



# ADVANCED ANGULAR

# SUMMARY



- Reminders
- Angular CLI
- Best practices
- Reactivity
- RxJS
- Standalone components
- Router
- Reactive Forms
- I18n
- SSR
- Animations
- Material
- Tests
- NgRx
- To go further

# LOGISTICS



- Schedule
- Lunch & breaks
- Other questions?





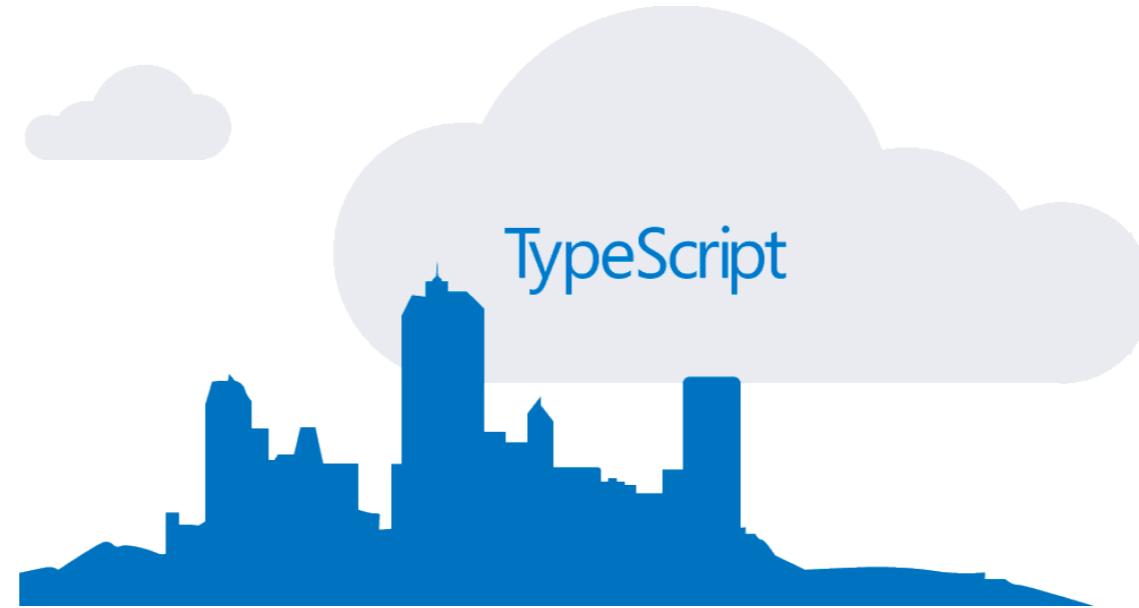
# REMINDERS

# SUMMARY



- *Reminders*
- Angular CLI
- Best practices
- Reactivity
- RxJS
- Standalone components
- Router
- Reactive Forms
- I18n
- SSR
- Animations
- Material
- Tests
- NgRx
- To go further

# TYPESCRIPT

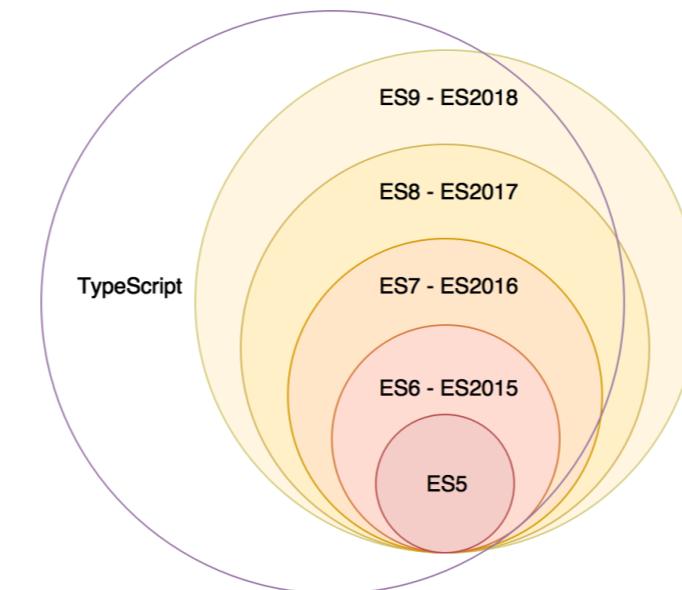


- Language created by **Anders Hejlsberg** in 2012
- Open-source project maintained by **Microsoft**
- Influenced by **Java** and **C#**
- Alternatives : CoffeeScript, Dart, Haxe or Flow

# TYPESCRIPT



- TypeScript is a superset of **JavaScript**
- Compile to **JavaScript**
- Support all versions of JavaScript (ES3 / ES5 / ES2015 / ...)
- Some functionalities have no impact on the generated JavaScript (like Interfaces)
- Every **JavaScript** program is a **TypeScript** program





# TYPESCRIPT - FEATURES

- Features of **ES2015+**
- Types
- Generics
- Interfaces
- Decorators
- Definitions files
- ...



# TYPESCRIPT - BASIC TYPES

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

```
const isVisible: boolean = true;  
let age: number = 32;
```

There are several places where type inference is used to provide type information when there is no explicit type annotation:

```
const isVisible = true; // is still of type boolean  
let age = 32;           // is still of type number
```

Some other types:

```
const name: string = 'Carl';  
const names: string[] = ['Carl', 'Laurent'];  
const notSure: any = 4;
```

# 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 - ARRAYS



Can be defined in two ways:

- Using Array literal notation
- Using Array constructor

```
let list: number[] = [1, 2, 3];  
  
let list: Array<number> = new Array<number>(1, 2, 3);  
  
let list: Array<number> = Array<number>(1, 2, 3); // The new operator is optional
```

- The result is the same, but **array literal notation** is more commonly used



# TYPESCRIPT - ENUMS

- Use the `enum` keyword
- Clarify a dataset
- Can be used as a type

```
enum Music { Rock, Jazz, Blues }; // is equivalent to: { Rock = 0, Jazz = 1, Blues = 2 }
let music: Music = Music.Jazz;
```

- Numeric values start at `0`
- Redefinition of the numeric value is allowed
- You can get back the string linked to the number

```
enum Music { Rock = 2, Jazz = 4, Blues = 8 };

let style: string = Music[4]; // --> Jazz
```

# TYPESCRIPT - ENUMS | ALTERNATIVE



- Use plain object and `as const` syntax

```
const Music = { Rock: 0, Jazz: 1, Blues: 2 } as const;  
  
type Music = typeof Music[keyof typeof Music]; // --> 0 | 1 | 2  
  
let music: Music = Music.Jazz;
```

- This solution is just `standard JavaScript` pseudo-enums with some quirky annotations
- Can still be used as a type
- But you can no longer get back the string linked to the number

# 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 {  
    public name!: string;  
  
    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**
- **private** alternative: use standard JavaScript private class features using hash **#** prefix
- **public** is the default scope
- Possibility to define **static** and **readonly** properties and **static** methods
- TypeScript provides a shortcut to link constructor arguments to class properties:

```
class Person {  
    constructor(public firstName: string) {}  
}  
  
// is equivalent to:  
class Person {  
    public firstName: string;  
  
    constructor(firstName: string) {  
        this.firstName = firstName;  
    }  
}
```

# TYPESCRIPT - CLASSES | INHERITANCE



- Inheritance system use keyword **extends**
- If no constructor is defined, the parent class constructor will be executed
- **super** keyword will call the parent implementation
- Parent class' properties and methods with **public** or **protected** scope are accessible

```
class Person {  
    constructor() {} // optional  
  
    speak() {}  
}  
  
class Child extends Person {  
    constructor() { super() } // optional  
  
    speak() { super.speak(); }  
}
```

# TYPESCRIPT - INTERFACES



- Can be used on classes with the `implements` keyword
- Can be also used to define: function signature, object shape, ...
- Used by the compiler to check object validity
- Have no impact on generated JavaScript
- Compiler throw an error while the interface contract is not respected
- Possibility to inherit an interface from another one

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

# 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

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

```
npm install <packageName>
```

Install a module globally on your system. Similar to an `apt-get install` on a linux system.  
Mostly used to install cli tools

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

Update a package

```
npm update <packageName>
```

Delete a package

```
npm remove <packageName>
```



# NPM - PACKAGE.JSON

- npm generate a **package.json** file which describes the project
- This file contains several informations:
  - **name**
  - **version**: should respect node-semver
  - **scripts**: can be run from the command line using `npm run <scriptName>...`
  - Dependencies: **dependencies**, **devDependencies**, **peerDependencies**
  - Meta data: **description**, **authors**, ...
  - ...

# 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>` / `npm i -D <packageName>`
- **peerDependencies:**
  - Needed for some modules to work but not installed with `npm install`
  - Typically used for libraries

# ANGULAR - HISTORIC

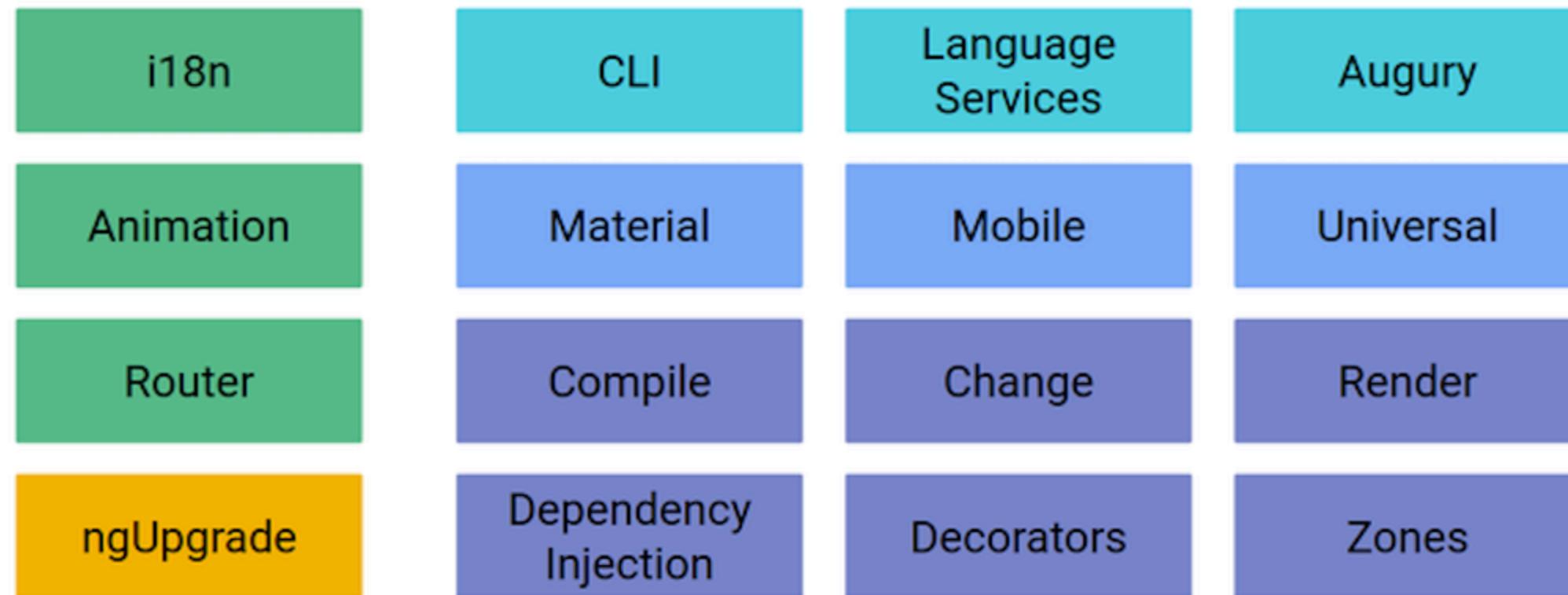


- Framework created by **Google** and announced in 2014
- Total rewrite of **AngularJS**
- Some concepts of **AngularJS** remain
- First release of **Angular 2.0.0** in September 2016
- Last major version **17.0.0** released in November 2023
- **Component oriented Framework**
- Documentation: <http://angular.io/> and <http://angular.dev/>

# ANGULAR - FRAMEWORK



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



# ANGULAR - PACKAGES



- All modules are scoped packages, imported as: `@angular/<packageName>`
- Some of the most commonly used packages:

```
import { animate, style, transition, trigger } from '@angular/animations';

import { AsyncPipe, DatePipe, NgClass, NgFor, NgIf } from '@angular/common';

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

import { Component, Directive, Injectable, Pipe, Provider } from '@angular/core';

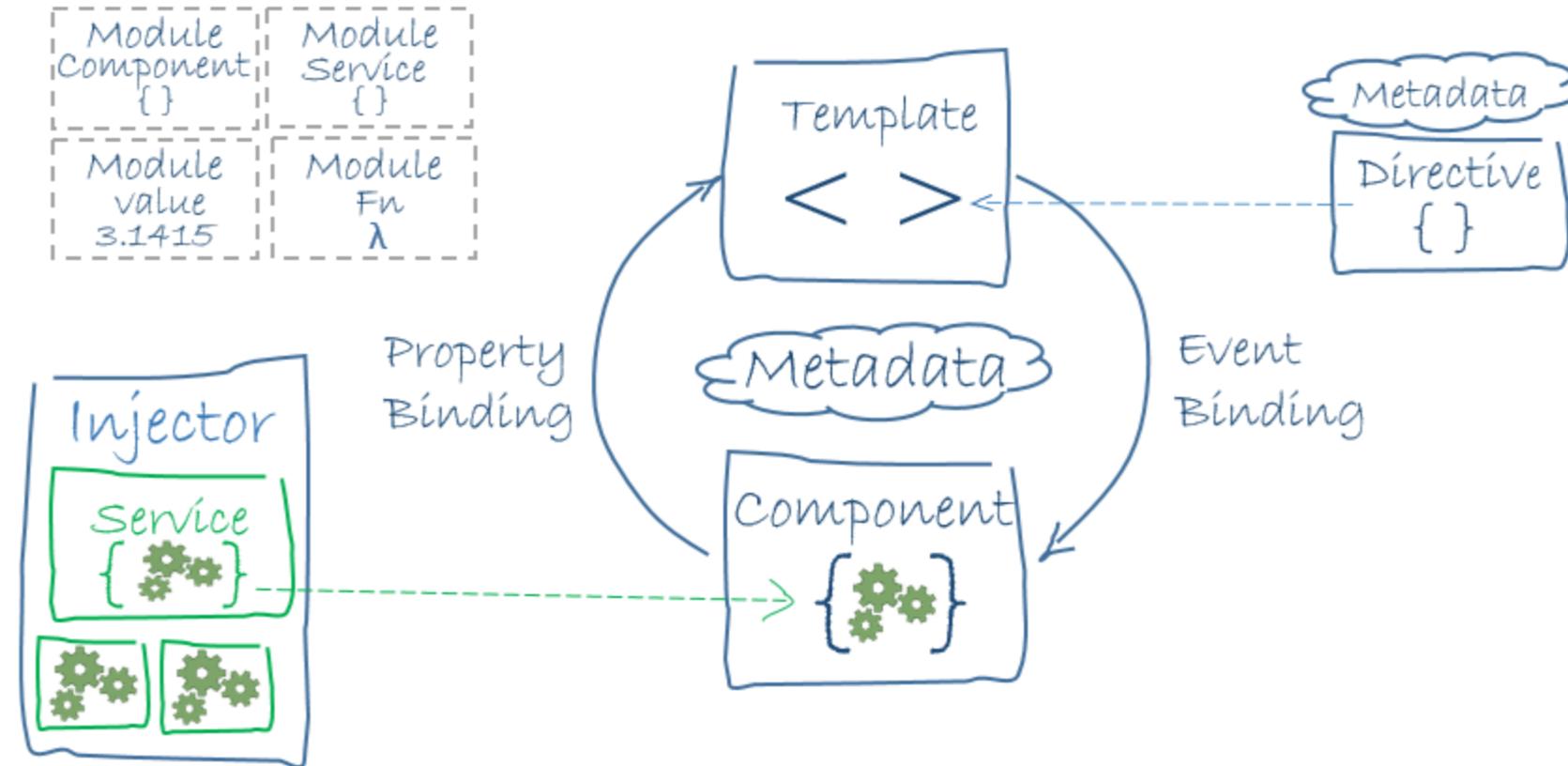
import { FormControl, FormGroup, FormsModule, ReactiveFormsModule } from '@angular/forms';

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

import { provideAnimations } from '@angular/platform-browser/animations';

import { ActivatedRoute, CanMatchFn, provideRouter, Router, Routes } from '@angular/router';
```

# ANGULAR - ARCHITECTURE





# ANGULAR - MODULES

- Help organize related things together

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

import { AppComponent } from './app.component';
import { LoginService } from './login/login.service';

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

- Optionals since Angular 15 when following **standalone components architecture**

# ANGULAR - COMPONENTS



- The main building block for Angular applications
- Each component consists of:
  - An **HTML template** that declares what renders on the page
  - A **TypeScript class** that defines behavior
  - A **CSS selector** that defines how the component is used in a template
  - Optionally, **CSS styles** applied to the template

# ANGULAR - COMPONENTS



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

@Component({
  selector: 'app-likes', // Usage in template: <app-likes [numberOfLikes]="3"></app-likes>
  template:
    <button (click)="toggleLike()">
      {{ numberOfLikes }}
      <i class="icon" [class.liked]="isLiked"> ❤ </i>
    </button>
  ,
})
export class LikesComponent {
  @Input() public numberOfLikes = 0;

  protected isLiked = false;

  protected toggleLike() {
    this.isLiked = !this.isLiked;
  }
}
```



# ANGULAR - DIRECTIVES

- 2 types: **Attribute** and **Structural** directives
- Common directives provided by the framework: **NgIf**, **NgFor**, **NgClass**, ...

Attribute directive example:

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

@Directive({
  selector: '[appHighlight]' // Usage in template: <div appHighlight></div>
})
export class HighlightDirective {

  constructor(private elementRef: ElementRef) {
    this.elementRef.nativeElement.style.backgroundColor = 'yellow';
  }
}
```



# ANGULAR - PIPES

- Common pipes provided by the framework: **DatePipe, CurrencyPipe, AsyncPipe, ...**

```
 {{ myDate | date | uppercase }} <!-- FRIDAY, APRIL 15, 1988 -->  
 {{ myPrice | currency : 'EUR' : true }} <!-- 53.12€ -->
```

- Custom pipe

```
import { PipeTransform, Pipe } from '@angular/core';  
  
@Pipe({  
  name: 'myLowerCase'  
})  
export class MyLowerCasePipe implements PipeTransform {  
  
  transform(value: string, param1: any, param2: any): string {  
    return value.toLowerCase();  
  }  
}
```

# ANGULAR - SERVICES



- A broad category encompassing any value, function or feature that an application needs
- Is typically a class with a narrow, well-defined purpose
- Should do something specific and do it well

```
import { Observable } from 'rxjs';
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { User } from './user.types';

@Injectable({
  providedIn: 'root'
})
export class UserService {
  constructor(private httpClient: HttpClient) {}

  getUsers(): Observable<User> {
    this.httpClient.get<User>('http://localhost/api/users');
  }
}
```

# ANGULAR - PROVIDERS



- A provider is an instruction to the **Dependency Injection system** on how to obtain a value for a dependency
- Most of the time, these dependencies are **services** that you create and provide
- Warning: you can't use TypeScript interface as injection token

# ANGULAR - PROVIDERS



```
@NgModule({
  providers: [
    UserService, // Most common case: a class to instanciate
    {
      provide: LoginService, // For an instance of that class
      useClass: LoginServiceImpl // You have to instanciate this class
    },
    {
      provide: ServerConfig, // For an instance of that class
      useFactory: serverConfigFactory, // Use this factory function
      deps: [AppService] // Inject these parameters to the factory
    }
  ]
})
export class AppModule {}

function serverConfigFactory (appService: AppService): ServerConfig {
  return appService.getConfig();
}
```





# Lab 1



# ANGULAR CLI

# SUMMARY



- Reminders
- *Angular CLI*
- Best practices
- Reactivity
- RxJS
- Standalone components
- Router
- Reactive Forms
- I18n
- SSR
- Animations
- Material
- Tests
- NgRx
- To go further :



# ANGULAR CLI

- A command-line interface tool that you use to **initialize**, **develop**, **scaffold**, and **Maintain** Angular applications directly from a command shell.
- Installation:

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

- Invoke the tool on the command line through the **ng** executable.
- Scaffold an Angular application, install dependencies and run development server:

```
ng new my-app  
cd ./my-app  
ng serve
```

- Alternatively, create a new Angular application without installing the CLI on your system:

```
npm init @angular my-app
```

# ANGULAR CLI



- When scaffolding an Angular application, all needed dependencies are installed:
  - **TypeScript**
  - **Webpack**
  - **Karma**
  - **Jasmine**
  - **Sass**
  - ...
- But applications can be configured to use alternative dependencies:
  - **EsbUILD** (instead of Webpack)
  - **Jest** (instead of Jasmine)
  - ...

# ANGULAR CLI



Several commands are available:

- `ng generate` generate the code of various angular building blocks:
  - `ng generate component` product
  - `ng generate directive` highlight
  - `ng generate pipe` upper-case
  - `ng generate service` user
- `ng serve`
- `ng build`
- `ng test`
- ...

# ANGULAR CLI - SCHEMATICS



- Used by the [Angular CLI](#).
- Template based code generator
  - Generate new code
  - Modify existing code
  - Install dependencies
- Use cases
  - Add a feature to the application (ESLint, Material, ...)
  - Install dependencies (related to these features)
  - Upgrade an application
  - ...

# ANGULAR CLI - SCHEMATICS | CONCEPTS



- Has access to the system package manager
- Uses a virtual file system

## Main blocks

- **Tree** { base, staging }
- **Rule**
- **Action** (CREATE, RENAME, OVERWRITE, DELETE)

# ANGULAR CLI - SCHEMATICS | EXAMPLE 1



- Add ESLint to an existing application

```
ng add @angular-eslint/schematics
```

- Install a bunch of `@angular-eslint/<packages>` packages
- Update `angular.json`
- Add `eslintrc.json`
- Add `lint` script in `package.json` (`npm run lint`)

Note that `Prettier` can be used in conjunction with `ESLint` to provide the best possible development experience

```
npm i -D prettier eslint-config-prettier prettier-plugin-organize-imports
```

Read this to configure ESLint to work nicely with Prettier:

<https://prettier.io/docs/en/install.html#eslint-and-other-linters>

# ANGULAR CLI - SCHEMATICS | EXAMPLE 2



- Add Angular Material to an existing application

```
ng add @angular/material
```

- Install `@angular/material` and `@angular/cdk` packages
- Add `provideAnimations()` in the list of `ApplicationConfig` providers
- Add the selected theme in `angular.json`
- Import `Roboto` font in `index.html`

# ANGULAR CLI - CONFIGURATION FILE



- The `angular.json` file at the root level of an Angular workspace
- A workspace can contains multiple projects
- Path values given in the configuration are relative to the root workspace directory

Learn more about the workspace configuration: <https://angular.io/guide/workspace-config>

# ANGULAR CLI - CONFIGURATION FILE



- The `projects` object contains the configuration of each project ("my-app" in this case)
- `projectType` property values are `application` or `library`

```
{  
  "projects": {  
    "my-app": {  
      "projectType": "application",  
      "schematics": { ... },  
      "root": "",  
      "sourceRoot": "src",  
      "prefix": "app",  
      "architect": {  
        "build": { ... },  
        "serve": { ... },  
        "extract-i18n": { ... },  
        "test": { ... }  
      }  
    }  
  }  
}
```





## Lab 2



# BEST PRACTICES

# SUMMARY



- Reminders
- Angular CLI
- *Best practices*
- Reactivity
- RxJS
- Standalone components
- Router
- Reactive Forms
- I18n
- SSR
- Animations
- Material
- Tests
- NgRx
- To go further

# BEST PRACTICES - DOMAIN-DRIVEN DESIGN (DDD)



DDD is an approach to help making **design decisions** on software:

- It focuses on the **domain** and its **logic**
- It provides **tools** for technical and domain experts to collaborate effectively (ubiquitous language, ...)

The folder tree of the Angular application should reflect DDD.

So, for example an e-commerce application should be organized like this:

```
app
  └── catalog
  └── basket
  └── checkout
  └── delivery
```



# BEST PRACTICES - FILE TYPE

- File name should be followed by the **file type**
- Don't hesitate to create your own types (**\*.utils.ts, \*.types.ts, ...**)

```
my-feature.component.ts  
my-feature.component.html  
my-feature.component.css  
my-feature.service.ts  
my-feature.config.ts  
my-feature.utils.ts  
my-feature.types.ts
```

- Test file ends with **.spec.ts** and should be located right next to the source file

```
my-feature.service.ts  
my-feature.service.spec.ts
```

# BEST PRACTICES - PIPES



- Use **pure pipes** as much as possible (this is the default behavior)
- Pure pipes are only executed on **pure changes**:
  - The input parameter reference has changed
- Impure pipes are less performant:
  - Executed on every change detection cycle

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

@Pipe({
  name: 'quote',
  pure: true // optional
})
export class QuotePipe implements PipeTransform {
  transform(value: string, quote = ''): string {
    return quote + value + quote;
  }
}
```

# BEST PRACTICES - CLASS PROPERTY ACCESSORS



- Don't make accessors for no reason:

```
class ProductComponent {  
    // Oh no!  
    private _total = 0;  
  
    get total() {  
        return this._total;  
    }  
  
    set total(value: number) {  
        this._total = value;  
    }  
}
```

```
class ProductComponent {  
    // Yes :)  
    total = 0;  
}
```

# BEST PRACTICES - HTML SEMANTIC



HTML semantic matter!

- Don't add `<div>` or whatever HTML element if not needed
- Use `<ng-container>` (and `<ng-template>`) instead:

```
<!-- Oh no! -->
<div *ngIf="basket">
  <h1>List of products</h1>
  <p *ngFor="let product of basket">{{ product.name }}</p>
</div>
```

```
<!-- Yes :) -->
<ng-container *ngIf="basket">
  <h1>List of products</h1>
  <p *ngFor="let product of basket">{{ product.name }}</p>
</ng-container>
```

# BEST PRACTICES - SERVICES



- A service should have only one **single responsibility**
- Remember not to subscribe to observables in services...
  - ...but in components only



# BEST PRACTICES - SERVICES 1/3

- Fetching and storing data in a component (so, no use of services at this point)

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

@Component({ /* ... */ })
class MyItemsComponent implements OnInit {
  protected myItems: MyItem[] = [];

  constructor(private apiService: ApiService) {}

  ngOnInit(): void {
    this.fetchMyItems();
  }

  private fetchMyItems(): Subscription {
    return this.apiService.getMyItems().subscribe( // <-- subscribe
      (myItems) => (this.myItems = myItems)
    );
  }
}
```

# BEST PRACTICES - SERVICES 2/3



- Fetching and storing data in a service
- Replacing `.subscribe(callbackFn)` with `.pipe(tap(callbackFn))`

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

@Injectable({ /* ... */ })
class MyItemsService {
  myItems: MyItem[] = [];

  fetch(): Observable<MyItem[]> {
    return this.apiService.getMyItems().pipe(
      tap((myItems) => (this.myItems = myItems)) // <-- tap
    );
  }
}
```

# BEST PRACTICES - SERVICES 3/3



- Consumming service data in a component

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

@Component({ /* ... */ })
class MyItemsComponent implements OnInit {
  protected get myItems(): MyItem[] { // <-- getter
    return this.myItemsService.myItems;
  }

  constructor(private myItemsService: MyItemsService) {}

  ngOnInit(): void {
    this.myItemsService.fetch().subscribe(); // <-- trigger the data fetching
  }
}
```

# BEST PRACTICES - CATCHING ERRORS



- Always handle errors of **observables** (and promises)

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

@Component({ /* ... */ })
class MyItemsComponent implements OnInit {
  protected get myItems(): MyItem[] {
    return this.myItemsService.myItems;
  }

  protected showMessage = false;

  constructor(private myItemsService: MyItemsService) {}

  ngOnInit(): void {
    this.myItemsService.fetch().subscribe({
      error: () => (this.showMessage = true)
    });
  }
}
```





## Lab 3



# REACTIVITY

# SUMMARY



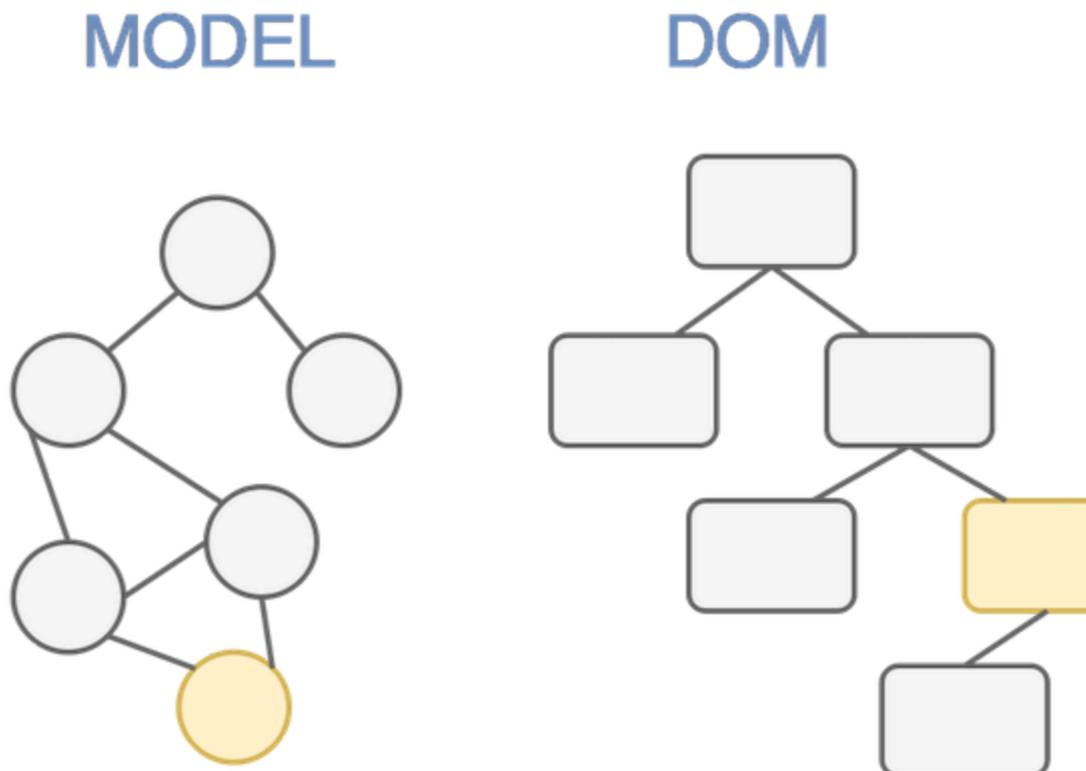
- Reminders
- Angular CLI
- Best practices
- *Reactivity*
- RxJS
- Standalone components
- Router
- Reactive Forms
- I18n
- SSR
- Animations
- Material
- Tests
- NgRx
- To go further

# REACTIVITY



*How data get displayed and updated in Angular?*

- Data binding: binding part of the **HTML** template to a **property** of an Angular component
- Event binding: binding DOM **event** to a **method** of an Angular component



# REACTIVITY - ASYNCHRONOUS TASK



*When apps typically update HTML?*

- On component initialization
- When running an event listener callback
- When getting response data from a server through an HTTP request
- When running macroTask such as setTimeout() or setInterval() callbacks
- When running microTask such as Promise.then() or Promise.catch() callbacks

In other words, it may be necessary to update the HTML every time an asynchronous task occurs



# REACTIVITY - ZONE.JS

*Who is responsible for telling Angular that an asynchronous task has occurred?*

- Zone.js intercepts asynchronous APIs through **monkey patching**
- Created by Brian Ford and inspired by Dart
- Zone.js implements Zones for JavaScript
- Angular has its own zone called **NgZone**
- A zone is an execution context that persists across asynchronous tasks

Zone.js is responsible for telling Angular when to trigger its *change detection* process

# REACTIVITY - CHANGE DETECTION



*How does Angular determine which part of the app state has changed?*

- Angular stores the current state of the app
- After being aware of an asynchronous task, Angular compares the new state of the app to the previous known state that was stored
- If the states differ, Angular updates the DOM (according to the different data bindings)

In summary, when an asynchronous task occurs, Zone.js tells Angular to run its change detection and update the DOM if necessary

# REACTIVITY - CHANGE DETECTION STRATEGY



- Every time Angular runs its change detection, it has to check the entire **components tree!**
- For large tree, this can lead to performance issues
- Performance can be improved by using **OnPush** strategy

OnPush change detection instructs Angular to run change detection for a component subtree only when:

- The root component of the subtree receives **new inputs** as the result of a **template binding**
- Angular **handles an event in the subtree's root component or any of its children** whether they are using OnPush change detection or not

# REACTIVITY - ONPUSH AND IMMUTABILITY



When using the **OnPush** strategy, remember to use immutable objects, otherwise changes to the component Input will not be detected.

- Using mutation

```
const person = { email: 'contact@zenika.com', name: 'Carl' };
person.name = 'Laurent';
```

The reference to the **person** object has not been changed. Only the **name** property has changed.

- Using immutable pattern

```
let person = { email: 'contact@zenika.com', name: 'Carl' };
person = { ...person, name: 'Laurent' };
```

This time, the **person** object is a completely different object

# REACTIVITY - ONPUSH AND IMMUTABILITY



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

@Component({
  selector: 'app-parent',
  template: `
    <app-child [childData]="parentData" />
    <button (click)="mutate()">Mutate</button> <button (click)="update()">Update</button>
  `,
})
export class ParentComponent {
  protected parentData = { name: 'Carl' };
  protected mutate() { this.parentData.name = 'Laurent' } // NO
  protected update() { this.parentData = { name: 'Laurent' } } // YES
}

@Component({
  selector: 'app-child',
  template: `<h1>{{ childData.name }}</h1>`,
  changeDetection: ChangeDetectionStrategy.OnPush,
})
export class ChildComponent {
  @Input({ required: true }) childData!: { name: string };
}
```

# REACTIVITY - CHANGEDETECTORREF



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

@Component({
  selector: 'app-child',
  template: `<h1 (click)="update()">{{ childData?.name }}</h1>`,
  changeDetection: ChangeDetectionStrategy.OnPush,
})
export class ChildComponent {
  protected childData?: { name: string };

  private httpClient = inject(HttpClient);
  private changeDetectorRef = inject(ChangeDetectorRef);

  update() {
    this.httpClient.get<{ name: string }>('/api/child-data').subscribe((childData) => {
      this.childData = childData;
      this.changeDetectorRef.markForCheck(); // <-- Manual triggering
    });
  }
}
```

# REACTIVITY - ASYNCPIPE



Subscribing to an observable in the template automatically triggers the change detection when value is emitted

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

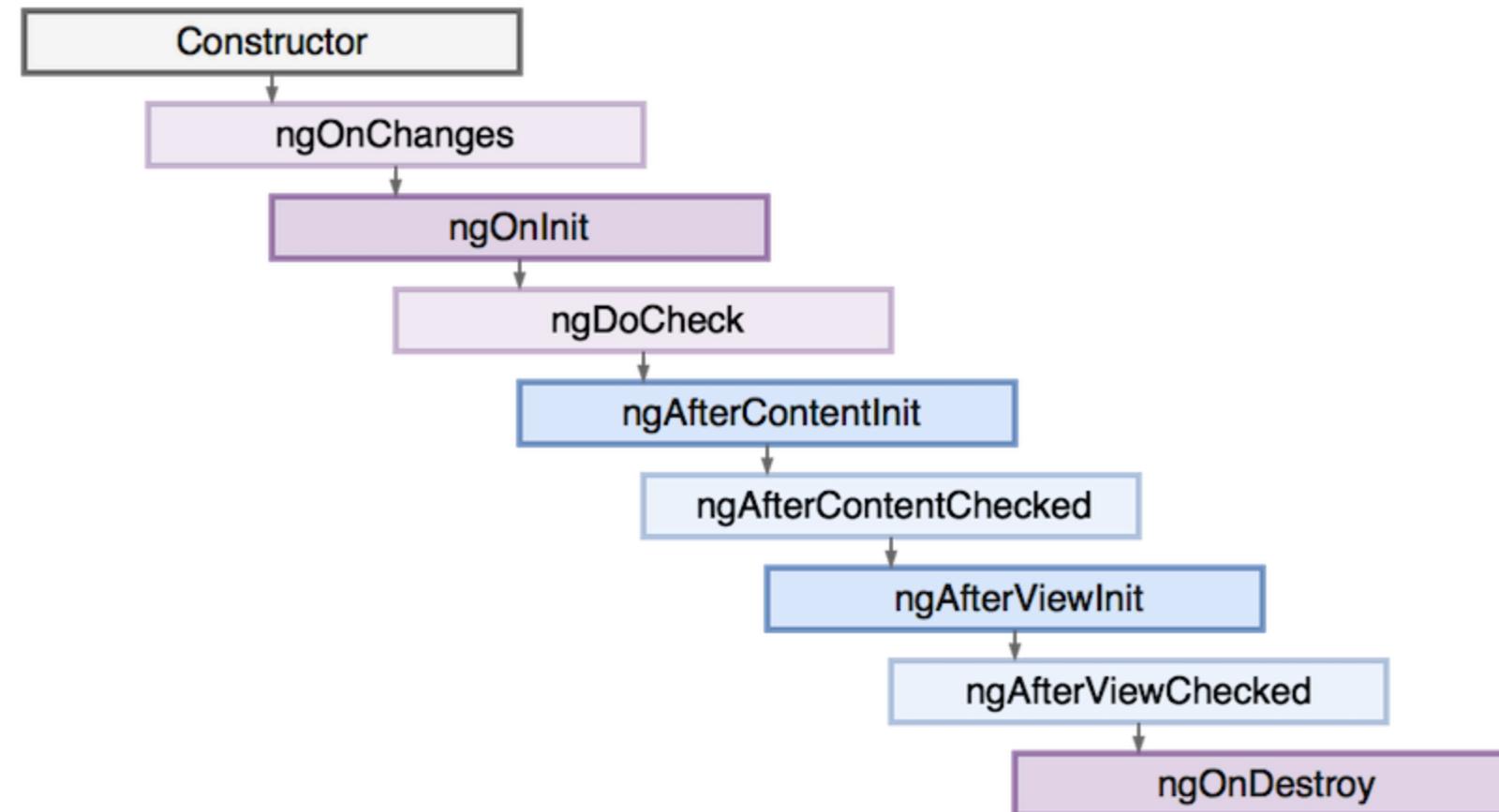
@Component({
  selector: 'app-child',
  template: `<h1>{{ (childData$ | async)?.name }}</h1>`,
  changeDetection: ChangeDetectionStrategy.OnPush,
})
export class ChildComponent {
  private httpClient = inject(HttpClient);

  protected childData$ = this.httpClient.get<{ name: string }>('/api/child-data');
}
```

# REACTIVITY - COMPONENT LIFECYCLE



- Angular creates, updates and destroys components or directives
- Code can be executed at each step of a component/directive lifecycle



# REACTIVITY - COMPONENT LIFECYCLE



- **ngOnChanges**: when an input/output binding value changes
- **ngOnInit**: initialize the component (called once, after the first ngOnChanges)
- **ngDoCheck**: developer's custom change detection
- **ngAfterContentInit**: after component's content is initialized
- **ngAfterContentChecked**: after every check of component content
- **ngAfterViewInit**: after a component's view is initialized
- **ngAfterViewChecked**: after every check of a component's view
- **ngOnDestroy**: just before destruction

# REACTIVITY - COMPONENT LIFECYCLE | OnInit



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

@Component({
  selector: 'app-child',
  template: `<h1>{{ email$ | async }}</h1>`,
  changeDetection: ChangeDetectionStrategy.OnPush,
})
export class ChildComponent implements OnInit {
  @Input({ required: true }) childData!: { name: string };

  protected email$: Observable<string>;
  private httpClient = inject(HttpClient);

  ngOnInit(): void {
    this.email$ = this.httpClient.get<string>(`/api/name/${this.childData.name}/email`);
  }
}
```

- Component inputs are only available from `ngOnInit` and later  
(like in the example above where `email$` depends on `childData` which is an `@Input`)

# REACTIVITY - SIGNALS



Later in the course:

- New way of implementing reactivity in Angular with *signals*
- Enabling *zoneless* applications with unmatched performance





**RXJS**

# SUMMARY



- Reminders
- Angular CLI
- Best practices
- Reactivity
- *RxJS*
- Standalone components
- Router
- Reactive Forms
- I18n
- SSR
- Animations
- Material
- Tests
- NgRx
- To go further

# 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, ...
- Angular uses the JavaScript implementation: **RxJS**
- Observables:
  - represent a stream of data that can be subscribed to
  - allowing multiple values to be emitted over time





- Observables are used everywhere in the Angular framework: **HTTP, Router, ...**
- **Best practices** of an Angular application can be achieved using observables
- Understanding observables is crucial to master the Angular framework
- To understand RxJS, you need to learn the following concepts:
  - **Observable**
  - **Observer**
  - **Subscription**
  - **Operators**
  - **Subjects**

# RXJS - OBSERVABLE & OBSERVER



```
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
});

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

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

- Use the **subscriber** to shape the behavior of the observable
- Use the **observer** to specify which events you want to listen to
- Subscriber and observer methods match: **next**, **complete** (and also **error**)

# RXJS - OBSERVABLE & OBSERVER



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

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

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

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

- Example of **error** event instead of **complete** event

# RXJS - OBSERVABLE & OBSERVER



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

const data$ = new Observable<number>((subscriber) => {
  subscriber.next(1);
  subscriber.next(2);
  subscriber.complete();
  subscriber.next(3);                                // <-- Value NOT emitted
});

const next = (data: number) => console.log(data); // <-- Function as observer

data$.subscribe(next);                            // <-- same as `data$.subscribe({ next })`;

// output: 1, 2
```

- Once the observable completes (or is in error), further calls to `next` are ignored
- You can use a function as observer to simply listen to "next" events

# RXJS - SUBSCRIPTION 1/3



- Example of an observable that completes itself properly (without memory leak)

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

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

  const interval = setInterval(() => {
    subscriber.next(++data); // <-- Emit next value every second

    if (data === 3) { // <-- Until this value
      clearInterval(interval); // <-- Cleanup interval to prevent memory leak
      subscriber.complete(); // <-- Then mark as complete
    }
  }, 1000);
});

data$.subscribe({
  next: (data: number) => console.log(data),
  complete: () => console.log('Done'),
}); // output: 1, 2, 3, Done
```



# RXJS - SUBSCRIPTION 2/3

- Example of an observable that never completes and have a **memory leak!**

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

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

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

});

const subscription: Subscription = data$.subscribe((data: number) => {
  console.log(data);
  if (data === 3) {
    subscription.unsubscribe();        // <-- Unsubscribe from data$
                                      //       but the observable still ticking...
  }
}); // output: 1, tick, 2, tick, 3, tick, tick, tick, ...
```

# RXJS - SUBSCRIPTION 3/3



- Example of an observable that never completes but cleans up itself properly

```
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 - OBSERVABLE SOURCE



- Observable can be created using **of** function:

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

const source$ = of('hello', 123);

source$.subscribe(console.log); // output: hello, 123
```

- Observable can be created from existing value (like Array or Promise) using **from** function:

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

const fromArray$ = from(['hello', 123]);

fromArray$.subscribe(console.log); // output: hello, 123

const fromPromise$ = from(new Promise((resolve) => resolve('Done!')));

fromPromise$.subscribe(console.log); // output: Done!
```

# RXJS - OBSERVABLE SOURCE



- Observable can be created using `fromEvent` function:

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

const fromDocumentClick$ = fromEvent(document, 'click');

fromDocumentClick$.subscribe((event: Event) => console.log(event));
```

- Observable that emits an error event can be created using `throwError` function:

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

const error$ = throwError(() => new Error('Oops!'));

error$.subscribe({
  error: (err: Error) => console.error(err.message) // output: Oops!
});
```

# RXJS - OPERATORS | SYNCHRONOUS



```
import {  
  Observable, filter, map // <-- "filter" and "map": synchronous transformations  
} from 'rxjs';  
  
const data$ = new Observable<number>((subscriber) => {  
  subscriber.next(1);  
  subscriber.next(2);  
  subscriber.next(3);  
  subscriber.next(4);  
  subscriber.complete();  
});  
  
data$.pipe(/* no operator */).subscribe(console.log); // output: 1, 2, 3, 4  
  
data$.pipe(filter((data) => data % 2 === 0)).subscribe(console.log); // output: 2, 4  
  
data$.pipe(map((data) => data * 10)).subscribe(console.log); // output: 10, 20, 30, 40  
  
data$.pipe(  
  filter((data) => data % 2 === 0),  
  map((data) => data * 10)  
).subscribe(console.log); // output: 20, 40
```

# RXJS - OPERATORS | ASYNCHRONOUS



```
import { Observable, concatMap } from 'rxjs'; // <-- "concatMap": asynchronous transformation

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

const fetchTodoFactory$ = (id: number) => new Observable<Todo>((subscriber) => {
  fetch(`https://jsonplaceholder.typicode.com/todos/${id}`)
    .then((response) => response.json())
    .then((todo: Todo) => {
      subscriber.next(todo);           // <-- Emit "next" event
      subscriber.complete();          // <-- Emit "complete" event
    })
    .catch((err) => subscriber.error(err)); // <-- Emit "error" event
});

todoId$.pipe(concatMap((id) => fetchTodoFactory$(id))).subscribe(console.log);

// output: { id: 1, title: 'delectus aut autem', completed: false }
// output: { id: 2, title: 'quis ut nam facilis et officia qui', completed: false }
```

# RXJS - MORE OPERATORS...



- **concatMap**  
Projects each source value to an Observable which is merged in the output Observable, in a serialized fashion waiting for each one to complete before merging the next.
- **mergeMap**  
Projects each source value to an Observable which is merged in the output Observable.
- **switchMap**  
Projects each source value to an Observable which is merged in the output Observable, emitting values only from the most recently projected Observable.
- **combineLatest**  
Combines multiple Observables to create an Observable whose values are calculated from the latest values of each of its input Observables.
- **debounceTime**  
Emits a notification from the source Observable only after a particular time span has passed without another source emission.

# RXJS - OPERATORS | CATCHERROR



- The `catchError` operator should:
  - return another observable
  - throw again to be handled by another `catchError` or the observer's `error` handler

```
import { interval, tap, catchError, of } from 'rxjs';

const source$ = interval(1000).pipe(
  tap((value) => {
    if (value > 3) throw new Error('Oops!');
  }),
  catchError(() => of('Fallback'))
);

source$.subscribe({
  next: console.log,
  error: console.error,
  complete: () => console.log('Done!')
});

// Output => 0, 1, 2, 3, Fallback, Done!
```

# RXJS - SUMMARY SO FAR



- 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 - SUBJECT



- A **Subject** implements both **Observable** and **Observer** interfaces

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

const subject$ = new Subject();

// Act as Observable
subject$.subscribe(/* ... */);
subject$.pipe(/* ... */);

// Act as Observer
subject$.next(/* ... */);
subject$.error(/* ... */);
subject$.complete(/* ... */);

// Can be converted into a simple Observable...
const observable$ = subject$.asObservable(/* ... */);

// ...hiding the Observer interface
observable$.next(/* ... */); // ✖ Property 'next' does not exist on type 'Observable'
```

# RXJS - SUBJECT



Unlike the observable:

- a subject implementation lives outside its instantiation (calling `next`, `error`, `complete`)
- a subject can emit stream events even before any subscription ("hot" observable)
- a subject is "multicast" (all subscribers share the same stream events)

```
const data$ = new Subject<string>();

data$.next('A');                                // <-- value is lost

data$.subscribe((data) => console.log(`#sub1(${data})`));
data$.next('B');                                // <-- value received by subscriber 1

data$.subscribe((data) => console.log(`#sub2(${data})`));

data$.next('C');                                // <-- value received by subscribers 1 and 2
data$.next('D');                                // <-- value received by subscribers 1 and 2
data$.complete();
// output: #sub1(B), #sub1(C), #sub2(C), #sub1(D), #sub2(D)
```

# RXJS - OBSERVABLE COMPARED TO SUBJECT



Unlike the subject:

- an observable implementation lives inside its instantiation (calling `next`, `error`, `complete`)
- an observable emits stream events only when subscribing ("cold" observable)
- an observable is "unicast" (each subscriber receive a new data stream)

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

const observable$ = new Observable<string>((subscriber) => {
  // This is where implementation takes place...
  subscriber.next('A');
  subscriber.next('B');
  subscriber.complete();
});

data$.subscribe((data) => console.log(`#sub1(${data})`));
data$.subscribe((data) => console.log(`#sub2(${data})`));

// output: #sub1(A), #sub1(B), #sub2(A), #sub2(B)
```

# RXJS - SUBJECT AS OBSERVER



- As an observer, a **Subject** can subscribe to an **Observable**!

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

const observable$ = new Observable<string>((subscriber) => {
  subscriber.next('A');
  subscriber.next('B');
  subscriber.complete();
});

const subject$ = new Subject<string>();
subject$.subscribe(console.log);           // output: A, B

// Doing this...
observable$.subscribe(subject$);          // <-- `subject$` acting as `Observer`

// ...is equivalent to
observable$.subscribe({
  next: (observable: string) => subject$.next(observable),
  complete: () => subject$.complete(),
  error: (err: unknown) => subject$.error(err),
});
```

# RXJS - SUBJECT | BEHAVIORSUBJECT



A variant of Subject that requires an initial value and emits its current value whenever it is subscribed to.

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

const data$ = new BehaviorSubject<string>('A'); // <-- Initial value

data$.subscribe((data) => console.log(`#sub1(${data})`)); // <-- #sub1 receive 'A'

data$.next('B'); // <-- #sub1 receive 'B'

data$.subscribe((data) => console.log(`#sub2(${data})`)); // <-- #sub2 receive 'B'

data$.next('C');
data$.next('D');

console.log(`#snapshot(${data$.value})`); // <-- and you have access to the instant value!

data$.complete();

// output: #sub1(A), #sub1(B), #sub2(B), #sub1(C), #sub2(C), #sub1(D), #sub2(D), #snapshot(D)
```

# RXJS - SUBJECT | REPLAYSUBJECT



A variant of Subject that "replays" old values to new subscribers by emitting them when they first subscribe.

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

const data$ = new ReplaySubject<string>(2); // <-- Number of events to replay

data$.next('A');

data$.subscribe((data) => console.log(`#sub1=${data}`)); // <-- #sub1 receive 'A'

data$.next('B'); // <-- #sub1 receive 'B'

data$.subscribe((data) => console.log(`#sub2=${data}`)); // <-- #sub2 receive 'A' and 'B'

data$.next('C');
data$.next('D');
data$.complete();

// output: #sub1(A), #sub1(B), #sub2(A), #sub2(B), #sub1(C), #sub2(C), #sub1(D), #sub2(D)
```

# RXJS - ANGULAR STATE MANAGEMENT 1/3



- Expose application data through service facade and observables

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

@Injectable({ providedIn: 'root' })
export class TodoService {
  private _todos$ = new BehaviorSubject<Todo[] | undefined>(undefined);
  todos$ = this._todos$.asObservable();
  get todosSnapshot() { return this._todos$.value; }

  constructor(private httpClient: HttpClient) {}

  fetch(): Observable<void> {
    return this.httpClient.get<Todo[]>('https://jsonplaceholder.typicode.com/todos').pipe(
      tap((todos) => {
        this._todos$.next(todos); // <-- Using `tap` operator for "side-effects"
      }),
      map(() => undefined), // <-- Force the consumer to use the `todos$` property
    );
  }
}
```

# RXJS - ANGULAR STATE MANAGEMENT 2/3



- Determine the appropriate place to trigger data fetching
- Don't forget to handle errors!

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

@Component({
  selector: 'app-root',
  template: `<h1>My todo list</h1>
    <app-todo-list />
    <p *ngIf="showError">Sorry, a problem occurred!</p>`
})
export class AppComponent implements OnInit {
  showError = false;

  constructor(private todoService: TodoService) {}

  ngOnInit() {
    this.todoService.fetch().subscribe({ error: () => (this.showError = true) });
  }
}
```

# RXJS - ANGULAR STATE MANAGEMENT 3/3



- Consume the service facade in your components

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

@Component({
  selector: 'app-todo-list',
  template:
    <p *ngFor="let todo of todos">
      {{ todo.title }} is completed: {{ todo.completed ? 'yes' : 'no' }}.
    </p>
})
export class TodoListComponent implements OnDestroy {
  protected todos: Todo[] = [];

  private subscription = this.todoService.todo$.subscribe((todos) => (this.todos = todos));

  constructor(private todoService: TodoService) {}

  ngOnDestroy() { this.subscription.unsubscribe(); }
}
```

# ANGULAR - ASYNC PIPE



- Use the `async` pipe to consume the facade directly in your components template

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

@Component({
  selector: 'app-todo-list',
  template: `
    <p *ngFor="let todo of todos$ | async">
      {{ todo.title }} is completed: {{ todo.completed ? 'yes' : 'no' }}.
    </p>
  `
})
export class TodoListComponent {
  protected todos$ = this.todoService.todos$;

  constructor(private todoService: TodoService) {}
}
```



# ANGULAR - ASYNC PIPE

- Use the "**as**" syntax to create a local template variable and reduce the number of subscriptions in your components template

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

@Component({
  selector: 'app-todo-list',
  template: `
    <ng-container *ngIf="todos$ | async as todos">
      <h2>There's {{ todos.length }} item(s) in your todo list.</h2>

      <p *ngFor="let todo of todos">
        {{ todo.title }} is completed: {{ todo.completed ? 'yes' : 'no' }}.
      </p>
    </ng-container>
  `
})
export class TodoListComponent {
  protected todos$ = this.todoService.todos$;

  constructor(private todoService: TodoService) {}
}
```

# ANGULAR - ASYNC PIPE



The **async** pipe provides many advantages:

- It automatically subscribes to the **Observable**
- It automatically triggers change detection when new data is emitted from the **Observable**
- It automatically unsubscribes from the **Observable** when the component is about to get destroyed





## Lab 4



# STANDALONE COMPONENTS

# SUMMARY



- Reminders
- Angular CLI
- Best practices
- Reactivity
- RxJS
- *Standalone components*
- Router
- Reactive Forms
- I18n
- SSR
- Animations
- Material
- Tests
- NgRx
- To go further

# STANDALONE COMPONENTS - DEFINITION



- A standalone component is a type of component which is not part of any Angular module
- Before the introduction of standalone components, a component had to be declared inside the **declarations** array of an **NgModule**

```
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';

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

- Modules are an additional layer of abstraction
- They are complex and difficult to understand
- With the introduction of standalone component, they are now optional
- The component becomes the central part of the framework

# STANDALONE COMPONENTS - DECLARATION



- To declare a component as standalone, add a `standalone` property to the component metadata:

```
@Component({
  selector: 'app-product',
  standalone: true,
  templateUrl: './product.component.html'
})
export class ProductComponent {}
```

- `Pipe` and `Directive` can also be declared as standalone
- Generate standalone component using the Angular CLI:

```
ng generate component product --standalone
```

# STANDALONE COMPONENTS - DEPENDENCIES



- Standalone component must declare their dependencies (child components, directives and pipes) directly by passing them to the `imports` array
- These dependencies can be either:
  - standalone components, directives and pipes
  - regular modules exporting components, directives and pipes

```
import { NgIf } from '@angular/common';
import { MatButtonModule } from '@angular/material/button';

@Component({
  selector: 'app-product',
  standalone: true,
  imports: [
    NgIf, // `ngIf` standalone directive
    MatButtonModule, // `mat-button` classic directive declared in ` MatButtonModule `
  ],
  template: `<button *ngIf="true" mat-button>Add to basket</button>
})
export class ProductComponent {}
```

# STANDALONE COMPONENTS - USAGE



- To use a standalone component in the rest of the application, there are 2 possibilities
  - It can be imported into another **standalone** component
  - It can be imported into a **@NgModule** to gradually adopt the new standalone component style
- Either way, the component must be added to the module/component **imports** section:

```
@Component({
  imports: [StandaloneComponent, StandaloneDirective, StandalonePipe],
})
export class AppComponent {}
```

```
@NgModule({
  imports: [StandaloneComponent, StandaloneDirective, StandalonePipe],
})
export class AppModule {}
```

# STANDALONE COMPONENTS - BOOTSTRAPPING



- With standalone component architecture style, angular modules are now completely optional
- It is possible to bootstrap an application using a standalone component
- In the `main.ts` file:

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

import { AppComponent } from './app/app.component';
import { appConfig } from './app/app.config';

bootstrapApplication(AppComponent, appConfig).catch((err) => console.error(err));
```

- With the following `app.config.ts`:

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

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



# STANDALONE COMPONENTS - DI

- You can configure **Dependency Injection** during the application bootstrapping

```
import { provideHttpClient } from '@angular/common/http';
import { ApplicationConfig } from '@angular/core';
import { provideRouter } from '@angular/router';

import { appRoutes } from './app.routes';

export const appConfig: ApplicationConfig = {
  providers: [provideRouter(appRoutes), provideHttpClient()], // <- Angular v16+
};
```

- It is possible to extract the providers defined within a module:

```
export const appConfig: ApplicationConfig = {
  providers: [importProvidersFrom(AppRoutingModule, HttpClientModule)], // <- Angular v15
};
```



# STANDALONE COMPONENTS - DI

- Angular and some libraries includes now `provideXYZ()` and `withXYZ()` functions
- These functions allow you to configure the providers without needing to extract them from an `NgModule`

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

import { appRoutes } from './app.routes';

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

# STANDALONE COMPONENTS - MIGRATION



Angular offers a schematic to migrate an existing application to standalone component architecture style:

```
ng generate @angular/core:standalone
```

3 steps are available:

- Convert all components, directives and pipes to standalone
- Remove unnecessary NgModule classes
- Bootstrap the project using standalone APIs

# STANDALONE COMPONENTS - BENEFITS



- Make it easier to learn Angular (no need to learn **NgModule** abstraction)
- Reduce bundle size with less boilerplate code and explicit dependencies imports
- Component creation steps are greatly simplified
- Simplified lazy-loading configuration (covered in next section)
- The component is now the centerpiece of the framework





## Lab 5



# ROUTER

# SUMMARY



- Reminders
- Angular CLI
- Best practices
- Reactivity
- RxJS
- Standalone components
- *Router*
- Reactive Forms
- I18n
- SSR
- Animations
- Material
- Tests
- NgRx
- To go further



# ROUTER - REMINDERS

- The **Router** enables navigation by interpreting a browser URL as an instruction to change the view
- **RouterModule** is imported from the **@angular/router** package
- **Routes** configuration are typically defined in a specific file: **app.routes.ts**
- Register these routes via **RouterModule.forRoot** method

```
// app.routes.ts
export const appRoutes: Routes = [
  { path: 'contacts', component: ContactListComponent },
  { path: 'contacts/:id', component: ContactComponent },
];

// app.module.ts
@NgModule({
  imports: [RouterModule.forRoot(appRoutes)]
})
export class AppModule {}
```

# ROUTER - REMINDERS



- Use the **RouterOutlet** directive to indicate the insertion point of the routed view
- Use the **RouterLink** directive to navigate between views
- Use the **RouterLinkActive** directive to add CSS classes on the active link

```
@Component({
  selector: 'app-root',
  template: `
    <nav>
      <a class="link" routerLink="/contacts" routerLinkActive="link-active">Contacts</a>
      <a class="link" routerLink="/contacts/5" routerLinkActive="link-active">Contact 5</a>
    </nav>

    <main>
      <router-outlet></router-outlet>
    </main>
  `
})
export class AppComponent {}
```

# ROUTER - ADVANCED CONFIGURATION



- Configure a fallback route via the `**` wildcard

```
const appRoutes: Routes = [
  { path: 'contacts/:id', component: ContactComponent },
  { path: '**', component: PageNotFoundComponent }
]
```

- Pass static data, accessible via the `ActivatedRoute` service

```
const appRoutes: Routes = [
  {
    path: 'contacts/:id',
    component: ContactComponent
  },
  {
    path: 'contacts-summary/:id',
    component: ContactComponent,
    data: { summary: true }
  },
];
```

# ROUTER - ADVANCED CONFIGURATION



- Use `ActivatedRoute` to be notified of the current route details

```
@Component({
  selector: 'app-contact'
})
export class ContactComponent implements OnInit {
  summary!: boolean;

  contact?: Contact;

  constructor(private activatedRoute: ActivatedRoute, private contactService: ContactService)
{}

ngOnInit() {
  this.summary = this.activatedRoute.snapshot.data['summary'] ?? false;

  this.activatedRoute.paramMap
    .pipe(switchMap((params: ParamMap) => this.contactService.get(params.get('id'))))
    .subscribe((contact: Contact) => (this.contact = contact));
}
}
```



# ROUTER - EVENTS

- The router emits events when redirecting from one view to another:
  - `NavigationStart`, `NavigationEnd`, `NavigationCancel`, `NavigationError`
  - `RoutesRecognized`, `RouteConfigLoadStart`, `RouteConfigLoadEnd`
- Each event is materialized by an `Observable`
- For debug, ability to display all these events in the console

```
RouterModule.forRoot(appRoutes, { enableTracing: true })
```

# ROUTER - EVENTS



- Router events example:

```
import { BehaviorSubject, filter, tap } from 'rxjs';
import { Injectable, inject } from '@angular/core';
import { NavigationEnd, Router } from '@angular/router';

@Injectable()
export class NavigationCounterService {
  counter$ = new BehaviorSubject(0);

  init() {
    return inject(Router).events.pipe(
      filter((event): event is NavigationEnd => event instanceof NavigationEnd),
      tap(() => this.counter$.next(this.counter$.value + 1))
    );
  }
}
```



# ROUTER - LAZY LOADING

- Structuring an application via **feature modules**
- A feature module is:
  - an Angular module
  - in which we will be able to define components, services, directives, ...
  - and define routes via the **RouterModule.forChild** method
  - it can be **lazy-loaded**
- The module is loaded when the user goes on one of its routes
- Reduce the JavaScript **bundle size** to load in the Browser



# ROUTER - LAZY LOADING

- Several loading strategies:
  - **NoPreloading**: Loading during navigation (default strategy)
  - **PreloadAllModules**: Pre-load the modules as soon as possible
  - **QuicklinkStrategy**: Pre-load only the routes associated with links on the current page (external lib)
- **@angular/cli** (via **webpack**) will be in charge of:
  - Creating a specific file for this module (**myfeature.module.chunk.js**)
  - Loading it in the browser when the user goes on one of its pages



# ROUTER - LAZY LOADING

- Configuring an `AdminModule` routes via `RouterModule.forChild` method

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



# ROUTER - LAZY LOADING

- Loading the **AdminModule** module on demand via **loadChildren**

```
@NgModule({
  imports: [
    RouterModule.forRoot([
      {
        path: 'admin',
        loadChildren: () => import('./admin/admin.module').then((module) =>
module.AdminModule),
      },
    ]),
  ],
})
export class AppModule {}
```



# ROUTER - LAZY LOADING

- If you want to add a user feedback while loading, you can reuse the events described previously

```
@Component({
  selector: 'app-root',
  template: `
    <router-outlet>
      <span class="app-loading" *ngIf="isLoading">Loading...</span>
    </router-outlet>
  `})
export class AppComponent {
  isLoading = false;

  constructor(router: Router) {
    router.events.subscribe((event: RouterEvent) => {
      if (event instanceof NavigationStart) { this.isLoading = true; }
      else if (event instanceof NavigationEnd) { this.isLoading = false; }
    });
  }
}
```

# ROUTER - STANDALONE COMPONENT



## *Lazy loading components*

- A standalone component can be lazy loaded without requiring a feature module
- Directly lazy-load the component with `loadComponent` instead of `loadChildren`

```
export const adminRoutes: Route[] = [
  {
    path: 'users',
    loadComponent: () =>
      import('./admin/admin-users/admin-users.component')
        .then((module) => module.AdminUsersComponent)
  }
];
```

# ROUTER - STANDALONE COMPONENT



## *Lazy loading routes*

- Multiple routes can be lazy-loaded at once by directly loading the routes config with `loadChildren`
- Each loaded route must use a standalone component

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

# ROUTER - STANDALONE COMPONENT



- A route can defined his own injector and configure the DI with scoped service

```
export const appRoutes: Routes = [
  {
    path: 'admin',
    loadChildren: () => import('./admin/admin-routes').then((module) => module.adminRoutes),
    providers: [AdminService]
  }
];
```



# ROUTER - GUARDS

- Classes to control redirections:
  - Check if a user has rights to access a page
  - Check there is no information loss
  - Load data before redirecting
  - ...
- A guard requires the implementation of one of these interfaces:
  - **CanMatch**: when matching a route to activate
  - **CanActivate**: when loading a route
  - **CanActivateChild**: when loading a child route
  - **CanDeactivate**: when leaving the route
  - **Resolve**: to get data before activating the route



# ROUTER - GUARDS

- Each implemented method can return a `boolean`, `Observable<boolean>` or `Promise<boolean>`

```
export interface CanActivate {  
    canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):  
        Observable<boolean> | Promise<boolean> | boolean;  
}
```

- The return value affects the redirection process:
  - `true`: the process continues
  - `false`: the process is stopped and the user stays on the page
- Best Practices:
  - Create several **Guards** and not only one who is in charge of many processing
  - Implement the guard in its own **TypeScript** file: `rights.guard.ts`

# ROUTER - GUARDS | CANACTIVATE



- Example: verify that the user is logged in before redirecting

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

  canActivate(route: ActivatedRouteSnapshot): boolean {
    if (this.authService.isLoggedIn()) { return true; }
    this.router.navigate([ '/login' ]);
    return false;
  }
}

@NgModule({
  imports: [
    RouterModule.forRoot([
      { path: 'admin', component: AdminComponent, canActivate: [AuthGuard] },
    ])
  ],
  providers: [AuthGuard]
})
export class AppRoutingModule {}
```

# ROUTER - GUARDS | CANDEACTIVATE



- Example: confirmation request before redirecting

```
@Injectable()
export class CanDeactivateGuard implements CanDeactivate<ContactFormComponent> {
  constructor(private dialogService: DialogService) {}

  canDeactivate(
    component: ContactFormComponent,
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): Observable<boolean> | boolean {
    if (component.hasUnsavedChanges) {
      // Assuming the following method returns an `Observable<boolean>`
      return this.dialogService.confirm('Discard changes?');
    }
    return true;
  }
}
```



# ROUTER - GUARDS | CANMATCH

- Example: multiple routes config is case guard returned **false**

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

import { basketEmptyGuard } from './basket/basket-empty.guard';

export const appRoutes: Routes = [
  {
    path: 'basket',
    loadComponent: () => import('./basket/basket.component').then((m) => m.BasketComponent),
    canMatch: [basketEmptyGuard],
  },
  {
    path: 'basket',
    loadComponent:
      () => import('./basket-empty/basket-empty.component').then((m) =>
      m.BasketEmptyComponent),
  },
];
```





## Lab 6



# FORMS

# SUMMARY



- Reminders
- Angular CLI
- Best practices
- Reactivity
- RxJS
- Standalone components
- Router
- *Reactive Forms*
- I18n
- SSR
- Animations
- Material
- Tests
- NgRx
- To go further



# FORMS

- There are 2 types of forms in Angular:
  - **Template-driven Forms**: Almost everything is done on the template side
  - **Reactive Forms**: Almost everything is done on the component side
- Both belong to the `@angular/forms` library and share a common set of form control classes

# FORMS - TEMPLATE-DRIVEN FORMS



- Enable template-driven forms by importing `FormsModule`

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

@NgModule({
  /* ... */
  imports: [FormsModule],
})
export class AppModule {}
```

# FORMS - TEMPLATE-DRIVEN FORMS



- Bind html form elements (input, select) to data model with **ngModel**

```
<form novalidate #signupForm="ngForm">
  <div>
    <label>Username *</label>
    <div>
      <input type="text" required [(ngModel)]="user.username" name="username"
#username="ngModel"/>
      <span *ngIf="(username.touched || username.dirty) && username.errors">
        <span *ngIf="username.errors.required">Username is required</span>
      </span>
    </div>
  </div>
  <div>
    <button (click)="save()" [disabled]="signupForm.invalid">Submit</button>
  </div>
</form>
```

# FORMS - TEMPLATE-DRIVEN FORMS



## *ngModel & two-ways data binding*

- `[ngModel]` syntax is used for **two-ways data binding**
- Disabled by default in Angular
- `()` to detect changes on our input `[]` to put data in it
- Enabled by default in AngularJS: performance issues
- Syntactic sugar instead of:

```
<input [ngModel]="user.username" (ngModelChange)="user.username=$event"/>
```

# FORMS - TEMPLATE-DRIVEN FORMS



- Init user in `ngOnInit()`
- Save user with `this.signupForm.value` (user to be saved)

```
import { ViewChild } from '@angular/core';
@Component({ /* ... */ })
export class SignupTemplateComponent implements OnInit {

  @ViewChild('signupForm') signupForm: NgForm;
  user: User;

  ngOnInit() {
    this.user = new User();
  }

  save() {
    if (this.signupForm.valid) { /* save user */ }
  }
}
```

🔥 **ViewChild**: get the first element or directive matching the selector from the view **DOM**

# FORMS - TEMPLATE-DRIVEN FORMS



To sum up:

- Two-way data bindings (using `[(NgModel)]` syntax)
- Easy to use: automatic track of the form and its data
- Easy to understand: minimal component code
- Asynchronous (it complicates unit testing)
- Keep the component class clean of form logic

Suitable for simple scenarii



# FORMS - REACTIVE FORMS

- Enable reactive forms by importing `ReactiveFormsModule`

```
import {NgModule} from "@angular/core";
import {ReactiveFormsModule} from "@angular/forms";

@NgModule({
  /* ... */
  imports: [ReactiveFormsModule],
})
export class AppModule {}
```

# FORMS - REACTIVE FORMS



- Bind form control objects to native html form control elements using `formGroup` and `formControlName`

```
<form novalidate [formGroup]="signupForm" (submit)="save()">
  <div [ngClass]="{{ 'has-error': (signupForm.get('username').touched
    || signupForm.get('username').dirty)
    && signupForm.get('username').invalid }}">
    <label>Username *</label>
    <div>
      <input type="text" name="username" formControlName="username"/>
      <span *ngIf="(signupForm.get('username').touched
        || signupForm.get('username').dirty)
        && signupForm.get('username').errors">
        <span *ngIf="signupForm.get('username').errors?.required">
          Username is required
        </span>
      </span>
    </div>
  </div>
  <div>
    <button [disabled]="signupForm.invalid">Submit</button>
  </div>
</form>
```

# FORMS - REACTIVE FORMS



- Create form control object
- Each field is reachable: `this.signupForm.get('username').value`
- Save user with `this.signupForm.value` (user to be saved)

```
@Component({ /* ... */ })
export class SignupReactiveComponent implements OnInit {
  signupForm: FormGroup;
  user: User;

  ngOnInit() {
    this.signupForm = new FormGroup({
      username: new FormControl('', Validators.required)
    });
  }

  save() {
    if (this.signupForm.valid) { /* save user */ }
  }
}
```

# FORMS - REACTIVE FORMS



To sum up:

- No data binding (respects immutability for data model)
- More flexible: for example, ability to add elements dynamically
- Easier unit testing: reactive forms are synchronous
- Needs more practice

Suitable for complex scenarii





## Lab 7



I18N

# SUMMARY



- Reminders
- Angular CLI
- Best practices
- Reactivity
- RxJS
- Standalone components
- Router
- Reactive Forms
- *I18n*
- SSR
- Animations
- Material
- Tests
- NgRx
- To go further



Building an application may involve the support of several languages

The localization process is carried out in several actions:

- Extracting text for translation into different languages
- Formatting data for a particular locale

# I18N - THE ANGULAR WAY



Angular comes with some Internationalized functionality such as:

- Displaying dates, numbers, percentages, and currencies
- Translating text in component templates
- Handling plural forms of words
- Handling alternative text



# I18N - LOCALIZE PACKAGE

Angular 9 has introduced a new package called `@angular/localize`:

```
ng add @angular/localize
```

Apply these tasks to localize your project:

- Mark text to translate
- Use the CLI to extract selected text into a source language file.
- Duplicate and translate language file for each language.
- Build your application for one or more locales.



# I18N - TRANSLATE TEXT IN TEMPLATE

- Prepare text for translation using Angular **i18n** attribute

```
<h1 i18n>Hello i18n!</h1>
```

- Add a description for translation

```
<h1 i18n="An introduction header">Hello i18n!</h1>
```

- Add an id

```
<h1 i18n="@introHeader">Hello i18n!</h1>
```

- Combine meaning, description and id

```
<h1 i18n="{meaning} | {description}@@{id}">Hello i18n!</h1>
```

# I18N - TRANSLATE ATTRIBUTE IN TEMPLATE



Some HTML element attributes includes text that could also be translated

- Add **i18n-\*** prefix to the attribute to mark attribute for translation

```

```

# I18N - TRANSLATE IN COMPONENT



The new `@angular/localize` package has introduced a function helper named `$localize`

This function can be used to translate text inside component, service, etc...

```
@Component({
  selector: 'app-header',
  template: `{{ header }}`
})
export class HeaderComponent {
  header = $localize`:@@introHeader:Hello i18n!`;
}
```



# I18N - TRANSLATION FILES

Angular uses a translation file for each language to handle translate, plural and alternate expression

- XLIFF 1.2 french translation file **messages.fr.xlf**

```
<trans-unit id="introHeader" datatype="html">
  <source>Hello i18n!</source>
  <target state="new">Bonjour i18n!</target>
</trans-unit>
```

- json format is also supported

```
{
  "locale": "fr",
  "translations": {
    "introHeader": "Bonjour i18n!"
  }
}
```

Note :

# I18N - TRANSLATION FILES



- Use the following Angular CLI command to extract text to a generated translation file for default language **messages.xlf**:

```
ng extract-i18n
```

- Command options
  - **--format** | Set the format of the output file (xlf, xlf2, json, arb, xmb)
  - **--out-file** | Set the name of the output file
  - **--output-path** | Set the path of the output directory
- Create a copy of translation source file for each locale
  - **messages.xlf → message.{locale}.xlf**

# I18N - DEFINE LOCALES FOR BUILD



Define locales you want to use for project with **i18n** project option in angular.json

```
"projects": {  
  "zenikaNgWebSite": {  
    ...  
    "i18n": {  
      "sourceLocale": "en-US",  
      "locales": {  
        "fr": "src/locale/messages.fr.xlf",  
        "es": "src/locale/messages.es.xlf"  
      }  
    },  
    "architect": {  
      ...  
    }  
  }  
}
```

# I18N - DEFINE LOCALES FOR BUILD



- Then, to build a localised version of the application, use the `localize` option with the locale that you have previously defined

```
"build": {  
  "builder": "@angular-devkit/build-angular:browser",  
  "options": {  
    "localize": ["fr"],  
    ...  
  },  
},
```

- To build for all locales previously defined in the i18n section, use :

```
"options": {  
  "localize": true,  
}
```

- In the dist folder, a build is created for each locale that needs to be built.



# I18N - ALTERNATIVES

- nxg-translate
- transloco
- angular-i18next

The main alternative is **ngx-translate**

- First step, add @ngx-translate/core to the project:

```
npm install @ngx-translate/core
```

# I18N - NGX-TRANSLATE



- Then, import **TranslateModule** to your module and configured using **TranslateService**

```
import { NgModule } from '@angular/core';
import { TranslateModule, TranslateService } from '@ngx-translate/core';

@NgModule({
  imports: [TranslateModule.forRoot()],
  bootstrap: [AppComponent]
})
export class AppModule {
  constructor(translate: TranslateService) {
    // Add translation for `en` lang
    translate.setTranslation('en', {
      HELLO: 'I <3 {{value}}'
    });

    // this language will be used as a fallback
    translate.setDefaultLang('en');
    translate.use('en');
  }
}
```

# I18N - NGX-TRANSLATE | PIPE



Now you can use **translate** pipe

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

@Component({
  selector: 'app',
  template: `<div>{{ 'HELLO' | translate:params }}</div>`
})
export class AppComponent {
  params = { value: 'zenika' };
}
```

# I18N - NGX-TRANSLATE | DIRECTIVE



You can also translate via `translate` directive

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

@Component({
  selector: 'app',
  template: `
    <div [translate]="'HELLO'" [translateParams]="params"></div>
    <div translate [translateParams]="params">HELLO</div>
  `
})
export class AppComponent {
  params = { value: 'zenika' };
}
```

*Best practice:* Please be consistent between pipe and directive

# I18N - NGX-TRANSLATE | SERVICE



You can get translation by using the service

```
import { Component } from '@angular/core';
import { TranslateService } from '@ngx-translate/core';

@Component({
  selector: 'app',
  template: `<div>{{message}}</div>`
})
export class AppComponent {
  params = { value: 'zenika' };
  message!: string;

  constructor(translate: TranslateService) {
    translate.get('HELLO', params)
      .subscribe((res: string) => {
        this.message = res;
      });
  }
}
```

# I18N - NGX-TRANSLATE | LOADERS



The translation file can be loaded dynamically over HTTP depending on the locale used:

```
import { TranslateLoader, TranslateModule } from '@ngx-translate/core';
import { TranslateHttpLoader } from '@ngx-translate/http-loader';

export function HttpLoaderFactory(http: HttpClient) {
  return new TranslateHttpLoader(http);
}

@NgModule({
  imports: [
    HttpClientModule,
    TranslateModule.forRoot({
      loader: {
        provide: TranslateLoader,
        useFactory: HttpLoaderFactory,
        deps: [HttpClient],
      },
    }),
  ],
})
export class AppModule {}
```





## Lab 8



**SSR**

# SUMMARY



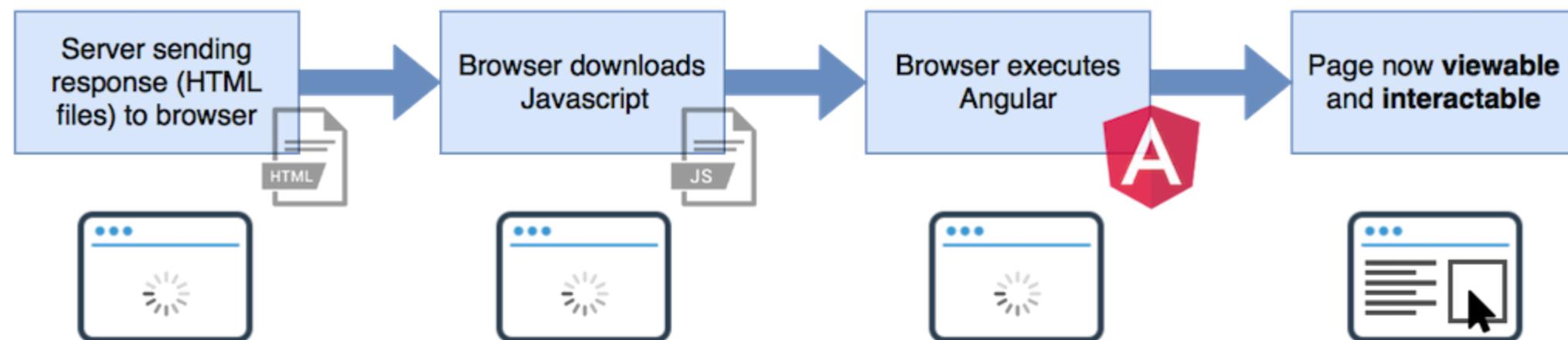
- Reminders
- Angular CLI
- Best practices
- Reactivity
- RxJS
- Standalone components
- Router
- Reactive Forms
- I18n
- *SSR*
- Animations
- Material
- Tests
- NgRx
- To go further



# SSR

- Angular SSR generates static application pages on the server (**Server-Side Rendering or SSR**)
- Normal Angular app executes in the browser
  1. user actions ask for update
  2. update DOM

## Client-Side Rendering



# SSR



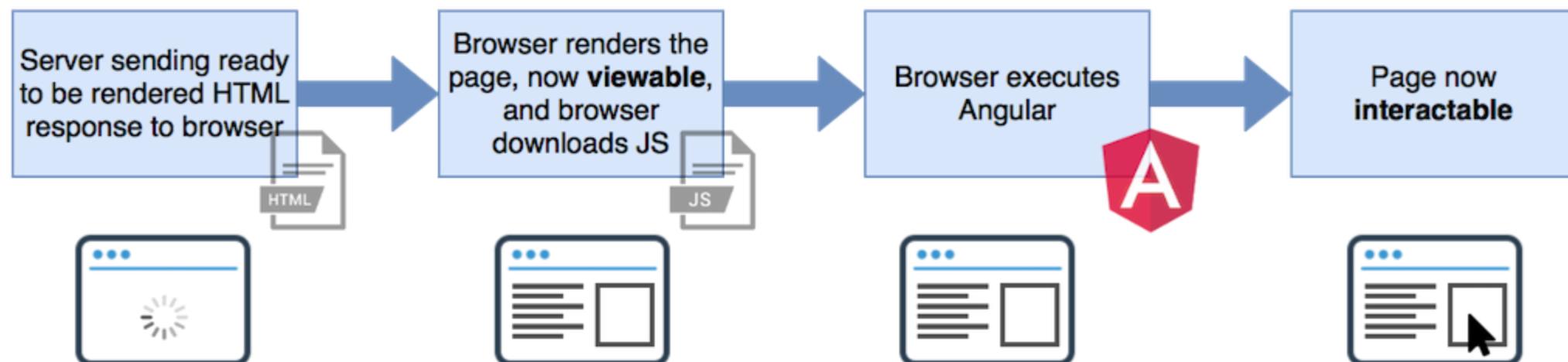
- Angular SSR can:
  - pre-generate HTML pages and served them later
  - generate and serve HTML pages in response to requests from browsers
  - handle transition from the server view to the client view in the browser
- Why use Angular SSR ?
  - Search Engine Optimization (SEO): facilitate web crawlers
  - Performance
  - Rendering the first page quickly





1. Browser receives initial response from server
2. **Server view** is displayed, events start being recorded, browser requests other files (JS, CSS, images, etc) asynchronously
3. Angular bootstrap, **client view** rendered in hidden **div**, browser replays events to adjust app state (clicks, etc)
4. Switch the visible **server view div** to the hidden **client view div**

## Server-Side Rendering





# SSR - SET UP

- Install:
  - `@angular/platform-server`: SSR server-side components
  - `@nguniversal/express-engine`: express engine for SSR apps
- To create the server-side app module, `app.server.module.ts`, run the following CLI command :

```
ng add @nguniversal/express-engine
```

- CLI command create `src/main.server.ts`:

```
export {AppServerModule} from './app/app.server.module';
```



# SSR - SET UP

- CLI command replaces **BrowserModule** import in `src/app/app.module.ts` with:

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

@NgModule({
  imports: [
    BrowserModule.withServerTransition({appId: 'my-app-id'}) // <-- Update to your app name
  ]
})
export class AppModule { }
```



# SSR - SET UP

- CLI command creates an **AppServerModule** in `src/app/app.server.module.ts`

```
import { AppModule } from './app.module';
import { AppComponent } from './app.component';

import { ServerModule } from '@angular/platform-server';

@NgModule({
  imports: [
    AppModule,
    ServerModule,
  ],
  bootstrap: [AppComponent]
})
export class AppServerModule { }
```



# SSR - SET UP

- CLI command creates a node server `server.ts`

```
import 'zone.js/dist/zone-node';

import { ngExpressEngine } from '@nguniversal/express-engine';
import * as express from 'express';
import { join } from 'path';
import { AppServerModule } from './src/main.server';
import { APP_BASE_HREF } from '@angular/common';
import { existsSync } from 'fs';
```

# SSR - SET UP



- CLI command creates a node server `server.ts`

```
export function app() {
  const server = express();
  const distFolder = join(process.cwd(), 'dist/zenika-ng-website/browser');
  const indexHtml = existsSync(join(distFolder, 'index.original.html')) ?
    'index.original.html' : 'index';

  server.engine('html', ngExpressEngine({
    bootstrap: AppServerModule,
  }));

  server.set('view engine', 'html');
  server.set('views', distFolder);
  server.get('*.*', express.static(distFolder, {
    maxAge: '1y'
  }));
  server.get('*', (req, res) => {
    res.render(indexHtml, { req, providers: [{ provide: APP_BASE_HREF, useValue: req.baseUrl }] });
  });

  return server;
}
```



# SSR - SET UP

- CLI command creates a node server `server.ts`

```
function run() {
  const port = process.env.PORT || 4000;
  const server = app();

  server.listen(port, () => {
    console.log(`Node Express server listening on http://localhost:${port}`);
  });
}
```



# SSR - CONFIG

- CLI adds a new tsconfig `src/tsconfig.server.json`:

```
{  
  "extends": "../tsconfig.json",  
  "compilerOptions": {  
    "outDir": "./out-tsc/app-server",  
    "module": "commonjs",  
    "types": [  
      "node"  
    ]  
  },  
  "files": [  
    "src/main.server.ts",  
    "server.ts"  
  ],  
  "angularCompilerOptions": {  
    "entryModule": "app/app.server.module#AppServerModule"  
  }  
}
```



# SSR - BUILD

- CLI command creates new target in `angular.json` config

```
{  
  "server": {  
    "builder": "@angular-devkit/build-angular:server",  
    "options": {  
      "outputPath": "dist/zenika-ng-website/server",  
      "main": "server.ts",  
      "tsConfig": "tsconfig.server.json"  
    },  
    "configurations": {  
      "production": {  
        "outputHashing": "media",  
        "fileReplacements": [  
          {  
            "replace": "src/environments/environment.ts",  
            "with": "src/environments/environment.prod.ts"  
          }  
        ],  
        "sourceMap": false,  
        "optimization": true  
      }  
    }  
  }  
}  
10 - 12}
```

# SSR - SERVE



- CLI command creates new target in `angular.json` config

```
{  
  "serve-ssr": {  
    "builder": "@nguniversal/builders:ssr-dev-server",  
    "options": {  
      "browserTarget": "zenika-ng-website:build",  
      "serverTarget": "zenika-ng-website:server"  
    },  
    "configurations": {  
      "production": {  
        "browserTarget": "zenika-ng-website:build:production",  
        "serverTarget": "zenika-ng-website:server:production"  
      }  
    }  
  }  
}
```

# SSR - RUN



- Add scripts in `package.json`:

```
"scripts": {  
  "dev:ssr": "ng run zenika-ng-website:serve-ssr",  
  "serve:ssr": "node dist/zenika-ng-website/server/fr/main.js",  
  "build:ssr": "ng build --prod && ng run zenika-ng-website:server:production",  
  "prerender": "ng run zenika-ng-website:prerender"  
}
```

- Build with:

```
npm run build:ssr
```

- Start the server with:

```
npm run serve:ssr
```





## Lab 9



# ANIMATIONS

# SUMMARY



- Reminders
- Angular CLI
- Best practices
- Reactivity
- RxJS
- Standalone components
- Router
- Reactive Forms
- I18n
- SSR
- *Animations*
- Material
- Tests
- NgRx
- To go further

# ANIMATIONS



- Angular animation module is built over **Web Animation API** standard
- Use of **polyfills** for non compatible browsers
- Angular animations benefit from **AoT**: performance is improved
- Use of metadata in **@Component** to declare animations

# ANIMATIONS



- Import utility functions for animations from `@angular/animations`:

```
@Component({
  selector: 'app',
  template: `<button (click)="toggle()">Open/Close</button>
            <div [@toggle]="toggleState">{{open}}</div>`,
  animations: [
    trigger('toggle', [
      state('open', style({ opacity: 1 })),
      state('close', style({ opacity: 0 })),
      transition('close <=> open', [ animate(1000) ])
    ])
  ]
})
export class AppModule {
  open: boolean = false;
  toggleState: string = "open";
  toggle() {
    this.open = !this.open;
    this.toggleState = this.open ? "open" : "close"
  }
}
```

# ANIMATIONS



## *States and Transitions*

- An animation relies on states: transition from a state A to a state B
- **state** is a value (string) that you can define and use from the template
- Styles can be associated with each **state**

```
state('open', style({
  backgroundColor: '#eeeeee',
  height: '*'
}),
state('close', style({
  backgroundColor: '#aaaaaa',
  height: 0
}))
```

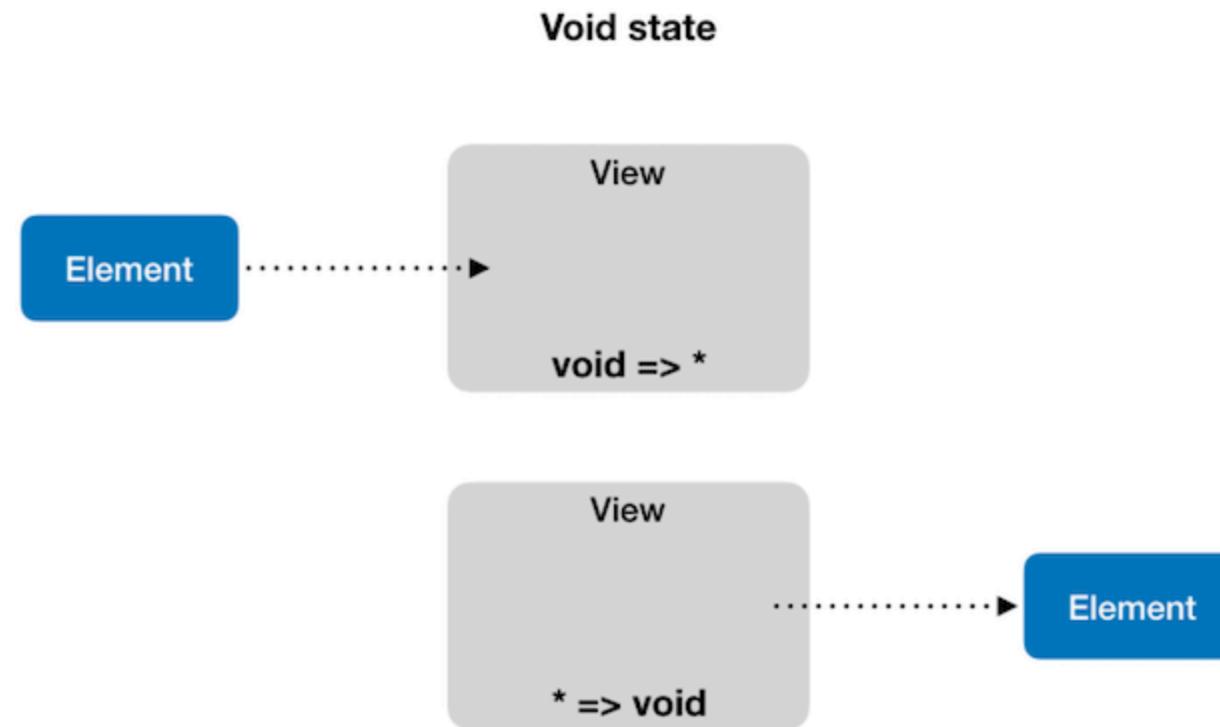
- Style is applied once the element has reached the state and keeps it as long as the state is the same

# ANIMATIONS



## *States and Transitions*

- state **\***: any animation state
- state **void**: element is not attached to view



# ANIMATIONS



## *States and Transitions*

Once the states are declared, we can define the transitions between the states

- A transition deals with the transit time between states

```
transition('open => close', animate(1000)),  
transition('close => open', animate(2000))
```

- If several transitions have the same timing configuration, we can use the syntax shortcut `<=>`

```
transition('open <=> close', animate('100ms ease-in'))
```

# ANIMATIONS



## *States and Transitions*

- Apply style during animation: use of **style** function as a second parameter of the **animate** function

```
transition('open <=> close',
  style({transform: 'scale(1)'}),
  animate('100ms ease-in', style({
    transform: 'scale(1.5)'
  }))
)
```

# ANIMATIONS



## *States and Transitions*

- Transitions can be defined using more generic **matchers** (`void`, `*`)

```
transition('void => *', [
  style({transform: 'translateY(-100%)'}),
  animate(1000)
]),
transition('* => void', [
  animate(2000, style({ transform: 'translateY(100%)'}))
])
```

- These transitions are generic ones: we can directly use the associated alias **enter** and **leave**

```
transition(':enter', [
  style({
    transform: 'translateY(-100%)'
  }),
  animate(1000)
]),
transition(':leave', [ animate(2000, style({ transform: 'translateY(100%)'}))])
```

# ANIMATIONS



## *Properties*

- The browser properties are all considered to be **animatable**
- **position, size, transforms, color, border...**

```
animate(2000, style({
  transform: 'translateY(100%)',
  padding: '50px',
  fontSize: '3em'
}))
```

- Some properties are calculated at runtime with **\*** property
  - for example: **height** and **width** properties depend on the size of the screen

```
transition(':enter', [
  style({height: 0}),
  animate(1000, style({height: '*'}))
])
```



# ANIMATIONS

## *Timing*

- Duration (in ms)

```
animate(2000)
animate('200ms')
animate('0.3s')
```

- Delay as a second parameter

```
animate('200ms 100ms')
animate('0.3s 100ms')
```

- Easing function

```
animate('200ms 100ms ease-in-out')
animate('200ms ease-out')
```

# ANIMATIONS



## *Keyframes*

- **Keyframes** allow advanced animations
- **Offset** sets the progress level of the animation: [0, 1]

```
animate(2000, keyframes([
  style({transform: 'translateX(-100%)', offset: 0}), /* animation beginning */
  style({transform: 'translateX(-50%)', offset: 0.3}), /* animation at 30% of ending */
  style({transform: 'translateX(0)', offset: 1}) /* animation ending */
]))
```

# ANIMATIONS



## *Parallel animations*

- Different types of animation can be applied in parallel
- Useful when we want several animations in a row but they are not related to each other

```
transition(':enter', [
  group([
    animate('0.1s ease',
      style({transform: 'translateX(0)'})
    ),
    animate('0.2s 0.1s ease',
      style({opacity: 1})
    )
  ])
])
```

# ANIMATIONS



## *Triggers*

- A **trigger** is defined in the **Component** metadata, and used in the template to trigger the animation

```
@Component({
  template: '<div [@toggle]="open ? true : false">Open/Close</div>',
  animations: [
    trigger('toggle', [
      state('true',
        style({height: '*'})),
      state('false',
        style({height: 0}))
    ),
    transition('true <=> false',
      animate(1000))
  ])
})
```





# Lab 10



# MATERIAL

# SUMMARY

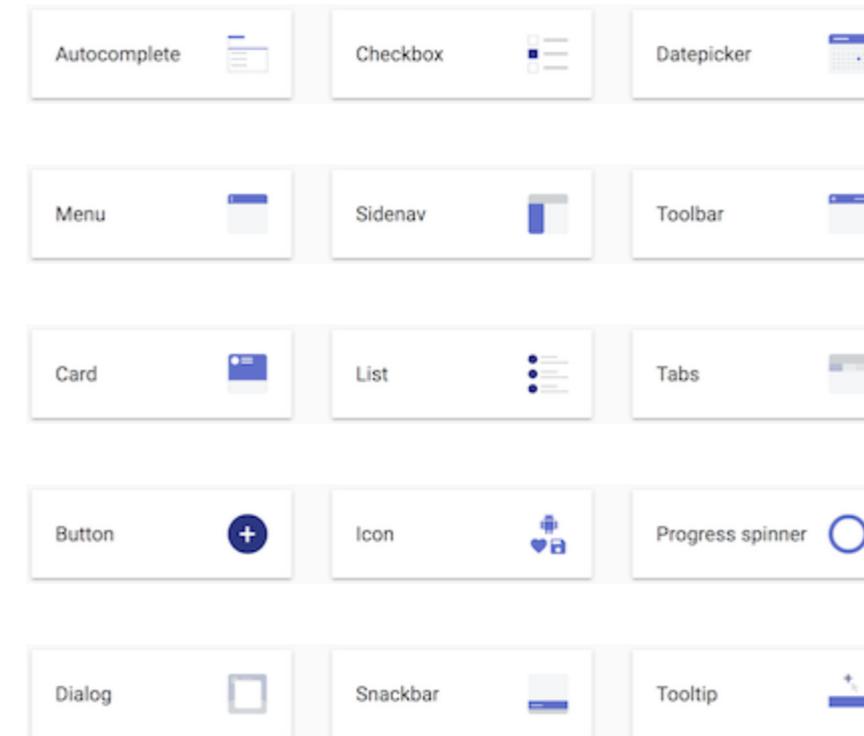


- Reminders
- Angular CLI
- Best practices
- Reactivity
- RxJS
- Standalone components
- Router
- Reactive Forms
- I18n
- SSR
- Animations
- *Material*
- Tests
- NgRx
- To go further



# MATERIAL

- Angular Library of UI elements that implements Material Design specifications
- Examples: Form Controls, Navigation, Layout, Buttons, Indicators, Popups, Modals, Data table...



- Available at <https://material.angular.io>



# MATERIAL

## Step 1: Install Angular Material

```
ng add @angular/material
```

## Step 2: Install dependencies

- Some Material components depend on it
- *Best Practice*: Install only those needed



# MATERIAL

## Step 3: Import the component modules

- You must import each module you want to use
- Import them in a special **NgModule**

```
import {MatButtonModule} from '@angular/material/button';
import {MatCheckboxModule} from '@angular/material/checkbox';

@NgModule({
  imports: [MatButtonModule, MatCheckboxModule],
  exports: [MatButtonModule, MatCheckboxModule]
})
export class DependenciesModule { }
```

# MATERIAL



## Step 4: Include a theme

What is a theme?

A theme is a set of colors that will be applied to components

- You can use a [pre-built theme](#)
- You can create your own theme



# MATERIAL

## Step 4: Include a pre-built theme

- Automatically added by `ng add @angular/material`
- include the CSS file in your `style.css`

```
@import '~@angular/material/prebuilt-themes/indigo-pink.css';
```

- Or include the CSS file in your `angular.json`

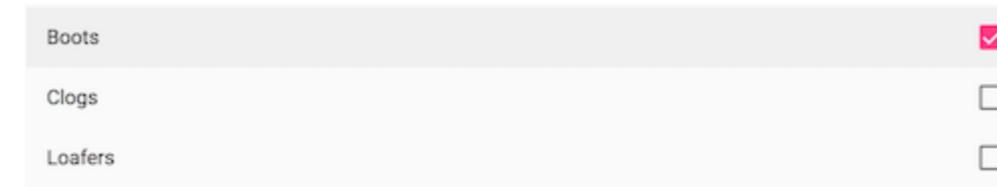
```
"styles": [
  "node_modules/@angular/material/prebuilt-themes/indigo-pink.css",
  "src/styles.css"
],
```



# MATERIAL - EXAMPLES



```
<mat-tab-group>
  <mat-tab label="Tab 1">Content 1</mat-tab>
  <mat-tab label="Tab 2">Content 2</mat-tab>
</mat-tab-group>
```



```
<mat-selection-list>
  <mat-list-option *ngFor="let shoe of typesOfShoes">{{shoe}}</mat-list-option>
</mat-selection-list>
```



# MATERIAL - ICONS

- Icon library available at <https://material.io/icons/>
- Load icon font in `index.html`:

```
<link href="https://fonts.googleapis.com/icon?family=Material+Icons" rel="stylesheet">
```

- used in template files as:

```
<mat-icon aria-hidden="false" aria-label="Example home icon" fontIcon="home"></mat-icon>
```



# MATERIAL

## *Dependency - Angular CDK*

- Angular CDK ([Component Development Kit](#)): Library to build advanced components without adopting the Material Design visual style
- Some Material components extends CDK components and apply style to them so that components style matches Material specifications

```
ng add @angular/cdk
```



# MATERIAL

## *Dependency - Angular animations module*

```
ng add @angular/animations
```

- Include the BrowserAnimationsModule

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

@NgModule({ /* ... */
  imports: [BrowserAnimationsModule],
})
export class AppModule { }
```

# MATERIAL



## *Dependency - Angular animations module*

- If you don't need these animations, use the **NoopAnimationsModule**: utility module that mocks the real animation module but doesn't actually animate

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

@NgModule({ /* ... */
  imports: [NoopAnimationsModule],
})
export class AppModule { }
```





## Lab 11



# TESTS

# SUMMARY



- Reminders
- Angular CLI
- Best practices
- Reactivity
- RxJS
- Standalone components
- Router
- Reactive Forms
- I18n
- SSR
- Animations
- Material
- *Tests*
- NgRx
- To go further

# TESTS - TYPES



3 types of test

- *Unit*: Test a small portion of code. Can be executed in memory.
- *Integration*: Test the interaction between different elements of your app.
- *End to End*: Test that simulates a user interacting with your application.



# UNIT TEST - TOOLS

By default

- *Karma* as a test runner
- *Jasmine* as a test framework

But free to install whatever tool you prefer

- *Jest*
- *Ava*
- ...

# UNIT TEST - ANGULAR ELEMENTS



Possibility to test all Angular elements :

- Components
- Services
- Directives
- Pipes
- ...

By default : *.spec.ts* files.

*The recommended way is to place these files next to the tested element.*

# TESTING PHILOSOPHIES



- Testing a service is as simple as testing a class
- Testing a component is testing both a class and/or a template



CLASS TESTING

DOM TESTING

# 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
- Tests dependent of your component class code

# DOM TESTING



## Pros

- Make sure your component behave exactly the way it should
- Tests are less likely to break for a change in your class code

## Cons

- Harder to setup
- Harder to write
- Most people are not used to this way of writing tests
- Makes your tests dependant of your template

# CLASS TESTING / DOM TESTING



Class testing :

```
it('should display duck names', () => {
  expect(component.duck).toEqual({ firstName: 'LouLou', lastName: 'Duck' });
});
```

Dom testing :

```
it('should display duck names', () => {
  const p = fixture.nativeElement.querySelector('p');
  expect(p.textContent).toContain('LouLou Duck');
});
```



# DEBUG ELEMENT

- Given by Angular through the fixture object
- Allow to write tests compatible with all supported platforms
- `fixture.nativeElement = fixture.debugElement.nativeElement`

```
it('should display user names in the template', () => {
  const pDe = fixture.debugElement.query(By.css('p'))
  const p = pDe.nativeElement

  expect(p.textContent).toContain('LouLou Duck');
});
```



# UNIT TEST - EXAMPLE

*Service :*

```
@Injectable({
  providedIn: 'root'
})
export class DuckService {
  getDuck(): { firstName: string, lastName: string } {
    return {
      firstName: 'Riri',
      lastName: 'Duck'
    }
  }
}
```



# UNIT TEST - EXAMPLE

*Component :*

```
<p>{{ duck?.firstName }} {{ duck?.lastName }}</p>
<button (click)="getDuck()">Get duck</button>
```

```
@Component({
  selector: 'app-duck-detail',
  templateUrl: './duck-detail.component.html',
})
export class DuckDetailComponent implements OnInit {
  duck: { firstName: string, lastName: string };

  constructor(private duckService: DuckService) {}

  ngOnInit() {
    this.getDuck();
  }

  getDuck() {
    this.duck = this.duckService.getDuck();
  }
}
```



# UNIT TEST - EXAMPLE

*Tests :*

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

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ DuckDetailComponent ],
      providers: [ DuckService ],
    }).compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(DuckDetailComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => { expect(component).toBeTruthy(); });

  it('should display duck names', () => {
    expect(component.duck).toEqual({ firstName: 'Riri', lastName: 'Duck' });
  });
});
```

# STUBS AND SPIES



Most of the time, you don't want your real services in your unit tests: *fake them*

## Pros

- Isolate your component tests : test your component and your component only
- Gives you better control of your test data
- Recommended way to test your component

## Cons

- Harder to setup
- Harder to write
- More verbose

# STUB



*Stub setup :*

```
const DuckServiceStub: Partial<DuckService> = {
  getDuck: () => ({ firstName: 'LouLou', lastName: 'Duck' })
};

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

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ DuckDetailComponent ],
      providers: [ { provide: DuckService, useValue: DuckServiceStub } ]
    }).compileComponents();
  });

  // ...
});
```



## *Stub tests :*

```
it('should display duck names', () => {
  expect(component.duck).toEqual({ firstName: 'LouLou', lastName: 'Duck' });
});

it('should get duck names when calling getDuck', () => {
  expect(component.duck).toEqual({ firstName: 'LouLou', lastName: 'Duck' });

  const duckService = TestBed.inject(DuckService);
  duckService.getDuck = () => ({ firstName: 'Fifi', lastName: 'Duck' });

  component.getDuck();

  expect(component.duck).toEqual({ firstName: 'Fifi', lastName: 'Duck' });
});
```



## *Spies setup :*

```
describe('DuckDetailComponent', () => {
  let component: DuckDetailComponent;
  let fixture: ComponentFixture<DuckDetailComponent>;
  let duckService: DuckService;
  let getDuckSpy: jasmine.Spy<DuckService['getDuck']>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ DuckDetailComponent ],
    }).compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(DuckDetailComponent);
    component = fixture.componentInstance;

    duckService = TestBed.inject(DuckService);
    getDuckSpy = spyOn(duckerService, 'getDuck').and.returnValue({firstName: 'Loulou',
lastName: 'Duck'});

    fixture.detectChanges();
  });
});
```



## *Spies tests :*

```
it('should display duck names', () => {
  expect(component.duck).toEqual({ firstName: 'LouLou', lastName: 'Duck' });
});

it('should get duck names when calling getDuck', () => {
  expect(component.duck).toEqual({ firstName: 'LouLou', lastName: 'Duck' });

  getDuckSpy.and.returnValue({ firstName: 'Fifi', lastName: 'Duck' });

  component.getDuck();
  expect(getDuckSpy.calls.count()).toEqual(2);
  expect(component.duck).toEqual({ firstName: 'Fifi', lastName: 'Duck' });
});
```

# TESTING ASYNCHRONOUS CODE



## *waitForAsync*

- Creates a specific async zone for your test
- Await all asynchronous tasks
- Can use fixture.whenStable()

# WAITFORASYNC



Testing with an *observable*

```
ngOnInit() {
  this.duckService.getDuck().subscribe(duck => (this.duck = duck));
}
```

```
let getDuckSpy: jasmine.Spy<DuckService['getDuck']>;

beforeEach(() => {
  const duckService = TestBed.inject(DuckService);
  getDucksSpy = spyOn(duckService, 'getDuck').and.returnValue(
    of({ firstName: 'LouLou', lastName: 'Duck' })
  );
});

it('should display duck names', waitForAsync(async () => {
  fixture.detectChanges();
  await fixture.whenStable();

  expect(component.duck).toEqual({ firstName: 'LouLou', lastName: 'Duck' });
}));
```

# WAITFORASYNC



*waitForAsync* sum up

- Make sure your test does not end before asynchronous tasks finish
- *whenStable* resolves when all asynchronous tasks are finished
- Works with *real time*: 5s to get data = 5s for your test to complete

# FAKEASYNC



Testing with an *observable*

```
let getDuckSpy: jasmine.Spy<DuckService['getDuck']>;  
  
beforeEach(() => {  
  const duckService = TestBed.inject(DuckService);  
  getDucksSpy = spyOn(duckService, 'getDuck').and.returnValue(  
    of({ firstName: 'LouLou', lastName: 'Duck' }).pipe(delay(2000))  
  );  
})  
  
it('should display user names', fakeAsync(() => {  
  expect(component.duck).toBeUndefined();  
  fixture.detectChanges(); // ngOnInit  
  
  tick(1000);  
  
  expect(component.duck).toBeUndefined();  
  
  tick(1000);  
  
  expect(component.duck).toEqual({ firstName: 'LouLou', lastName: 'Duck' });  
}));
```

# FAKEASYNC



*FakeAsync* sum up

- *simulate* time
- 5s delay for your data != 5s for your test
- does not work with *XHR requests*
- only works for some macroTasks:
  - setTimeout
  - setInterval
  - requestAnimationFrame
  - webkitRequestAnimationFrame
  - mozRequestAnimationFrame

# END TO END TEST



## *Concept*

- Test like the *final user* on a production-like environment.
- Test *scenarios* in your app (example: a user story).
- This kind of test will start a browser and simulate a final user.

# END TO END TEST



*Protractor* was the default tool for e2e testing in Angular app for a long time.

*It was deprecated in Angular 12.*

- created when WebDriver APIs were not yet a standard
- end of Protractor development planed at the end of 2022
- robust alternatives have emerged since then

Popular alternatives are now recommended:

- *Cypress*
- *PlayWright*
- *Puppeteer*

# CYPRESS



## *First look*

```
describe('My First Test', () => {
  it('Visits the Kitchen Sink', () => {
    cy.visit('https://example.cypress.io')
  })
})
```

# CYPRESS - SELECTORS



The first thing to do in order to interact with an element is to *select* it

```
cy.get('.className');
cy.get('#myId');
cy.get('h1')
```

- Similar in syntax to jquery selectors
- Get *all* matching elements

To get *one* specific element

```
cy.get('.className').first()
cy.get('.className').eq(1)
```



# CYPRESS - SELECTORS

*cy.get()*

✗

```
const inputElement = cy.get('input')
```

The *get* command returns a promise-like object



```
cy.get('input').then(inputElement)
```

# CYPRESS - SELECTORS



What happens if Cypress does not find your element?

```
cy.get('input').then(inputElement => // Do something)
```

Cypress *automatically retries* the query until either:

- the element is found
- a timeout is reached (default to 4s)



# CYPRESS - SELECTORS

Querying by *text content*

```
cy.contains('something')
```

Can be combine with *cy.get()*

```
cy.get('main').contains('something')
```

# CYPRESS - INTERACTING WITH ELEMENTS

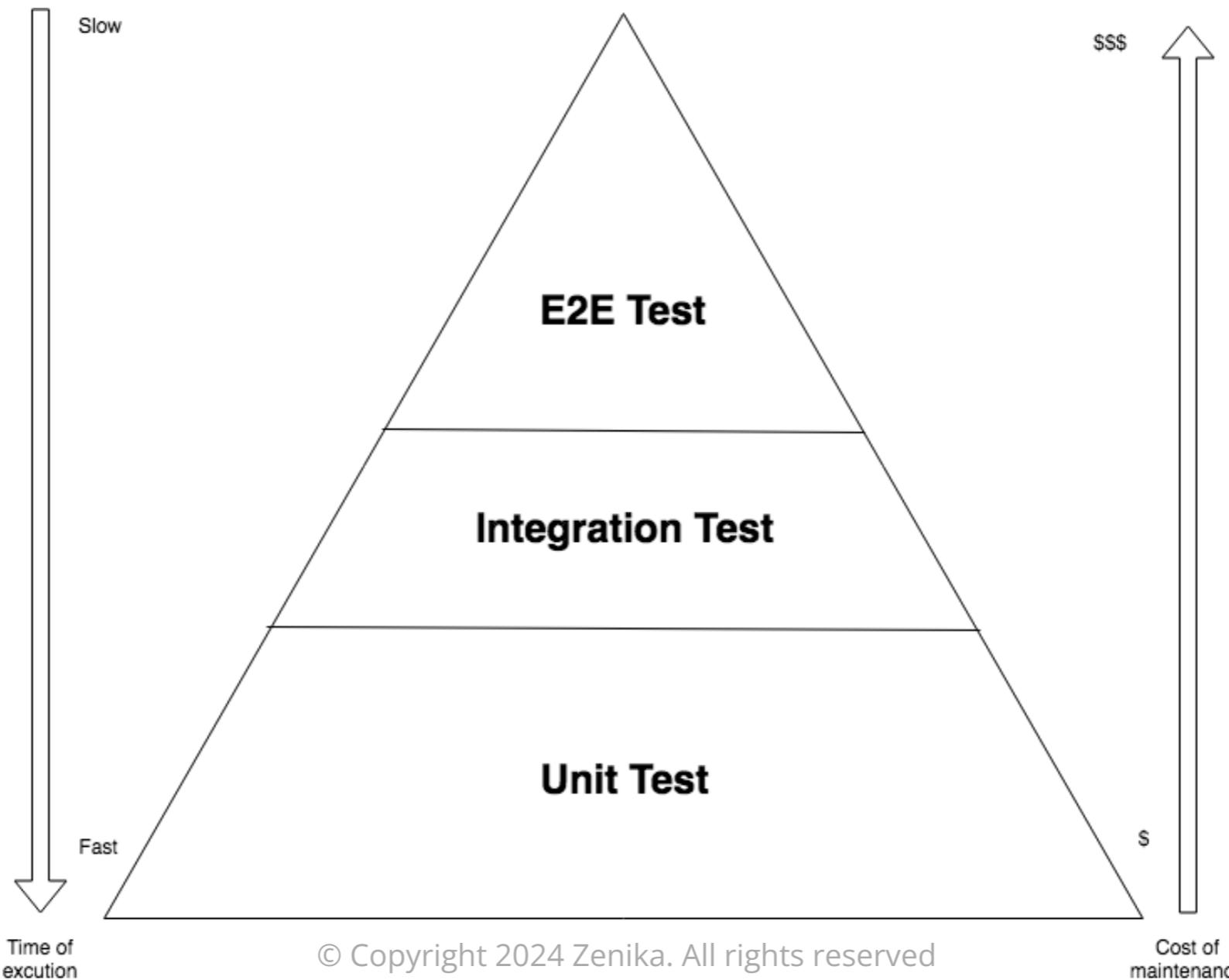


Interaction with elements is made through commands

```
cy.get('button').click()  
  
cy.get('input').type('This is an excellent post.')  
  
cy.get('input').clear();  
  
cy.get('.checkbox').check()  
  
cy.get('.checkbox').uncheck()  
  
cy.get('select').select('user-1')  
  
cy.get('input').focus()
```



# PYRAMID OF TEST







## Lab 12



**NGRX**

# SUMMARY



- Reminders
- Angular CLI
- Best practices
- Reactivity
- RxJS
- Standalone components
- Router
- Reactive Forms
- I18n
- SSR
- Animations
- Material
- Tests
- *NgRx*
- To go further

# NGRX - REDUX



## *What is Redux?*

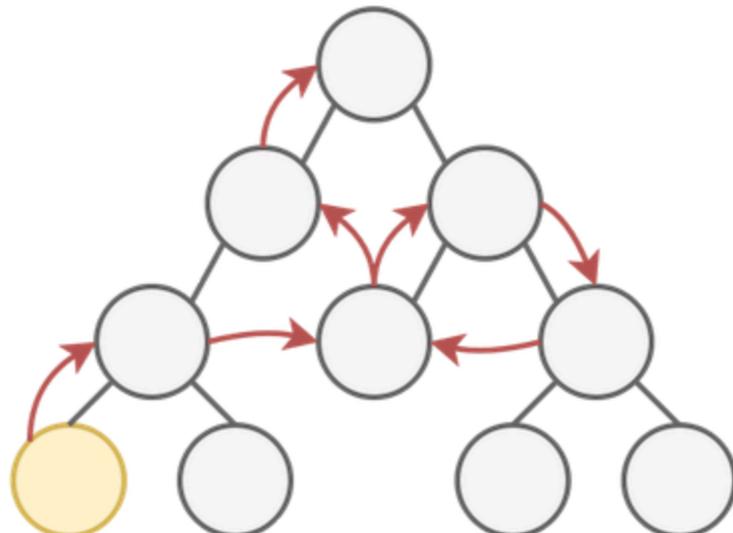
- Alternative to MVC
- Redux is a state container: it **stores** all the states of an app
- One-way data flow: you can only change the state by dispatching an **action**

# NGRX - REDUX

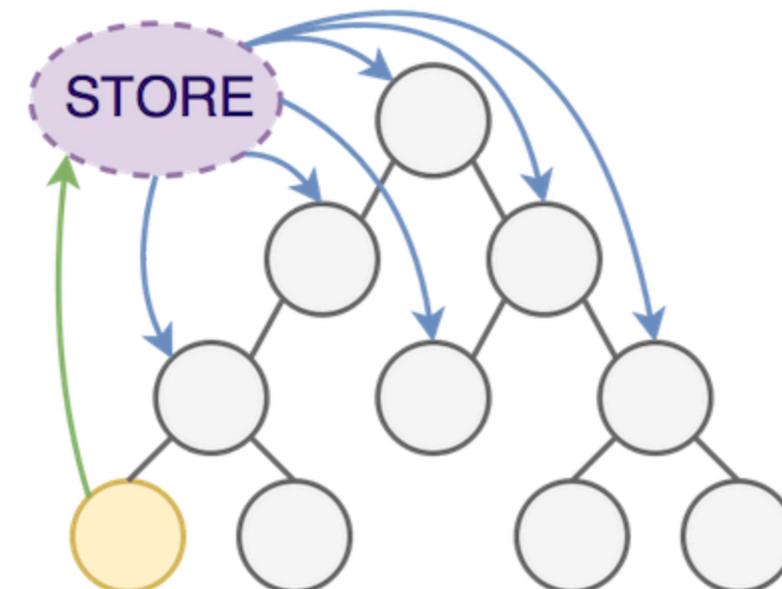


*What is Redux?*

Without Redux



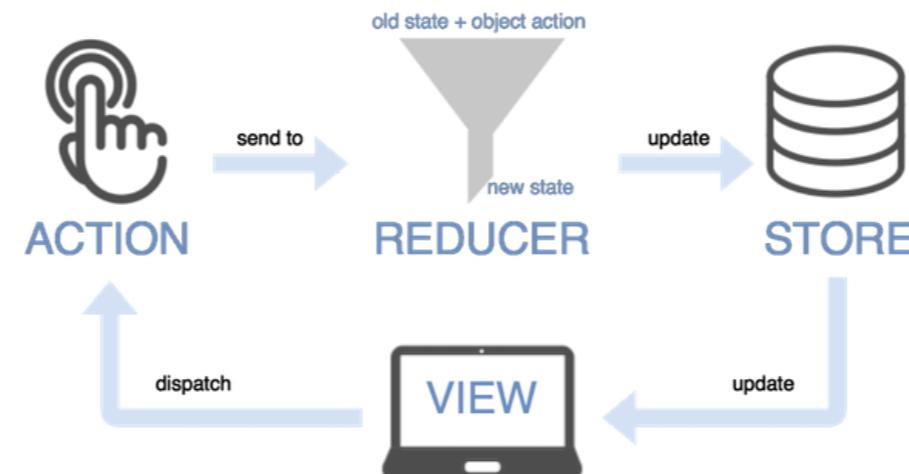
With Redux



# NGRX - REDUX



1. User requests state changes (click events...) on the app
2. An **action** is dispatched
3. The **reducer** takes the old state and an action object and emit a new state
4. The state is updated in the **store**
5. The view is also updated with the new store



# NGRX - REDUX



- **action example:**

```
export function addProduct(newProduct) {
  return {
    type: ADD_PRODUCT,
    product: newProduct
  }
}
```

- **reducer example:**

```
function productReducer(state = initState, action) {
  switch (action.type) {
    case ADD_PRODUCT:
      return {
        ...state,
        product: action.product
      };
    default:
      return state
  }
}
```

# NGRX VS OTHER LIBRARIES



- Redux can be used with any JavaScript technology
- Several libraries takes Redux principles and adapt them to Angular:
  - [ngrx/store](#)
  - [ngxs](#)

# NGRX/STORE



- Install it with:

```
ng add @ngrx/store
```

- Initializing the store in `src/app.module.ts`:

```
import { StoreModule } from "@ngrx/store";
import { catalogReducer } from "./catalog.reducer";

@NgModule({ /* ... */
  imports: [
    StoreModule.forRoot({ catalog: catalogReducer })
  ],
})
export class AppModule { }
```

# NGRX/STORE



- Create a file for all actions used to mutate the **catalog** store. Let's name it **catalog.actions.ts**

```
import { createAction } from '@ngrx/store';

export const addProduct = createAction('[Catalog] Add', props<{ product: string }>());
```

- Create a file for reducer (**catalog.reducer.ts**). This file will be in charge of mutating your store

```
import { createReducer, on } from '@ngrx/store';
import { addProduct } from './catalog.actions';

export const initialState = { products: []};

export const catalogReducer = createReducer(initialState,
  on(addProduct, (state, action) => ({
    ...state,
    products: [...state.products, action.product]
  })
),
);
```



- Access one of the states with `select()`:

```
import { addProduct: addProductAction } from "./catalog.actions"; // import the product type
import { State, Store, select } from "@ngrx/store";

@Component({ /* ... */ })
export class AppComponent {
  public products$: Observable<string[]>

  constructor(private store: Store<State>) {
    this.products$ = store.select('catalog').pipe(map(state => state.products));
  }

  addProduct() {
    this.store.dispatch(addProductAction(this.product));
  }
}
```

- `addProduct()`: is triggered in template (html)
- `dispatch()`: dispatch an `action`



- User click triggers `addProduct()` in `src/app/app.component.html`:

```
<ul>
  <li *ngFor="let product of products$ | async"> {{ product }} </li>
</ul>
<form>
  <div class="form-group">
    <input type="text" [(ngModel)]="product" name="product">
  </div>
  <button type="submit" (click)="addProduct()">Add product</button>
</form>
```

- `product` is two-way bound: value is retrieved in `src/app/app.component.ts` and sent with the action

# NGRX/STORE



- Create selector to use `select()` more appropriately:

```
export const getCatalogFeature = createFeatureSelector<{products: string[]}>('catalog');
export const getProducts = createSelector(
  getCatalogFeature,
  (state: {products: string[]}) => state.products
);
```

```
import { getProducts } from './store/catalog.selector';
import { State, Store, select } from "@ngrx/store";

@Component({ /* ... */ })
export class AppComponent {
  /* ... */

  constructor(private store: Store<{products: string[]}>) {
    this.products$ = store.select(getProducts);
  }

  /* ... */
}
```



# NGRX/STORE - EFFECTS

- Handle side effects of an action with another

```
import { StoreModule } from '@ngrx/store';
import { catalogReducer } from './catalog.reducer';
import { EffectsModule } from '@ngrx/effects';

@NgModule({ /* ... */
  imports: [
    StoreModule.forRoot({ catalog: catalogReducer }),
    EffectsModule.forRoot([ProductsEffects])
  ],
})
export class AppModule {}
```



# NGRX/STORE - EFFECTS

- Handle side effects of an action with another

```
@Injectable()
export class ProductsEffects {

  fillProducts$ = createEffect(() => this.actions$.pipe(
    ofType(ROOT_EFFECTS_INIT),
    mergeMap(() => this.restService.getProducts()),
    map((products: Product[]) => ProductsActions.Fill({ products }))
  ));

  constructor(private actions$: Actions, private restService: RestService) {}
}
```

# NGRX/STORE - TESTING



- You can use the `provideMockStore` method if you need to mock your store

```
import { TestBed } from '@angular/core/testing';
import { Store } from '@ngrx/store';
import { provideMockStore, MockStore } from '@ngrx/store/testing';
import { cold } from 'jasmine-marbles';
import { AuthGuard } from '../guards/auth.guard';

describe('Auth Guard', () => {
  let guard: AuthGuard;
  let store: MockStore<{ loggedIn: boolean }>;

  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [
        AuthGuard,
        provideMockStore({ loggedIn: false }),
      ],
    });
    store = TestBed.inject<Store>(Store);
    guard = TestBed.inject<AuthGuard>(AuthGuard);
  });

  it('should ...', () => {
    expect(guard.canActivate(store)).toEqual(false);
  });
});
```

# NGRX/STORE - TESTING



- The `jasmine-marbles` module can be used if you want to test the stream of the observable

```
it('should return false if the user state is not logged in', () => {
  const expected = cold('(a|)', { a: false });

  expect(guard.canActivate()).toBeObservable(expected);
});

it('should return true if the user state is logged in', () => {
  store.setState({ loggedIn: true });

  const expected = cold('(a|)', { a: true });

  expect(guard.canActivate()).toBeObservable(expected);
});
```







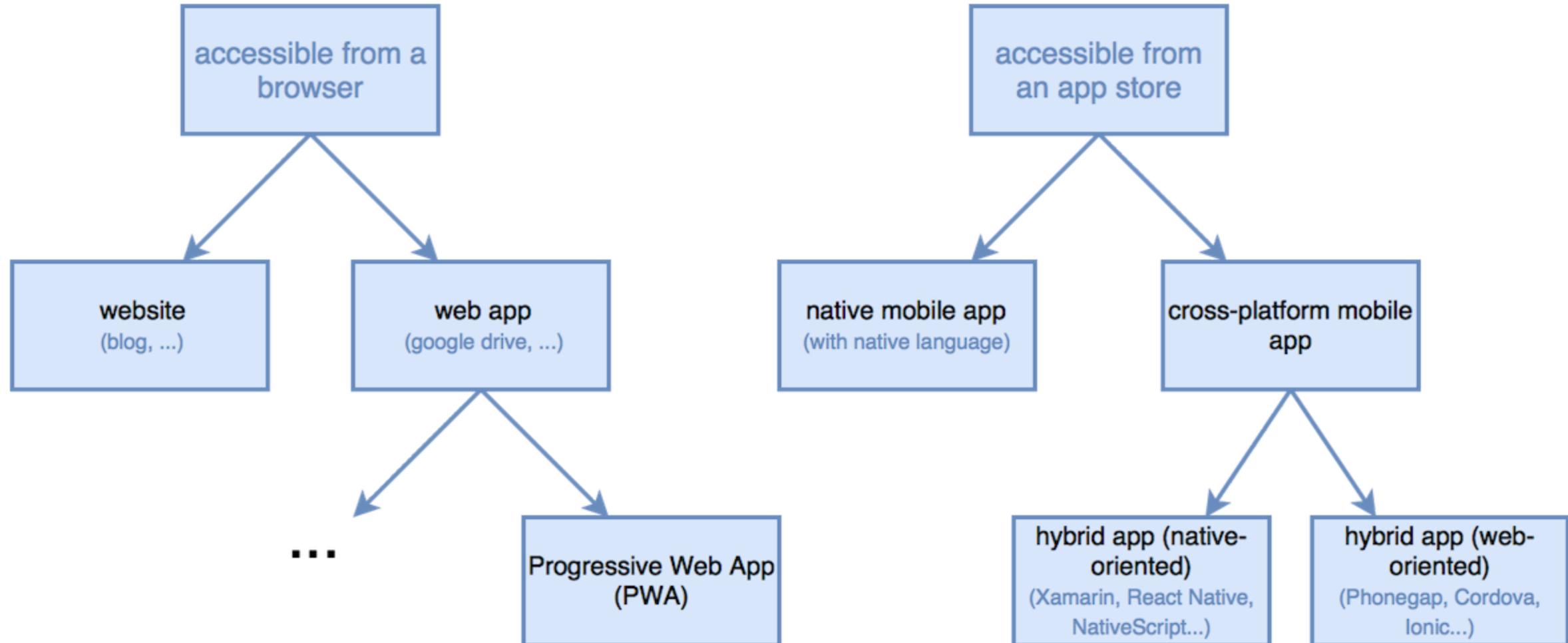
**TO GO FURTHER**

# SUMMARY



- Reminders
- Angular CLI
- Best practices
- Reactivity
- RxJS
- Standalone components
- Router
- Reactive Forms
- I18n
- SSR
- Animations
- Material
- Tests
- NgRx
- *To go further*

# TO GO FURTHER - PROGRESSIVE WEB APP



# TO GO FURTHER - PROGRESSIVE WEB APP



*Progressive Web Apps (PWA)* are a mix of *Mobile web apps* and *Native mobile apps*

- As a **mobile web app**, a PWA:
  - can be accessed in the browser
  - searches engine access
  - is responsive
  - use https
  - has a small size
- As a **native mobile app**, a PWA:
  - responds quickly to user
  - looks and feels like an app
  - can send push notifications
  - works offline
  - is accessible from the home screen
- So, it can be developed with Angular

# TO GO FURTHER - PROGRESSIVE WEB APP



## *Service Workers*

- Service Workers are scripts running in the background and in response to events, triggered by PWA to:
  - work offline
  - send push notifications
  - update content in the background
  - cache content

## *App Shell*

- The mobile web app loads initially an empty shell of the user interface, and then loads the app content

# TO GO FURTHER - PROGRESSIVE WEB APP



- Use `@angular/cli` if you want to convert your application into a PWA.

```
ng add @angular/pwa
```

- This command will add the files you need for a PWA
  - manifest.webmanifest
  - ngsw-config.json
  - some icons

# TO GO FURTHER - PROGRESSIVE WEB APP



- This command will add the files you need for a PWA (suite)
  - the index.html file has been updated in order to import the manifest file.
  - the module **ServiceWorkerModule** is now imported in the main module of your application
- The only last thing to do is to update some of these files based on your needs.

# TO GO FURTHER - NATIVESCRIPT



*NativeScript* is an OpenSource framework used to build *native mobile apps* with technology you already know

- Cross-platform: iOS and Android
- You can develop your app with either:
  - Angular
  - TypeScript
  - JavaScript
  - Vue (thanks to a community-developed plugin)
- available at <https://www.nativescript.org/>

# TO GO FURTHER - IONIC



*Ionic* is an OpenSource SDK (Software Development Kit) for *hybrid mobile app*

- Original release (Ionic 1) was built on top of AngularJS
- Recent release has support for Angular, React and Vue.js (beta)
- Ionic is built on top of **Apache Cordova**: it packages your HTML5 app to a native app
- **Cross-platform**: iOS and Android

```
npm install -g ionic  
ionic start myApp tabs
```

```
cd myApp  
ionic serve
```

# TO GO FURTHER - WEB COMPONENTS



Web component is a way to create a new HTML element that can be used in your HTML page. It allows to use components in other frameworks, like Angular, React, Vue.js, or even with no framework at all.

One of the key features of the Web Components standard is the ability to create custom elements that encapsulate your functionality on an HTML page, rather than having to make do with a long, nested batch of elements that together provide a custom page feature.

Features:

- *Custom elements*: Create custom html tags
- *Shadow DOM*: A web component hide its content from the rest of the page
- *attributes and events*: Custom elements can have attributes and fire events (Input and Output in angular)

# TO GO FURTHER - WEB COMPONENTS - USAGE



An angular component can be registered as a web component to the browser.

Use the library `@angular/element` to create custom elements.

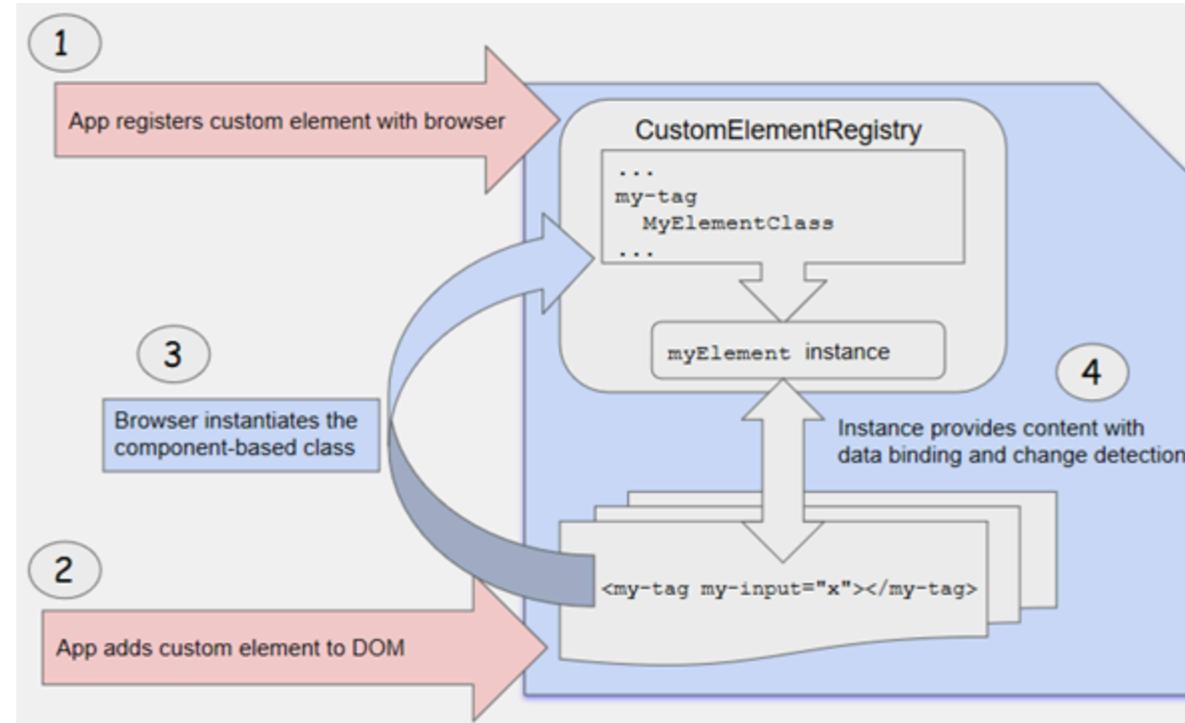
Use the function `createCustomElement` to register a web component.

```
import { createCustomElement } from '@angular/elements';

// Code in the AppComponent or the module constructor (must be executed only once)
export class AppComponent {
  constructor(injector: Injector) {
    // Convert `MyComponent` to a custom element.
    const MyElement = createCustomElement(MyComponent, {injector});
    // Register the custom element with the browser.
    customElements.define('my-element', MyElement);
  }
}
```

Avoid using the `@Component selector` as the custom-element tag name. This can lead to unexpected behavior, due to Angular creating two component instances for a single DOM element: One regular Angular component and a second one using the custom element.

# TO GO FURTHER - WEB COMPONENTS - SCHEMA





# TO GO FURTHER - WEB COMPONENTS - ENCAPSULATION

To use shadow DOM, you need to use the **encapsulation** option.

```
@Component({  
  selector: 'app-my-component',  
  template: `<h2>ShadowDom</h2>`,  
  styles: ['h2 { color: blue; }'],  
  encapsulation: ViewEncapsulation.ShadowDom,  
})  
export class MyComponent {}
```

The HTML and style will be separated from the html and style of the rest of the app who include the webcomponent

# TO GO FURTHER - WEB COMPONENTS - USE



To use a web component in an Angular application, you need to say to Angular to be ok with custom tag name which does not match an actual Angular component, by adding the custom element schema:

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

@NgModule({
  [...]
  schemas: [CUSTOM_ELEMENTS_SCHEMA]
})
export class AppModule { }
```

Using this schema you will have no error in your application if you mess up an tag name which is supposed to match an Angular component. For this reason, it is a good practice to create a wrapper component in a separated module with this schema.





**THANK YOU**



# ANNEXES

# BLOG ANGULAR



- <https://blog.ninja-squad.com/>
- <https://alligator.io/angular/>
- <https://vsavkin.com/>
- <https://blog.angular-university.io/>

# ANGULAR TIPS & TRICKS



- [Performance Checklist](#)
- [Cheatsheet](#)
- [Angular References](#)
- [Compodoc](#)
- [Storybook](#)
- [Augury](#)
- [Angular update](#)
- [Top 10 Angular Best Practices to Adapt](#)
- [Useful Angular Features You've Probably Never Used](#)



# TEST

- [Testing strategies with Angular](#)
- [A Comprehensive guide to unit-testing with Angular and Best Practices](#)
- [Replace jasmine per jest](#)
- [ng-mocks](#)
- [ngx-speculoos](#)
- [One trick for 3 times faster ng-test](#)

# I18N



- Internationalization with `@angular/localize`
- Locl - i18n tools suite for Angular



# DESIGN

- [Angular Material](#)
- [Ngx-Bootstrap](#)
- [PrimeNg](#)
- [Ant Design](#)

# FORMS

- [Angular Reactive Forms: Tips and Tricks](#)
- [Creating a material custom form field control](#)

# RXJS



- [Rxmarbles](#)
- [Rx Visualizer](#)
- [RxJS Operators for Dummies: forkJoin, zip, combineLatest, withLatestFrom](#)

# UNIVERSAL



- Git Repository

# REDUX / NGRX / NGXS



- [How to explain Redux to a 5-year-old](#)
- [Comprehensive Introduction to @ngrx/store](#)
- [Angular Service Layers: Redux, RxJs and Ngrx Store - When to Use a Store And Why?](#)
- [NGRX VS. NGXS VS. AKITA VS. RXJS: FIGHT!](#)

# SSR



- [Demystifying SSR, CSR, universal and static rendering with animations](#)