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

- Mulitples 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

ÉVOLUTION DES STANDARDS

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

JAVASCRIPT

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

GÉNÉRALITÉS

- **scripting**, **multi-plateformes**, **orienté objet**
- interprété, **dynamique**
- spécification **ECMAScript**
  - 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("La valeur de f est: " + f); // "La valeur de f est undef
console.log("La valeur de g est: " + g); // ReferenceError
```

Variables et types

Déclaration / Évaluation

```
if (!undefined) {
  console.log("undefined se comporte comme le booléen false");
}

var a;
a + 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 = 5"

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

TYPE *OBJECT*

Variables et types

Nombres

```
// décimaux, pas de type spécifique pour les entiers
1234567890;
42;
// octaux
0755;
0644;
// binaires
0b1010;
0B0111;
// hexadécimaux
0X10;
0xAF;
```

Variables et types

Number

```
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

Math

```
Math.PI

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

Variables et types

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("cdbbdbsbz");       // 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)
```

ÉVALUATION RAPIDE

Opérateurs

Logiques: Petit test

```
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

Logiques: Petit test

```
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

Logiques: Petit test

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

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
for (var value in 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);
}
```

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); // newArr reçoit [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

Paramètres "*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; i < arr.length; 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", "foo", "baz", "qux"];

arr.indexOf("foo");                         // 0
arr.indexOf("foo", 2);                      // 2

arr.forEach(console.log);

arr = arr.map(x => x.toUpperCase());   // ["FOO", "BAR", "FOO", "BAZ"

arr.filter(x => x.startsWith("B"));    // ["BAR", "BAZ"]

arr.every(x => typeof x === "string"); // true

arr.some(x => x.startsWith("Q"));      // true

arr.reduce(function(acc, input) { return acc + input.length; }, 0); /
```

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**

Programmation Objet

Prototype

```
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 notMe = new Person("I-has", "Money", 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, notMe);
var a4 = new Car("Audi", "A4", 2017, notMe);

model3.owner.firstname; // "I-has"
```

Programmation Objet

Prototype

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

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

new

- creates **new instance** and set its **prototype** to the prototype of the **constructor**
- **invoke** the constructor with the new object **bound to *this***
- return **value** or **this**

Programmation Objet

"*new does all the magic*"

```
var conceptCar = Car("LeEco", "LeSEE", 2016, notMe);
conceptCat;   // undefined

function Car(maker, model, year, owner) {
  // ...
  return this;
}

var conceptCar = Car("LeEco", "LeSEE", 2016, notMe);
conceptCat;   // Window
```

Programmation Objet

Connaître le type d'un objet

```
model3 instanceof Car;    // true
model3 instanceof Object; // true
```

JSON

- JavaScript Object Notation
- format d'échange de données, **sérialiser**
- 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
```

ECMASCRIPT : OÙ EN EST-ON ?

- Les navigateurs supportent couramment ES5.
- Support partiel de ES6 (publié en juin 2015).
- ES7 (juin 2016) : nouvelle approche avec une nouvelle spécification par an.

ECMASCRIPT 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

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
```

Export via un alias :

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

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

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

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
    }
}
```

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()
```

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
```

PORTÉE 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}`)
})
```

- 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

```
function times(a, b = 1) {
  return a*b;
}

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

Portée lexicale du this

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

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" ]
```

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

ÉVOLUTION DU FRAMEWORK ANGULAR

- Rappels sur AngularJS
- Angular
- TypeScript

RAPPELS SUR ANGULAR

- Publié en 2009 par Google
- Axé autour de l'augmentation du HTML :
  - par des composants (version "maison" des Web Components)
  - par des directives qui altèrent des éléments existants
- Version actuelle du CLI : 1.6.4 (mars 2017)

DÉMARRER AVEC ANGULAR (BOOTSTRAP)

- Vient sous forme de lignes de commande

```
ng install -g @angular/cli

ng new mon-application
cd mon-application
ng serve  //serveur embarqué
ng build  //pour la distribution
```

ANGULAR : APERÇU

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

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

```
<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>
```

ANGULAR

- Octobre 2016 : sortie de la version finale de Angular 2
- 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
- Outillage officiel avec notamment `@angular/cli`
- Le framework officiel est découpé en grands modules `@angular/*`

APERÇU

```
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'; }
 }
}
```

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

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

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

Mais aussi

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

//Tuple
let x: [string, number];
x = ["hello", 10]; // OK
x = [10, "hello"]; // 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");
}
```

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;
}
```

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);
```

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) { }
}
```

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
- mais aussi une interface peut étendre une classe !

GENERICS

- Une classe peut définir des types paramétrés

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

- 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);
    });
  });
```

LES COMPOSANTS

- Principes généraux
- Templates
- Style
- Cycle de vie

PRINCIPES GÉNÉRAUX

- Annotation `@Component`
- Classe "contrôleur"

TEMPLATE : NOUVELLES SYNTAXES

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>
```

Boucle :

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

Condition :

```
<button *ngIf="isAdmin">Secret button</button>
```

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>
```

Variable locale au template :

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

Two-ways binding :

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

Transclusion :

```
<ng-content select="slotName"></ng-content>
```

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>
```

STYLES DU COMPOSANT

- Styles isolés par défaut
- Utilisation de `ViewEncapsulation` pour altérer ce comportement
- Utilisation de `:host` et `:host-context(context)`

EXEMPLE

```
:host {
 display: block;
 border: 1px solid black;
}

:host-context(.error) {
 border-color: red;
}

p {
 font-size: 1.2em;
}
```

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

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

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

OUTPUT

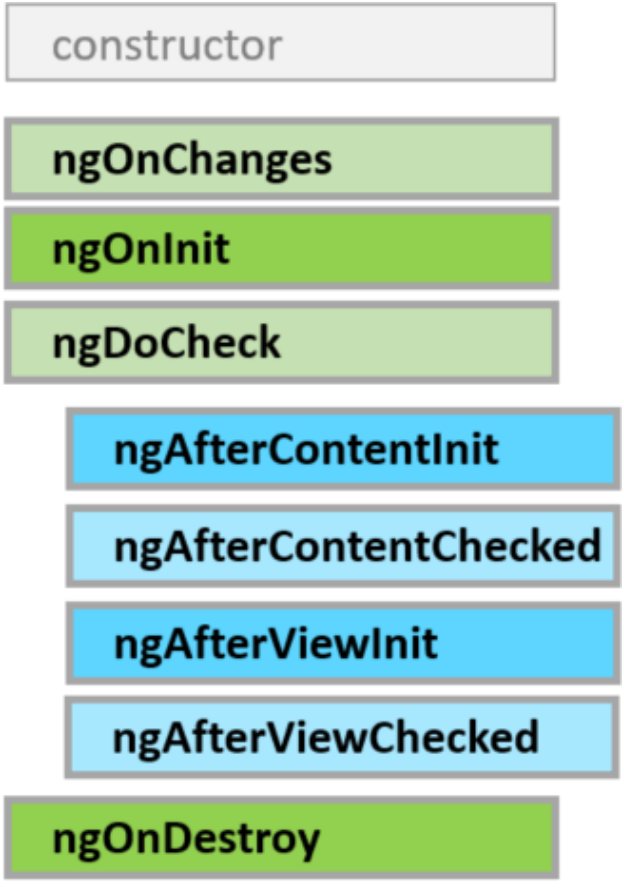
```
class TaskComponent {
 @Output('complete') onTaskComplete = new EventEmitter<boolean>();

 completeTask(isComplete: boolean) {

  this.onTaskComplete.emit(isComplete);
 }
}
```

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

CYCLE DE VIE DES COMPOSANTS



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

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.

NgOnInit

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

NgDoCheck

Appelé à chaque détection de changement.

Permet de détecter manuellement des changements qui ne seraient pas gérés par Angular.

NgAfterContentInit

Appelé quand le contenu externe (transclusion) a été injecté dans le composant.

Appelé une fois, après le premier appel à `ngDoCheck`.

NgAfterContentChecked

Appelé à chaque détection de changement du contenu externe (transclusion).

Appelé une fois après `ngAfterContentInit`, puis après chaque appel à `ngDoCheck`.

`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.

NgAfterViewChecked

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

`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.

INJECTION DE DÉPENDANCE

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

PRINCIPES DU MÉCANISME D'INJECTION

Objectif : **déclarer** ce dont on a besoin plutôt que de le **construire** soi-même.

```
class Car {
 public engine: Engine;
 public tires: Tires;
 public description = 'No DI';

 constructor() {
   this.engine = new Engine();
   this.tires = new Tires();
 }
}
```

```
class Car {
 public description = 'DI';

 constructor(
   public engine: Engine,
   public tires: Tires
 ) { }
}
```

On déplace la création des dépendances vers le consommateur :

```
new Car(new Engine(), new Tires());
```

Besoin d’une factory :

```
CarFactory.createCar(); // returns a Car instance
```

Factory générique : l’injecteur.

```
let car = injector.get(Car);
```

ANNOTATIONS ET DÉCORATEURS

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

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

CONFIGURATION DE L'INJECTEUR

```
providers: [ CarsService ]

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

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

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

InjectionToken

```
export interface AppConfig {
 apiEndpoint: string;
 title: string;
}

export const APP_DI_CONFIG: AppConfig = {
 apiEndpoint: 'api.example.com',
 title: 'Dependency Injection'
};

export let APP_CONFIG = new InjectionToken<AppConfig>(cfg);
```

```
// Dans le module :
providers: [
  ...
  { provide: APP_CONFIG, useValue: APP_DI_CONFIG },
  ...
]

// Utilisation :
constructor(
  @Inject(APP_CONFIG) config: AppConfig
) {
 this.title = config.title;
}
```

LE ROUTAGE

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

CARACTÉRISTIQUES DU ROUTEUR

- Centré autour des fragments d'URL
- 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

DÉCLARER SES ROUTES

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

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

Exemples :

```
{ path: 'hello', component: HelloWorldComponent }

{ path: '', pathMatch: 'full', redirectTo: '/hello' }

{ path: '**', component: NotFoundComponent }
```

GESTION DES PARAMÈTRES

```
{ 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))
}
```

DIRECTIVES POUR 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>
```

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

RÉSOLUTION DE DONNÉES

```
@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);
 }
}
```

Route :

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

Composant :

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

OUTLETS NOMMÉS

```
<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)
```

GUARDS

```
{
    path: 'admin',
    component: AdminPageComponent,
    canActivate: [IsAdmin]
}

@Injectable()
export class IsAdmin implements CanActivate {
  constructor(private authService: AuthService) { }

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

LAZY LOADING

AppModule :

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

AdminModule :

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

LES REQUÊTES HTTP

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

PROMESSES

Une promesse est un contrat sur la résolution future d'un action asynchrone.

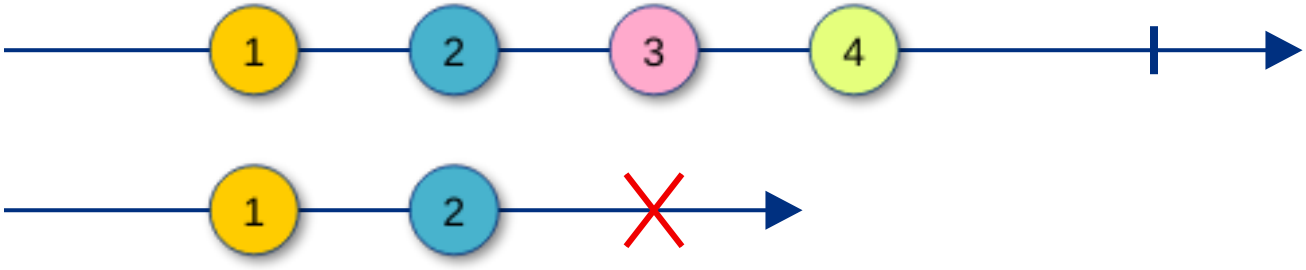
```
const prom = new Promise((resolve, reject) => {
    doSomethingLong(result => {
        if (result.isError) {
            reject(result.errorMessage);
        } else {
            resolve(result.data);
        }
    })
})

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

OBSERVABLES

Un observable est un flux de données avec une notion temporelle, 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#...)



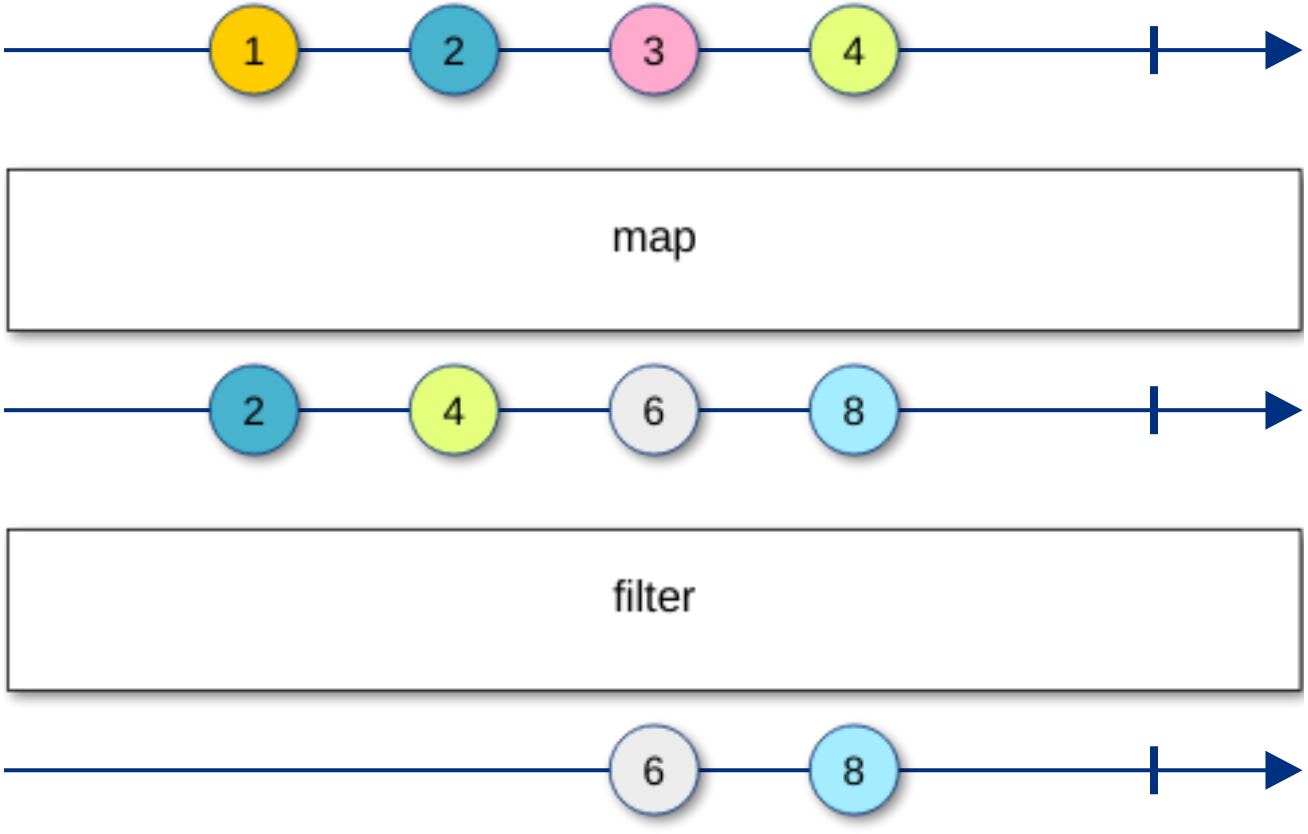
- reactivex.io/rxjs
- rxmarbles

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

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

OPÉRATEURS

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



LE SERVICE Http

```
class HttpClient {

 get(url: string, options?: RequestOptionsArgs): Observable<Response>

 post(url: string, body: any, options?: RequestOptionsArgs): Observable<Response>

 put(url: string, body: any, options?: RequestOptionsArgs): Observable<Response>

 delete(url: string, options?: RequestOptionsArgs): Observable<Response>

 patch(url: string, body: any, options?: RequestOptionsArgs): Observable<Response>

 head(url: string, options?: RequestOptionsArgs): Observable<Response>

 options(url: string, options?: RequestOptionsArgs): Observable<Response>

  ...
}
```

```
http.get('http://swapi.co/api/people/')
 .pipe(pluck('results', 'name'))
 .subscribe(names => console.log(names));
```

```
{
  "count": 87,
  "next": "http://swapi.co/api/people/?page=2",
  "previous": null,
  "results": [
    {
      "name": "Luke Skywalker",
      "height": "172",
      "mass": "77",
     ...
   },
   ...
 ]
}
```

Typage de la réponse

```
interface SwapiCharacter {
  name: string
  height: string
  mass: string
}

interface SwapiResponse {
  count: number
  results: SwapiCharacter[]
}

http.get<SwapiResponse>('http://swapi.co/api/people/')
```

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 });
```

FORMULAIRES ET ÉVÉNEMENTS

- *Template-driven forms*
- *Reactive forms*
- Intéractions utilisateur

TEMPLATE-DRIVEN FORMS

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

→ Similaire à AngularJS

```
<form #form="ngForm" (ngSubmit)="submit(form.value)">
    <div>
        <label>Firstname:</label>
        <input type="text" name="firstname" ngModel required>
    </div>
    <div>
        <label>Lastname:</label>
        <input type="text" name="lastname" ngModel required>
    </div>
    <button type="submit">Submit</button>
</form>
```

REACTIVE FORMS

Dans cette approche, la structure et les validations sont construites côté TypeScript : plus de contrôle, et template plus simple.

→ adapté à des formulaires complexes

```
export class UserFormComponent {
 form: FormGroup;

 constructor(fb: FormBuilder) {
   this.form = fb.group({
     name: ['', Validators.required],
     isAdmin: [false, Validators.required],
     mail: ['', Validators.email],
   });

   this.form.valueChanges
     .subscribe(() => console.log(
        'form modified!'
      ));
 }
}
```

```
<form [formGroup]="form">
 <label>Name:</label>
 <input type="text" formControlName="name">

 <label>Admin:</label>
 <input type="radio" formControlName="isAdmin" [value]="true" > Yes
 <input type="radio" formControlName="isAdmin" [value]="false"> No

 <label>E-mail:</label>
 <input type="text" formControlName="mail">
</form>
```

Exemple de condition :

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

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

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

INTÉRACTIONS UTILISATEUR

Dans le template, on utilise le nom des événements DOM standard :

```
<button (click)="doSomething($event)"
    (mouseenter)="highlightButton($event.target)"
>Test</button>
```

Au niveau du composant lui-même :

```
constructor(private el: ElementRef) { }

@HostListener('mouseenter') onMouseEnter() {
 this.el.nativeElement.style.backgroundColor = 'yellow';
}
```

EXEMPLE AVANCÉ : autocomplete

```
@Component({
  selector: 'app-auto-complete',
  template: `
    <input #txt type="text" />
    <ul>
      <li *ngFor="let country of countries$ | async">
        {{ country }}
      </li>
    </ul>`
})
export class AutoCompleteComponent implements AfterViewInit
{
  @ViewChild('txt', {read: ViewContainerRef})
  textInput: ViewContainerRef;

  countries$: Observable<string>;

  constructor(private http: Http) { }
```

...

...

```
  ngAfterViewInit(): void {
    const api = 'https://restcountries.eu/rest/v2';
    const el = this.textInput.element.nativeElement;

    this.countries$ = fromEvent(el, 'keyup').pipe(
      debounceTime(300),
      pluck('target', 'value'),
      distinctUntilChanged(),
      switchMap(value =>
        this.http
          .get(`${api}/name/${value}?fields=name`)
          .pipe(
            map(item => item.name)
            catchError(() => of([]))
          )
      )
    );
  }
}
```

TESTS UNITAIRES

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

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

```
describe("A suite is just a function", () => {
 let a;

 it("and so is a spec", () => {
   a = true;

   expect(a).toBe(true);
 });
});
```

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

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

```
describe('AppComponent', () => {
  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;
  });

  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');
  });
})
```

TEST DOUBLE POUR UNE DÉPENDANCE

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

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

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

  userService = TestBed.get(UserService);

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

INTERCEPTER DES APPELS À UN SERVICE

```
@Component({
 selector: 'twain-quote',
 template: '<p class="twain"><i>{{quote}}</i></p>'
})
export class TwainComponent  implements OnInit {
 intervalId: number;
 quote = '...';
 constructor(private twainService: TwainService) { }

 ngOnInit(): void {
   this.twainService.getQuote()
    .then(quote => this.quote = quote);
 }
}
```

```
beforeEach(() => {
 TestBed.configureTestingModule({
   declarations: [ TwainComponent ],
   providers:    [ TwainService ],
 });

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

 twainService = fixture.debugElement.injector.get(TwainService);

 spy = spyOn(twainService, 'getQuote')
   .and.returnValue(Promise.resolve(testQuote));

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

TEST D'UN APPEL ASYNCHRONE

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

  fixture.whenStable().then(() => { // wait for async getQuote
    fixture.detectChanges();        // update view with quote
    expect(el.textContent).toBe(testQuote);
  });
}));
```

Ou en plus lisible :

```
it('should show quote after getQuote promise (fakeAsync)',
  fakeAsync(() => {
    fixture.detectChanges();
    tick();                       // wait for async getQuote
    fixture.detectChanges(); // update view with quote
    expect(el.textContent).toBe(testQuote);
  })
);
```

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');
  });
})
```

TESTS DE ROUTAGE

```
it('should tell ROUTER to navigate when hero clicked',
  inject([Router], (router: Router) => {

    const spy = spyOn(router, 'navigateByUrl');

    // trigger an action which calls navigateByUrl
    testClick();

    // args passed to router.navigateByUrl()
    const navArgs = spy.calls.first().args[0];

    // expecting to navigate to id of the component's first item
    const id = comp.items[0].id;
    expect(navArgs).toBe('/item/' + id,
      'should nav to ItemDetail for first item'
    );
  })
);
```

TESTS DE BOUT EN BOUT

- Pilotage avec Protractor
- Assertions avec Jasmine

PROTRACTOR

Moteur d'exécution de tests via un webdriver : permet d'automatiser des actions qui simulent un véritable utilisateur.

```
browser.get('/');

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

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

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

http://www.protractortest.org/#/api

MISE EN PRODUCTION

- Outils de build
- Build avec `@angular/cli`

OUTILS DE BUILD

- Grunt / Gulp
- Webpack
- `@angular/cli`

Optimisations pour la production :

- minification / tree-shaking
- AOT

BUILD AVEC `@angular/cli`

Flag	--dev	--prod
--aot	false	true
--environment	dev	prod
--output-hashing	media	all
--sourcemaps	true	false
--extract-css	false	true