Advanced Angular

Summary

- Reminders
- Tests
- Angular best practices
- Angular architecture
- Internationalization
- Design
- Forms
- Router
- RxJS

Summary

- Industrialization
- Angular Universal
- Ngrx
- To go further

Logistics

- Schedule
- Lunch & breaks
- Other questions?

Reminders

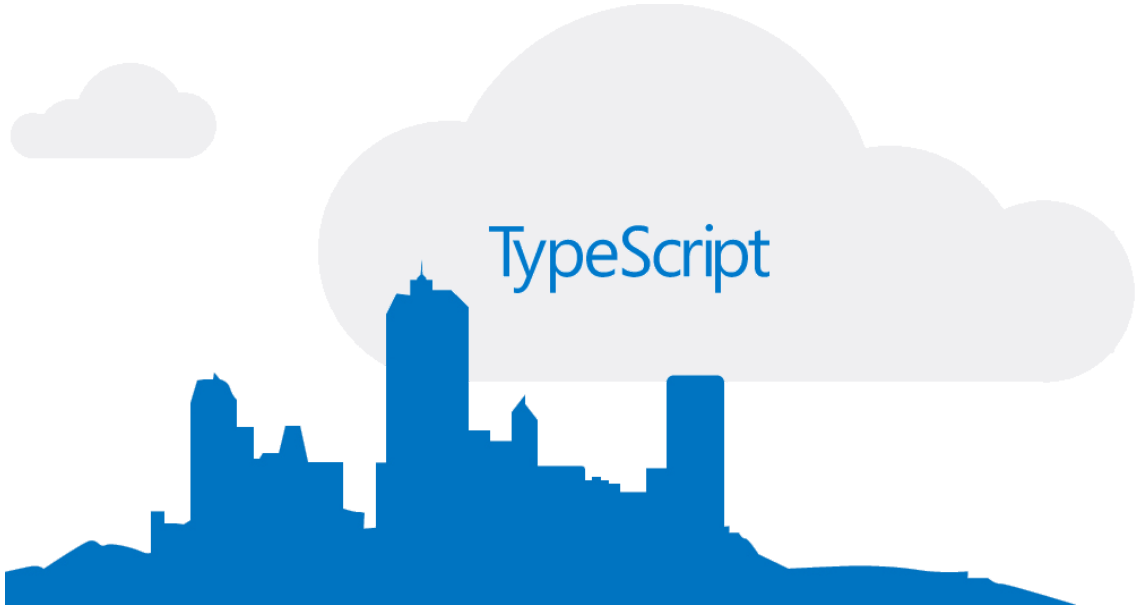
Summary

- *Reminders*
- Tests
- Angular best practices
- Angular architecture
- Internationalization
- Design
- Forms
- Router
- RxJS

Summary

- Industrialization
- Angular Universal
- Ngrx
- To go further

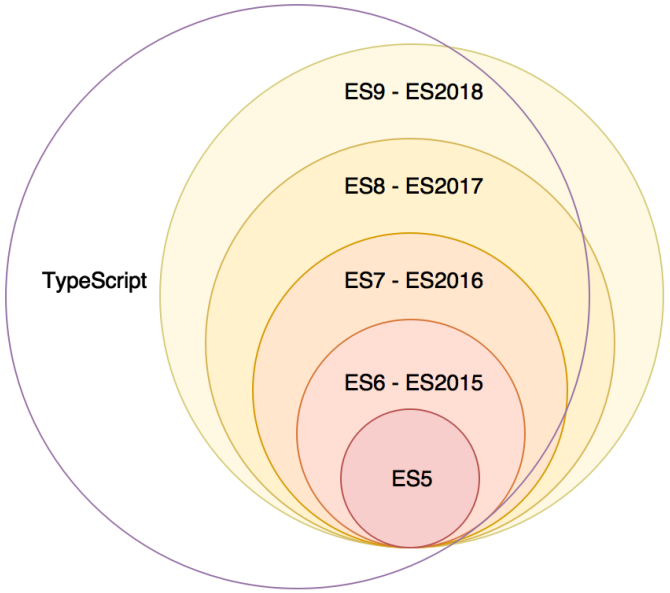
Typescript



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

Typescript

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



TypeScript - features

- Features of **ES2015+**
- Types
- Generic
- Interfaces
- Definitions files
- Mixins
- Decorators

Basic types

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

```
let age: number = 32;
const isAdult: boolean = false;
```

- boolean : `const isDone: boolean = false;`
- number : `const height: number = 6;`
- string : `const name: string = 'Carl';`
- array : `const names: string[] = ['Carl', 'Laurent'];`
- any : `const notSure: any = 4;`

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 => {  };
```

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

```
function getFullName(name: string = 'Dupont', forename?: string) {}
```

Arrays

- Arrays can be defined in two ways : literally or with the constructor

```
let list: number[] = [1, 2, 3];
let list: Array<number> = new Array<number>(1, 2, 3);
```

- This will produce exactly the same code

Enums

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

```
enum Music { Rock, Jazz, Blues };
let c: Music = Music.Jazz;
```

- Numeric values start at `0`
- Redefinition of the numeric value is allowed

```
enum Music { Rock = 2, Jazz = 4, Blues = 8 };
let c: Music = Music.Jazz;
```

- You can get back the string linked to the number

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

Classes

- **Classes** and **Interfaces** are similar to those in Object Oriented Programming
- Generated JavaScript will use `prototype`
- **Classes** are composed of one constructor, methods and properties
- Explicitly defining a constructor is optional
- Properties / methods are accessible with `this` object

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

  constructor() {}

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

const person = new Person();
```

Classes

- 3 scopes : `public`, `private` et `protected`
- `public` is the default scope
- Possibility to define `static` and `readonly` properties / methods
- TypeScript provides a shortcut to link constructor arguments to class properties

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

// ===

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

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 / methods with `public` or `protected` scope are accessible

```
class Person {
  constructor() {}

  speak() {}
 }

class Child extends Person {
  constructor() { super() } // optional

  speak() { super.speak(); }
}
```

Interfaces

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

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

class TrumpetPlay implements Musician {
  play(): void {
    console.log('Play!');
  }
}
```

Generics

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

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

let numericLog = new Log<number>();

numericLog.log(5); // Correct
numericLog.log('hello'); // Incorrect
```

npm

- Node.js includes **npm** (node package manager)
- The main way to share modules in JavaScript
- **npm** is the most popular package manager of all time

npm commands

- Main command :
  - `npm install packageName` : download the module and install it in `./node_modules` directory
  - `npm install -g packageName` : install the module globally on your system. Similar to an `apt-get install` on a linux system. Mostly used to install cli tools
  - `npm update packageName` : update a package
  - `npm remove packageName` : delete a package

package.json

- `npm` generate a `package.json` file which describes the project
- This file contains several informations:
  - `name`
  - `version`: should respect node-semver
  - `Description`: `description`, `authors`, ...
  - `scripts`: scripts `main`, `test`, ...
  - `dependencies`: `dependencies`, `devDependencies`, `peerDependencies`

package.json : dependencies

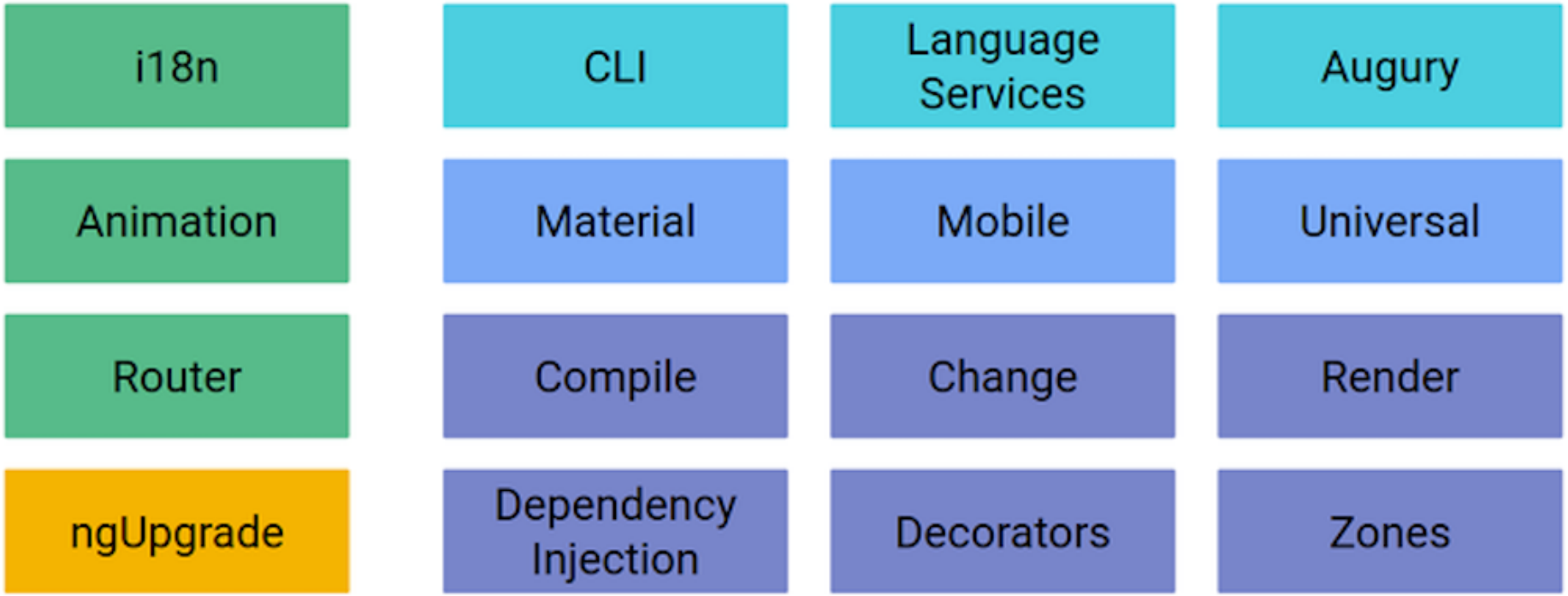
- `dependencies`: dependencies required to run your project
- `devDependencies`: dependencies required to develop your project. `--save-dev`
- `peerDependencies`: dependencies needed for some modules to work but not installed with `npm install`

Angular - Intro

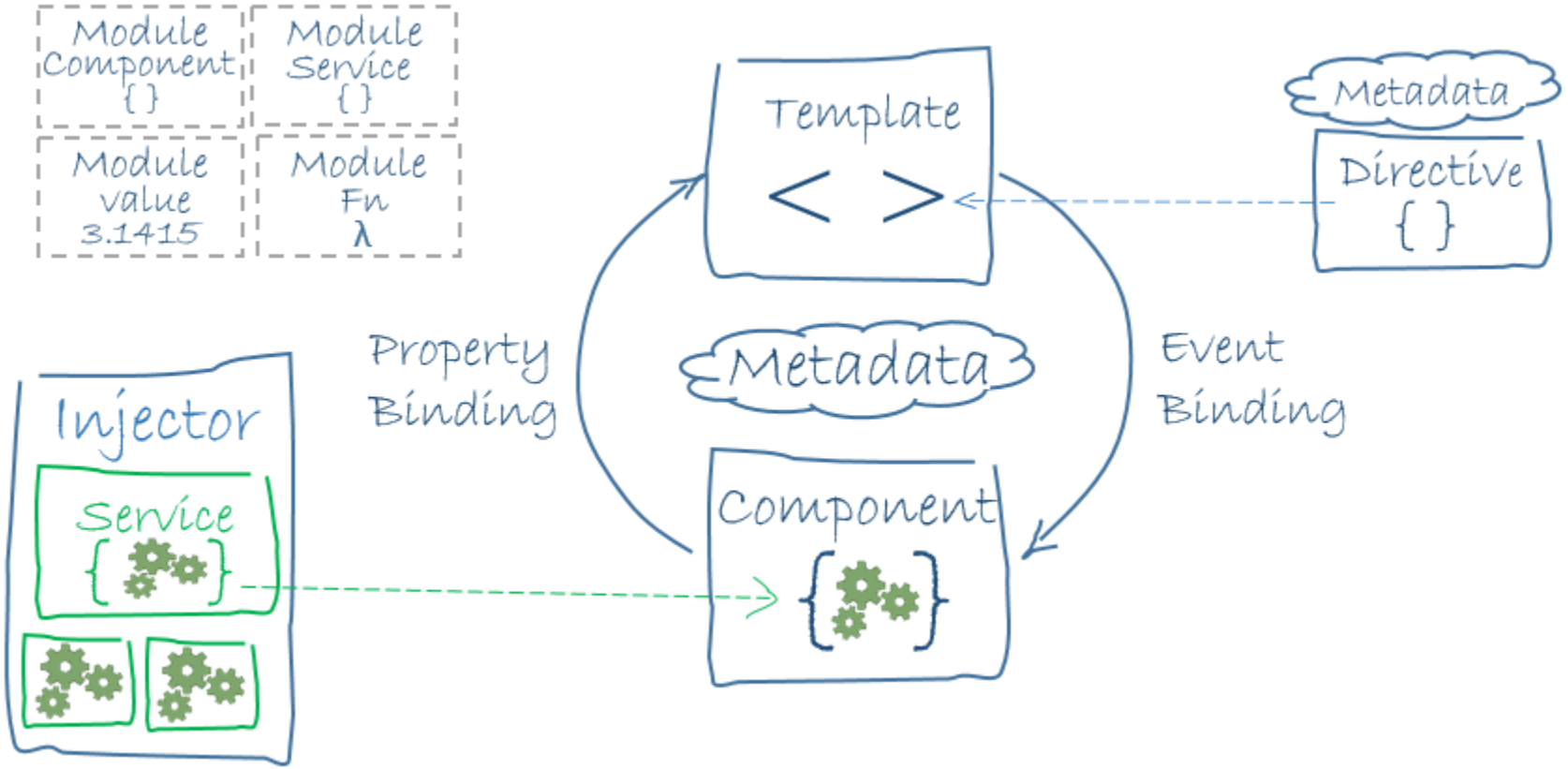
- Framework created by **Google** and announced in 2014
- Total rewrite of **AngularJS**
- Some concepts of **AngularJS** remain
- First release of `Angular 2.0.0` in September 2016
- Last major version `10.0.0` released in June 2020
- **Component** oriented Framework
- http://angular.io/

Angular - The FrameWork

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



Angular - Architecture



Angular - components

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

@Component({
  selector: 'like', // <like [nbLike]="3"></like>
  template: `
    <button (click)="clickLike()">
      <span>{{ nbLike }}</span>
      <i class="icon" [ngClass]="{'liked': isLiked}">❤</i>
    </button>
  `,
})
export class LikeComponent {
  @Input() nbLike: number;
  isLiked = false;

  clickLike() {
    this.isLiked = !this.isLiked;
  }
}
```

Angular - modules

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

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

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

Angular - services

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

import { Logger } from './logger-service';

@Injectable({
  providedIn: 'root'
})
export class UserService {
  constructor(private logger: Logger) {}

  getUsers(): void {
    this.logger.log('getUsers called!');
  }
}
```

Angular - providers

- Description of the Angular injector on how to get an instance of an element

```
@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: (appService: AppService) => {  // Use this factory function
          return appService.getConfig()
      },
      deps: [ AppService ] // Inject these parameters to the factory
    }
  ]
})
export class AppModule {}
```

Warning, you can't use TypeScript interfaces as identifier

Angular - pipes

```
{{ myVar | date | uppercase}}
<!-- FRIDAY, APRIL 15, 1988 -->

{{ price | currency:'EUR':true }}
<!-- 53.12€ -->
```

```
import { isString, isBlank } from '@angular/core/src/facade/lang';
import { PipeTransform, Pipe } from '@angular/core';

@Pipe({
  name: 'mylowercase'
})
export class MyLowerCasePipe implements PipeTransform {

  transform(value: any, param1:string, param2:string): string {
    if (!isString(value)) throw new Error('MyLowerCasePipe value should be a
string');

    return value.toLowerCase();
  }
}
```

Angular CLI

- Scaffold your Angular application with:
  - `TypeScript, Webpack, Karma, Protractor, CSS preprocessors, ...`

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

```
ng new ApplicationName
```

```
cd ApplicationName
```

```
npm start
```

Angular CLI

Several commands are available:

- `ng generate`: generate code of different angular element
  - `ng generate component Product`
  - `ng generate pipe UpperCase`
  - `ng generate service User`
  - `ng generate directive myNgIf`
- `ng lint`
- `ng test`
- `ng e2e`
- `ng build`

Lab 1

Tests

Summary

- Reminders
- *Tests*
- Angular best practices
- Angular architecture
- Internationalization
- Design
- Forms
- Router
- RxJS

Summary

- Industrialization
- Angular Universal
- 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 an 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 toll you prefer

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

Unit test - Angular elements

Possiblity to test all Angular elements :

- Components
- Services
- Directives
- Pipes
- Etc.

By default : ***.spec.ts*** files.

The recommanded 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

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

Debug Element

- Given by Angular throught 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(() => {
    TestBed.configureTestingModule({
      declarations: [ DuckDetailComponent ]
    }).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
- Recommanded way to test your component

Cons

- Harder to setup
- Harder to write
- More verbose

Stub

Stub setup :

```
const UserServiceStub: Partial<DuckService> = {
  getUser: () => ({ firstName: 'Loulou', lastName: 'Duck' })
};

describe('DuckDetailComponent', () => {

  // ...

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

  // ...

});
```

Stub

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

  component.getDuck();

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

Spies

Spies setup :

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

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

  beforeEach(async(() => {
    const duckService = jasmine.createSpyObj<DuckService>('DuckService',
['getDuck']);
    getDuckSpy = duckService.getDuck.and.returnValue({ firstName: 'Loulou',
lastName: 'Duck' });

    TestBed.configureTestingModule({
      declarations: [ DuckDetailComponent ],
      providers: [ { provide: DuckService, useValue: duckService }]
    }).compileComponents();
  }));

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

Spies

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

Async

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

Async

Common use case : ***compileComponents***

```
beforeEach(async(() => {
  TestBed.configureTestingModule({
    declarations: [ DuckComponent ],
    providers: [ { provide: DuckService, useValue: duckService }]
  })
  .compileComponents();
}));
```

Async

Testing with an ***observable***

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

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

beforeEach(() => {
  const duckService = jasmine.createSpyObj<DuckService>('DuckService',
['getDuck']);
  getDuckspy = duckService.getDuck.and.returnValue(
    of({ firstName: 'Loulou', lastName: 'Duck' })
  );
});

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

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

Async

Async 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 = jasmine.createSpyObj<DuckService>('DuckService',
['getDuck']);
  getDuckspy = 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 (exemple: a user story).
- This kind of test will start a browser and simulate a final user.

End To End Test

Protractor is :

- an End to End Test framework made by Angular
- for Angular Apps
- based on WebDriverJS, also called Selenium webdriver
- runs tests in a real browser, interacting with it as an user would

Protractor

With ***Jasmine***

```
describe('Protractor Demo App', () => {

  it('should add one and two', () => {
    browser.get('http://juliemr.github.io/protractor-demo/');
    element(by.model('first')).sendKeys(1);
    element(by.model('second')).sendKeys(2);

    element(by.id('gobutton')).click();

    expect(element(by.binding('latest')).getText()).toEqual('5');
  });

});
```

Protractor - selectors

Selectors

```
by.css('className');
by.id('myId');
```

DOM element

```
element(by.css('some-css')); // One element (the first found)
element.all(by.css('some-css')); // A list of element
```

Chain

```
element(by.css('some-css')).element(by.css('some-other-css'));
element(by.css('some-css')).all(by.css('some-other-css'));
```

Protractor - actions

Actions

```
// Click on the element.
el.click();

// Write text in an input
el.sendKeys('my text');

// Clear the text of an input
el.clear();

// Get the value of an attribute
el.getAttribute('value');

// Submit a form
el.submitForm()
```

Protractor - browser

Browser

```
browser.get('http://fakeurl.com');

browser.getTitle(); // Get the title of the current page

// Wait 5 sec for a button to be clickable
const button = element(by.css('.myButton'));
browser.wait(
  protractor.ExpectedConditions.elementToBeClickable(button),
  5000
);

browser.pause(); // Very useful to debug test

browser.takeScreenshot(); // Take screenshot of the web page
```

End to End tests - Page Object

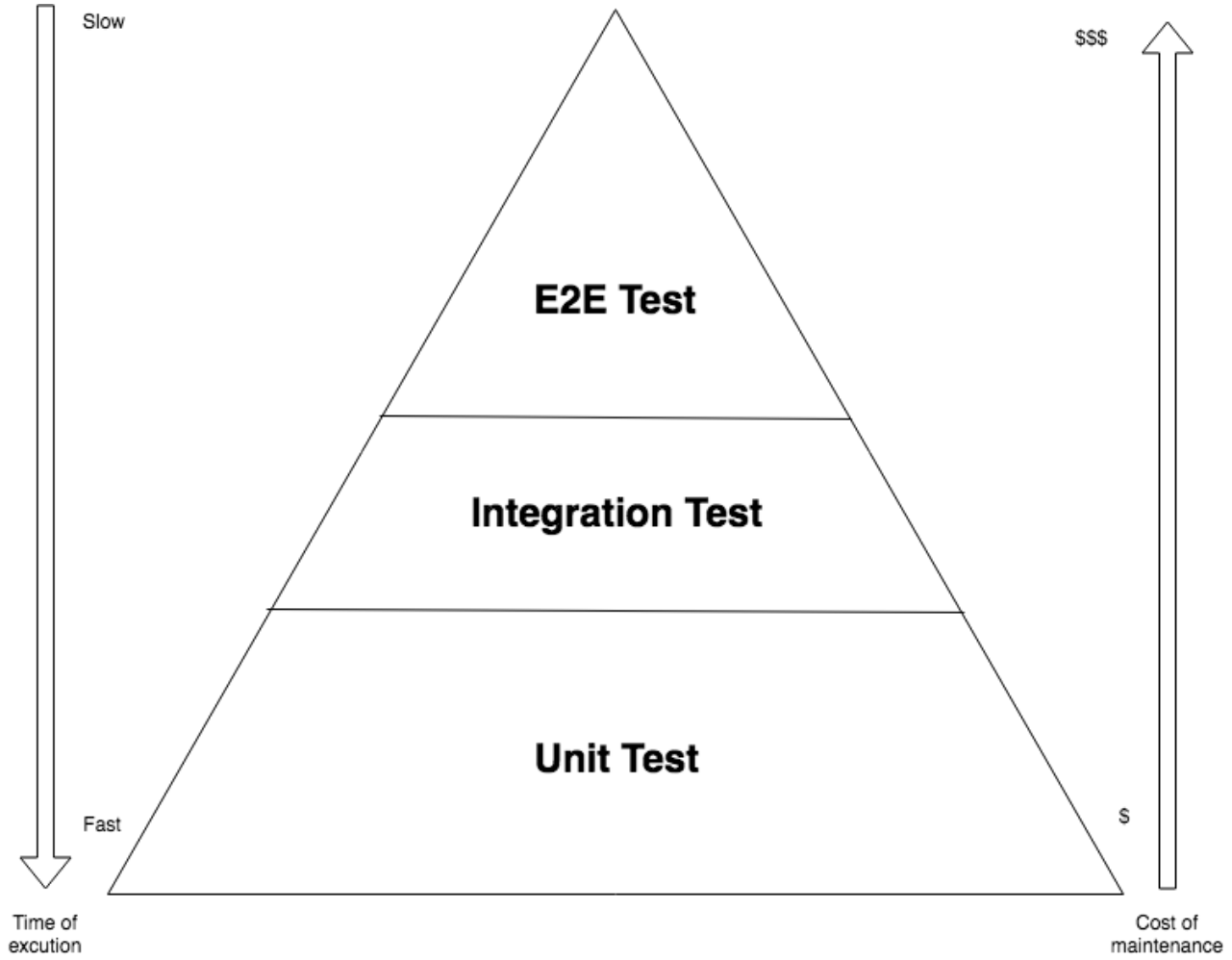
Write the API of each page of your website

```
class LoginPage {
  open(path) {
    browser.get('/login');
  }

  connectAsRoot() {
    element(by.model('user')).sendKeys('root');
    element(by.model('password')).sendKeys('root');

    browser.element('#login').submitForm();
  }
}
```

End to End tests - configuration

- ***Protractor*** by default
- Free to install whatever tool you prefer.
- ***Examples***
  - ***Cypress*** : very popular testing tool for integration tests - developer friendly.
  - ***Cucumber*** : BDD oriented testing tool - uses DSL to simplify writing tests.

Pyramid of test



Lab 2

Angular best practices

Summary

- Reminders
- Tests
- *Angular best practices*
- Angular architecture
- Internationalization
- Design
- Forms
- Router
- RxJS

Summary

- Industrialization
- Angular Universal
- Ngrx
- To go further

Angular best practices - John Papa

- For AngularJS, **John Papa** released the most popular **best practices guide**
- With Angular release, John Papa was integrated to the Angular Team: it results in **Angular best practices**
- `@angular/cli` respects all these best practices

Angular best practices

File extensions

- File name should be followed by the **file type**

```
service-name.service.ts
pipe-name.pipe.ts
module-name.module.ts
directive-name.directive.ts
component-name.component.ts
component-name.component.html
component-name.component.css
```

- Test file must be in the same folder than the component
- File name should be ended by `spec.ts`

```
service-name.service.spec.ts
component-name.component.spec.ts
```

Angular best practices

Domain-Driven Design: - is an approach to help making design decisions on softwares - focus on the domain and its logic - provides practices to technical and domain experts to make them collaborate (Ubiquitous Language...)

File organization

- You should respect DDD, so for example an e-commerce application should look like this:

```
├── Shop
├── Basket
├── Payment
└── Delivery
```

Angular best practices

Modules

- Split your code in multiple modules
- One module should represent one domain point as it is recommended in DDD

Naming

- Be consistent when you set variable name or file name (for example, always set name to plural)

Lifecycle

- Do not initialize variables in constructor component: always use `ngOnInit`

Angular best practices

Pipe

- Make **pure pipes** as much as possible
- Pure pipes are only using **pure changes**: reference changes on input parameters value
- Impure pipes are not performant:
  - Angular creates multiple instances of impure pipes
  - Angular executes impure pipes during every change detection cycle of a component
- Pipes are pure by default

```
@Pipe({
  name: 'myCustomPipe',
  pure: true // optional
})
export class MyCustomPipe { }
```

Angular best practices

Service

- Service should have only one responsibility
- Subscribe to an `Observable` only in components, never in services
- Use @Injectable when you need to use the dependency injection system
- Never use string as service identifier, use `injectionToken` instead

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

export const API_URL = new InjectionToken<string>("ProductService");

@NgModule({
  provider: [{provide: API_URL, useValue: 'api.url.com'}]
})
```

```
@Component({/* ... */})
class CustomComponent {
   constructor(private @Inject(API_URL) apiUrl) { }
}
```

Angular best practices

Catch errors

- Always handle errors of promises and observables!
- You should also catch Angular bootstrap error:

```
platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => {
    document.write('Please refresh the page');
    console.error(err);
  })
```

Accessors

- Never make accessors for no reason

```
class productComponent {
  private this._total: number = 0

  public get total():number { return this._total: }
  public set total(t: number): void { this._total = t; }
}
```

Angular best practices

Semantics

- HTML semantic matter:
  - never add `div` or whatever element when not needed
  - respect hierarchy for html tags (no `div` in `p`)
- There are `ng-container` and `ng-template`, use it

```
<ng-container *ngIf="basket">
    <div *ngFor="let product of basket">
        <div>{{product}}</div>
    </div>
</ng-container>
```

Lab 3

Angular architecture

Summary

- Reminders
- Tests
- Angular best practices
- *Angular architecture*
- Internationalization
- Design
- Forms
- Router
- RxJS

Summary

- Industrialization
- Angular Universal
- Ngrx
- To go further

Main bricks

- All modules are **scoped packages**, imported as : `@angular/somepackagename`

```
├── animations
├── common
│   └── http
├── core
├── forms
├── platform-browser
├── platform-browser-dynamic
└── router
...
```

Main bricks

core

- Core elements of Angular: NgModule, Injectable, Directive, Component, Pipe, ...

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

@NgModule(/* ... */)
export class AppModule { }
```

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

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

Main bricks

common

- Directives and pipes commonly needed and provided with the framework as an optional dependency :
  - `NgIf`
  - `NgFor`

forms

- Contains all directives for forms
- Use `FormsModule` when you want to build template driven forms
- Use `ReactiveFormsModule` when you want to build reactive forms

Main bricks

platform-browser

- Library for using Angular in a web browser
- Ahead-of-Time pre-compiled version of app sent to the browser
- Use `BrowserModule` when you want to run your app in a browser
- Needs to be imported only once
- `@angular/cli` imports it by default in the root module
- `BrowserModule` 'includes' `CommonModule` (+ other things)

Main bricks

platform-browser-dynamic

- Library for using Angular in a web browser with JIT (Just-In-Time) compilation
- App compiled on client-side

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
    enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule); // = angular.startApp()
```

Main bricks

http

- Don't use `@angular/http` but `@angular/common/http`
- Used when you need your app to talk to the server
- Import `HttpClientModule` in the root module before using `HttpClient`

router

- Library that provides elements used for navigation from one client-generated view to another
- Those elements are directives, services to access params, ...

Main bricks

animations

- Allow us to animate a component without deep knowledge of animation techniques or CSS.

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

@Component({
  animations: [
    trigger('buttonState', [
      state('inactive', style({
        backgroundColor: '#eee',
        transform: 'scale(1)'
      })),
      state('active',   style({
        backgroundColor: '#cfd8dc',
        transform: 'scale(1.1)'
      })),
      transition('inactive => active', animate('100ms ease-in')),
      transition('active => inactive', animate('100ms ease-out'))
    ])
  ]
})
```

Change detection & Zone.js

- During rendering, DOM is generated to display the app to the user
- When a change happens, the application state might have changed, so the DOM needs to be updated
- Someone needs to tell Angular to update the DOM

Change detection & Zone.js

- Changes are asynchronous and can be:
  - **Events** (click, input, submit, ...)
  - **XMLHttpRequests** (when fetching data from a remote service)
  - **Timers** (setTimeout(), setInterval())
- Problem: accessing the DOM is expensive

But who notifies Angular?

Change detection & Zone.js

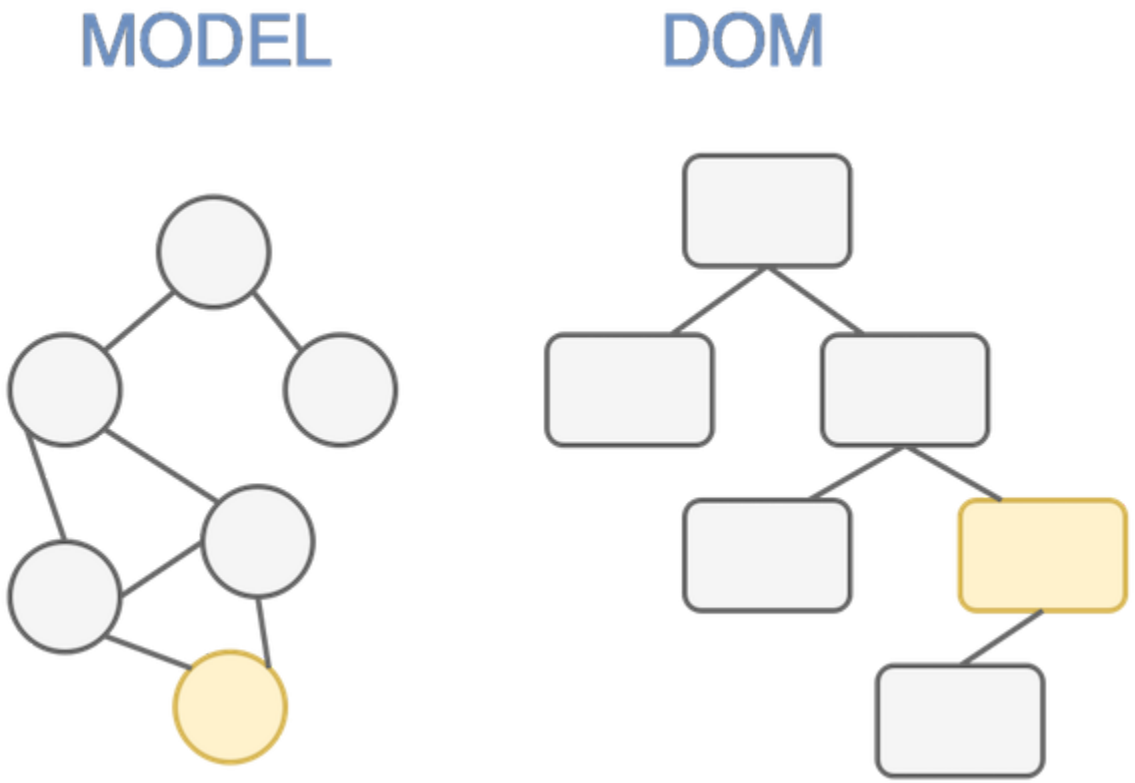
What's a zone?

A zone is an execution context that persists across async tasks. Each time code enters or exits a zone, it can perform an operation.

- **Zone.js** implements Zones for JavaScript
- created by Brian Ford and inspired by Dart
- Angular has its own zone called **NgZone**

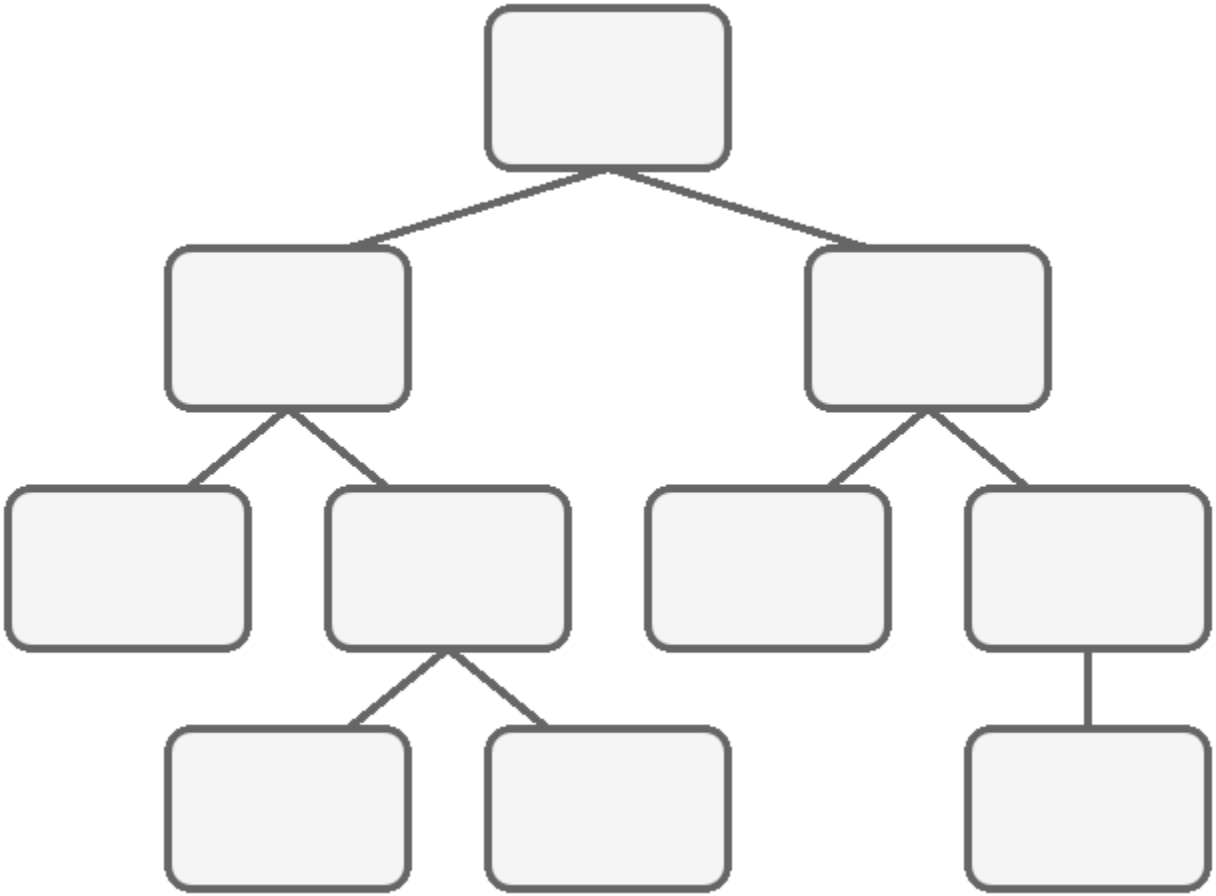
Change detection & Zone.js

1. changes requested (event)
2. zone checks for data changes & update info via data bindings



Change detection & Zone.js

- In Angular, each component has its own change detector
- Change detection is directional



Performance

Angular has to check every component every single time an event happens

How make it faster & more efficient?

Only run change detection for the parts of the application that changed their state. This can be done with: - **Immutable objects** *-* **Observables**

Performance - Immutable Objects

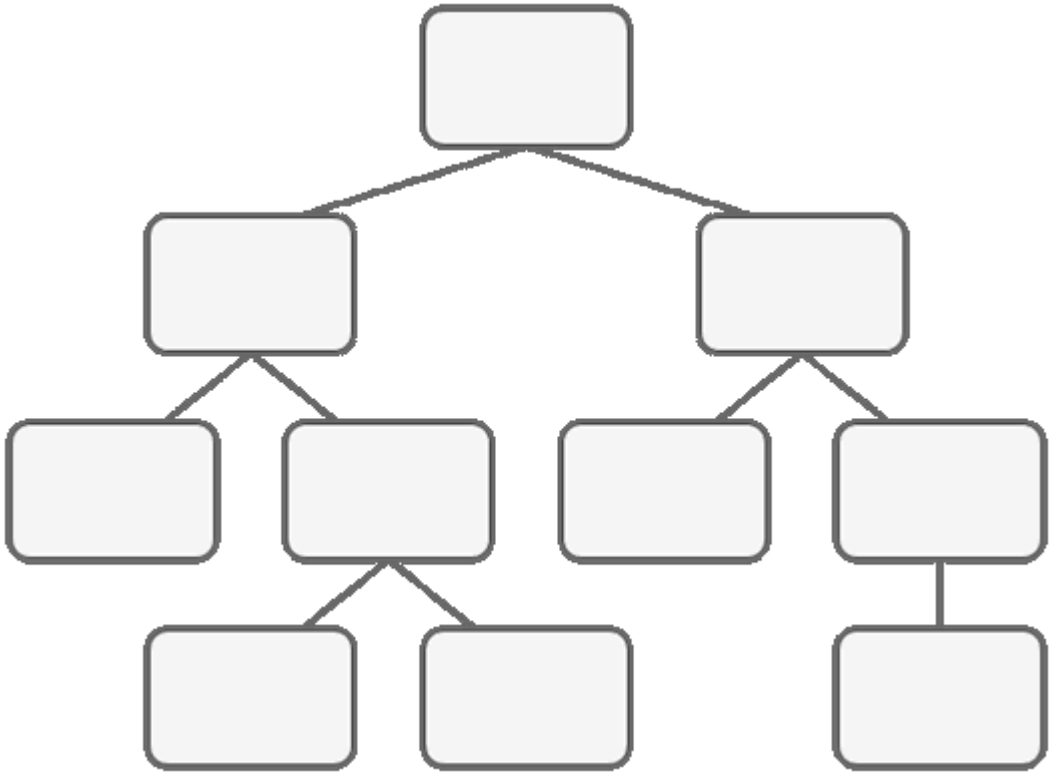
> *If a component has **immutable** input properties and only depends on these input properties, this component can change if and only if one of its input properties changes.*

- Angular can skip change detection for the component sub-tree.
- Specify a component change detection policy to only update when it changes.

```
let person = { name: 'Carl' };
person.name = 'Laurent'; // not immutable

const person1 = { name: 'Carl' };
const person2 = { ...person1, name: 'Laurent' }; // immutable
```

Performance - Immutable Objects

```
@Component({
  template: `<h2>{{person.name}}</h2>`,
  changeDetection: ChangeDetectionStrategy.OnPush
})
class PersonComponent {
  @Input() person;
}
```



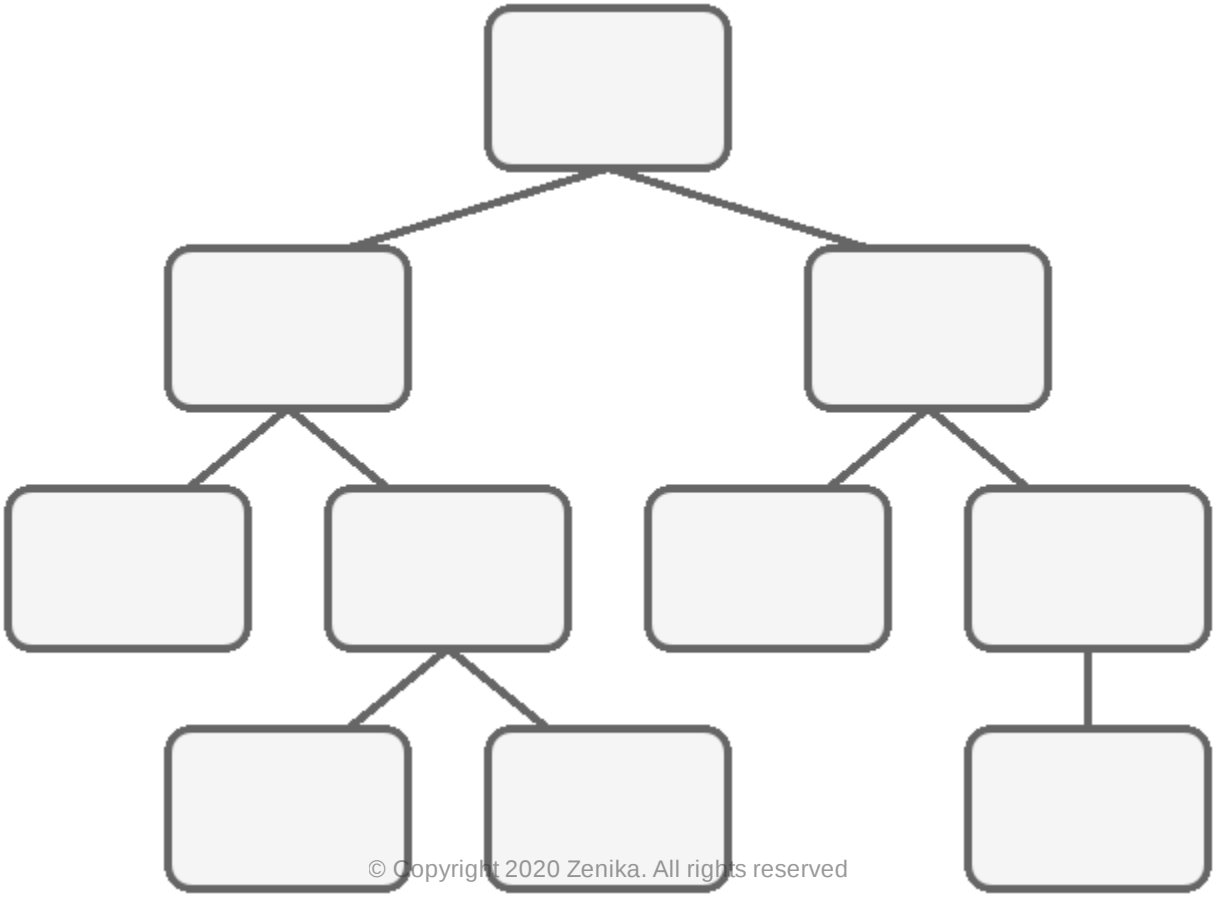
Performance - Observables

*If a component has **observable** input properties and only depends on these input properties, this component can change if and only if one of its input properties changes.*

```
@Component({
    changeDetection: ChangeDetectionStrategy.OnPush
})
class LikeComponent {
  @Input() addItemStream:Observable<any>;
  counter = 0;
  constructor(private cd: ChangeDetectorRef) {}
  ngOnInit() {
      this.addItemStream.subscribe(() => {
        this.counter++; // application state changed
        this.cd.markForCheck(); // marks path
      })
    }
  }
}
```

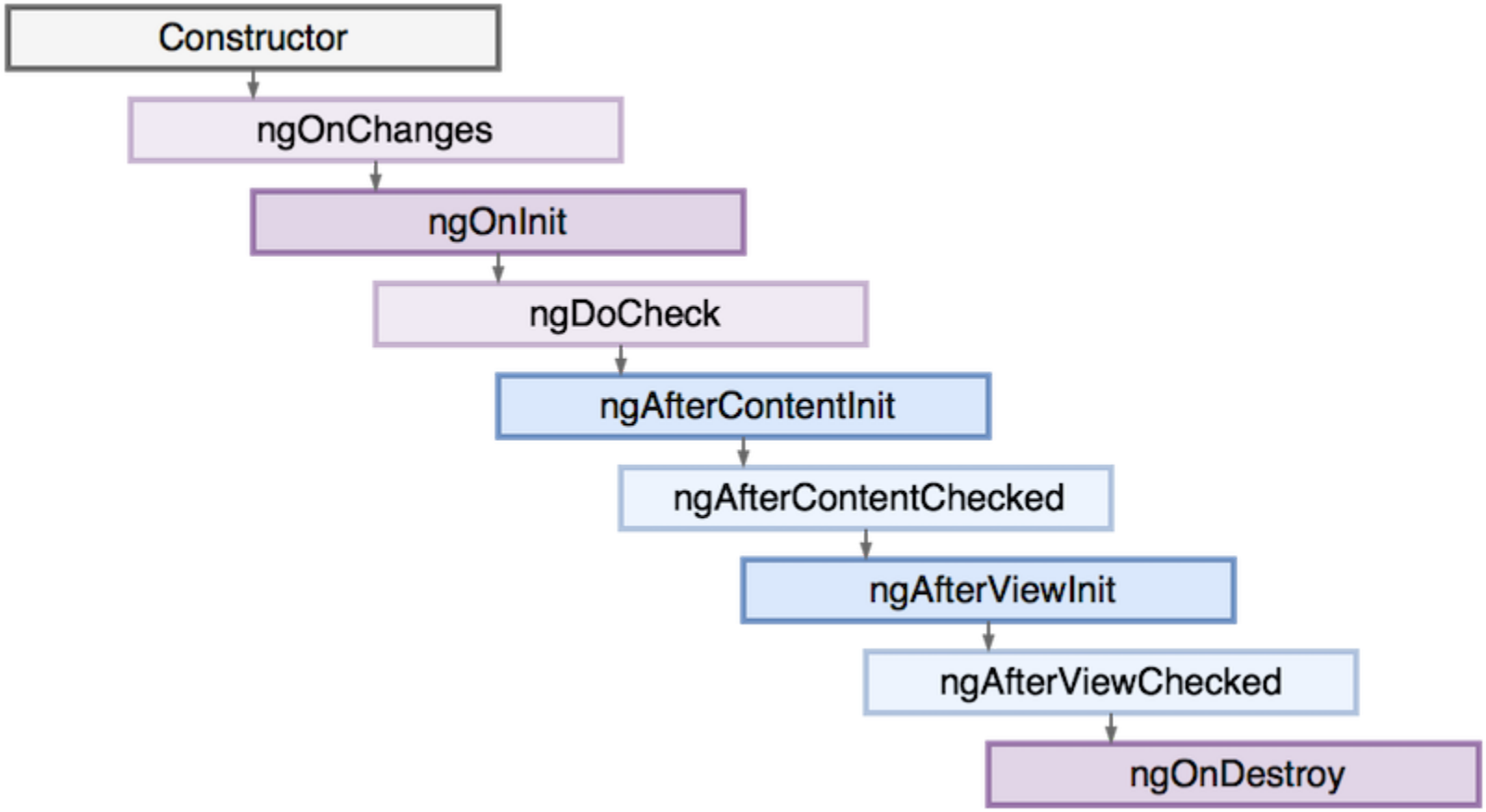
Performance - Observables

1. Observable event occurs
2. Mark path until root for check
3. Perform change detection



Components Lifecycle

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



Components 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 content initialized.
- `ngAfterContentChecked`: after every check of component content.
- `ngAfterViewInit`: after a component's views are initialized.
- `ngAfterViewChecked`: after every check of a component's views.
- `ngOnDestroy`: just before destruction.

Components Lifecycle

- Example with `ngOnInit`

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

@Component({ selector: 'user', /* ... */ })
export class UserComponent implements OnInit {

  @Input() data: User;
  products: Product[];

  ngOnInit(): void {
    this.products = this.getProducts(this.data.id);
  }

  getProducts(id){ ... }
}
```

- ***Best Practice***: don't do initialization in the constructor

Lab 4

Internationalization

Summary

- Reminders
- Tests
- Angular best practices
- Angular architecture
- *Internationalization*
- Design
- Forms
- Router
- RxJS

Summary

- Industrialization
- Angular Universal
- Ngrx
- To go further

The Angular way

Angular came with some Internationalized functionality such as :

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

Add localize package

Angular 9 has introduced a new package called @angular/localized

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

If @angular/localize is not installed, the Angular CLI may generate an error when you try to build a localized version of your app.

Templates translation

- Prepare text for translation using Angular `i18n` attribute

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

- You can add a description or meaning for translation

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

- You also may add an id

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

Templates in Typescript

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

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

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

Translation files

Angular use 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>
```

- Use angular CLI to extract source file with commande : `shell script ng xi18n`

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"
      }
    }
    ...
}
```

Define locales for build

Then create a configuration for each locale that you will use for build or serve
:- `ng serve --configuration=fr`

```
"build": {
  ...
  "configurations": {
    ...
    "fr": {
      "localize": ["fr"],
      ...
    }
  }
},
"serve": {
  ...
  "configurations": {
    ...
    "fr": {
      "browserTarget": "zenikaNgWebSite:build:fr"
    }
  }
}
```

Text translation: Angular way

- The angular way for translation is still under development and is not very well documented.
- Moreover usage is a little complexe and doesn't support some important feature like :
  - Runtime translation
  - JSON format

Let's find an alternative !

Text translation: ngx-translate

- The main alternative is ngx-Translate
  - Very similar to `pascalprecht.translate` for AngularJS

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

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

Text translation: ngx-translate

- Then, import TranslateModule to your module

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

@NgModule({
    imports: [TranslateModule.forRoot()],
    bootstrap: [AppComponent]
})
export class AppModule { }
```

ngx-translate

`ngx-translate` need to be configured through a service

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

@Component({/*...*/})
export class AppComponent {
    constructor(translate: TranslateService) {
        translate.setTranslation('en', {
          HELLO: 'I <3 {{value}}'
        });

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

        translate.use('en');
    }
}
```

ngx-translate: pipe

Now you can use `translate` pipe

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

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

ngx-translate: directive

You can also use 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: 'AYA'};
}
```

Best practice : Please be consistent between pipe and directive

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: 'Florent'};
    message: string;

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

best practice : Use translate service in a service for better performances

ngx-translate: loaders

You can fetch translate file according the fr

```
import {NgModule} from '@angular/core';
import {HttpClientModule, HttpClient} from '@angular/common/http';
import {TranslateModule, TranslateLoader} from '@ngx-translate/core';
import {TranslateHttpLoader} from '@ngx-translate/http-loader';

@NgModule({
    imports: [
        HttpClientModule,
        TranslateModule.forRoot({
            loader: {
                provide: TranslateLoader,
                useFactory: (http: HttpClient) => {
                    return new TranslateHttpLoader(http, './assets/i18n/',
'.json');
                },
                deps: [HttpClient]
            }
        })
    ]
})
export class AppModule {}
```

Lab 5

Design

Summary

- Reminders
- Tests
- Angular best practices
- Angular architecture
- Internationalization
- *Design*
- Forms
- Router
- RxJS

Summary

- Industrialization
- Angular Universal
- Ngrx
- To go further

Design - 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

Design - 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"
  }
}
```

Design - 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**

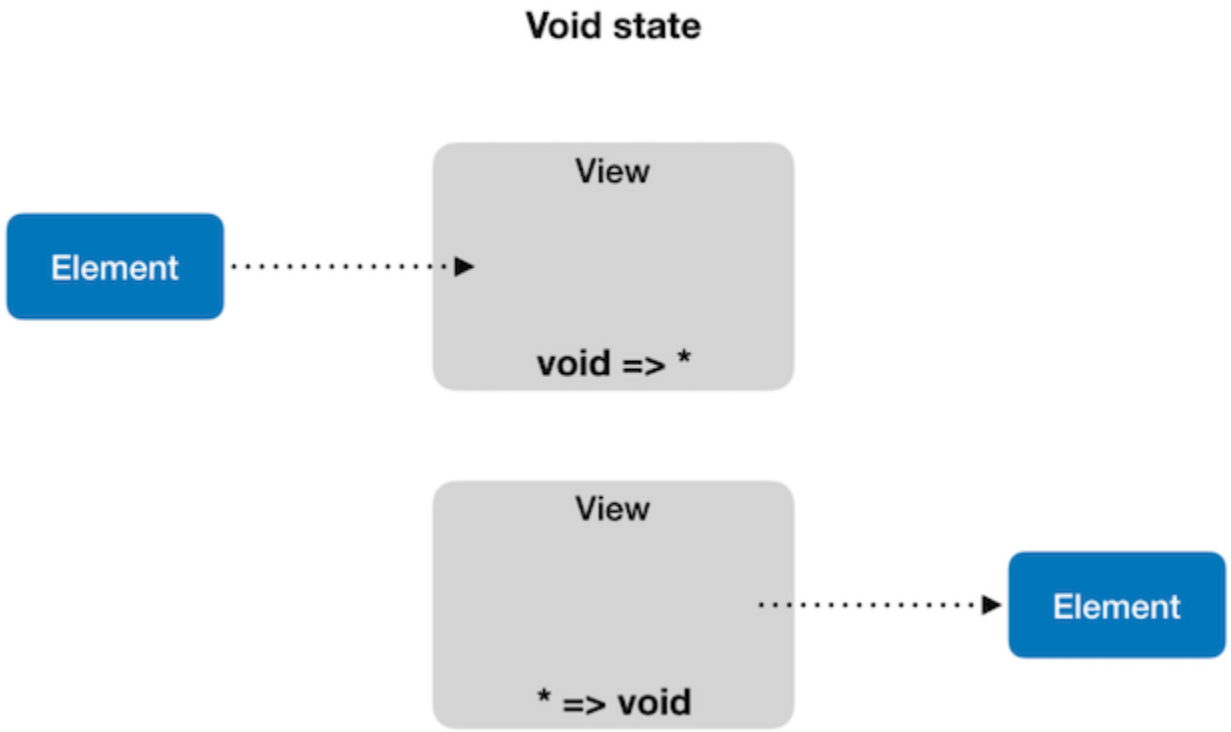
```
state('open', style({
  backgroundColor: '#eeeeee',
  height: '*'
})),
state('close', style({
  backgroundColor: '#aaaaaa',
  height: 0
}))
```

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

Design - Animations

States and Transitions

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



Design - 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'))
```

- 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)'
  }))
)
```

Design - 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%)'}))])
```

Design - 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: '*'}))
])
```

Design - 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')
```

Design - 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 */
]))
```

Design - 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})
    )
  ])
])
```

Design - 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)
      )
    ])
  ]
})
```

Design - Component Style

- Component style can be defined in two ways:
  - styles

```
@Component({
  selector: 'app-root',
  template: `<h1>App Works</h1>`,
  styles: [`
    h1 { font-weight: normal; }
  `]
})
export class AppComponent { }
```

  - styleUrls

```
@Component({
  selector: 'app-root',
  template: `<h1>App Works</h1>`,
  styleUrls: ['./app.component.css']
})
export class AppComponent { }
```

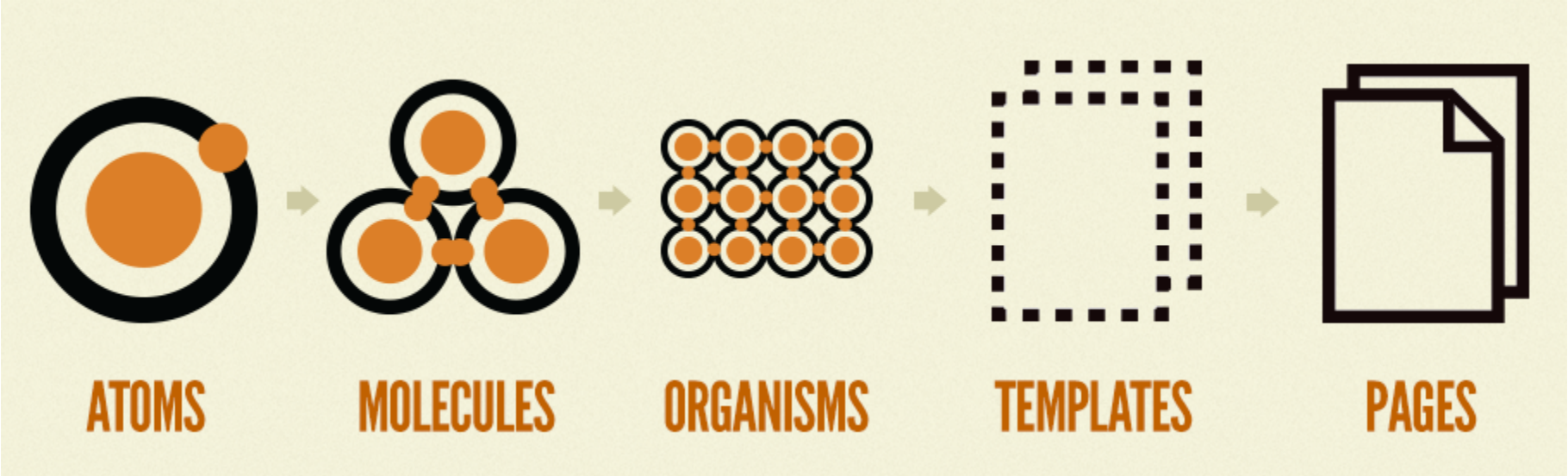
Design - Component Style

Atomic Design

- Each basic component:
  - lives in complete isolation: renders itself, obtains information by itself
  - does not affect other components if it breaks
  - can be reused with other components to form more complex composite components and user interfaces

Design - Component Style

Atomic Design

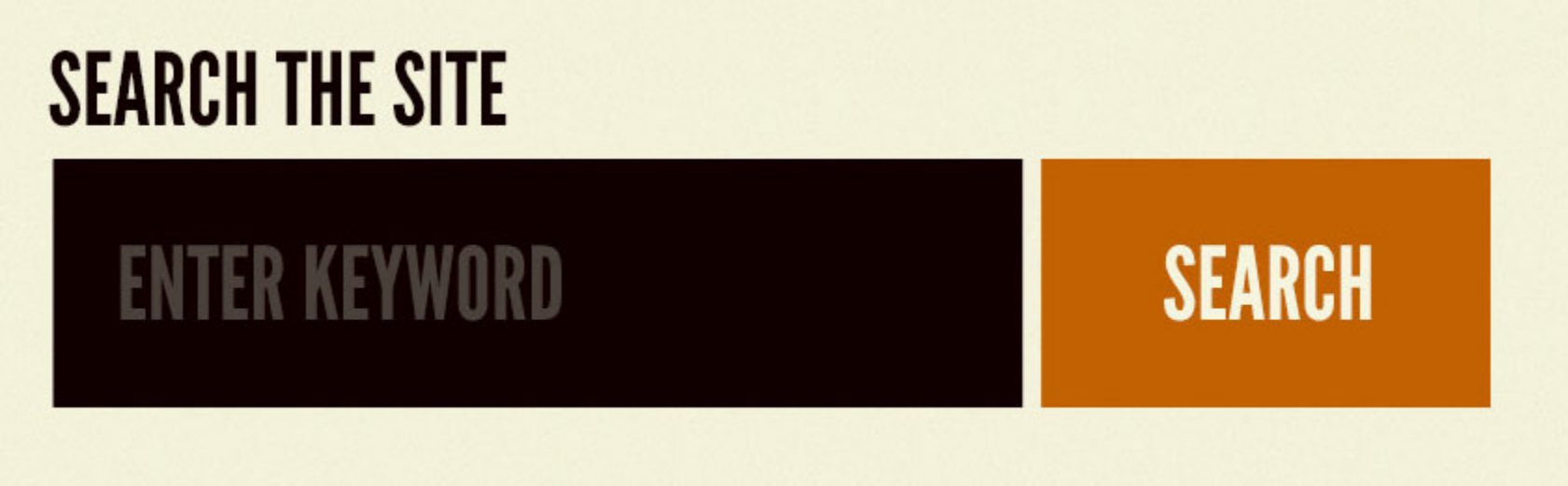


- **Atoms**: most basic building blocks (inputs, buttons, labels, color palettes, fonts, animations)

Design - Component Style

Atomic Design

- **Molecules**: simple combinations of atoms bonded together

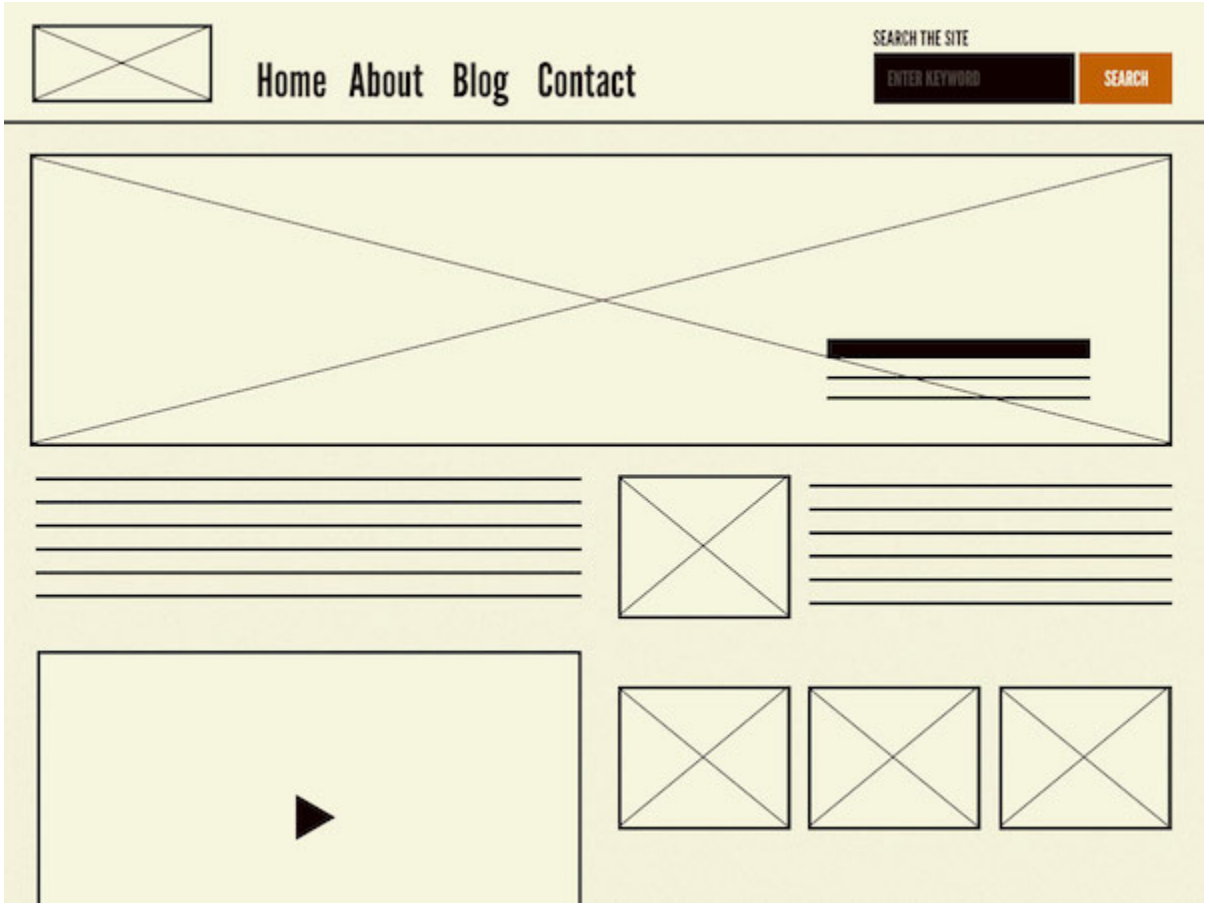


- **Organisms**: groups of molecules joined together to represent relatively complex component

Design - Component Style

Atomic Design

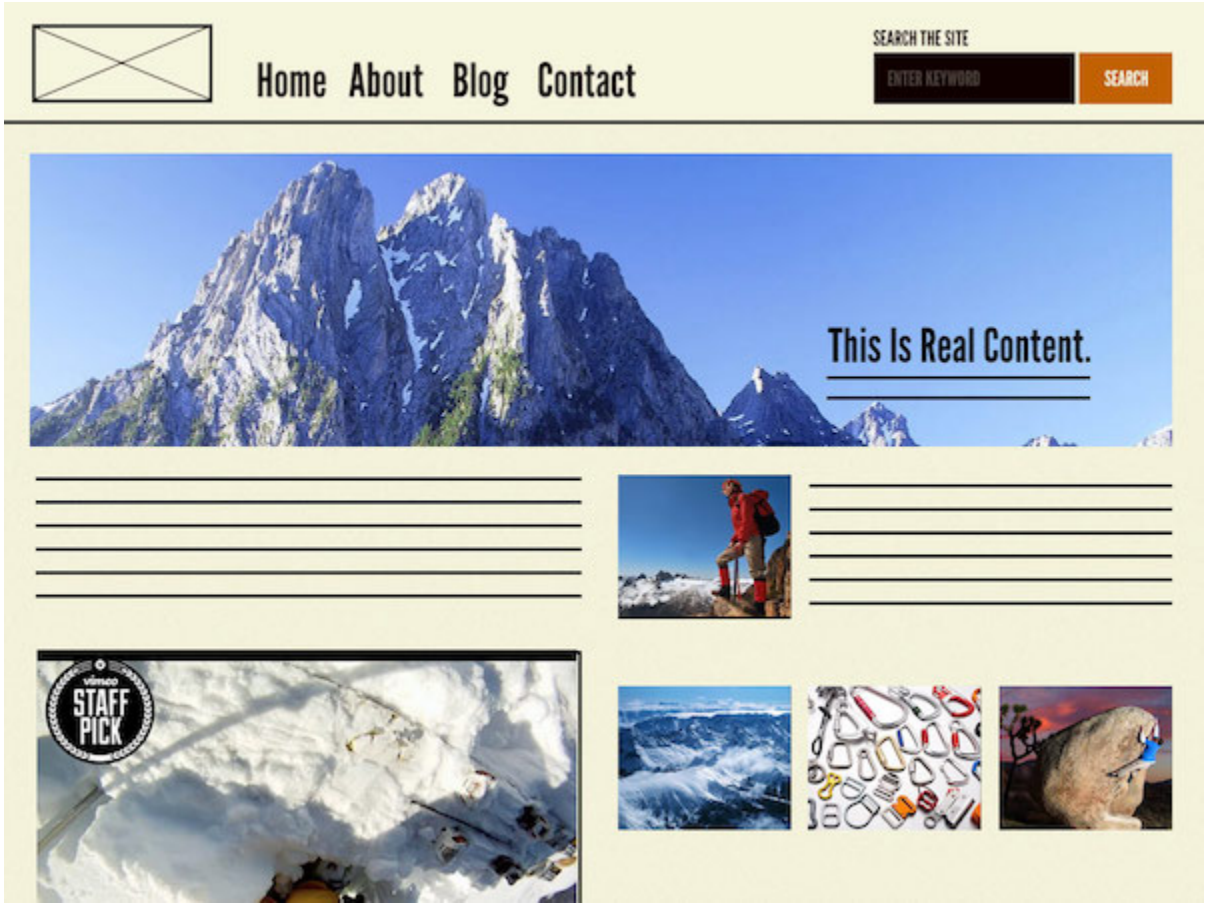
- **Templates**: Organisms put together form templates



Design - Component Style

Atomic Design

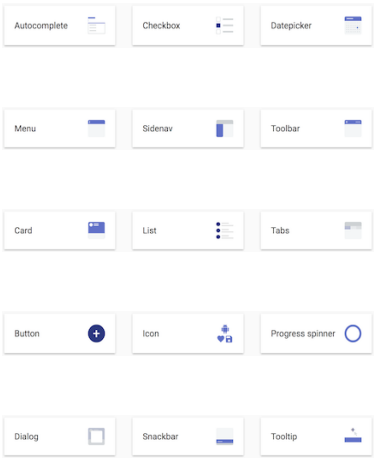
- **Pages**: With specific information, templates become pages



Design - Angular 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

Design - Angular 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

Design - Angular 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
```

Design - Angular Material

Dependency - HammerJS

- Used for gesture in some Material components (slider, tooltip)

```
npm install hammerjs
```

- Import it on your app's entry point (`src/main.ts`)

```
import 'hammerjs';
```

Design - Angular 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 { }
```

Design - Angular 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 { }
```

Design - Angular Material

Step 3: Import the component modules

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

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

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

Design - Angular 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**
  - include the CSS file in your `style.css`

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

- You can create your own theme

Design - Angular 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>
```

Design - Angular 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:

```
<i class="material-icons">face</i>
```

Lab 6

Forms

Summary

- Reminders
- Tests
- Angular best practices
- Angular architecture
- Internationalization
- Design
- *Forms*
- Router
- RxJS

Summary

- Industrialization
- Angular Universal
- 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**

Forms

- Use of several forms, but they are similar to each other
- Model can change regularly, which leads to frequently update every single form

Building these types of form can be time-consuming and costly

> **Solution: generate forms dynamically!**

Forms - Dynamic Forms

- Generate object model that can describe all scenarii

```
export class QuestionBase<T>{
  value: T;
  key: string;
  text: string;
  required: boolean;
  order: number;
  controlType: string;
  constructor(options: { /* optional fields */ } = {}) {
    this.value = options.value;
    this.controlType = options.controlType || '';
  }
}

export class TextboxQuestion extends QuestionBase<string> {
  type: string;
  controlType = 'textbox';
  constructor(options: {} = {}) {
    super(options);
    this.type = options['type'] || '';
  }
}
```

Forms - Dynamic Forms

- Define a simple service for transforming the questions to a FormGroup

```
@Injectable()
export class QuestionControlService {
  constructor() { }

  toFormGroup(questions: QuestionBase<any>[] ) {
    let group: any = {};

    questions.forEach(question => {
      group[question.key] = question.required ?
      new FormControl(question.value || '', Validators.required) :
      new FormControl(question.value || '');
    });
    return new FormGroup(group);
  }
}
```

- Once the complete model defined, you can create a component for the dynamic form

Forms - Dynamic Forms

- Create form control object on init with `QuestionControlService`
- Save user with `this.form.value` (user to be saved)

```
@Component({
    /* ... */
})
export class DynamicFormComponent implements OnInit {
  @Input() questions: QuestionBase<any>[] = [];
  form: FormGroup;

  constructor(private qcs: QuestionControlService) {  }

  ngOnInit() {
    this.form = this.qcs.toFormGroup(this.questions);
  }

  onSubmit() {
    /* save user */
  }

  get isValid() { return this.form.controls[this.question.key].valid; }
}
```

Forms - Dynamic Forms

- `ngSwitch` determines which type of question to display
- `formControlName` and `formGroup` are directives defined in `ReactiveFormsModule`

```
<form (ngSubmit)="onSubmit()" [formGroup]="form">

  <div *ngFor="let question of questions">
    <label [attr.for]="question.key">{{question.label}}</label>
    <div [ngSwitch]="question.controlType">
      <input *ngSwitchCase="'textbox'" [id]="question.key"
             [formControlName]="question.key" [type]="question.type">
      <!-- other types -->
    </div>
    <div *ngIf="!isValid">{{question.label}} is required</div>
  </div>

  <div>
    <button type="submit" [disabled]="!form.valid">Save</button>
  </div>

</form>
```

Forms - Dynamic Forms

To sum up:

- Dynamic data binding of metadata
- Dynamic fields validation
- Handle change of data model

> **Suitable when several similar form**

Lab 8

Router

Summary

- Reminders
- Tests
- Angular best practices
- Angular architecture
- Internationalization
- Design
- Forms
- *Router*
- RxJS

Summary

- Industrialization
- Angular Universal
- Ngrx
- To go further

Router - Reminders

- **RouterModule** import from **@angular/router**
- Configure your application's routes in a specific file: `app.routes.ts`
- Register these routes using **forRoot** method

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

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

Router - Reminders

- Use of **router-outlet** directive to indicate the insertion point of the page

```
@Component({
  selector: 'app',
  template: `
    <h1>{{title}}</h1>
    <router-outlet></router-outlet>
  `
})
export class AppComponent implements OnInit { }
```

- **routerLink** directive is used on links for redirection

```
<a routerLink="/contacts">Contacts</a>
```

Router - Advanced Configuration

- Configure a ***fallback*** route

```
const routes: Routes = [
  { path: 'contacts', component: ContactsComponent }
  { path: '**', component: PageNotFoundComponent }
]
```

- Define static data, accessible via the **ActivatedRoute** service

```
const routes: Routes = [
  {
    path: 'contacts',
    component: ContactsComponent,
    data: { title: 'Your contact list' }
  }
];
```

- Ability to add CSS classes on a link if the corresponding URL is active

```
<a routerLink="/contacts" routerLinkActive="active">Contacts</a>
```

Router - Advanced Configuration

- Notification thanks to `Observable` and `ActivatedRoute` service when parameters have been changed

```
@Component({ selector: 'app' })
export class AppComponent {

  user: User;

  constructor(private route: ActivatedRoute, private userService: UserService){

    this.route.paramMap.pipe(
      switchMap((params: ParamMap) =>
        this.userService.getUser(params.get('id'))
      )
    ).subscribe((user: User) => this.user = user);

    //this.route.queryParamMap
  }
}
```

Router - Events

- Router issues events when redirecting from one page 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(
    routes,
    { enableTracing: true }
)
```

Router - Events

- Example of using events for main component update

```
@Component({
  selector: 'app',
  template: `
    <h1>{{title}}</h1>
    <router-outlet></router-outlet>
  `
})
export class AppComponent implements OnInit {
  constructor(private router: Router) {}

  title: string;

  ngOnInit() {
    this.router.events.subscribe(event => {
      if (event instanceof NavigationEnd) {
        this.title = this.router.routerState.snapshot.root.data['title'];
      }
    });
  }
}
```

Router - Lazy Loading

- Stucturing 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 `forChild` method
  - it can be ***lazy-loaded***

Router - Lazy Loading

- Reduce the JavaScript ***bundle*** size to load
- The module is loaded when the user goes on one of its pages
- Configure the router with `loadChildren` property
- Several loading strategies:
  - `PreloadAllModules`: Pre-load the modules as soon as possible
  - `NoPreloading`: Loading during navigation (default strategy)
  - `QuicklinkStrategy`: Pre-load only the routes associated with links on the current page
- `@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

- Chargement à la demande du module `AdminModule`
- Loading the `AdminModule` module on demand

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

- Configuring `AdminModule` routes using `forChild` method

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

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="loader" *ngIf="loading"></span>
    </router-outlet>`
})
export class AppComponent {
  loading = false;
  constructor(router: Router) {
    router.events.subscribe((event: RouterEvent) => {
        if (event instanceof NavigationStart) {
          this.loading = true;
        } else if (event instanceof NavigationEnd) {
          this.loading = false;
        }
      }
    );
  }
}
```

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:
  - `CanLoad`: when loading a **feature module**
  - `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 - 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 - CanDeactivate

- Example: confirmation request before redirecting

```
@Injectable()
export class CanDeactivateGuard implements CanDeactivate<ContactFormComponent> {

  canDeactivate(
    component: ContactFormComponent,
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): Observable<boolean> | boolean {

    if (!component.previousValue ||
      component.previousValue.name === component.nextValue.name) {

      return true;
    }

    return component.dialogService.confirm('Discard changes?');

  }
}
```

Lab 9

RxJS

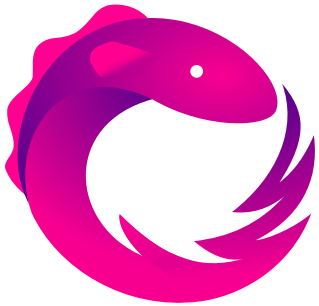
Summary

- Reminders
- Tests
- Angular best practices
- Angular architecture
- Internationalization
- Design
- Forms
- Router
- *RxJS*

Summary

- Industrialization
- Angular Universal
- Ngrx
- To go further

What is Reactive Programming?



ReactiveX is a library for composing asynchronous and event-based programs by using observable sequences. The whole library is based on ***Observables*** *&* ***Observers****. Observables emit events that will be handled by Observers.*

Advantages of using Reactive Programming

- Provides the same features of Promises

Advantages of using Reactive Programming

- Provides the same features of Promises
- Using ***streams***

Advantages of using Reactive Programming

- Provides the same features of Promises
- Using ***streams***
- Lazy triggering

Advantages of using Reactive Programming

- Provides the same features of Promises
- Using ***streams***
- Lazy triggering
- Is cancelable

Advantages of using Reactive Programming

- Provides the same features of Promises
- Using ***streams***
- Lazy triggering
- Is cancelable
- Provides powerful operators

Frameworks

Reactive programming is a usable pattern in many languages. A lot of frameworks use it for a while.
A short list :

- Meteor.js
- RxJava
- ProAct.js
- Reactor
- Shiny
- ... many more. *https://en.wikipedia.org/wiki/Reactive_programming*

Data streams

Reactive programming is programming with asynchronous data streams.

Data streams

We can create data streams of anything (numbers, strings) and listen to this stream.

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

new Observable(subscriber => {
  subscriber.next('Rxjs'); // string
  subscriber.next(2018); // number
  subscriber.next('training'); // string
});
```

We can see that as a sequence of events ordered in time. A stream can emit a value, an error or a completed signal.

Our Observable can emit multiple values and these values can be anything we want.

From a source

In some case we will need to create an Observable from an existing value, like an array or a Promise. To do so, we will use an other method from Observable. A method called ***from***.

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

const myArray = [1, 2, 3]
const myObservableArray = from(myArray)

const myPromise = new Promise(resolve => resolve('Hello RxJS!'))
const myObservablePromise = from(myPromise)
```

Of a variable amount of values

Also we can create an Observable from a variable amount of values. These values can be from different types. For that we will use ***of***.

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

const myVariableAmountObservable = of(1, 'Hello World', {magicKey: 42})
```

Subscribe to a stream

RxJS provide us ***subscribe*** method to get our data. With it, we will be able to listen to the stream. The stream is the subject (Observable) being observed.

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

const myObservable = new Observable(subscriber => {
  subscriber.next('Hello');
  subscriber.next('Observable');
  subscriber.next(2);
});

myObservable.subscribe(response => {
    console.log(response)
})

// output =>
// 'Hello'
// 'Observable'
// 2
```

Subscribe to a stream

As we said, Observable let us operate in three differents actions. onNext by default (like the previous examples), onError and onCompleted.

Subscribe to a stream

As we said, Observable let us operate in three differents actions. onNext by default (like the previous examples), onError and onCompleted.

- ***onError*** : will be called when Observable indicates that it has failed to generate the expected data or has encountered some other error. This stops the Observable and it will not make further calls to onNext or onCompleted.

Subscribe to a stream

As we said, Observable let us operate in three differents actions. onNext by default (like the previous examples), onError and onCompleted.

- ***onError*** : will be called when Observable indicates that it has failed to generate the expected data or has encountered some other error. This stops the Observable and it will not make further calls to onNext or onCompleted.

- ***onCompleted*** : is called after it has called onNext for the final time.

Subscribe to a stream - Example

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

const myObservable = new Observable(subscriber => {
  subscriber.next('Hello');
  subscriber.next('Observable');
  subscriber.next(2);
});

myObservable.subscribe(
  next => console.log('onNext: %s', next),
  error => console.error('onError: %s', error),
  () => console.log('onCompleted')
)
```

What are operators

Operators are methods provided by Observable. They allow us to do operations on an Observable but ***they don't modify*** the Observable source, they return a new Observable.

An Operator is essentially a pure function which takes one Observable as input and generates another Observable as output.

Operators types

- There are differents types of operators for different purposes. Some for creation, transformation, filtering and many others.
  - map
  - filter
  - flatMap
  - merge
  - delay
  - etc.
- They are all listed on RxJS official documentation.
  http://reactivex.io/documentation/operators.html

Map

Apply projection with each value from source.

```
import { of } from 'rxjs';
import { map } from 'rxjs/operators';

const myObservable = of(1, 2, 3);

myObservable.pipe(
  map(x => x * 10)
)
.subscribe(console.log);
```

What will it render?

Map

```
import { of } from 'rxjs';
import { map } from 'rxjs/operators';

const myObservable = of(1, 2, 3);

myObservable.pipe(
  map(x => x * 10)
)
.subscribe(console.log);

// => 10
// => 20
// => 30
```

Filter

Filter operator help us to filter values. Only values that pass the provided condition will be emitted.

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

const myArray = [1, 2, 3, 4, 5];

from(myArray).pipe(
  filter(element => element > 3)
)
.subscribe(console.log)
```

What will it render?

Filter

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

const myArray = [1, 2, 3, 4, 5];

from(myArray).pipe(
  filter(element => element > 3)
)
.subscribe(console.log)

// => 4
// => 5
```

Error handling

Since our observable pipelines become more complex, we need now to catch our possible errors. Fortunately, Observers take an error callback to receive any unhandled errors in an observable stream.

```
import { interval } from 'rxjs';
import { map } from 'rxjs/operators';

const source = interval(500).pipe(
  map(value => {
    if (value > 5) {
      throw new Error("Error detected!");
    }
    return value;
  })
);

source.subscribe(
  value => console.log(value),
  err => console.error(err.message),
  () => console.log(`We're done here!`)
);

// 0 1 2 3 4 5 Error detected!
```

Error handling - catchError

Catch should return another observable or throw again to be handled by the next catch operator or the observer's error handler if there's no additional catch operator.

```
import { interval, of } from 'rxjs';
import { map, catchError } from 'rxjs/operators';

const source = interval(500).pipe(
  map(value => {
    if (value > 5) {
      throw new Error("Error detected!");
    }
    return value;
  }),
  catchError(error => of(5))
);

source.subscribe(
  value => console.log(value),
  err => console.error(err.message),
  () => console.log(`We're done here!`)
);

// 0 1 2 3 4 5 5 We're done here!
```

Lab 10

Industrialization

Summary

- Reminders
- Tests
- Angular best practices
- Angular architecture
- Internationalization
- Design
- Forms
- Router
- RxJS

Summary

- *Industrialization*
- Angular Universal
- Ngrx
- To go further

Industrialization - angular-cli configuration

- `@angular/cli` configuration takes place in `angular.json`
- `projects`: Main configuration of the project
- For each project, an `architect` object is defined with the configuration of the different tools you will use
  - `build / serve`: Configuration about how to build / serve the application
  - `e2e`: End-to-end tests configuration
  - `lint`: Configuration for TSLint
  - `test`: Unit tests configuration
  - `extract-i18n`: Configuration for the internationalisation

Industrialization - angular-cli configuration

projects

```
"projects": {
  "zenikaNgWebSite": {
    "projectType": "application",
    ...
  }
},
```

- `zenikaNgWebSite`: the name of your project
- `projectType`: you project can be an application or a library

Industrialization - angular-cli configuration

- For every project, you sill have this configuration object

```
"zenikaNgWebSite": {
    "projectType": "application",
    "schematics": {},
    "root": "",
    "sourceRoot": "src",
    "prefix": "app",
    "architect": {
      "build": {},
      "serve": {},
      "test": {},
      "lint": {},
      "e2e": {}
    }
}
```

Industrialization - build

- Global configuration of the application

```
"build": {
    "builder": "@angular-devkit/build-angular:browser",
    "options": {
      "outputPath": "dist/zenikaNgWebSite",
      "index": "src/index.html",
      "main": "src/main.ts",
      "polyfills": "src/polyfills.ts",
      "tsConfig": "tsconfig.app.json",
      "aot": false,
      "assets": [
        "src/favicon.ico",
        "src/assets"
      ],
      "styles": [
        "src/styles.scss"
      ],
      "scripts": []
    },
    "configurations": { ... }
}
```

Industrialization - build

- Global configuration of the application

```
"build" : {
   "configurations": {
     "production": {
       "fileReplacements": [
         {
           "replace": "src/environments/environment.ts",
           "with": "src/environments/environment.prod.ts"
         }
       ],
       "optimization": true,
       "outputHashing": "all",
       "sourceMap": false,
       "extractCss": true,
       "namedChunks": false,
       "aot": true,
       "extractLicenses": true,
       "vendorChunk": false,
       "buildOptimizer": true,
       "budgets": [...]
     }
   }
}
```

Industrialization - build

- You can also set size thresholds in your configuration to ensure that your application stay within size boundaries.

```
"budgets": [
  {
    "type": "initial",
    "maximumWarning": "2mb",
    "maximumError": "5mb"
  }
]
```

- Multiple values are available for the `type` parameter : `bundle`, `initial`, `allScript`, `all`, `anyComponentStyle`, `anyScript`, `any`

Industrialization - e2e tests

e2e

```
  "e2e": {
    "builder": "@angular-devkit/build-angular:protractor",
    "options": {
      "protractorConfig": "e2e/protractor.conf.js",
      "devServerTarget": "zenikaNgWebSite:serve"
    },
    "configurations": {
      "production": {
        "devServerTarget": "zenikaNgWebSite:serve:production"
      }
    }
  }
  ...
```

- `protractorConfig`: config file name (path included)

Industrialization - lint

lint

```
  "lint": {
    "builder": "@angular-devkit/build-angular:tslint",
    "options": {
      "tsConfig": [
        "tsconfig.app.json",
        "tsconfig.spec.json",
        "e2e/tsconfig.json"
      ],
      "exclude": [
        "**/node_modules/**"
      ]
    }
  },
  ...
```

- `tsConfig`: location of the `tsconfig` file name
- `exclude`: files to ignore

Industrialization - unit tests

test

```
  "test": {
     "builder": "@angular-devkit/build-angular:karma",
     "options": {
       "main": "src/test.ts",
       "polyfills": "src/polyfills.ts",
       "tsConfig": "tsconfig.spec.json",
       "karmaConfig": "karma.conf.js",
       "codeCoverage": true
     }
   }
...
```

- `karmaConfig`: location of the karma config file
- `codeCoverage`: A code coverage report will be generated
- You could also enable `codeCoverage` with `ng test --watch=false --code-coverage`

Industrialization - Ahead-of-Time (AOT) Compilation

Ahead-of-Time (AOT) Compilation

- *It transforms TypeScript and HTML code into JavaScript code at* **build time**
- *It happens* **before** *the browser downloads and runs that code*

- Opposed to **Just-in-Time (JIT) Compilation**, which compiles your app in the browser at runtime
- When you run `ng build` or `ng serve`, it uses the **JIT Compilation**
- To force **AOT Compilation**, use `ng build --aot` or `ng serve --aot`
- In production, **AOT Compilation** should be used: when using `--prod` in `ng build --prod`, **AOT** is enable by default

Industrialization - Ahead-of-Time (AOT) Compilation

Why use AOT?

- Pre-compiled version of the app is downloaded by the browser: it can render the app **faster**
- This already-compiled app does not need the Angular compiler anymore: the **Angular framework size is smaller**
- External HTML templates and CSS style sheets are inlined: there are **fewer asynchronous requests**

Angular Universal

Summary

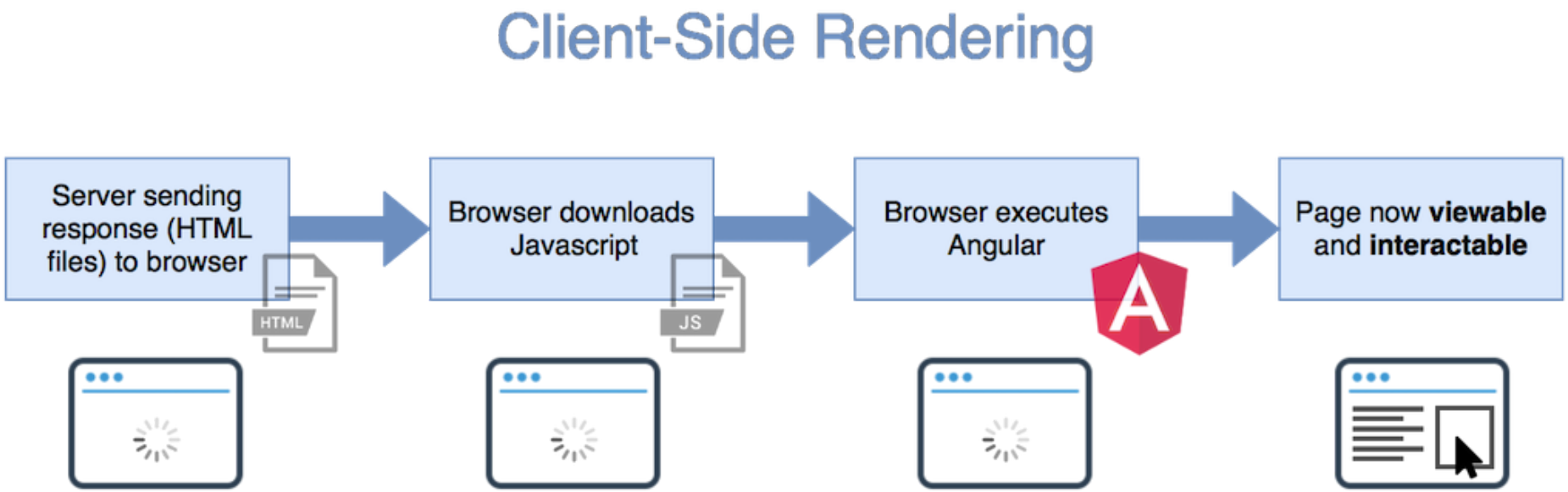
- Reminders
- Tests
- Angular best practices
- Angular architecture
- Internationalization
- Design
- Forms
- Router
- RxJS

Summary

- Industrialization
- *Angular Universal*
- Ngrx
- To go further

Angular Universal

- **Angular Universal** 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

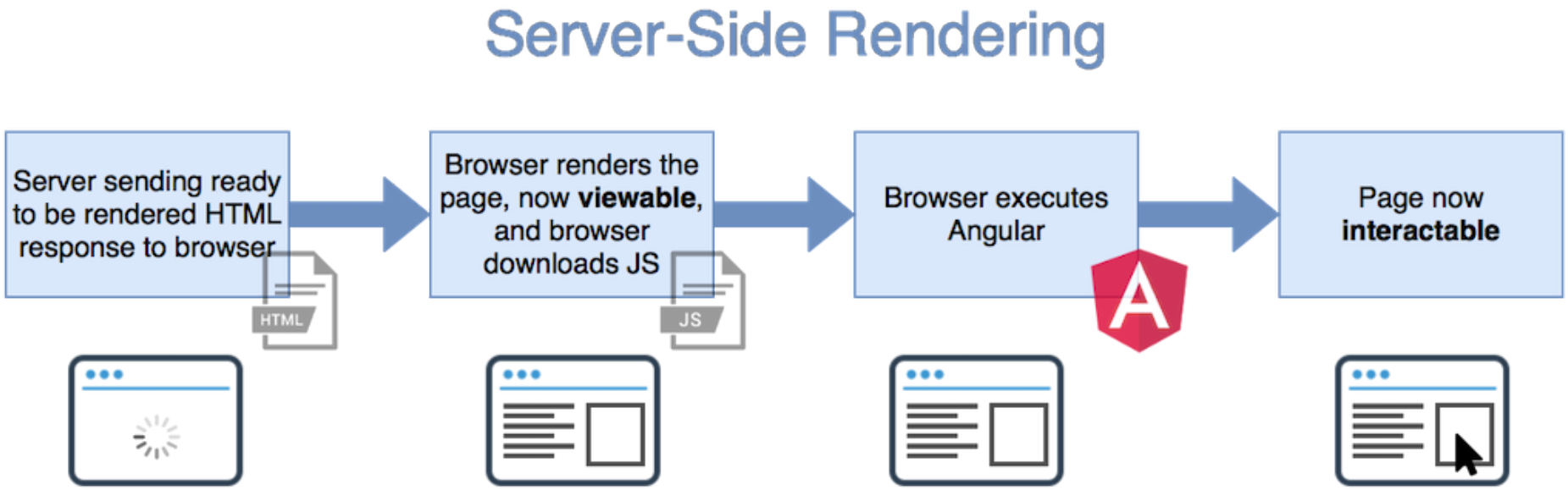


Angular Universal

- **Angular Universal** 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 Universal** ?
  - Search Engine Optimization (SEO): facilitate web crawlers
  - Performance
  - Rendering the first page quickly

Angular Universal

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`



Angular Universal - Set up

- Install:
  - `@angular/platform-server`: Universal server-side components
  - `@nguniversal/express-engine`: express engine for Universal 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';
```

Angular Universal - 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 { }
```

Angular Universal - 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 { }
```

Angular Universal - 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';
```

Angular Universal - 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;
}
```

Angular Universal - 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}`);
  });
}
```

Angular Universal - 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"
  }
}
```

Angular Universal - 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
     }
   }
 }
}
```

Angular Universal - 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"
        }
    }
  }
}
```

Angular Universal - 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 11

Ngrx

Summary

- Reminders
- Tests
- Angular best practices
- Angular architecture
- Internationalization
- Design
- Forms
- Router
- RxJS

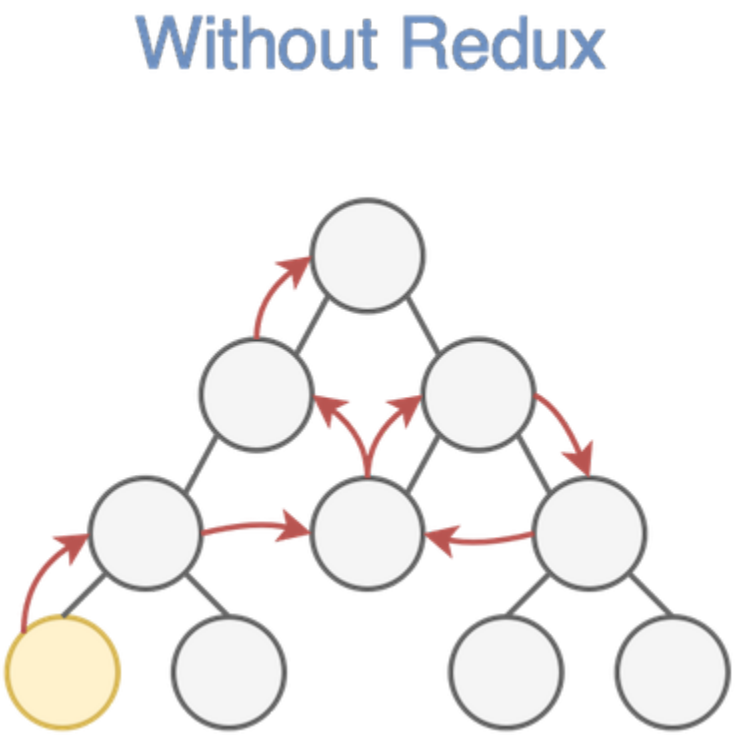
Summary

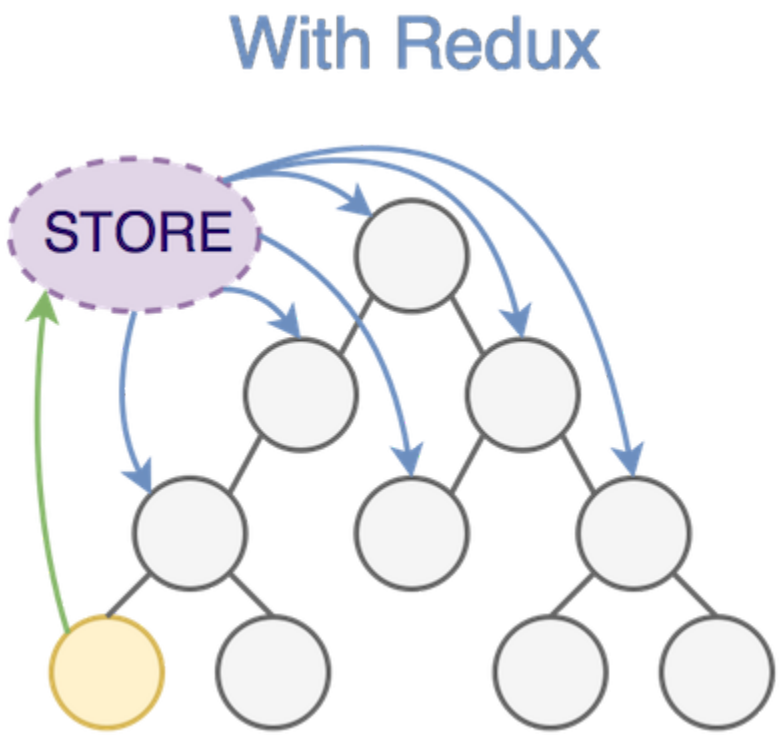
- Industrialization
- Angular Universal
- *Ngrx*
- To go further

Ngrx - Redux

What is Redux?

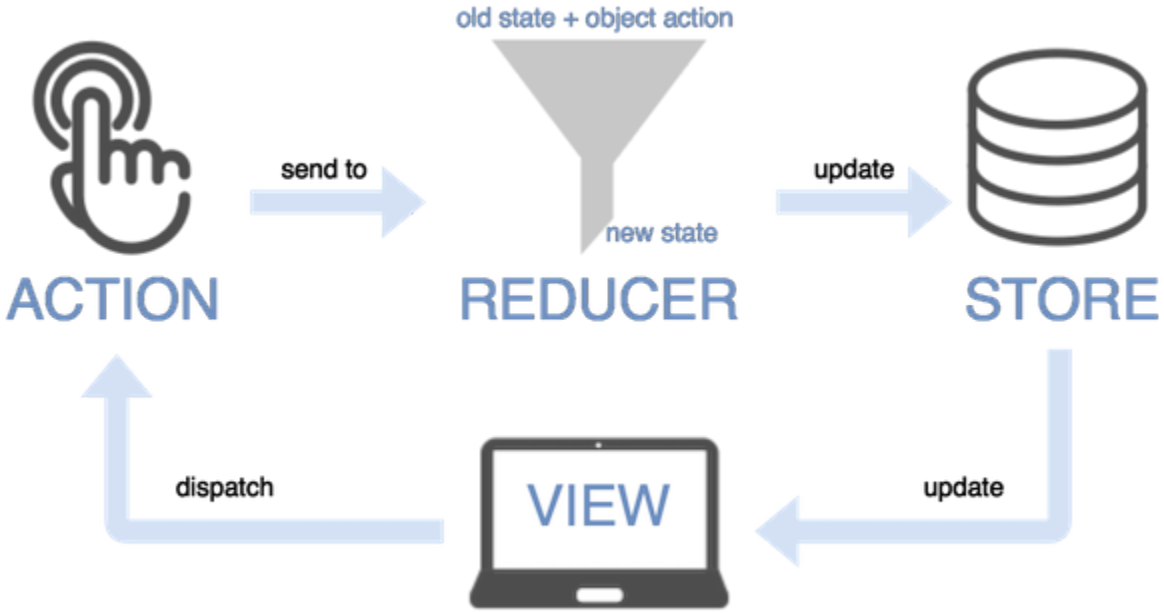
- Alternative to MVC
- Redux is a state container: it **stores** all the states of an app
- One-way data flow: you can only change the state by dispatching an **action**





Ngrx - Redux

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`: reimplementation of redux
  - `angular-redux/store`: Angular bindings for redux, not a new implementation

Ngrx/store

- Install it with:

```shell script ng add @ngrx/store
```

```
<br>

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

```typescript
import {StoreModule} from "@ngrx/store";
import {counterReducer} from "./products.reducer";

@NgModule({ /* ... */
  imports: [
    StoreModule.provideStore({ products: productsReducer })
  ],
})
export class AppModule { }
```
```

Ngrx/store

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

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

export const addProduct = createAction('[Products] Add');
```

- Create a file for reducer (`products.reducer.ts`). This file will in charge of mutating your store

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

export const initialState = [];

export const counterReducer = createReducer(initialState,
  on(addProduct, state => [...state, product]),
);
```

Ngrx/store

- Access one of the states with `select()` in `src/app/app.component.ts`:

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

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

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

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

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

Ngrx/store

- 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 in the action

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.get<Store>(Store);
    guard = TestBed.get<AuthGuard>(AuthGuard);
  });
});
```

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);
});
```

Lab 12

To go further

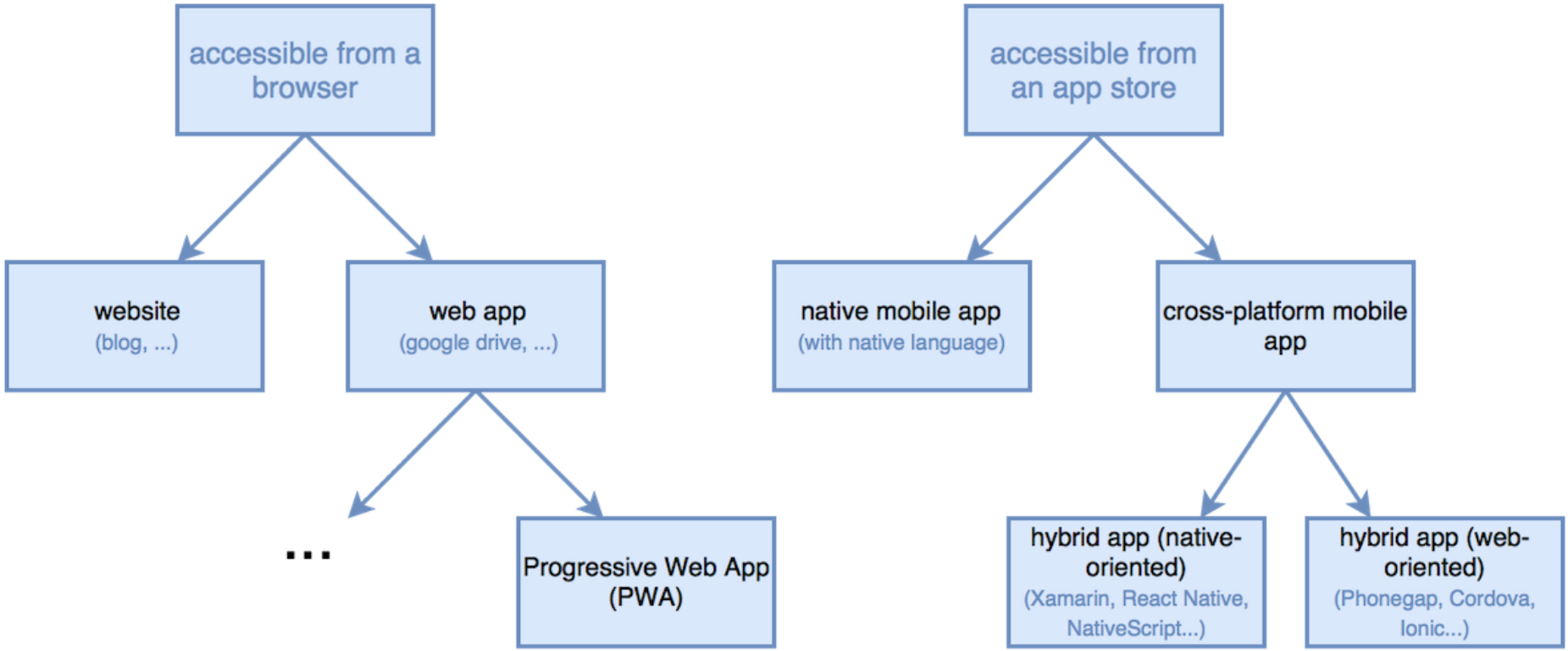
Summary

- Reminders
- Tests
- Angular best practices
- Angular architecture
- Internationalization
- Design
- Forms
- Router
- RxJS

Summary

- Industrialization
- Angular Universal
- 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 allows 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
  - 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 an native app
- **Cross-platform**: iOS and Android

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

cd myApp
ionic serve
```