

# CONCEPTION D'APPLICATIONS WEB AVEC ANGULAR

```
@Component({  
    selector: 'formation',  
    template: '<p>{{ objective }}</p>'  
})  
export class FormationComponent {  
    objective: string = 'Découvrons Angular';  
}
```

# PRÉSENTATION

- Du formateur
  - De **DocDoku**
- 

## À PROPOS DE VOUS

- Expérience avec les frameworks front-end
- Ce que vous attendez de la formation
- Vos projets à venir utilisant Angular

# AGENDA

1. Évolution des standards
2. Évolution du framework  
Angular
3. Les composants
4. Injection de dépendances
5. Le routage
6. Les requêtes HTTP
7. Formulaires et événements
8. Tests unitaires
9. Tests de bout en bout
10. Mise en production

# DÉROULEMENT DES TP

**Objectif** : réalisation d'une application complète

- Multiples composants
- Routage
- Formulaires
- Requêtes HTTP (opérations CRUD)
- Intégration de composants tiers
- Tests

# MODALITÉS

- Horaires : 9h -> 17h30
- Pauses : 15 minutes matin et après midi
- Déjeuner : 1h30

# 1 - ÉVOLUTION DES STANDARDS

- JavaScript : la base
- ECMAScript : où en est-on ?
- ECMAScript 6 : les grandes lignes
- Les modules natifs
- Les WebComponents

# 1.1 JAVASCRIPT

- Généralités
- Variables et types
- Opérateurs
- Structures de contrôle
- Fonctions
- Collections
- Programmation Objet
- JSON

# GÉNÉRALITÉS

- Scripting, multi-plateformes, orienté objet
- Interprété, dynamique
- Spécification ECMAScript
- Implémentations
  - SpiderMonkey (Firefox)
  - V8 (Chrome, Opera)
  - Chakra (Edge)
  - Nitro (WebKit, Safari)

# VARIABLES ET TYPES

- Déclaration /  
Évaluation

```
var a = 42;                      // variable globale
b = 42;                          // variable globale
var c = null, d = [], e, x = "foo" ; // plusieurs variables

var f;
console.log(f); // undefined
console.log(g); // Error: ReferenceError

var k;
console.log(k + 2); // NaN

var n = null;
console.log(n * 32); // 0
```

# VARIABLES ET TYPES

- Portée des variables

```
var fn = function() {  
  if (true) {  
    var x = 42;  
  }  
  console.log("x = " + x);  
}  
  
fn(); // "x = 42"  
  
console.log(x); // ReferenceError
```

# VARIABLES ET TYPES

- Portée des variables  
("Hoisting")

```
(function() {  
    var x = 5;  
  
    console.log("x is " + x + " and y is " + y);  
  
    var y = 7;  
}());  
  
// "x is 5 and y in undefined"
```

# VARIABLES ET TYPES

## ■ Types primitifs

- Boolean
- Null
- Undefined
- Number
- String
- Symbol

<https://www.keithcirkel.co.uk/metaprogramming-in-es6-symbols/>

## ■ Autres types

- Object
- Function

# VARIABLES ET TYPES

## ■ Nombres

```
var a = 42; // décimaux et entiers
var b = 0755; // octaux
var c = 0b1010; // binaires
var d = 0xAF; // hexadécimaux
```

```
Number.MAX_VALUE;
Number.MIN_VALUE;
Number.POSITIVE_INFINITY;
Number.NEGATIVE_INFINITY;
Number.NaN;

Number.parseFloat(str);
Number.parseInt(str);
Number.isFinite(nbr);
Number.isInteger(nbr);
Number.isNaN(nbr);
```

# VARIABLES ET TYPES

## ■ Boolean

```
Boolean( false );           // false
Boolean( 0 );               // false
Boolean( 0.0 );             // false
Boolean( "" );              // false
Boolean( null );            // false
Boolean( undefined );       // false
Boolean( NaN );             // false
Boolean( "false" );          // true
Boolean( "0" );              // true
```

# VARIABLES ET TYPES

## ■ L'API

### Math

```
Math.PI
```

```
Math.abs(nbr);
Math.min(nbr [, nbr]);
Math.max(nbr [, nbr]);
Math.random();
```

# VARIABLES ET TYPES

## ■ L'API

### Date

```
var now = new Date();
var d = new Date("December 25, 2015 13:30:00");
var d = new Date(2015, 11, 25, 13, 30, 0, 0);
var d = new Date(2015, 11, 25);

d.getFullYear();
d.getHours();
d.getDate();
// ...
d.setMonth(nbr);
d.setTime(nbr);
```

# VARIABLES ET TYPES

- *String*: "wrapper\*" autour du primitif

```
var foo = "foo";
var bar = new String("bar");

typeof foo           // "string"
typeof bar          // "object"

console.log(foo);    // "foo"
console.log(bar);    // [ "b", "a", "r" ]

foo.length == bar.length // true
```

# VARIABLES ET TYPES

## ■ String

```
concat();
trim();
indexOf();
charAt();
substring();
toLowerCase();
toUpperCase();
split();
```

# VARIABLES ET TYPES

## ■ RegExp

```
var re = /ab+c/;  
var re = new RegExp("ab+c");  
  
// caractères spéciaux  
\ // interpréter ou pas le caractère suivant  
^ // début de séquence  
$ // fin de séquence  
* // expression précédente répétée 0+ fois  
+ // expression précédente répétée 1+ fois  
? // expression précédente présente 0 ou 1 fois  
. // n'importe quel caractère sauf le saut de ligne
```

# VARIABLES ET TYPES

## ■ RegExp

```
/d(b+)d/.exec("cdbbdbbz");           // true

var text = "one two three four";

text.match(/\w+\s/);                  // [ "one " ]
text.match(/\w+\s/g);                // [ "one ", "two ", "three " ]

text.split(/\s+/);                  // [ "one", "two", "three", "four" ]

text.replace(/\s+/, "-");            // "one-two three four"
text.replace(/\s+/g, "-");           // "one-two-three-four"
```

# OPÉRATEURS

## ■ Arithmétiques

```
+    // Addition
-    // Soustraction
*    // Multiplication
/    // Division (retourne une valeur en virgule flottante)
%    // Modulo (retourne le reste de la division entière)

-    // Négation unaire (inverse le signe)
++   // Incrémentation (en forme préfixée ou postfixée)
--   // Décrémentation (en forme préfixée ou postfixée)
```

# OPÉRATEURS

## ■ Binaires

```
&      // And   : et binaire
|      // Or    : ou binaire
^      // Xor   : ou binaire exclusif
~      // Not   : inverse tous les bits

<<     // décalage à gauche avec remplissage à droite avec 0)
>>     // décalage à droite avec conservation du bit de signe
```

# OPÉRATEURS

## ■ Affectation

```
=      // Affectation
+=     // Ajoute et affecte
-=     // Soustrait et affecte
*=     // Multiplie et affecte
/=     // Divise et affecte

&=    // Et binaire puis affectation
|=    // Ou binaire puis affectation
^=    // Ou exclusif binaire puis affectation
<<=   // Shift left puis affectation
>>=   // Shift right puis affectation
```

# OPÉRATEURS

- Affectation par décomposition  
(tableau)

```
var arr = ["foo", "bar", "baz", "qux"];  
  
var [foo, bar] = arr;  
var [foo, bar, ...rest] = arr;  
[foo, bar] = [bar, foo]
```

## Affectation par décomposition (objet)

```
var o = {p: 42, q: true};  
var {p, q} = o;  
var {p: toto, q: truc} = o;
```

# OPÉRATEURS

## ■ Comparaison

```
==      // Égal à
!=      // Différent de
>      // Supérieur à
>=     // Supérieur ou égal à
<      // Inférieur à
<=     // Inférieur ou égal à
===    // Identiques (égaux et du même type)
!==    // Non-identiques
```

# OPÉRATEURS

## ■ Logiques

```
&&      // AND (opérateur logique ET)
||       // OR (opérateur logique OU)
!        // NOT (opérateur logique NON)
```

# OPÉRATEURS

## Évaluation rapide

```
var a1 = true && true;
var a2 = true && false;
var a3 = false && true;
var a4 = false && (3 == 4);
var a5 = "Chat" && "Chien";
var a6 = false && "Chat";
var a7 = "Chat" && false;
```

# OPÉRATEURS

## Évaluation rapide

```
var o1 = true || true;
var o2 = false || true;
var o3 = true || false;
var o4 = false || (3 == 4);
var o5 = "Chat" || "Chien";
var o6 = false || "Chat";
var o7 = "Chat" || false;
```

# OPÉRATEURS

## Évaluation rapide

```
var n1 = !true;  
var n2 = !false;  
var n3 = !"Chat";
```

# STRUCTURES DE CONTRÔLE

- if/else if/else
- try/catch/finally
- switch case
- for/for var  
in/forEach
- while/do while

# STRUCTURES DE CONTRÔLE

- if /  
else

```
if (expression1) {  
    ...  
}  
else if (expression2) {  
}  
else {  
}
```

- Opérateur  
ternaire/conditionnel

```
var resultat = (expression) ?  
    "expression true" : "expression false";
```

# STRUCTURES DE CONTRÔLE

## ■ Switch

```
switch (expression) {  
    case "foo":  
        // ...  
        break;  
    case 22 :  
        // ...  
        break;  
    default:  
        // ...  
        break;  
}
```

# STRUCTURES DE CONTRÔLE

## ■ Boucles

*for*

```
// boucle for
for (initialisation ; condition; instructions) {
    // ...
}

// boucle for .. in
for (var property in object) {
    console.log(object[property]);
}

// boucle for .. of. Attention, pas sur IE!
for (var value of array) {
    console.log(value);
}
```

# STRUCTURES DE CONTRÔLE

## ■ Boucles

*while*

```
// boucle while
while(condition) {
    // ...
}

// boucle do .. while
do {
    // ...
} while(condition);
```

# STRUCTURES DE CONTRÔLE

## ■ Exceptions

```
try {
    // some code
} catch(exception) {
    // Catch the exception
} finally {
    // Toujours exécuté
}
```

## ■ Throw

```
throw "Erreur2"; //type String
throw 42;        //type Number
```

# FONCTIONS

- Mot clé ***function***
- **Nom** de la fonction
- Liste d'**arguments**
- **Instructions**  
JavaScript

```
function cow(say) {  
    console.log("Mooh " + say);  
}
```

# FONCTIONS

## ■ Les paramètres sont passés par valeur

- primitif : la **valeur** est passée
- objet : la **valeur de la référence** est passée

```
function foo(anObject) {  
    anObject.field_1 = "foo";  
  
    anObject = {field_1: "bar"};  
}  
  
var myObject = {field_1: ""};  
console.log(myObject.field_1); // ""  
  
foo(myObject);  
console.log(myObject.field_1); // "foo"
```

# FONCTIONS

- Expression de fonction

```
var square = function(n) { return n * n };
var x = square(4); // x reçoit la valeur 16

// récursivité
var factorial = function fac(n) { return n<2 ? 1 : n * fac(n-1) };
var x = factorial(3); // x reçoit la valeur 6
```

# FONCTIONS

- Fonction d'ordre supérieur (Higher-order function)

```
function map(fn, arr) {  
  var ret = [], i;  
  for (i = 0; i != arr.length; i++) {  
    ret[i] = f(arr[i]);  
  }  
  return ret;  
}  
  
var myArr = [0, 1, 2, 5, 10];  
  
var newArr = map(square, myArr);  
  
console.log(newArr) // [0, 1, 4, 25, 100]
```

# FONCTIONS

- Fermetures  
(Closures)

```
function outer(x) {  
    function inner(y) {  
        return x + y;  
    }  
    return inner;  
}  
  
ret = outer(5)(15); // renvoie 20
```

# FONCTIONS

## ■ Paramètre

"*arguments*"

```
function times() {  
    var newArr=[ ];  
    for(var i=0;i<arguments.length;i++)  
        newArr.push(arguments[i]*2);  
  
    return newArr;  
}  
  
var arr = times(1, 2, 3); // [2, 4, 6]
```

# COLLECTIONS

- Array - ensemble **ordonné** de valeurs

```
var arr = new Array("foo", "bar", "baz", "qux");
var arr = ["foo", "bar"];

arr.length;      // 2
arr[0];          // "foo"
arr[1];          // "bar"
arr[100];        // undefined
arr["length"];   // 2

arr[0] = "baz";
arr[1] = "qux";
[arr[0], arr[1]] = [arr[1], arr[0]]
```

# COLLECTIONS

- Array -  
*length*

```
var arr = [];
arr[0] = "foo";
arr[1] = "bar";
arr.push("baz");
arr.length;      // 3
```

```
var arr = Array(2);
arr.push("foo");
arr.push("bar");
arr[10] = "baz";
arr.length;      // 11
```

```
arr.length = 2;  // [undefined, undefined]
```

# COLLECTIONS

- Parcourir un tableau

```
var arr = ["foo", "bar", , "qux"];  
  
for (var i = 0, len = arr.length; i < len; ++i) {  
    console.log(arr[i]);  
}  
  
var divs = documents.getElementsByTagName('div');  
for (var i = 0, div; div = divs[i]; ++i) {  
    doSomething(div);  
}
```

# COLLECTIONS

- Parcourir un tableau

```
var arr = ["foo", "bar", , "qux"];  
  
arr.forEach(el => console.log(el));  
  
arr[2] = "baz";  
  
arr.forEach(el => console.log(el));
```

# COLLECTIONS

- Méthodes de  
Array

```
var arr = ["foo", "bar"];  
  
arr = arr.concat("baz", "qux"); // ["foo", "bar", "baz", "qux"]  
  
arr.join(" - "); // "foo - bar - baz - qux"  
  
arr.pop(); // ["foo", "bar", "baz"]  
  
arr.push("qux"); // ["foo", "bar", "baz", "qux"]  
  
arr.reverse(); // ["qux", "baz", "bar", "foo"]  
  
arr.sort(); // ["bar", "baz", "foo", "qux"]
```

# COLLECTIONS

```
var arr = ["foo", "bar"];  
  
arr.indexOf("foo");           // 0  
arr.indexOf("foo", 2);        // 2  
  
arr.forEach(console.log);  
  
arr = arr.map(x => x.toUpperCase()); // ["FOO", "BAR"]  
  
arr.filter(x => x.startsWith("B")); // ["BAR"]  
  
arr.every(x => typeof x === "string"); // true  
  
arr.some(x => x.startsWith("B"));      // true  
  
arr.reduce(function(acc, input) {  
  return acc + input.length;  
}, 0); // 6
```

# PROGRAMMATION OBJET

- Ensemble de **propriétés**
- Associe un nom (**clé**) à une **valeur**
- Valeur fonction => **méthode**

# PROGRAMMATION OBJET

- Accés aux propriétés

```
var car = new Object();
car.maker    = "Tesla Motors, Inc";
car.model    = "Model 3";
car["year"]  = 2017;

var maker = "maker";
console.log(car[maker]); // "Tesla Motors, Inc"
console.log(car.model); // "Model 3"
console.log(car.year); // 2017
console.log(car.sales); // undefined
```

# PROGRAMMATION OBJET

## ■ Initialisateur

d'objets

```
var car = {  
  "_id": Math.round(Math.random()*1000),  
  maker: "Tesla Motors, Inc",  
  model: "Model 3",  
  year: 2017,  
  42: "You bet!",  
  engine: {  
    nb: 3,  
    type: "electric"  
  }  
};
```

# PROGRAMMATION OBJET

- **prototype** : objet dont un autre objet **hérite** les propriétés
- Recherche de propriétés dans la *chaîne de prototypes*
- Fin de la chaîne de prototypes : **null**

```
var car = {  
    maker: "Tesla Motors, Inc",  
    model: "Model 3",  
    year: 2017  
};  
  
Object.getPrototypeOf(car); // [object Object]  
Object.getPrototypeOf(Object.getPrototypeOf(car)); // null
```

# PROGRAMMATION OBJET

## ■ Constructeur -

### Fonction

```
function Person(firstname, lastname, age) {  
    this.firstname = firstname;  
    this.lastname = lastname;  
    this.age = age;  
}  
  
var person = new Person("John", "Doe", 30);
```

# PROGRAMMATION OBJET

## ■ Constructeur -

### Fonction

```
function Car(maker, model, year, owner) {  
    this.maker = maker;  
    this.model = model;  
    this.year = year;  
    this.owner = owner;  
}  
  
var model3 = new Car("Tesla Motors, Inc", "Model 3", 2017, person);  
var a4 = new Car("Audi", "A4", 2017, person);  
  
model3.owner.firstname; // "John"
```

# PROGRAMMATION OBJET

## ■ Prototype

```
var model3 = new Car("Tesla Motors, Inc", "Model 3", 2017, person);
var a4 = new Car("Audi", "A4", 2017, person);

model3.describe = function() {
  console.log(this.model + " by " + this.maker);
};

model3.describe(); // "Model 3 by Tesla"
a4.describe();    // TypeError: a4.describe is not a function

Car.prototype.describe = function() {
  console.log(this.model + " by " + this.maker);
};

a4.describe();    // "A4 by Audi"
```

# PROGRAMMATION OBJET

- Opérateur *new*
  - Crée une nouvelle instance et définit son prototype avec celui du constructeur
  - Invoque le constructeur avec le nouvel objet référencé par *this* et le retourne

```
function Car(model) {  
    this.model = model;  
}  
  
var conceptCar = new Car('Twingo');  
  
conceptCar instanceof Car;    // true  
conceptCar instanceof Object; // true
```

# JSON

- **JavaScript Object Notation**
- Format d'échange de données, **sérialisé**
- Syntaxe **basée** sur JavaScript
- Nombres, booléens, chaînes, null, tableaux et objet
- Voir <http://json.org/> pour la spécification

```
JSON.parse(str);          // valeur
JSON.stringify(valeur); // str
```

# 1.2 ECMASCIPT : OÙ EN EST-ON ?

Fini les noms ES5, ES6, ... place aux dates.

- ES5 : Décembre 2009
- ES 2015 (ES6) : Juin 2015
- ES 2016 : Juin 2016
- ES 2017 : Juin 2017
- ES 2018 : Juin 2018
- ESnext : En cours de développement

<https://tc39.github.io/ecma262/>

# EVOLUTION D'ECMASCRIPT

- ES 2016 : nouvelle approche avec une nouvelle spécification par an
- Les navigateurs supportent couramment ES5

## Problème

- Support partiel de ES6 (publié en juin 2015)

<https://caniuse.com/#search=es6>

## Solutions ?

- Polyfills
- Transpileurs

# 1.3 ECMASCIPT 6 : LES GRANDES LIGNES

- Modules
- Classes
- Portée lexicale avec `let` et  
`const`
- Promesses
- Déstructuration
- Fonctions fléchées
- `Map` et `Set`
- Chaînes de caractères "template"
- Opérateurs rest et spread

# PORTEE LEXICALE AVEC LET ET CONST

- Définition de constante (valeur non modifiable)

```
const PI = 3.141593
PI > 3.0
```

- Les variables définies par **var** ont pour portée la fonction
- **let** introduit la portée par bloc

```
let callbacks = []
for (let i = 0; i <= 2; i++) {
    callbacks[i] = function () { return i * 2 }
}
callbacks[0]() === 0
callbacks[1]() === 2
callbacks[2]() === 4
```

# PROMESSES

- Gestion des traitements asynchrones
- Concurrencé aujourd'hui par l'API Observable

```
function msgAfterTimeout (msg, who, timeout) {  
    return new Promise((resolve, reject) => {  
        setTimeout(() => resolve(` ${msg} Hello ${who} !`), timeout)  
    })  
}  
msgAfterTimeout("", "Foo", 100).then((msg) =>  
    msgAfterTimeout(msg, "Bar", 200)  
) .then((msg) => {  
    console.log(`done after 300ms:${msg}`)  
})
```

# PROMESSES

- Pour en finir avec l'enchevêtrement de callback

```
function fetchAsync (url, timeout, onData, onError) {  
    ...  
}  
let fetchPromised = (url, timeout) => {  
    return new Promise((resolve, reject) => {  
        fetchAsync(url, timeout, resolve, reject)  
    })  
}  
Promise.all([  
    fetchPromised("http://backend/foo.txt", 500),  
    fetchPromised("http://backend/bar.txt", 500),  
    fetchPromised("http://backend/baz.txt", 500)  
]).then((data) => {  
    let [ foo, bar, baz ] = data  
    console.log(`success: foo=${foo} bar=${bar} baz=${baz}`)  
}, (err) => {  
    console.log(`error: ${err}`)  
})
```

# DÉSTRUCTURATION

- Affectation par déstructuration
  - de tableau
  - d'objet

```
var list = [ 1, 2, 3 ]
var [ a, , b ] = list
[ b, a ] = [ a, b ]
```

```
var { op, lhs, rhs } = getMyObject();
```

# FONCTIONS FLÉCHÉES

- Syntaxe raccourcie

```
var arr = [  
  "Hydrogen",  
  "Helium",  
  "Unobtainium"  
];  
  
var arr2 = arr.map(function(s){ return s.length });  
  
var arr3 = arr.map( s => s.length );
```

# PARAMÈTRES PAR DÉFAUT

## ■ ES5

```
function times(a,b) {  
  b = typeof b === 'number' ? b : 1;  
  return a*b;  
}
```

## ■ ES6

```
const times = (a, b = 1) => a * b;
```

```
times(); // 5  
times(5, 2); // 10
```

# PORTEE LEXICALE DU THIS

- Référence le **this** du contexte parent

```
this.nums = [1,2,3,4,5, ...]
this.fives = [];

this.nums.forEach((v) => {
  if (v % 5 === 0)
    this.fives.push(v)
}

console.log(this.fives) // 0, 5, 10, ...
```

# MAP ET SET

- Au delà des tableaux, deux nouvelles structures pour gérer les collections

```
var sayings = new Map();
sayings.set("dog", "woof");
sayings.set("cat", "meow");
sayings.set("elephant", "toot");
sayings.size; // 3
sayings.get("fox"); // undefined
sayings.has("bird"); // false
sayings.delete("dog");

for (var [key, value] of sayings) {
  console.log(key + " goes " + value);
}
// "cat goes meow"
// "elephant goes toot"
```

# SET

```
var s = new Set();
s.add(1);
s.add("foo");
s.add("bar");

s.has(1); // true
s.delete("foo");
s.size; // 2

for (let item of monEnsemble) console.log(item);
// 1
// "bar"
```

# CHAÎNES DE CARACTÈRES "TEMPLATE"

- Template  
Literals

```
var server = { name: "NodeJS" }
var config = { host: '192.168.1.33', port: 1337 }
var message = `Starting ${server.name},
running on host ${config.host}:${config.port}`
```

# OPÉRATEURS REST ET SPREAD

- Généralisation de la variable **arguments**
- Reste des paramètres et décomposition

```
function f (x, y, ...a) {  
    return (x + y) * a.length  
}  
f(1, 2, "hello", true, 7) === 9
```

```
var params = [ "hello", true, 7 ]  
var other = [ 1, 2, ...params ] // [ 1, 2, "hello", true, 7 ]  
  
function f (x, y, ...a) {  
    return (x + y) * a.length  
}  
f(1, 2, ...params) === 9  
  
var str = "foo"  
var chars = [ ...str ] // [ "f", "o", "o" ]
```

# 1.4 LES MODULES NATIFS

## ■ Exports nommés

:

```
//---- lib.js ----
export const sqrt = Math.sqrt;
export function square(x) {
  return x * x;
}
export function diag(x, y) {
  return sqrt(square(x) + square(y));
}

//---- main.js ----
import { square, diag } from './lib';
console.log(square(11)); // 121
console.log(diag(4, 3)); // 5
```

# LES EXPORTS

- Export via un alias

:

```
//---- main.js ----
import * as lib from 'lib';
console.log(lib.square(11)); // 121
console.log(lib.diag(4, 3)); // 5
```

# LES EXPORTS

## ■ Export par défaut

:

```
//---- myFunc.js ----  
export default function () { ... };  
  
//---- main1.js ----  
import myFunc from 'myFunc';  
myFunc();
```

```
//---- MyClass.js ----  
export default class { ... };  
  
//---- main2.js ----  
import MyClass from 'MyClass';  
let inst = new MyClass();
```

# LES CLASSES

- Création d'une classe

```
class Shape {  
    constructor (id, x, y) {  
        this.id = id  
        this.move(x, y)  
    }  
    move (x, y) {  
        this.x = x  
        this.y = y  
    }  
}
```

# LES CLASSES

- Avec de l'héritage

```
class Rectangle extends Shape {  
    constructor (id, x, y, width, height) {  
        super(id, x, y)  
        this.width = width  
        this.height = height  
    }  
}  
class Circle extends Shape {  
    constructor (id, x, y, radius) {  
        super(id, x, y)  
        this.radius = radius  
    }  
}
```

# LES CLASSES

- Membres statiques

```
class Rectangle extends Shape {  
    ...  
    static defaultRectangle () {  
        return new Rectangle("default", 0, 0, 100, 100)  
    }  
}  
class Circle extends Shape {  
    ...  
    static defaultCircle () {  
        return new Circle("default", 0, 0, 100)  
    }  
}  
var defRectangle = Rectangle.defaultRectangle()  
var defCircle = Circle.defaultCircle()
```

# LES CLASSES

- Définition de propriétés avec getter et setter

```
class Rectangle {  
    constructor (width, height) {  
        this._width = width  
        this._height = height  
    }  
    set width (width) { this._width = width }  
    get width () { return this._width }  
    set height (height) { this._height = height }  
    get height () { return this._height }  
    get area () { return this._width * this._height }  
}  
var r = new Rectangle(50, 20)  
r.area === 1000
```

# 1.5 LES WEB COMPONENTS

- Composants graphiques **réutilisables**
- Extension du langage HTML par le développeur
- Utilisable sans écrire de code JS (juste du HTML)
- **Isolation** du DOM via les Shadow DOM
  - Simplifie les css
  - Moins de craintes d'impacts

# WEB COMPONENTS

- De nouvelles balises

```
<template> <content> <shadow>
```

- Nouvelle pseudo classe

```
:unresolved
```

- Une API pour gérer les composants personnalisés

```
Document.registerElement()  
Document.createElement()
```

# EXEMPLE D'USAGE

- Ajout de sémantique aux templates

```
class MyElement {  
  constructor( ... ) { ... }  
}
```

```
let elements = window.customElements;  
elements.define('my-element', MyElement, { extends: 'div' });
```

- Utilisation

```
let e = document.createElement('my-element');
```

```
<my-element></my-element>
```

# 2 - ÉVOLUTION DU FRAMEWORK ANGULAR

- Rappels sur AngularJS
- Angular
- TypeScript

## 2.1 RAPPELS SUR ANGULARJS

- Publié en 2009 par Google
- ES5 / ES6
- Axé autour de l'augmentation du HTML :
  - par des **composants** (version “maison” des Web Components)
  - par des **directives** qui altèrent des éléments existants
- Data binding
- Modularité

# ANGULARJS : APERÇU

Anciennement ...

```
class HelloWorld {
  $onInit() {
    if (!this.name) { this.name = 'world'; }
  }
}

angular.module('app', [])
.component('helloWorld', {
  bindings: {
    name: '@'
  },
  template: `<p>Hello, {{ $ctrl.name }}!</p>`,
  controller: HelloWorld
})
```

# ANGULARJS : APERÇU

Anciennement ...

```
<body ng-app="app">
<h1>Example app</h1>
<hello-world></hello-world>
<hello-world name="John"></hello-world>
</body>
```

```
<!-- Rendu : -->
<body>
<h1>Example app</h1>
<p>Hello, world!</p>
<p>Hello, John!</p>
</body>
```

## 2.2 ANGULAR

- Evolution d'Angular
  - Octobre 2016 : sortie de la version finale de Angular 2
  - Décembre 2016 : version 4
  - Novembre 2017 : version 5
  - **Mai 2018 : version 6 (6.1.9)**
  - Octobre 2018 : version 7
  - Avril 2019 : version 8
- Adoption du Semantic versioning : Angular v4 et les suivantes deviennent simplement **Angular**

# ANGULAR : SYNTAXE ET ÉCOSYSTÈME

- Microsoft apporte **TypeScript**
  - Langage recommandé pour Angular
- Architecture orientée composants
- Outilage officiel avec notamment **@angular/cli**
- Le framework officiel est découpé en grands modules  
**@angular/\***

# DÉMARRER AVEC ANGULAR (BOOTSTRAP)

Vient sous forme de lignes de commande

- Installer le  
CLI

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

- Créer un  
projet

```
ng new mon-application
cd mon-application
ng serve //serveur embarqué
ng build //pour la distribution
ng help // afficher les commandes disponibles
```

# ANGULAR : APERÇU

## Nouvelles syntaxes

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

@Component({
  selector: 'hello-world',
  template: `<p>Hello, {{ name }}!</p>`
})
export class HelloWorldComponent implements OnInit {
  @Input() name: string;

  ngOnInit() {
    if (!this.name) { this.name = 'world'; }
  }
}
```

# ANGULAR : APERÇU

## Nouvelles syntaxes

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

@Component({
  selector: 'app-root',
  template: `
    <h1>Example</h1>
    <hello-world [name]="userNamed"></hello-world>
  `
})
export class AppComponent {
  userNamed = 'John';
}
```

## 2.3 TYPESCRIPT

### Objectif

- Sur-ensemble des standards ECMAScript
- Aide au développement (complétion automatique, refactoring, ...)

### Grâce à l'apport de nouvelles fonctionnalités

- Typage
- Interfaces
- Generics
- ...

# TYPAGE

- Variables, arguments de fonctions, retour de fonctions...
- Le **typage** qui manque tant à JavaScript

## Les types de base

```
let isDone: boolean = false;
let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;
let color: string = "blue";
color = 'red';
```

# TYPAGE

- Types avancés

```
//Array
let list: number[] = [1, 2, 3];

//Tuple
let x: [string, number];
x = ["hello", 10]; // Transpilation OK
x = [10, "hello"]; // Transpilation Error

//Enum
enum Color {Red, Green, Blue}
let c: Color = Color.Green;
```

# TYPAGE

- Paramètres et retour de fonctions

```
function sum(x:number, y:number):number {
    return x+y;
}

function warnUser(): void {
    alert("This is my warning message");
}
```

# TYPAGE

- Assertion de type (équivalent du cast Java)

```
let someValue: any = "this is a string";
let strLength: number = (<string>someValue).length;

let someValue: any = "this is a string";
let strLength: number = (someValue as string).length;
```

# TYPAGE

- Types implicites

```
let n = 1                      // let n: number = 1

let s = "Hello World"          // let s: string = "Hello World"

n = s;                         // COMPILATION ERROR
s = n;                         // COMPILATION ERROR

function f() {                  // function f(): string {
    return "hello" ;
}
```

Lorsqu'il s'agit de types basiques, le compilateur associe par défaut les types des variables non annotées.

# INTERFACE

- Sur la base des nouveautés d'ECMAScript 2015,  
TypeScript ajoute la notion d'**interface**

```
interface ServerConfig {  
    host: string;  
    port: number;  
  
    start():void;  
}  
  
class ChatServer implements ServerConfig{  
    // ...  
}
```

# INTERFACE

- Définition inline d'interface

```
function printLabel(labelledObj: { label: string }) {  
    console.log(labelledObj.label);  
}  
  
let myObj = {size: 10, label: "Size 10 Object"};  
printLabel(myObj);
```

# INTERFACE

- Définition de fonctions

```
interface SearchFunc {  
    (source: string, subString: string): boolean;  
}
```

- Définition de constructeurs

```
interface ClockConstructor {  
    new (hour: number, minute: number);  
}  
  
class Clock implements ClockConstructor {  
    currentTime: Date;  
    constructor(h: number, m: number) { }  
}
```

# GENERICs

- Une classe peut définir des **types paramétrés**

```
class GenericNumber<T> {  
    zeroValue: T;  
    add: (x: T, y: T) => T;  
}
```

# GENERICs

- Mais également au niveau des paramètres de méthodes

```
function randomElem<T>(theArray: T[]): T {
  let randomIndex = Math.floor(Math.random()*theArray.length);
  return theArray[randomIndex];
}
```

- Ceux-ci peuvent être inférés ou spécifiés à l'appel

```
function getAsync<T>(url: string): Promise<T[]> {
  return fetch(url)
    .then((response: Response) => response.json());
}

getAsync<Movie>("/movies")
  .then(movies => {
    movies.forEach(movie => {
      console.log(movie.title);
    });
  });

```

} );

# RÉSUMÉ

## Comme en Java

- Une classe peut être abstraite
- Des méthodes peuvent être abstraites
- Une classe peut implémenter plusieurs interfaces
- Une interface peut étendre une autre interface

## Et en plus

- Une interface peut étendre une classe  
!

# 3 - LES COMPOSANTS

- Principes généraux
- Templates
- Input/Output
- Style
- Détection des changements
- Cycle de vie

# 3.1 PRINCIPES GÉNÉRAUX

- Annotation  
`@Component`
- Classe "contrôleur"
- Template
- Inputs et outputs
- CSS isolé

# COMPOSANT DE BASE

- Création d'une classe

```
@Component({  
  selector: 'my-component',  
  templateUrl: './my.component.html',  
  styleUrls: ['./my.component.css']  
})  
export class MyComponent implements OnInit {  
  ngOnInit() {  
    // ...  
  }  
}
```

- Utilisation

```
<my-component></my-component>
```

# MEMBRES DE CLASSE ET MÉTHODES

```
@Component( ... )
export class MyComponent implements OnInit {

    myVar: string = 'world'; // publique (par défaut en Typescript)

    constructor(private someService: MyService){}

    ngOnInit() {
        this.doSomething(this.myVar);
        this.someService.sayHello();
    }

    doSomething(s: string): void{
        // ...
    }
}
```

```
<p>Hello {{ myVar }}</p>
<button (click)="someService.sayHello()">Click me !</button>
```

## 3.2 TEMPLATE : NOUVELLES SYNTAXES

- Garde la philosophie d'angular 1.x
- Nouvelles syntaxes
- Nouvelles directives
- Suppressions de quelques directives

# TEMPLATES

- Binding dans une chaîne

:

```
<p>{{ maVariable }}</p>
```

- Input de composant, valeur "en dur"

:

```
<mon-comp value="test"></mon-comp>
```

- Input de composant, valeur d'une variable

:

```
<mon-comp [value]="maVariable"></mon-comp>
```

- Output de composant

:

```
<mon-comp (selected)="onSelection($event)"></mon-comp>
```

# TEMPLATES

## ■ Boucle

:

```
<li *ngFor="let item of items">{{ item.name }}</li>
```

## ■ Condition

:

```
<button *ngIf="isAdmin; else not-admin">Secret button</button>
<ng-template #not-admin>content here...</ng-template>
```

# TEMPLATES

## ■ Switch

:

```
<div [ngSwitch]="currentUser.group">
  <user-modo  *ngSwitchCase="'modo'"
    [user]="currentUser"
  ></user-modo>
  <user-admin *ngSwitchCase="'admin'"
    [user]="currentUser"
  ></user-admin>
  <user-normal *ngSwitchDefault
    [user]="currentUser"
  ></user-normal>
</div>
```

# TEMPLATES

- Variable locale au template

:

```
<input type="text" #localtext> <p>{{ localtext.value }}</p>
```

- Two-ways binding

:

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

- Transclusion

:

```
<!-- my-component.html -->
<ng-content select="my-slot"></ng-content>
```

```
<my-component>
  <div my-slot>...</div>
</my-component>
```

# TEMPLATES

- Binding d'attribut

:

```
<tr><td [attr.colspan]="spanValue">One-Two</td></tr>
```

- Binding de style

:

```
<p [style.borderColor]="isError ? 'red' : 'green'">Content</p>
```

- Binding de classe

:

```
<p [class.error]="isError">Content</p>
```

## 3.3 INPUT / OUTPUT

Un composant est une **boite noire** : il communique via ses inputs et ses outputs :

- Input : pour passer des données au composant
- Output : pour réagir à un changement d'état du composant

# INPUT

- Gestion des attributs du composant

```
class TaskComponent {  
    @Input() public done: boolean;  
    @Input('taskName') public name: string;  
}
```

- Passage de paramètres

```
<task [taskName]="item.name" [done]="item.done"></task>
```

# OUTPUT

- Notifier des changements

```
class TaskComponent {  
  @Output('complete') onTaskComplete = new EventEmitter<boolean>();  
  
  completeTask(isComplete: boolean) {  
  
    this.onTaskComplete.emit(isComplete);  
  }  
}
```

- Appel interne

```
<task (complete)="onTaskCompleteInComponent($event)"></task>
```

## 3.4 STYLES DU COMPOSANT

- Styles isolés par défaut
  - le style d'un composant ne peut altérer rendu d'un autre composant
- Utilisation de l'annotation `@ViewEncapsulation` pour altérer ce comportement
- Utilisation de `:host` et `:host-context(context)` dans la feuille de style

# VIEWENCAPSULATION

- Trois types d'encapsulations
  - None : pas d'encapsulation
  - Emulated : Encapsulation du CSS par Angular
  - Native : Shadow DOM + scope CSS

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

@Component({
  ...
  encapsulation: ViewEncapsulation.None
})
class MyComponent {}
```

# :HOST

- Sélecteur CSS portant sur l'élément custom dans son shadow DOM : [MDN :host](#)
- Angular réinterprète ce sélecteur

my-super-button.html

```
<div>
  <button>...</button>
</div>
```

my-super-button.css

```
:host > div {
  padding: 0 0.5rem
}
```

# :HOST-CONTEXT

- Surcharge de style lors d'un usage ciblé

```
<my-component>
  <my-super-button></my-super-button>
</my-component>
```

```
<other-component>
  <my-super-button></my-super-button>
</other-component>
```

```
// my-super-button.scss
:host-context(my-component) button { color: red }
:host-context(other-component) button { color: blue }
```

# DÉTECTION DES CHANGEMENTS

- Comment la magie du binding opère-t-elle ?
- Angular rafraîchit le DOM dès que les données "bindées" sont susceptibles d'avoir changées
- Plus précisément suite à :
  - Evènements utilisateur (click, change, input...)
  - Requêtes XHR, fetch
  - Timer (setTimeout, setInterval)

# FONCTIONNEMENT INTERNE

- Angular exploite la librairie zone.js (intégrée au framework)
- Les zones permettent de créer un contexte d'exécution des fonctions asynchrones
  - Hooks = fonctions s'exécutant avant, après ces callbacks
  - `NgZone.runOutsideAngular` (exécute une fonction en dehors des hooks)
  - Instance de `NgZone` obtenue par injection

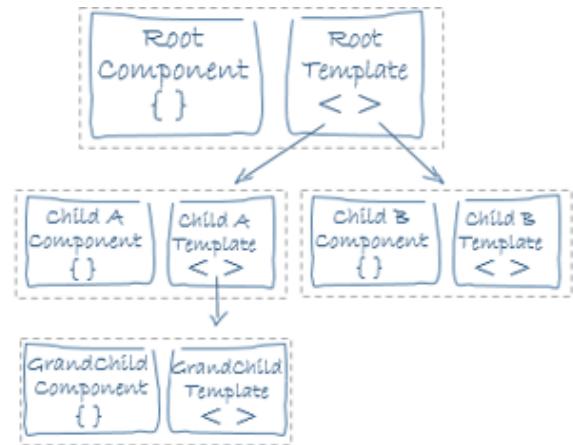
# CONFIGURER LA DÉTECTION DES CHANGEMENTS

- Configuration propre au composant
- Attribut `changeDetection` sur le décorateur `@Component`
  - `ChangeDetectionStrategy.Default`
    - Le composant est redessiné dès qu'un évènement survient
  - `ChangeDetectionStrategy.OnPush`
    - Le composant est redessiné dès qu'un de ses inputs change

# METTRE EN PLACE UNE STRATÉGIE ONPUSH

- Si les références restent les mêmes le composant ne sera pas rafraîchi
- Les changements internes des objets input ne sont pas détectés
- Les inputs doivent donc être immuables
- ImmutableJS peut aider à appliquer cette stratégie

# L'ARBRE DE COMPOSANTS

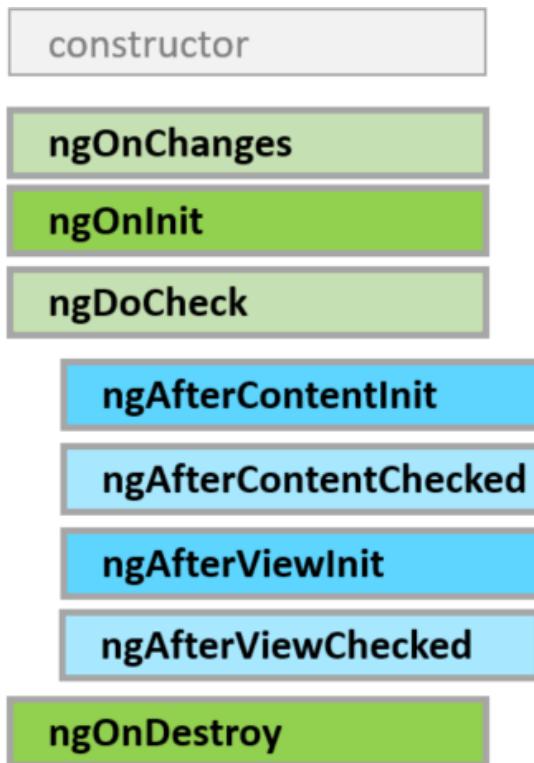


- Un composant non rafraîchi = tous ceux en dessous ne le sont pas non plus

# FORCER LE RAFRAÎCHISSEMENT D'UN COMPOSANT

- `ApplicationRef.tick()`
  - L'arbre complet des composants est rafraîchi
- `NgZone.run(callback)`
- `ChangeDetectorRef.detectChanges()`
  - Seul le composant et ses enfants sont rafraîchis
- Les références vers les objets `ApplicationRef`, `NgZone`, `ChangeDetectorRef` sont obtenues par injection

# 3.5 CYCLE DE VIE DES COMPOSANTS



- Interfaces à implémenter pour connecter les *lifecycle hooks*.
- Réaction aux changements d'inputs
- Nettoyage avec destruction
- Initialisation

# NgOnInit

- Appelé une fois à l'initialisation du composant, quand les inputs et outputs ont été liés à **this**.

```
ngOnInit() {  
    // La vue est initialisée  
    console.log('Hello from component...');  
    this.doSomething();  
}
```

# NgOnChanges

- Appelé quand les inputs du composant changent.
- En paramètre : un objet **SimpleChanges** qui contient les anciennes et nouvelles valeurs des inputs qui ont changé.
- Appelé une fois avant **ngOnInit**, puis à chaque changement.

```
ngOnChanges(changedInputs: SimpleChanges) {  
    for (let propName in changedInputs) {  
        let input = changedInputs[propName];  
        console.log(input.currentValue);  
        console.log(input.previousValue);  
    }  
}
```

# NgDoCheck

- Appelé à chaque détection de changement.
- Permet de détecter manuellement des changements qui ne seraient pas gérés par Angular.

```
ngDoCheck() {  
    if (this.user.name !== this.oldUserName) {  
        this.changeDetected = true;  
        this.doSomething();  
    }  
}
```

# NgAfterContentInit

- Appelé quand le contenu externe (transclusion) a été injecté dans le composant.
- Appelé une fois, après le premier appel à **ngDoCheck**.

```
ngAfterContentInit() {  
  // Le contenu est initialisé  
  console.log('AfterContentInit');  
  this.doSomething();  
}
```

# NgAfterContentChecked

- Appelé à chaque détection de changement du contenu externe (transclusion).
- Appelé une fois après **ngAfterContentInit**, puis après chaque appel à **ngDoCheck**.

```
ngAfterContentChecked() {  
    // Détection d'un changement...  
    if (this.oldUserName === this.user.name) {  
        console.log('AfterContentChecked (no change)');  
    } else {  
        this.oldUserName = this.user.name;  
        console.log('AfterContentChecked changes detected');  
        this.doSomething();  
    }  
}
```

# NgAfterViewInit

- Appelé après l'initialisation de la vue du composant et de celles de ses enfants.
- C'est le bon moment pour analyser le DOM et appeler des méthodes sur les éléments du DOM du composant.

```
ngAfterViewInit() {  
  console.log('AfterViewInit');  
  this.doSomething();  
}
```

# NgAfterViewChecked

- Appelé après chaque détection de changements sur la vue du composant et de celles de ses enfants.

```
ngAfterViewChecked() {  
  console.log('AfterViewChecked');  
  this.doSomething();  
}
```

# NgOnDestroy

- Appelé avant que le composant soit détruit.
- C'est le bon moment pour faire du nettoyage, comme par exemple se désabonner d'émetteurs d'évènements.

```
ngOnDestroy() {  
  console.log('OnDestroy');  
  this.cleanUp();  
}
```

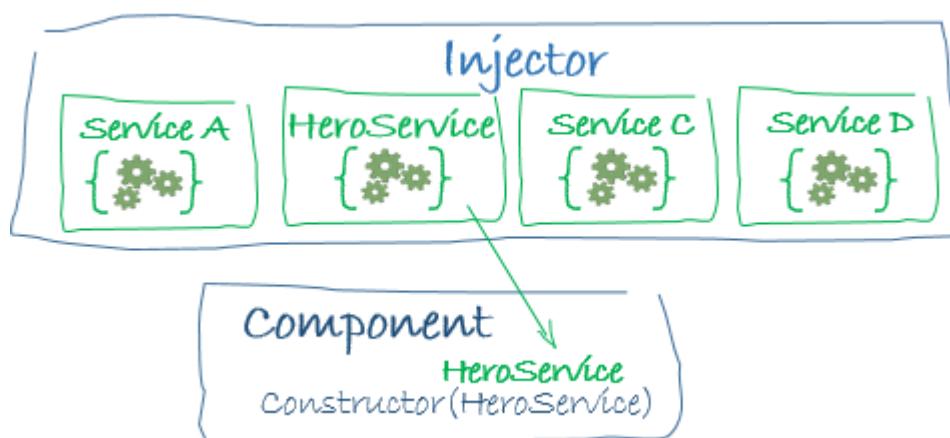
# 4 - INJECTION DE DÉPENDANCE

- Principes du mécanisme d'injection
- Annotations et décorateurs
- Configuration de l'injecteur

# 4.1 PRINCIPES DU MÉCANISME D'INJECTION

## Objectif

Déclarer ce dont on a besoin plutôt que de le construire soi-même.



# INJECTION DE DÉPENDANCES

- Valeur injectée
  - **Singleton**
  - Nouvelle instance via une factory maison
- Surcharge de services existants
- Partage de données via des services

Besoin de MyService ?

```
@Component( ... )
class MyComponent {
  constructor(private service: MyService) {
    console.log(this.service)
  }
}
```

# HIÉRARCHIE D'INJECTEURS

- Angular a plusieurs injecteurs
- Structure **arborescente** qui suit l'arborescence des components
- Possibilité de configurer les différents points d'injection
  - Pour créer des services seulement si besoin
  - Pour gérer plusieurs instances niveau composant

## 4.2 ANNOTATIONS ET DÉCORATEURS

- Création d'un service injectable

```
// Dépendance injectable
@Injectable()
export class CarsService { ... }

// Déclaration auprès de l'injecteur, au niveau du module
@NgModule({
  ...
  providers: [
    CarsService,
    ...
  ],
  ...
})
export class AppModule { }
```

# ANNOTATION INJECTABLE

- "Provided in" 'root'
  - Injection dans le module racine
  - Même instance tout au long du cycle de vie de l'application

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

@Injectable({
  providedIn: 'root',
})
export class MyService {
  constructor() { }
}
```

# ANNOTATION INJECTABLE

- "Provided in" module
  - Injection dans un module donné
  - Même instance tout au long du cycle de vie du module
  - Permet de ne pas instancier un service dès le chargement de l'application

```
import { Injectable } from '@angular/core';
import { SomeModule } from './some.module';

@Injectable({
  providedIn: SomeModule,
})
export class MyService {
  constructor() { }
}
```

# ANNOTATION INJECTABLE

- Injection au niveau component
  - Limite le scope du service
  - Plusieurs instances de services

```
import { Component } from '@angular/core';
import { MyService } from './my.service';

@Component({
  providers: [ MyService ],
  ...
})
export class MyComponent { }
```

# 4.3 CONFIGURATION DE L'INJECTEUR

- Au niveau du module

```
providers: [ CarsService ]  
  
// Raccourci pour :  
providers: [ {provide: CarsService, useClass: CarsService} ]
```

- Providers alternatifs : surcharge d'un provider existant, ou remplacement complet

```
providers: [  
  {provide: Http, useClass: CustomHttpService}  
]
```

# INJECTOR ET INJECTIONTOKEN

- Définir une classe injectable à l'exécution

```
class MyService {  
  constructor(private someDep: MyDep) { ... }  
}  
  
const MY_SERVICE_TOKEN = new InjectionToken<MyService>('...', {  
  providedIn: 'root',  
  factory: () => new MyService(inject(MyDep)),  
});  
  
const myService = injector.get(MY_SERVICE_TOKEN);
```

# SURCHARGE

- Service

```
@Injectable()
export class BetterLogger extends Logger {
  constructor() { super(); }
  log(message: string) {
    // ...
  }
}
```

- Dans le module

```
providers [
  ...
  [{ provide: Logger, useClass: BetterLogger }]
]
```

# 5 - LE ROUTAGE

- Généralités
- Caractéristiques du routeur
- Déclarer ses routes
- Gestion des paramètres
- Directives pour les templates
- Résolution de données
- *Outlets nommés*
- *Guards*
- *Lazy loading*

# 5.1 GÉNÉRALITÉS

Navigation dans Single Page App (SPA) :

- clic sur lien → modifier partie page plutôt que charger
- nécessité de gérer l'historique pour permettre
  - d'aller à la page précédente ou suivante
  - de recharger la page et conserver l'état de la page

Routage = mécanisme pour synchroniser URL et affichage

# ROUTAGE AVEC FRAGMENT DE L'URL

```
<a href="#about">About</a>
```

```
window.onhashchange = function() {  
    if (window.location.hash === '#about')  
        ...  
}
```

⚠ une seule page du point de vue Google Bot

# ROUTAGE AVEC history.pushState

```
<a href="/about"
    onclick="goto('about')"
>About
```

```
function goto(state) {
  if (state === 'about') ... // afficher la
    // changer l'URL
  window.history.pushState(..., '/about');
  // Empêcher le navigateur de suivre le li
  return false;
}
```

⚠ serveur doit gérer **/about** (sinon rafraîchir → 404)

- soit en faisant rendu Angular avec Node.js  
    ⇒ application indexable par moteur recherche
- soit en redirigeant vers **index.html**  
    ⇒ indexable mais mauvaise gestion 404

## 5.2 CARACTÉRISTIQUES DU ROUTEUR

- Le routeur insère des composants dans des **outlets**
- Un outlet **principal** (obligatoire) + des outlets nommés
- Routes imbriquées
- *Guards* pour protéger une route ou ses enfants
- *Lazy loading* de modules de l'application

# LE ROUTER D'ANGULAR

- Balise `<base href>`
  - Le premier élément de la balise `<head>`
  - Permet de configurer le router

```
<head>
  <base href="/" />
  ...
</head>
```

# LE ROUTER D'ANGULAR

- Module de routage

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

const routes: Routes = [
  ...
];

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

# LE ROUTER D'ANGULAR

- Composant RouterOutlet
  - Obligatoire pour le routage

```
<router-outlet></router-outlet>
```

- Directive RouterLink
  - Génère les URLS pour le routage

```
<a routerLink="/users">User list</a>
```

- Directive RouterLinkActive
  - Permet d'associer une classe CSS

```
<a routerLink="/users" routerLinkActive="my-class">User list</a>
```

# 5.3 DÉCLARER SES ROUTES

- Routes statiques/dynamiques
- Passages de paramètres
- Redirections

```
const routes: Routes = [  
  { path: 'users', component: UserListComponent },  
  { path: 'user/:id', component: UserDetailComponent },  
  {  
    path: 'other',  
    component: OtherComponent,  
    data: { title: 'Some title' }  
  },  
  { path: '',  
    redirectTo: '/home',  
    pathMatch: 'full'  
  },  
  { path: '**', component: PageNotFoundComponent }  
];
```

# INTERFACE ROUTE

- Définition de l'interface

```
interface Route {  
  path : string  
  component : Type<any>  
  pathMatch : string  
  redirectTo : string  
  outlet : string  
  canActivate : any[]  
  data : Data  
  resolve : ResolveData  
  children : Routes  
  ...  
}
```

# ROUTER OUTLET

- Un outlet principal
- Des outlets nommés optionnels
- Permet de séparer certains composants

```
<router-outlet></router-outlet>
<router-outlet name='left'></router-outlet>
<router-outlet name='right'></router-outlet>
```

```
{
  path: 'users', component: UsersComponent, children: [
    {path: '', component: LeftComponent, outlet: 'left'},
    {path: '', component: RightComponent, outlet: 'right'}
}
```

# ROUTES STATIQUES ET DYNAMIQUES

## ■ Exemples

:

```
{ path: 'hello', component: HelloWorldComponent }
{ path: 'user/:id', component: UserComponent }
```

## ■ Navigation programmatique

```
this.router.navigate(['/hello']);
this.router.navigate(['/user', 42]);
```

## 5.4 GESTION DES PARAMÈTRES

- Récupération de la valeur du paramètre dans l'URL

```
// Route
{ path: 'user/:id', component: UserComponent }

// Dans le composant :
constructor(private route: ActivatedRoute) { }

ngOnInit() {
  this.route.params
    .map(params => params.id)
    .subscribe(id => this.loadUser(id))
}
```

# 5.5 DIRECTIVES POUR LES TEMPLATES

- Cration de liens dans les templates

```
<a [routerLink]=["/users"]>Users list</a>
<a [routerLink]=[ '/user', user.id]>{{ user.name }}</a>

<nav>
  <a routerLink="/crisis-center" routerLinkActive="active">
    Crisis Center
  </a>
  <a routerLink="/heroes" routerLinkActive="active">Heroes</a>
</nav>
```

# 5.6 RÉSOLUTION DE DONNÉES

- Création d'un resolver

```
@Injectable()
export class UserResolver implements Resolve<User> {
  constructor(private us: UsersService, private router: Router) {}

  resolve(route: ActivatedRouteSnapshot): Promise<User> {
    const id = route.params['id'];

    return this.us.getUser(id);
  }
}
```

# RÉSOLUTION DE DONNÉES

## ■ Route

:

```
{  
  path: 'user/:id',  
  component: UserComponent,  
  resolve: { user: UserResolver }  
}
```

## ■ Composant

:

```
ngOnInit() {  
  this.route.data  
    .subscribe((data: { user: User }) => {  
      this.user = data.user  
    });  
}
```

# 5.7 OUTLETS NOMMÉS

## ■ Template

:

```
<router-outlet></router-outlet>
<router-outlet name="sidepanel"></router-outlet>

<a [routerLink]="[{ outlets: { sidepanel: ['details'] } }]">
  Show details
</a>
```

## ■ Route

:

```
{
  path: 'details',
  component: DetailsComponent,
  outlet: 'sidepanel'
}

// URL : http://.../main-route(sidepanel:details)
```

# 5.7 ROUTE GUARDS

- **Hooks** de changements de routes
- Accepter ou refuser la transition
- Exemples d'utilisation

CanActivate : vérifier l'accès à la route

CanActivateChild : vérifier l'accès à une route enfant

CanDeactivate : demander confirmation pour laisser du travail en cours non sauvé

CanLoad : vérifier l'état de ressources

# ROUTE GUARDS

- Exemple d'autorisation pour un admin

```
// Route
{
  path: 'admin',
  component: AdminPageComponent,
  canActivate: [IsAdmin]
}
// Validateur
@Injectable()
export class IsAdmin implements CanActivate {
  constructor(private authService: AuthService) { }

  canActivate(): boolean {
    return this.authService.currentUser.isAdmin;
  }
}
```

# 5.8 LAZY LOADING

## Objectifs

- Ne charger que le nécessaire lors du chargement de la page

## Coté Angular

- Découpage des modules en "chunks" par Angular
- Plus rapide au démarrage
- Possibilité de charger toutes les dépendances de façon asynchrone à l'exécution
- Configuration de mode de preload pour chaque module

# LAZY LOADING

- Chargement "on-demand"

```
RouterModule.forRoot([
  { path: 'home', component: HomeComponent },
  { path: '', pathMatch: 'full', redirectTo: '/home' },
  { path: 'admin',
    loadChildren: 'app/admin/admin.module#AdminPageModule'
  },
])
```

- AdminPageModule  
(children)

```
RouterModule.forChild([
  { path: '', component: AdminPageComponent }
])
```

# 6 - LES REQUÊTES HTTP

- Promesses
- Observables
- Le service `Http`
- Authentification

# 6.1 PROMESSES

*Contrat sur la résolution future d'un action asynchrone.*

- Objet encapsulant une opération asynchrone
- Peut être écrit dans un style synchrone
- Débarrasse du "callback-hell"
- Peut être enchainée avec une autre promesse
- Facilité de lancements d'opérations séquentiellement ou parallèlement

```
doThis()
  .then(doThat, logIfError)
  .then(doThatAgain)
  .then(console.log)
  .catch((e) => console.log('OUCH!', e));
```

# PROMESSES

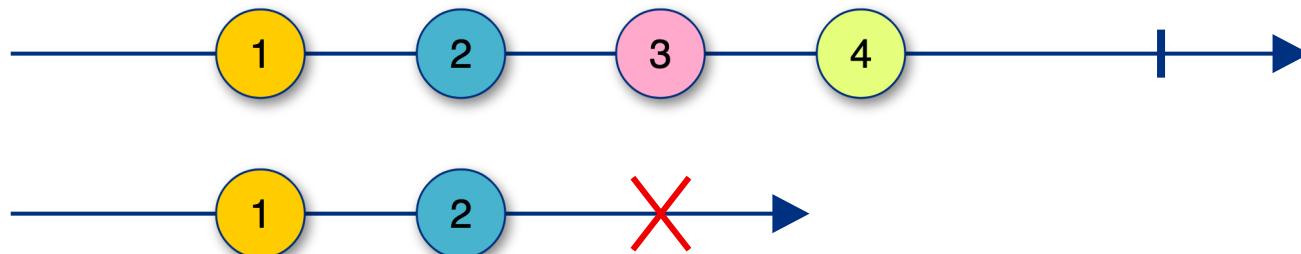
- Chaque promesse utilise les paramètres de retour de la précédente

```
const prom = new Promise((resolve, reject) => {
  doSomethingLong(result => {
    if (result.isError) {
      reject(result.errorMessage);
    } else {
      resolve(result.data);
    }
  })
}

prom
  .then(data => { console.log(data) })
  .catcherrMsg => { console.error(errMsg) })
```

## 6.2 OBSERVABLES

- Flux de données auquel on peut s'abonner
  - Plus complexe que les Promises (plus large)
  - Traite aussi les flux infinis (bus d'évènements)
  - API multi-langages (Java, C#...)



- [reactivex.io/rxjs](http://reactivex.io/rxjs)
- [rxmarbles](http://rxmarbles.com)
- [learnrxjs](http://learnrxjs.com)

# UTILISATION BASIQUE

- Création d'un Observable

```
const obs = Observable.create((o: {}) => {
  doSomethingLong(result => {
    if (result.isError) { o.error(result.errorMessage); }
    else { o.next(result.data); }
    o.complete();
  })
})
```

- Abonnement

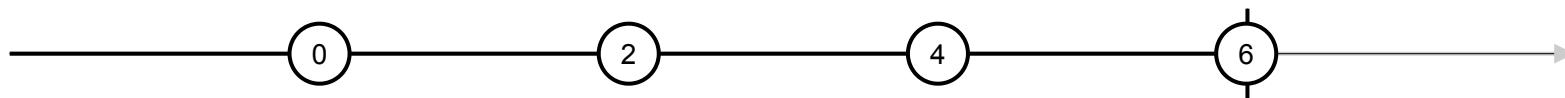
```
obs.subscribe(
  value => console.log(value),
  err => console.error(err),
  () => console.log('Completed')
)
```

# LES "MARBLE DIAGRAMS"

- Diagrammes temporels
- De la gauche vers la droite

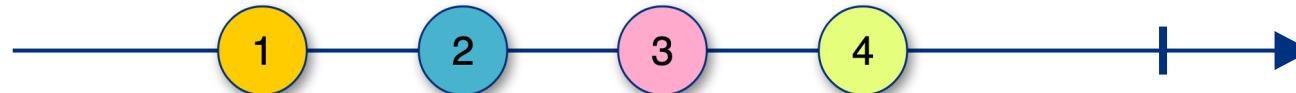
```
const { interval } = Rx;
const { take , map } = RxOperators;

interval(1000).pipe(
  take(4),
  map((x) => x*2)
)
```



# OPÉRATEURS

Les opérateurs permettent de dériver un observable à partir d'un (ou de plusieurs) observable(s) source(s).



map



filter



# 6.3 LE SERVICE HttpClient

- Pour consommer des web services

```
class HttpClient {  
    request(first: string | HttpRequest<any>, url?: string, options: {...}): Observable<any>  
  
    delete(url: string, options: {...}): Observable<any>  
  
    get(url: string, options: {...}): Observable<any>  
  
    head(url: string, options: {...}): Observable<any>  
  
    jsonp<T>(url: string, callbackParam: string): Observable<T>  
  
    options(url: string, options: {...}): Observable<any>  
  
    patch(url: string, body: any | null, options: {...}): Observable<any>  
  
    post(url: string, body: any | null, options: {...}): Observable<any>  
  
    put(url: string, body: any | null, options: {...}): Observable<any>  
}
```

# EXEMPLE

- Requête de la liste des utilisateurs

```
http.get('http://localhost:8000/users')
  .subscribe((users: string) => console.log(users));
```

- Réponse JSON

```
[  
  {  
    "name": "Luke Skywalker",  
    "height": "172",  
    "mass": "77",  
    ...  
  },  
  ...  
]
```

# ENVOI DE DONNÉES

- Gestion des headers

```
const httpOptions = {  
  headers: new HttpHeaders({  
    'Content-Type': 'application/json',  
    'Authorization': 'my-auth-token'  
  })  
};
```

- Crédit du service

```
export class MyService {  
  public createUser(user:User):Observable<User> {  
    return this.http.post<User>('/users', user, httpOptions)  
      .pipe(  
        catchError(this.handleError(user))  
      );  
  }  
}
```

# CLIENT HTTP

- Typage de la réponse

```
interface User {  
    name: string  
    height: string  
    mass: string  
}  
  
interface UserResponse {  
    users: User[]  
}  
  
http.get<UserResponse>('http://localhost:8000/users')
```

# 6.4 AUTHENTIFICATION

- Par cookie en cross-domain

:

```
http.get(url, { withCredentials: true });
```

- Par header

:

```
const headers = new Headers();
headers.append('Authorization', token);

http.get(url, { headers });
```

# RÉCUPÉRATION DE HEADERS

- Récupération de la réponse
- Analyse des headers
- **HttpResponse**

```
export class MyService {  
  createUser(): Observable<HttpResponse<User>> {  
    return this.http.post<User>('/users', { observe: 'response' });  
  }  
}  
  
service.createUser().subscribe(resp => {  
  const keys = resp.headers.keys();  
  if(resp.headers.has('jwt')){  
    let token = resp.headers.get('jwt');  
    // ...  
  }  
});
```

# 7 - FORMULAIRES ET ÉVÉNEMENTS

- *Template-driven forms*
- *Reactive forms*
- Directives

# 7.1 TEMPLATE-DRIVEN FORMS

- Utilisation de ngModel pour lier au composant
- Utilisation des classes CSS ajoutées par Angular selon l'état et la validité :
  - ng-untouched
  - ng-touched
  - ng-pristine
  - ng-dirty
  - ng-valid
  - ng-invalid

→ Similaire à AngularJS

# LA DIRECTIVE NGFORM

- Crée un FormGroup
- Tracke l'état et la validité des données
- Gère la liste d'éléments du formulaire
- Actif sur les tags <form> quand le module **FormsModule** est importé

## Utilisation de base

```
<form #f="ngForm" (ngSubmit)="onSubmit(f)">
```

```
onSubmit(f: NgForm) {  
    // ...  
}
```

# TEMPLATE-DRIVEN FORMS

- Exemple d'un formulaire de login

```
<form #f="ngForm" (ngSubmit)="submit(f)">
  <div>
    <label>Login:</label>
    <input type="text" name="login" ngModel required>
  </div>
  <div>
    <label>Password:</label>
    <input type="password" name="password" ngModel required>
  </div>
  <button type="submit">Submit</button>
</form>
```

```
onSubmit(f: NgForm) {
  alert(f.value.login);
}
```

## 7.2 REACTIVE FORMS

- Structure et validations construites côté  
**TypeScript**
    - Plus de contrôle
    - Template plus simple.
  - API NgForm
    - Crédit/suppression d'éléments
    - Gestion de groupes
- adapté à des formulaires complexes

# REACTIVE FORMS

Nécessite le module **ReactiveFormsModuleModule**

```
import { ReactiveFormsModuleModule } from '@angular/forms';
//...

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

# REACTIVE FORMS

- API FormBuilder, FormGroup et FormControl

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

@Component( ... )
export class MyFormComponent {
  myForm = new FormGroup({
    login: new FormControl(' ', Validators.required),
    password: new FormControl(' ', Validators.required),
  });

  constructor(){ ... }
}
```

# REACTIVE FORMS

- Le template est simplifié

```
<form [formGroup]="myForm">
  <label> Login:
    <input type="text" formControlName="login">
  </label>

  <label> Password:
    <input type="password" formControlName="password">
  </label>
</form>
```

# CONDITIONS DANS LES TEMPLATES

- Vérification des FormControl

```
<p *ngIf="!form.controls['name'].hasError('required')  
      && form.controls['name'].dirty">  
  Name is mandatory  
</p>
```

On peut aussi tester l'état du formulaire dans son ensemble

```
<p *ngIf="form.valid">...</p>
```

## 7.3 DIRECTIVES

```
<a-component customAttribute="someValue"></a-component>
```

Attribut custom que l'on donne à un élément d'un template

- pour l'apparence de l'élément : **attribute directive**
  - gérer les classes CSS ou le style de l'élément
  - modifier le comportement de l'élément en fonction d'événements utilisateur
- pour modifier la structure du DOM : **structural directive**
  - ex.: **\*ngIf**, **\*ngFor**

# ATTRIBUTE DIRECTIVE : STYLES

Directive qui positionne une classe CSS

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

@Directive({
  selector: '[myappFoo]'
})
export class FooDirective {
  constructor(renderer: Renderer2, hostElement: ElementRef) {
    renderer.addClass(hostElement.nativeElement, 'foo');
  }
}
```

```
<p myappFoo>...</p>
```

# ATTRIBUTE DIRECTIVE : ÉVÉNEMENTS UTILISATEUR

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

@Directive({
  selector: '[myappHighlight]'
})
export class HighlightDirective {
  constructor(private el: ElementRef) { }
  @Input('myappHighlight') highlightColor: string;

  @HostListener('mouseenter') onMouseEnter() {
    this.highlight(this.highlightColor || 'red');
  }
  @HostListener('mouseleave') onMouseLeave() {
    this.highlight(null);
  }

  private highlight(color: string) {
    this.el.nativeElement.style.backgroundColor = color;
  }
}
```

# ATTRIBUTE DIRECTIVE : ÉVÉNEMENTS UTILISATEUR

Utilisation de la directive **myappHighlight**

```
<div>
  <input type="radio" name="colors" (click)="color='lightgreen'">Green
  <input type="radio" name="colors" (click)="color='yellow'">Yellow
  <input type="radio" name="colors" (click)="color='cyan'">Cyan
</div>
<p [appHighlight]="color">Highlight me!</p>
```

```
export class MyComponent {
  color: string;
}
```

# 8 - TESTS UNITAIRES

- Outils de test
- Tests de composants
- Tests de services
- Tests de routage

# 8.1 OUTILS DE TEST

- **Karma** : moteur de test
  - Se charge de déterminer quels tests exécuter, et de lancer un navigateur réel ou simulé comme environnement de test
- **Jasmine** : framework d'assertion
  - Fournit les fonctions et objets permettant de décrire les conditions de test et d'effectuer les vérifications

# TESTS UNITAIRE

- Structure d'une suite de tests

```
describe("Some tests suite", () => {  
  
  it("should pass test 1", () => {  
    expect(true).toBe(true);  
  });  
  
  it("should pass test 2", () => {  
    expect(true).toBe(true);  
  });  
  
  //...  
});
```

# TESTS UNITAIRE

- Lancements des tests
  - Build l'application en "watch mode" puis lance karma et chrome

```
ng test
```

```
10% building modules 1/1 modules 0 active
...INFO [karma]: Karma v1.7.1 server started at http://0.0.0.0:9876
...INFO [launcher]: Launching browser Chrome ...
...INFO [launcher]: Starting browser Chrome
...INFO [Chrome ...]: Connected on socket ...
Chrome ....: Executed 3 of 3 SUCCESS (0.135 secs / 0.205 secs)
```

- Headless mode (console seulement)

```
ng test --browsers ChromeHeadless
```

# CONVENTIONS DES FICHIERS DE TEST

- Noms et chemins

```
src/app/my.component.ts  
src/app/my.component.spec.ts
```

- Utilisation de ".spec.ts" pour que l'outil de test identifie le fichier comme testable
- Fichier de test dans le même dossier que l'implémentation

# ANGULAR TESTBED

- Outil que facilite le test et les mocks
- Permet d'émuler un module et de le configurer
  - `configureTestingModule`

```
describe("Some tests suite", () => {

  let moduleMetadata = {
    // .. @NgModule options
    providers: [ ... ],
    ...
  }

  beforeEach(() => {
    TestBed.configureTestingModule(moduleMetadata);
  });

});
```

## 8.2 TESTS DE COMPOSANTS

Le module `@angular/core/testing` fournit les outils pour :

- Simuler la compilation du template du composant
- Piloter sa détection de changements pour décrire un comportement dynamique dans les tests

# IMPLEMENTATION DU COMPOSANT

Un simple composant qui affiche un titre

```
@Component({
  selector: 'app-root',
  template: '<h1>{{ title }}</h1>'
})
export class AppComponent {
  title = 'app works!';
}
```

# TEST DU COMPOSANT

Initialisation avec beforeEach (exécuté avant chaque 'it')

```
describe('AppComponent test suite', () => {  
  
  let comp: AppComponent;  
  let fixture: ComponentFixture<AppComponent>;  
  let el: HTMLElement;  
  
  beforeEach(() => {  
    TestBed.configureTestingModule({declarations: [AppComponent]});  
    fixture = TestBed.createComponent(AppComponent);  
    comp = fixture.componentInstance;  
    el = fixture.debugElement.query(By.css('h1'))!.nativeElement;  
  });  
  
  ...  
});
```

# TEST DU COMPOSANT

## Tests d'affichage

```
...  
  
it('should display the title', () => {  
  fixture.detectChanges();  
  expect(el.textContent).toContain(comp.title);  
});  
  
it('should display a different test title', () => {  
  comp.title = 'Test title';  
  fixture.detectChanges();  
  expect(el.textContent).toContain('Test title');  
});  
}
```

# SURCHARGE DE SERVICE

```
beforeEach(() => {
  userServiceStub = {
    isLoggedIn: true,
    user: { name: 'Test User' }
  };

  TestBed.configureTestingModule({
    declarations: [ UserComponent ],
    providers: [ {provide: UserService, useValue: userServiceStub } ]
  });
}

fixture = TestBed.createComponent(UserComponent);
comp    = fixture.componentInstance;

userService = TestBed.get(UserService);

de = fixture.debugElement.query(By.css('.welcome'));
el = de.nativeElement;
});
```

# INTERCEPTER DES APPELS À UN SERVICE

```
beforeEach(() => {
  let testUser = {name: 'test'};

  TestBed.configureTestingModule({
    declarations: [ UserComponent ],
    providers:    [ UserService ],
  });

  fixture = TestBed.createComponent(UserComponent);
  comp    = fixture.componentInstance;

  userService = fixture.debugElement.injector.get(UserService);

  spy = spyOn(userService, 'getUser')
    .and.returnValue(Promise.resolve(testUser));

  de = fixture.debugElement.query(By.css('.welcome'));
  el = de.nativeElement;
});
```

# TEST D'UN APPEL ASYNCHRONE

```
it('should show name after getUser promise (async)', async(() => {
  fixture.detectChanges();

  fixture.whenStable().then(() => {
    fixture.detectChanges();
    expect(el.textContent).toBe(testUser.name);
  });
}));
```

Ou en plus lisible avec fakeAsync:

```
it('should show name after getUser promise (fakeAsync)',
  fakeAsync(() => {
    fixture.detectChanges();
    tick();
    fixture.detectChanges();
    expect(el.textContent).toBe(testUser.name);
  })
);
```

## 8.3 TESTS DE SERVICES

- Un service est une simple classe
- Peut être testé sans outil spécifique à Angular

```
describe('FancyService without the TestBed', () => {  
  
  let service: FancyService;  
  
  beforeEach(() => {  
    service = new FancyService();  
  });  
  
  it('#getValue should return real value', () => {  
    expect(service.getValue()).toBe('real value');  
  });  
  
})
```

# GESTION DES DÉPENDANCES

- Les dépendances peuvent aussi être instanciées sans l'aide de TestBed

```
export class MyService {  
  constructor(private myDep1: MyDep){ .. }  
}
```

```
describe('MyService with dependencies', () => {  
  
  let service: MyService;  
  
  beforeEach(() => {  
    service = new MyService(new MyDep());  
  });  
  
  ...  
})
```

## 8.4 TESTS DE ROUTAGE

- Injection du router

```
let router: Router;

beforeEach(inject([Router], (router: Router) => {
    let router = router;
}));

...
```

- Vérification de redirection

```
it('navigate to "" redirects you to /home', fakeAsync(() => {
    router.navigate(['']);
    tick();
    expect(location.path()).toBe('/home');
}));
```

# 9 - TESTS DE BOUT EN BOUT

- Pilotage avec Protractor
- Assertions avec Jasmine

# 9.1 PROTRACTOR

**Moteur** d'exécution de tests via un webdriver

- Permet d'automatiser des actions qui simulent un véritable utilisateur.
- Attend automatiquement pour les opérations asynchrones
- Multiple drivers
- API browser
- API element

# CONFIGURATION

- Fichier statique dans le dossier e2e :

**protractor.conf.js**

- Chemins des fichiers de test
- Implémentation browser
- Url de base de l'application
- ...

```
exports.config = {
  allScriptsTimeout: 11000,
  specs: [ './src/**/*e2e-spec.ts' ],
  capabilities: { browserName: 'chrome' },
  directConnect: true,
  baseUrl: 'http://localhost:4200/'}
```

# API BROWSER

- Naviguer vers une page

```
let url = '/users/233';

browser.get(url);

expect(browser.getCurrentUrl()).toBe(url);

browser.get('/other-url');
...
```

# API BY ET ELEMENT

- Permettent de récupérer des éléments du DOM

```
var myDiv = element(by.css('#myDiv'));
```

- Manipuler des éléments du DOM
  - Lecture de contenu
  - Lancements d'évenements (souris/clavier)

```
element(by.css('app-root h1')).getText();
```

```
element(by.css('app-root button')).click();
```

```
element(by.css('app-root input')).sendKeys('John');
```

# API BY ET ELEMENT

- Différents sélecteurs  
(locators)

```
element(by.css('app-root h1'));
element(by.model('userLogin'));
element(by.binding('user.firstname'));
element(by.repeater('user in users'));
element(by.tagName('div'));
element(by.xpath('//ul/li/a'));
// ...
```

## 9.2 JASMINE

- Même framework que les tests unitaires

```
describe('User test suite', () => {  
  
  browser.get('/users/22');  
  
  it('should show user name', () => {  
    let text = element(by.model('userName')).getText();  
    expect(text).toEqual('Joe');  
  });  
  
});
```

# ORGANISATION ET FACTORISATION DES TESTS

## Problème

Duplication de code via les sélecteurs, complexité

## Solution

- Page objects
  - Création de classes représentant les pages
  - API métier fonctionnelle

# PAGE OBJECTS

## Création d'un page object

```
var UserPage = function() {
  var userName = element(by.model('userName'));

  this.get = async function() {
    await browser.get('/user');
  };

  this.getDisplayedName = function() {
    return userName.getText();
  };

  this.accessToMyAccount = async function(name) {
    accountLink.click();
  };

  ...
};

module.exports = new UserPage();
```

# RÉUTILISATION

L'écriture est plus naturelle et ré-utilisable

```
var userPage = require('./UserPage');

describe('User test suite', () => {

  userPage.get();

  it('should show user name', () => {
    expect(userPage.getDisplayedName()).toEqual('Joe');
  });
});
```

# 10 - MISE EN PRODUCTION

- Outils de build
- Gestion des environnements
- Servir son application

# 10.1 OUTILS DE BUILD

## ng build

Optimisations pour la production :

- Minification / Tree-shaking
- AOT

```
ng build
```

- Gestion du chemin racine dans l'URL
  - Changement du tag `<base href="/">`

```
ng build --base-href /myUrl/
```

Doc `ng build` : <https://github.com/angular/angular-cli/wiki/build>

# BUILD ET TREE-SHAKING

- Elimination de code mort ou non atteignable
- Crédit à l'optimisation
- Compilation pour lazy loading

```
ng build --aot --build-optimizer --vendor-chunk
```

# NG BUILD FLAGS

```
ng build --prod
```

- Build avec la conf "production" de `angular.json`

```
"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  
}
```

-  Mettre `vendorChunk` à  
`true`

# 10.2 GESTION DES ENVIRONNEMENTS

- Fichiers d'environnements

:

```
src/environments/environment.prod.ts  
src/environments/environment.ts
```

- Contenu du fichier

```
export const environment = {  
  production: false,  
  myURL: 'http://somewhere/',  
  otherValue: 42  
};
```

# 10.3 SERVIR SON APPLICATION

- N'importe quel server

- web

- Apache

- Nginx

- IIS

- Lite-server

- AWS

- Github pages

- ...

Le dossier `./dist` contient les fichiers de production

# CONFIGURATION DU SERVEUR WEB

## Routage :

- Fallback sur index.html
- Support des "deep links"

```
http://myapp.com/users/22
```

vs

```
http://myapp.com/#/users/22
```

## Enjeux

- Robots
- Référencement
- Lisibilité des URLs

# PRODUCTION ET CORS

*Cross-Origin Resource Sharing*

=> partage des ressources entre origines multiples

## Problème

Les appels réseaux sont refusés par le navigateur si le serveur d'API n'autorise pas CORS

- Règle:

`protocole, domaine, port`

**Solutions** : proxy, headers CORS

# AJOUT DES HEADERS CORS

Le serveur est censé répondre à une pré-requête OPTIONS

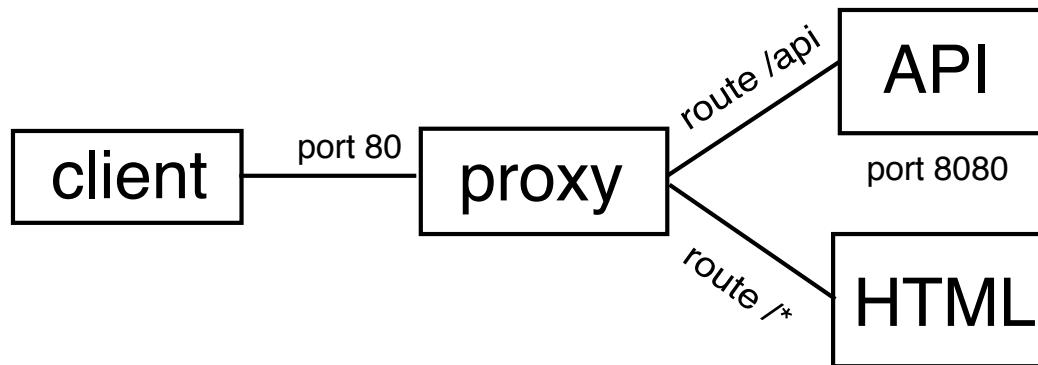
```
Access-Control-Allow-Origin: http://www.myapp.com  
Access-Control-Allow-Methods: GET, POST, PUT, DELETE
```

```
Access-Control-Allow-Credentials: ...  
Access-Control-Expose-Headers: ...  
Access-Control-Max-Age: ...  
Access-Control-Allow-Headers: ...
```

Les requêtes HTTP vers le serveur sont alors possibles depuis la page servie sur <http://www.myapp.com>

# SOLUTION AVEC UN PROXY

Pour servir les sources et l'API sur le même protocole/domaine/port



- Nginx
- Apache
- HAProxy
- ...

# PROXY WEBPACK

Pour éviter de configurer son serveur en CORS lors du développement :

## package.json

```
"scripts": {  
  "start": "ng serve --proxy-config config/proxy.conf.json",  
  // ...  
},
```

## proxy.conf.json

```
{  
  "/rest": {  
    "target": "http://localhost:61686",  
    "logLevel": "debug"  
  }  
}
```

# FIN DE LA FORMATION

---

Merci pour votre attention

# DOCUMENTATIONS OFFICIELLES

- <https://angular.io/docs>
- <https://karma-runner.github.io/2.0/index.html>
- <http://www.protractortest.org/#/api>
- <https://jasmine.github.io>
- <https://www.typescriptlang.org/docs>

# MÉMOS ET BEST PRACTICES

- <https://angular.io/guide/cheatsheet>
- <https://angular.io/guide/styleguide>
- <https://angular.io/resources>
- <http://a2.hubwiz.com/docs/ts/latest/guide/cheatsheet.html>