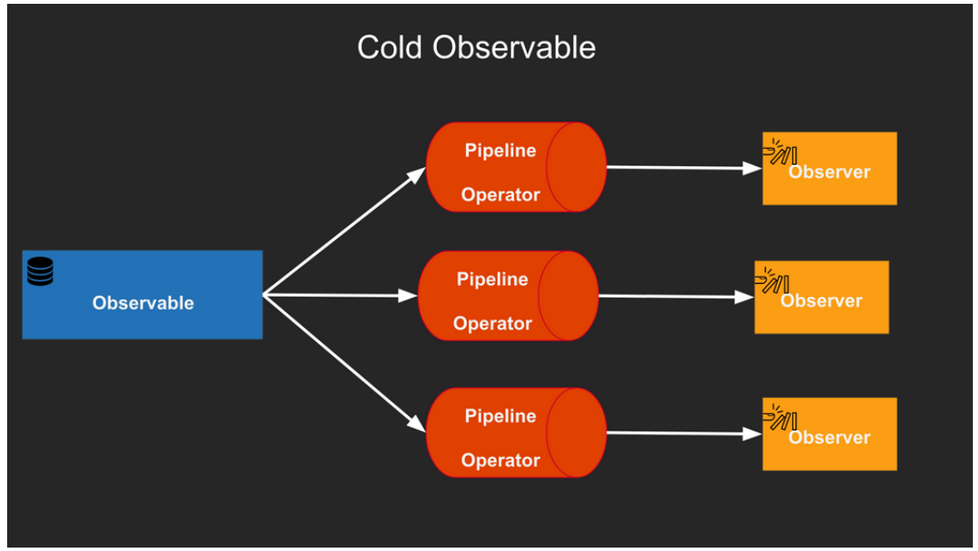
- Observables
  - represent a stream of data that can be subscribed to
  - allowing multiple values to be emitted over time
- Refers to a paradigm called ReactiveX (http://reactivex.io/)
  - an API for asynchronous programming with observable streams
  - implemented in all major programming languages: RxJava, Rx.NET, RxJS, ...

Angular relies a lot on the notion of Observable, powered by the library RxJS.

RXJS - OBSERVABLE & OBSERVER

An `observable` is a stream. Data navigate in it. There can be no data, one data or multiple data going throught it as time goes by.

An `observer` is a listener. It `subscribes` to an `observable` to be notified when new data are received.



HTTP - ADDING A LISTENER

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

@Component({
  selector: 'app-todo',
  template: '<p>{{ todo?.title }}</p>'
})
export class TodoComponent implements OnInit {
  todo?: Todo;

  constructor(private httpClient: HttpClient) {}

  ngOnInit(): void {
    this.httpClient
      .get<Todo>('https://jsonplaceholder.typicode.com/todos/1')
      .subscribe(todo => this.todo = todo);
  }
}
```

HTTP - NO LISTENER

What whould happen if there is no listener on your Observable ?

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

@Component({
  selector: 'app-todo',
  template: '<p>{{ todo?.title }}</p>'
})
export class TodoComponent implements OnInit {
  todo?: Todo;

  constructor(private httpClient: HttpClient) {}

  ngOnInit(): void {
    this.httpClient.get<Todo>('https://jsonplaceholder.typicode.com/todos/1');
  }
}
```

HTTP - NO LISTENER

`no subscribe === no request sent`.

The http request is the data source of your Observable.

```
// Creates an Observable but do not trigger the http request
this.httpClient.get<Todo>('https://jsonplaceholder.typicode.com/todos/1');
```

Also, `x subscribe === x http requests sent`

Lab 7

RXJS - OPERATORS

What are operators ?

> Operators are the essential pieces that allow complex asynchronous code to be easily composed in a declarative manner.

- Functions
- Takes an `Observable` as an input, creates a new `Observable` as an output
- Are commonly used to manipulate data going through an `Observable`
- Can be chained

RXJS - OPERATORS | FILTER

```
// ...
import { filter } from 'rxjs';

export class TodoComponent implements OnInit {
  todo?: Todo;

  constructor(private httpClient: HttpClient) {}

  ngOnInit(): void {
    this.httpClient
      .get<Todo>('https://jsonplaceholder.typicode.com/todos/1')
      // Filter the todo : only keep it if it is not completed
      .pipe(filter(todo => todo.completed === false))
      .subscribe(todo => this.todo = todo);
  }
}
```

RXJS - OPERATORS | TAP

```
// ...
import { filter } from 'rxjs';

export class TodoService {
  todo?: Todo;

  constructor(private httpClient: HttpClient) {}

  getTodo(id: number): Observable<Todo> {
    this.httpClient
      .get<Todo>(`https://jsonplaceholder.typicode.com/todos/${id}`)
      // See the response Todo and keep track of it without changing it
      .pipe(tap(todo => this.todo = todo));
  }
}
```

Lab 7

HTTP - TESTING 1/2

- Angular provides `HttpClientTestingModule` and `HttpTestingController` for mocking the Http module

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

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

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientTestingModule],
    });

    service = TestBed.inject(ApiService);
    httpTestingController = TestBed.inject(HttpTestingController);
  });
});
```

HTTP - TESTING 2/2

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

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

describe('ApiService', () => {
  // ...

  it('should fetch the products', () => {
    const responseMock: Product[] = [{ id: 'ID_1' } as Product, { id: 'ID_2' } as Product];

    service.fetchProducts().subscribe((products) => expect(products).toEqual(responseMock));

    const req = httpTestingController.expectOne('http://localhost:8080/api/products');
    expect(req.request.method).toEqual('GET');
    req.flush(responseMock);

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

ROUTER

SUMMARY

- Introduction
- Reminders
- Getting started with Angular
- Components
- Unit testing
- Directives
- Services
- Pipes
- Http
- *Router*
- Forms

ROUTER

- In simple terms, the router allows to:
  - Display different views
  - At a defined insertion point
  - Depending on the browser's URL
- The Angular router offers many features:
  - Nested routes management
  - Possibility to have multiple insertion points
  - Guards system to allow/deny route access
  - Asynchronous views loading
- To use the router in your app, import the following module in your `AppModule`:

```
import { RouterModule } from `@angular/router`;
```

ROUTER

- The router is Component oriented
- The principle is to associate the components to be loaded according to the URL

In a nutshell, you need to:

- Use the `RouterModule.forRoot` function to add routing capabilities to your app
- Use the `RouterOutlet` directive to define the insertion point
- Navigate between pages via the `RouterLink` directive

ROUTER

- Here's an example:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'contacts', component: ContactsComponent },
  { path: 'contact/:id', component: ContactComponent }
];

@NgModule ({
  imports: [RouterModule.forRoot(routes)],
})
export class AppModule {}
```

ROUTER - ROUTEROUTLET

- Directive to use via the `<router-outlet />` element
- Defines the insertion point
- Ability to name the insertion point via a `name` attribute
- Naming outlets is useful when you have multiple views to display for the same route

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

@Component ({
  selector: 'app-root',
  template: `
    <header>My Awesome App</header>
    <router-outlet></router-outlet>
    <footer>Copyright Zenika</footer>
  `
})
export class AppComponent {}
```

ROUTER - ROUTERLINK

- Allows you to navigate from one route to another
- `RouterLink` takes an array of path *segments*
  - Segments are then concatenated to form the URL

```
@Component ({
  selector: 'app-root',
  template: `
    <nav>
      <ul>
        <li><a routerLink="/contacts/1"> Link 1 </a></li>
        <li><a [routerLink]="['/contact', 2]"> Link 2 </a></li>
        <li><a [routerLink]="['/contact', id]"> Link 3 </a></li>
      </ul>
    </nav>
    <route-outlet />
  `
})
export class AppComponent {
  id = 3;
}
```

ROUTER - NESTED ROUTEROUTLET

- Nesting multiple `RouterOutlet` to define a hierarchy of views

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

const routes: Routes = [
  {
    path: 'contact/:id',
    component: ContactComponent,
    children: [
      { path: 'view', component: ViewContactComponent },
      { path: 'edit', component: EditContactComponent },
    ],
  },
];

RouterModule.forRoot(routes);
```

- The `ContactComponent` component template must contain a `<router-outlet />` element in order to insert the `ViewContactComponent` or `EditContactComponent` components.

ROUTER - STRATEGIES FOR GENERATING URLS

- @angular/router offers two possible strategies for URLs
- The configurations are done by the system of injection of dependencies
- `PathLocationStrategy` (default policy)

```
router.navigate(['contacts']); // -> http://example.com/contacts
```

- `HashLocationStrategy`

```
router.navigate(['contacts']); // -> http://example.com/#/contacts
```

- `PathLocationStrategy` is the recommended solution today
- If your application is not deployed to the root of your domain
  - Need to add a parameter: `APP_BASE_HREF` or the `<base href='/'>` tag in your `index.html`

ROUTER - STRATEGIES FOR GENERATING URLS

- Configure the implementation to use

```
import { HashLocationStrategy, LocationStrategy } from '@angular/common';

@NgModule ({
  providers: [{ provide: LocationStrategy, useClass: HashLocationStrategy }]
})
export class AppModule {}
```

- Configure the base url of the application (if different from /)

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

@NgModule ({
  providers: [{ provide: APP_BASE_HREF, useValue: '/my/app' }],
})
export class AppModule {}
```

ROUTER - RETRIEVING URL PARAMETERS

- Using the `ActivatedRoute` and `params` service
- The API is in the form of a flow of the value of the parameters over time

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

@Component ({
  template: '<main> <router-outlet /> </main>'
})
export class ContactComponent implements OnInit {
  constructor(private route: ActivatedRoute) {}

  ngOnInit() {
    this.route.params.subscribe((params: Params) => {
      const id = Number(params.id); // The parameters are always string
      //...
    });
  }
}
```

ROUTER - RETRIEVING URL PARAMETERS

- If you are sure that the parameter can not change
- The `snapshot` property gives values at a time T

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

@Component ({
  template: '<main> <router-outlet/> </main>'
})
export class ContactComponent {
  constructor(private route: ActivatedRoute) {}

  ngOnInit(): void {
    const snapshot: ActivatedRouteSnapshot = this.route.snapshot;
    const id = Number(snapshot.params.id);
    //...
  }
}
```

ROUTER - GUARDS

- Ability to interact with the lifecycle of the navigation
- CanActivate interface allows to prohibit or to authorize the access to a route

```
import { Injectable } from '@angular/core';
import {
  CanActivate, Router, Routes, ActivatedRouteSnapshot, UrlTree
} from '@angular/router';
import { AuthService } from './auth.service';
import { AdminComponent } from './admin.component';

@Injectable()
export class AuthGuard implements CanActivate {
  constructor(private authService: AuthService, private router: Router) {}

  canActivate(route: ActivatedRouteSnapshot) {
    if (this.authService.isLoggedIn()) return true;
    return this.router.parseUrl('/login');
  }
}

const routes: Routes = [{ path: 'admin', component: AdminComponent, canActivate: [AuthGuard] }];
```

ROUTER - LAZY LOADING

- Allows to divide the size of the *bundle* JavaScript to load to start
- Each section of the site is isolated in a different `NgModule`
- The module will be loaded when the user will visit one of its pages
- Automatic creation of `chunk` via Webpack thanks to @angular/cli
- Configuring the router with the `loadChildren` property
- Separate the elements (components, services) of each module
- Several loading strategies
  - `PreloadAllModules`: Pre-load the modules as soon as possible
  - `NoPreloading`: Loading during a navigation (default strategy)

ROUTER - LAZY LOADING

- Load on demand of the `AdminModule` module

```
const routes: Routes = [{
  path: 'admin',
  loadChildren: () => import('./admin/admin.module').then(mod => mod.AdminModule)
}];

@NgModule ({ imports: [RouterModule.forRoot(routes)] }) export class AppModule {}
```

- Configuring `AdminModule` routes through the `forChild` method

```
const adminRoutes: Routes = [{
  { path: '', component: AdminHomeComponent },
  { path: 'users', component: AdminUsersComponent }
}];

@NgModule ({
  declarations: [AdminHomeComponent, AdminUsersComponent],
  imports: [RouterModule.forChild(adminRoutes)]
})
export class AdminModule {}
```

Lab 8

FORMS

SUMMARY

- Introduction
- Reminders
- Getting started with Angular
- Components
- Unit testing
- Directives
- Services
- Pipes
- Http
- Router
- *Forms*

FORMS - MODULES

Angular provides 2 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

Any form can be created using either of the following technique, 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 training focuses solely on Template-driven forms.

FORMS - MODULES

- Import the `FormsModule` in your app
- Use the provided directives like `ngModel`

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

@NgModule({
  imports: [FormsModule],
  declarations: [AppComponent],
})
export class AppModule {}

@Component({
  selector: 'app-root',
  template: `<input ngModel />`,
})
export class AppComponent {}
```

FORMS - GETTING STARTED

- 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 3 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

- 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 like `<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

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

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

  <input
    ngModel #emailModel="ngModel" name="email" placeholder="Your email" type="email" required
  />
  <!-- ... -->
</form>
```

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

Later, you'll discover why these variables are important and how they'll be used...

But for now let's take a look at how the assignment of these variables works.

FORMS - ACCESSING NGFORM & NGMODEL

- 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: 'appDoSomething' exportAs: 'doSomethingExportedName' })
export class DoSomethingDirective {}

@Component({
  selector: 'app-root',
  template: '<div appDoSomething #myDirective="doSomethingExportedName" #myDiv></div>',
})
export class AppComponent {}
```

- Here, the template variable `myDirective` holds the `DoSomethingDirective` 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 - NGFORM

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

- 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',
  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 AppComponent {
  submitForm(userFormValue: { name: string; email: string; message: string }) { /* ... */ }
}
```

FORMS - NGFORM

- By the way, you can use `@ViewChild` decorator to retrieve a template variable on the component class side!
- Here's an example where we log the `NgForm` status (`'INVALID'` or `'VALID'`) in the console:

```
import { AfterViewInit, Component, ViewChild } from '@angular/core';
import { takeUntilDestroyed } from '@angular/core/rxjs-interop';
import { NgForm } from '@angular/forms';

@Component({
  selector: 'app-root',
  template: `<form #userForm="ngForm"> <!-- ... --> </form>`,
})
export class AppComponent implements AfterViewInit {
  @ViewChild('userForm') userForm!: NgForm;

  ngAfterViewInit(): void {
    this.userForm.statusChanges?.pipe(takeUntilDestroyed()).subscribe(console.log);
    // output: INVALID, ..., VALID, ... (depending on the current form state)
  }
}
```

FORMS - NGMODEL

Now let's take a closer look at the `NgModel` directive.

- First use-case: `ngModel` directive lets you achieve two-way data binding easily
- Works even outside a `<form>` element (`name` attribute is not mandatory in this case)

```
@Component({
  selector: 'app-root',
  template: `
    <div>{{ data }}</div>

    <input [(ngModel)]="data" />

    <input [ngModel]="data" (ngModelChange)="data = $event" />

    <input #inputRef [value]="data" (input)="data = inputRef.value" />
  `,
})
export class AppComponent {
  data = '';
}
```

FORMS - NGMODEL

- Second use-case: `ngModel` keeps track of the input state
- The directive adds special CSS classes to reflect that state:
  - `ng-untouched`/`ng-touched`, `ng-pristine`/`ng-dirty`, `ng-invalid`/`ng-valid`

```
@Component({
  selector: 'app-root',
  template: `
    <!-- 1. Initial state -->
    <input required ngModel class="ng-untouched ng-pristine ng-invalid" />

    <!-- 2. After the user has entered and leaved 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 AppComponent {}
```

FORMS - NGMODEL

- You can also define your own CSS classes and bind them using the `NgModel` properties:
  - `untouched`/`touched`, `pristine`/`dirty`, `invalid`/`valid`

```
@Component({
  selector: 'app-root',
  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 AppComponent {}
```

FORMS - VALIDATORS

- 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

- Use the `.errors` property on the `NgModel` directive the track the validation errors
- Here's an example with a form field that is required and must be a valid email

```
<input name="email" ngModel #emailModel="ngModel" required type="email" />

"{{ emailModel.errors | json }}"

<!--
  Depending on the field value, output might be:
    - "null"
    - "{ required: true }"
    - "{ email: true }"
-->
```

FORMS - VALIDATORS

- Use the `.hasError` method on the `NgModel` directive to check the presence of a particular error:

```
<input name="email" ngModel #emailModel="ngModel" required type="email" />

<span *ngIf="emailModel.hasError('required')" style="color:red">
  The email is required
</span>

<span *ngIf="emailModel.hasError('email')" 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, Validator } from '@angular/forms';

@Directive({
  selector: '[appStartWith][ngModel]',
  providers: [{ provide: NG_VALIDATORS, useExisting: StartWithValidator, multi: true }],
})
export class StartWithValidator implements Validator {
  @Input({ required: true }) appStartWith!: string;

  validate(control: AbstractControl) {
    if (typeof control.value !== 'string' || !control.value.startsWith(this.appStartWith)) {
      return { startWith: this.appStartWith };
    }
    return null;
  }
}
```

FORMS - VALIDATORS | CUSTOM 2/2

- Here's an example of how to use this custom validator:

```
<input ngModel #model="ngModel" appStartWith="hello" />

<span *ngIf="model.errors?.['startWith'] as expectedValue" style="color:red">
  The value should start with: {{ expectedValue }}.
</span>
```

Lab 9

THANK YOU