

Formation Angular 11.X

Janvier 2021
Créeé et animée par Vincent Caillierez

1. Introduction

Présentations, Angular,
TypeScript/ES6, Quickstart

Module

Copyright 2016-2019 - Toute reproduction interdite

Présentations

Participants, Formateur, Formation

Lecon

Copyright 2016-2019 - Toute reproduction interdite

Vincent Caillerez

- **Mon profil**

- Développeur web full stack depuis une quinzaine d'années.
- **Entre 2005 et 2012 :** Spécialisé dans le CMS **Drupal**.
- **Depuis 2014 :** Spécialisé dans les technos frontend, notamment **Angular**.
- **Mon job aujourd'hui :** Formateur Angular indépendant + consulting/développement freelance.

VOUS



- **Je m'appelle...**
- Je viens de... (**ville, région** ou **pays** d'origine)
- Les **3 technologies** sur lesquelles j'ai le plus travaillé dans ma vie sont...
- Quand j'entends Angular, **la première chose qui me vient à l'esprit** est...

QUIZ #1

Connaissez-vous Angular ?

<https://kahoot.it/>

Introduction à Angular

Qu'est-ce que c'est ?
Points forts, Points faibles

Qu'est-ce qu'Angular ?

- Angular est un framework JavaScript pour créer des **applications monopages** (*Single Page Applications*), **web** et **mobiles**.
- Angular gère le **front-end / l'UI** de l'application, via une interface très réactive et potentiellement complexe.
 - **Tâches typiques front-end** : Afficher les données, rafraîchir l'UI en temps réel, récolter saisies utilisateur, déclencher une action ou un traitement sur le serveur...
 - **Tâches typiques back-end** : lire/enregistrer les données dans une base, authentifier l'utilisateur, traiter un paiement en ligne, redimensionner des images, générer des pdf...
- Back-end et front-end communiquent via des **requêtes HTTP**.

Principales caractéristiques

- **Plusieurs langages supportés.** ES5, ES6/TypeScript et Dart.
- **Complet.** Inclut **toutes les briques nécessaires** à la création d'une appli professionnelle. Routeur, requêtage HTTP, gestion des formulaires, internationalisation...
- **Modulaire.** Le framework lui-même est découpé en **modules** correspondant aux grandes aires fonctionnelles (core, forms, router, http...). Vos applis doivent être organisées en **composants** et en **modules**.
- **Tout est composant.** Composant = brique de base de toute appli Angular.

Points forts d'Angular

- **Moderne.** Conçu pour le **web de demain**. Utilise les évolutions récentes du langage JavaScript (ES6 et TypeScript). Architecture inspirée des **Web Components** (“l’avenir du web”)
- **Rapide.** D’après les benchmarks, Angular est aujourd’hui **5 fois plus rapide** que la version 1⁽¹⁾.
- **Facile.** **Moins de concepts spécifiques** à Angular à maîtriser. Utilise davantage les standards. (Comparé à AngularJS 1.x.)
- **Soutenu par Google.** Ce point peut être discuté...

Points faibles d'Angular

- **Trop “usine à gaz”.** Trop “entreprise”. Certains développeurs reprochent à Angular de nécessiter un outillage trop lourd (TypeScript, IDE compatible...) et de vouloir en faire trop.
- **Ré-écriture complète d'AngularJS.** De nombreux développeurs hésitent à sauter le pas et perdre l'investissement qu'ils ont fait dans AngularJS.
- **Pas de support IE8.** En plus de supporter les dernières versions de Chrome, Edge, Firefox, IE et Safari, Angular a aussi été testé pour fonctionner avec les anciens navigateurs comme IE9+ et Android 4.1+. Voir <https://angular.io/docs/ts/latest/guide/browser-support.html>.

Ressources utiles

- **Documentation :**
 - **Site officiel Angular** (doc, API...) : <https://angular.io/>
NE PAS CONFONDRE AVEC AngularJS : <https://angularjs.org/>
 - **Livre** Ninja Squad "[Deviens un Ninja avec Angular](#)" (FR)
- **Communauté :**
 - **News** (EN) : <http://www.ng-newsletter.com/>
 - **Questions** (EN) : <http://stackoverflow.com/questions/tagged/angular>

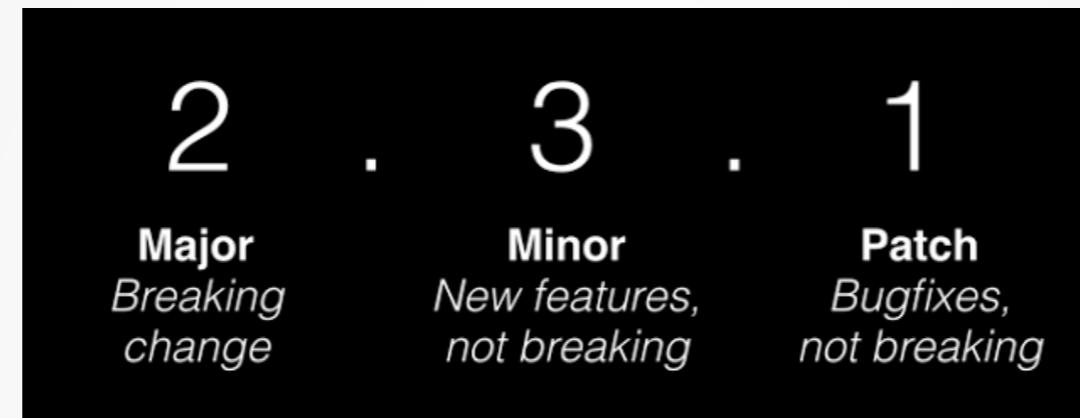
AngularJS, 2, 4...

Lecon

Copyright 2016-2019 - Toute reproduction interdite

Angular 2, 4, 5... ?

- À partir de décembre 2016, Angular est passé au **versionnage sémantique**, c. à d. que les numéros de version doivent avoir un sens :



- **AVANT** : AngularJS 1 —> Angular 2 = la v2 est **totalement incompatible** avec la v1 (ré-écriture complète du framework).
- **MAINTENANT** : Angular 3 —> Angular 4 = au moins un “breaking change” a été introduit, mais la v4 reste **majoritairement compatible** avec la v3, et la v4 respecte des phases de dépréciation.
- Il faut juste dire “Angular” sans préciser de numéro de version.
Pour info, cette formation est basée sur **Angular 11.x**.

Outilage

Prérequis, Angular CLI, appli squelette

Prérequis



node & npm
(installer les librairies)



IDE
(coder avec TypeScript et Angular)



Navigateur
(débuggeur)

VS Code - Raccourcis clavier

- **Quick Open** : ⌘P
- **Command Palette**: ⌘P
- IntelliSense : Ctrl + Space
- Delete Line: ⌘K
- Comment Block: ⌘:
- Formatting: Entire doc : ⌘F — Current Selection: ⌘K
⌘F

Source : <https://code.visualstudio.com/docs/getstarted/tips-and-tricks>

VS Code - Configuration

- **Extensions à installer**
 - TSLint
 - Angular Language Service
- **Réglages VS Code**
 - Files: Auto Save = onFocusChange
 - Files: Trim Trailing Whitespace = YES
 - Quote Style = “single” en JS et TS



- **Outil en ligne de commande** permettant de :
 - Générer un **squelette d'application** Angular
 - Générer rapidement des **bouts de code** (composant, module, route...)
 - Faire tourner un **serveur de développement**, d'**exécuter les tests**, de **déployer l'application**.
- **Paramétrable** via le fichier⁽¹⁾:

PROJECT_ROOT/**angular.json**

CLI - Principales commandes

- Installer Angular-CLI :

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

- Générer une nouvelle appli Angular + Démarrer serveur de dev :

```
ng new PROJECT_NAME  
cd PROJECT_NAME  
ng serve
```

- Générer du code :

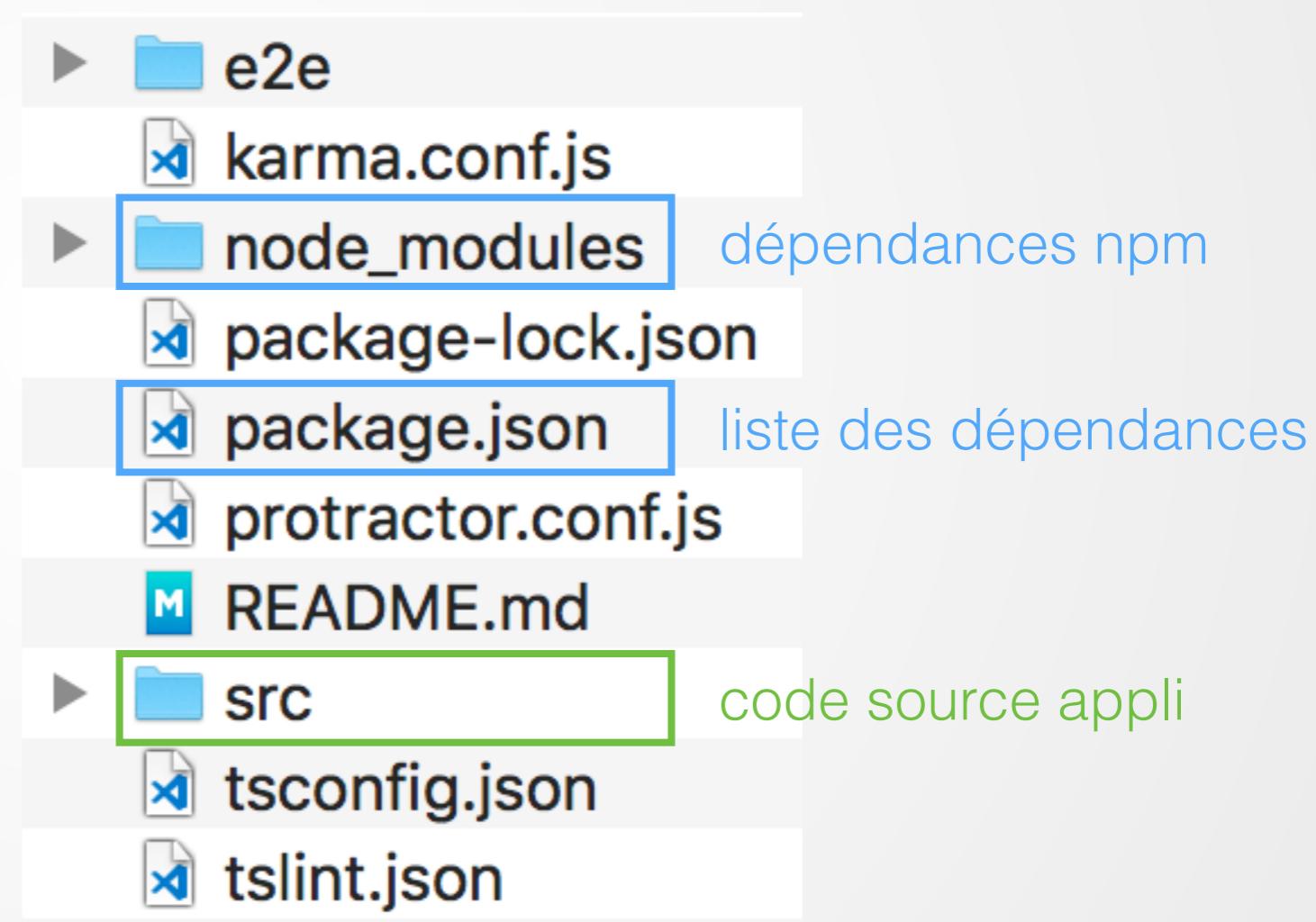
```
ng g component my-new-component  
ng g service my-new-service  
ng g module my-module
```

CLI - Fichiers importants

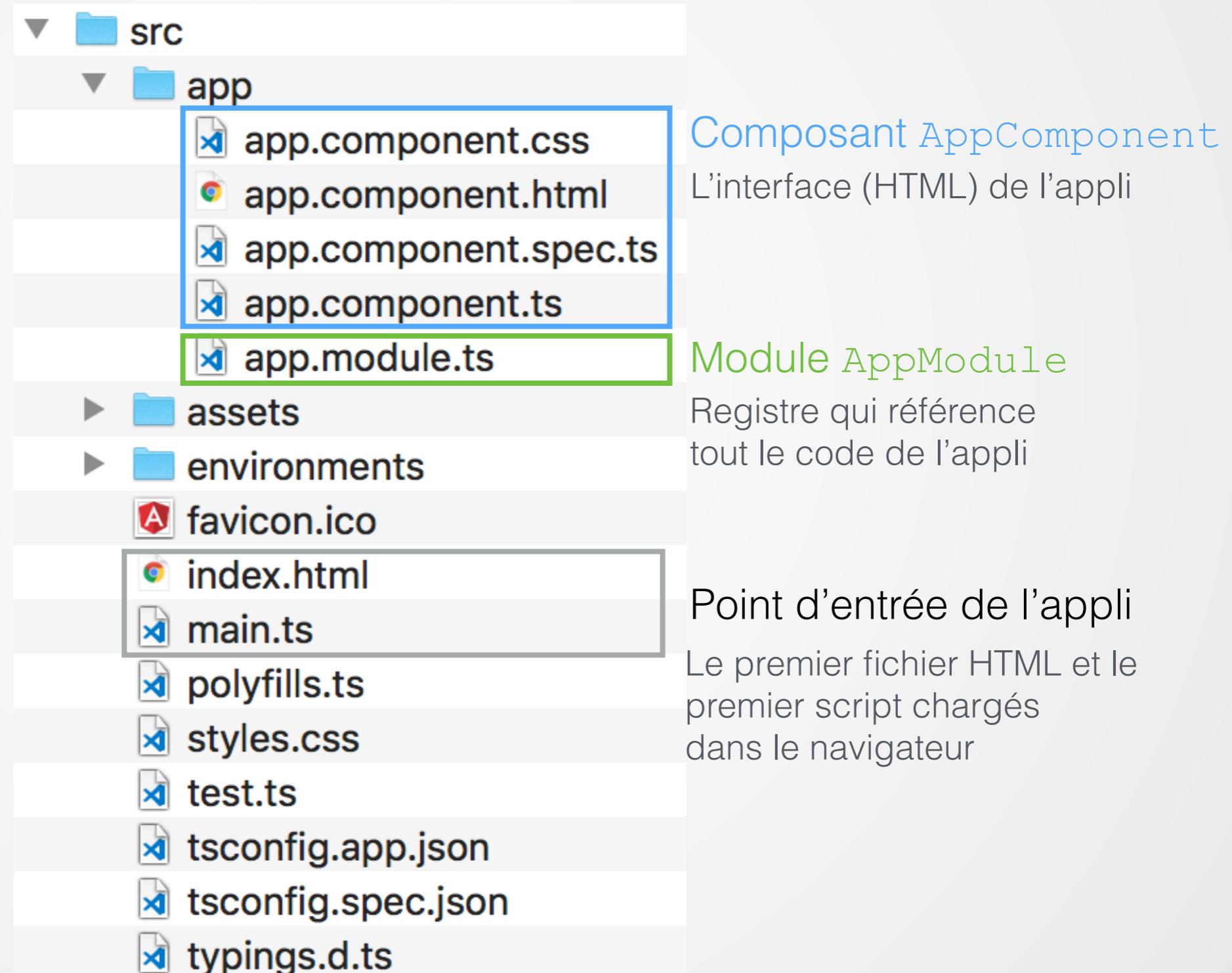
```
ng new PROJECT_NAME
```



génère un squelette d'appli
+ télécharge les dépendances



CLI - Appli Angular



CLI - Workflow de dévt

- Lancer l'application en local (serveur de dévt + live-reload) dans un terminal **que vous laisserez tourner** :
`ng serve`
- Modifier le code dans votre IDE + Enregistrer.
- Le navigateur doit se rafraîchir **automatiquement** et refléter les changements.
- Remarque : Le CLI compile automatiquement le TS en JS. Ce n'est **pas à votre IDE de le faire**.

EXO 0

- Prérequis logiciels + Installation SuperQuiz

QUIZ #2

Connaissez-vous vraiment JavaScript ?

<https://kahoot.it/>

Révisions JS

Variables, Objets, Tableaux, Fonctions

Déclarer une variable en JS :

var, let ou const ?

- Utiliser **const** par défaut.
 - ATTENTION. Les valeurs déclarées avec const ne sont pas immutables (c. à d. pas constantes). const signifie juste qu'on ne peut pas ré-assigner la variable.

```
const contact = {name: 'Pierre'};  
contact.name = 'Vincent'; // OK  
contact = {name: 'Paul'}; // PAS OK
```

- Utilisez **let** uniquement si vous devez ré-assigner la variable :

```
let html = '';  
html = '<p>' + userName + '</p>';
```

- Vous pouvez oublier **var**.

Révisions JS

Syntaxes compactes

- **Assigner une valeur par défaut :**

// OUI

```
const msg = message || 'Salut';
```

// NON

```
const msg;
if (message) {
  msg = message;
} else {
  msg = 'Salut';
}
```

- **Opérateur ternaire :**

// OUI

```
const allowed = age > 18 ? 'yes' : 'no';
```

// NON

```
const allowed;
if (age > 18) {
  allowed = 'yes';
} else {
  allowed = 'no';
}
```

- **Renvoyer true ou false :**

// OUI

```
return age > 18
  && user.role == 'admin';
```

// NON

```
if (age > 18 && user.role == 'admin') {
  return true;
} else {
  return false;
}
```

Révisions JS - Objets

- **Objet JS** = Simple collection de **paires clé-valeur**.
Équivalent d'un "hash" ou "tableau associatif" dans d'autres langages.
- **Créer** un objet :

```
let contact = {} ; // Notation littérale  
let contact = {name: 'Pierre', age: 18} ; // Décl. + Affect
```

- **Accéder aux propriétés** d'un objet :

```
// Syntaxe point  
contact.name = "Pierre";  
const name = contact.name;
```

```
// Syntaxe crochet  
contact["name"] = "Pierre";  
const name = contact["name"];
```

Passage par valeur vs Passage par référence

- En JavaScript, les **valeurs primitives** sont passées **PAR VALEUR**. Ainsi, si une fonction change une valeur primitive qu'elle a reçue, le changement **n'est pas reflété en dehors** de la fonction :

```
function double(number) {  
    number = number * 2;  
}
```

- En revanche, les **objets** sont passés **PAR RÉFÉRENCE**. Si une fonction change un objet qu'elle a reçu, le changement **est reflété en dehors** de la fonction :

```
function myFunc(theObject) {  
    theObject.make = 'Toyota';  
}  
  
var mycar = {make: 'Honda', model: 'Accord', year: 1998};  
  
var x = mycar.make; // x reçoit la valeur "Honda"  
myFunc(mycar);  
var y = mycar.make; // y reçoit la valeur "Toyota"
```

D'où l'importance de :

```
const obj = { a: 1 };
const copie1 = Object.assign({}, obj);
const copie2 = { ...obj }; // spread
```

- En **copiant un objet original dans un objet vide** avec `Object.assign()`, on “casse” la référence à l’objet original et on évite les effets de bord.
- **Fréquent dans Angular.** On affiche une liste d’objets que l’utilisateur peut éditer dans un formulaire. Il commence à éditer un objet puis il clique sur “Annuler”. PROBLÈME : l’objet affiché dans la liste a quand même été modifié, car c’était le même objet que celui édité dans le formulaire.

Révisions JS - Tableaux (1/2)

- **Créer** un tableau :

```
const contacts = [] ; // Notation littérale  
const contacts = ['Pierre', 'Paul', 'Joe'] ; // Décl. + Affectation
```

- **Itérer** sur les éléments d'un tableau - `Array.forEach()` :

```
contacts.forEach(function(contact, index) {  
    console.log(contact);  
});
```

- **Ajouter** un élément à la fin d'un tableau - `Array.push()` :

```
contacts.push(c) ; // Modifie le tableau original  
contacts.concat(c) ; // Renvoie un nouveau tableau
```

- **Retirer** un élément dans un tableau - `Array.splice()` :

```
contacts.splice(start, deleteCount);
```

Révisions JS - Tableaux (2/2)

- **Trouver une valeur** précise dans un tableau :

```
const foundIndex = contacts.indexOf('Pierre');
const foundIndex = contacts.findIndex(contact => contact.id == 10);
const foundContact = contacts.find(contact => contact.id == 10);
```

- **Transformer** tous les éléments d'un tableau -
Array.map() :

```
const contacts = ['Pierre', 'Paul', 'Joe'];
const formalContacts = contacts.map(function(contact) {
  return 'Monsieur ' + contact;
});

// SYNTAXE ÉQUIVALENTE avec fonction flèche (return implicite)
const formalContacts = contacts.map(contact => 'Monsieur ' + contact);
```

Révisions JS - Fonctions

- **Fonction de callback = fonction passée en argument** à une autre fonction. Très fréquent en JS (et Angular).

Syntaxe 1

```
function hey() {  
    alert('coucou');  
}  
  
setTimeout(hey, 500);
```

Déclaration de fonction

Syntaxe 2

```
setTimeout(function() {  
    alert('coucou');  
}, 500);
```

Expression de fonction

Syntaxe 3

```
setTimeout(() => alert('coucou'), 500);
```

Fonction flèche

Fonctions flèche 1/2 (ES6)

- **Syntaxe raccourcie** pour les **fonctions de callback** :

```
const names = ['pif', 'paf', 'pouf'];

// AVANT (ES5)
names.forEach(function(name) {
    console.log(name);
});

// APRÈS (ES6)
names.forEach( (name) => {
    console.log(name);
});
```

- Disparition de **function** + apparition de **=>** (*fat arrow*) entre les paramètres et le corps de la fonction.

Fonctions flèche 2/2 (ES6)

- **Encore plus court !** Si la fonction flèche n'a qu'**un seul paramètre**, on peut supprimer les `()`. Si elle n'a qu'**une seule instruction**, on peut supprimer les `{ }`⁽¹⁾:

```
names.forEach( (name) => { console.log(name) ; } ) ;  
// devient  
names.forEach( name => console.log(name) ) ;
```

- **Dans le corps d'une fonction flèche :**
 - **this** n'est pas bindé à la fonction elle-même, mais au contexte dans lequel la fonction est définie⁽²⁾.
 - Le **return** est implicite en l'absence d'accolades.

Fonction - Erreur fréquente #1

- Confondre le fait de **référencer** une fonction et le fait d'**invoyer** une fonction :

```
function hey() {  
    alert('coucou');  
}
```

// **CORRECT (référence)**

```
setTimeout(hey, 500);
```

// **INCORRECT (invocation)**

```
setTimeout(hey(), 500);
```

Fonction - Erreur fréquente #2

- Croire que les **noms de paramètres** dans une fonction de callback sont “**magiques**” (i.e. chargés de sens) :

```
const items = [ 'Fraise', 'Pomme', 'Banane' ];
```

// CORRECT

```
items.forEach(function(item, index) {  
  console.log(item);  
});
```

// CORRECT AUSSI

```
items.forEach(function(toto, titi) {  
  console.log(toto);  
});
```

Révision JS

Templates chaîne (ES6)

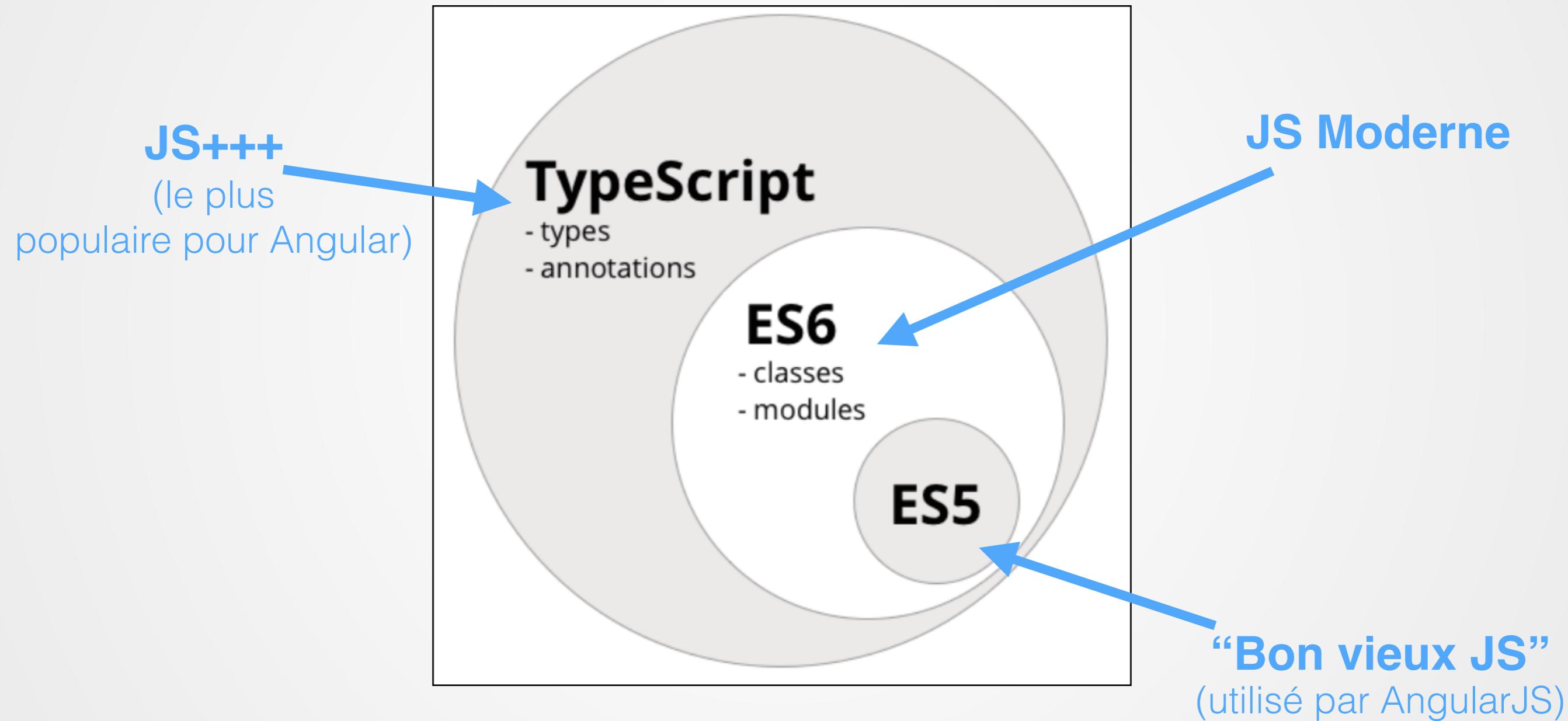
- Permettent de remplacer les **concaténations de chaînes** par une **écriture plus lisible**.
- **Syntaxe** : Chaîne de caractères délimitée par des **back-ticks** (apostrophes inversées) au lieu des simples/doubles quotes habituels.
- **Caractéristiques** :
 - Peuvent **s'étendre sur plusieurs lignes**.
 - Peuvent contenir des **guillemets simples ou doubles**, et des **variables interpolées** grâce à la syntaxe **`${var}`** :

```
// ES5
const fullname = 'Madame ' + firstname + ' ' + lastname;
// ES6
const fullname = `Madame ${firstname} ${lastname}`;
```

Introduction à TypeScript

Types facultatifs, Compilation obligatoire

Langages supportés par Angular



Pssst. Il y a aussi Dart, mais il n'est pas très utilisé.

TypeScript - Introduction

- TypeScript est un **langage de programmation libre et open-source** développé par **Microsoft** (depuis 2012).
- Il a pour but d'**améliorer et sécuriser la production de code JavaScript**, notamment grâce au **typage statique (optionnel)** des variables et des fonctions.
- Le code TypeScript n'est **pas interprété nativement** par les navigateurs ou environnements JavaScript. Il doit donc être **transpilé en JavaScript (ES5)** pour pouvoir être exécuté.
- Site officiel : <https://www.typescriptlang.org/>

TypeScript - Bénéfices

- Les erreurs de type peuvent être détectées par l'IDE, bien avant l'exécution du code. On produit donc du **code plus robuste**.
- Les types sont une **manière de documenter le code** :
 - Quand on utilise une **fonction**, on sait exactement le type de params qu'elle attend et la valeur de retour qu'elle renvoie.
 - Quand on utilise une **librairie tierce-partie**, l'IDE nous assiste en proposant les propriétés et les méthodes disponibles sur chaque objet.
- Angular tire pleinement partie de certaines syntaxes TypeScript - classes, décorateurs... - pour produire du **code plus expressif et plus compact**.

TypeScript - Surensemble de JS

```
// JavaScript ES5 - Valide en TypeScript
var prenoms = ['Pierre', 'Paul', 'Jo'];

prenoms.map(function(prenom) {
    return 'Monsieur ' + prenom;
});
```

```
// Ajout des syntaxes ES6
const prenoms = ['Pierre', 'Paul', 'Jo'];

prenoms.map((prenom) => {
    return 'Monsieur ' + prenom;
});
```

```
// Ajout des syntaxes TypeScript
const prenoms: string[] = ['Pierre', 'Paul', 'Jo'];

prenoms.map((prenom: string) => {
    return 'Monsieur ' + prenom;
});
```

TypeScript - Compilation

TypeScript



JavaScript (ES5)



```
const API_KEY = '466fgjH$DITO';

class Person {
    name: string;

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

```
var API_KEY = '466fgjH$DITO';
var Person = (function () {
    function Person(theName) {
        this.name = theName;
    }
    return Person;
}());
```

ng serve / tsc

Syntaxes TypeScript :

- const
- class
- string

Les syntaxes TypeScript ont disparu.

Syntaxes TypeScript/ES6

Types, Interfaces, Classes, Modules,
Fonctions fléchées, Templates chaîne

Types - 1/2 (ts)

- Les **types** sont au cœur de TypeScript (ils ont donné son nom au langage), mais ils sont **facultatifs**.
- Ils permettent de **spécifier le type d'une variable** en l'annotant avec la syntaxe **variable: type**.

- Exemples :

```
let age: number;
let is_customer: boolean;

// Typage + Affectation de valeur
let name: string = 'Vince';

// Tableau - Syntaxe 1
let list: number[] = [1, 2, 3];
// Tableau - Syntaxe 2
let list: Array<number> = [1, 2, 3];
```

Types - 2/2 (ts)

- Quand on n'est **pas certain** du type - any :

```
let notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // okay, definitely a boolean
```

- Type dans une **déclaration d'une fonction**⁽¹⁾ :

```
function isAdmin(username: string): boolean {
    // Ici, code qui teste si l'utilisateur username
    // est admin ou pas
    return true;
}

// Si la fonction ne renvoie rien :
function warnUser(): void {
    alert("This is my warning message");
}
```

Inférence de type (ts)

- Rappelez-vous que les types sont **facultatifs**.
- En cas d'**absence d'annotation de type**, le compilateur TypeScript s'appuie sur les **valeurs utilisées** ou le **contexte** pour déduire le type des variables (SI C'EST POSSIBLE).

- **Types primitifs⁽¹⁾** :

```
const num = 3;           // `num` est de type number
const name = 'Paul';    // `name` est de type string
```

- **Meilleur type commun⁽²⁾** :

```
const x = [0, 1, null]; // number ou null ?
```

- En conséquence, le code suivant passe en JavaScript mais plante s'il est exécuté en TypeScript :

```
var foo = 123;
foo = "hello"; // Error: cannot assign `string` to `number`
```

Interfaces (ts)

- Une interface est un moyen de **créer son propre type** et de **lui donner un nom**.
- Pour créer une interface, on utilise le mot-clé interface.
- Une interface peut être utilisée **partout où un type peut être utilisé**.
- Exemple - Utilisation d'une interface Todo pour décrire un **objet** :

```
interface Todo {  
    text: string;  
    done: boolean;  
}  
  
// Ailleurs dans le code  
let my_todo: Todo;
```

Points virgules

Propriétés facultatives (ts)

- **Dans une interface décrivant un objet**, on peut marquer certaines propriétés comme **facultatives** en plaçant un ? juste après leur nom :

```
interface SquareConfig {  
    color?: string;  
    width: number;  
}  
function createSquare(config: SquareConfig): {color: string; area: number} {  
    let newSquare = {color: "white"};  
    if (config.color) {  
        newSquare.color = config.color;  
    }  
    newSquare.area = config.width * config.width;  
    return newSquare;  
}  
  
let mySquare = createSquare({width: 200});
```

- Pattern populaire pour implémenter le **pattern “objet d’options”** (e.g. valeurs d’initialisation) et que toutes les options ne sont pas définies.

Classes - Syntaxe (ES6/ts)

Déclaration :

mot-clé **class** Person {

propriété **name**: string;

constructeur
constructor(theName: string) {
 this.name = theName;
}

méthode
sayMyName () {
 console.log('Mon nom est ' + **this.name**);
}
}

Instanciation :

```
const p: Person = new Person('Vince');  
p.sayMyName();
```

Interface ou Classe ?

- **Interface**

- Sert juste à **imposer une certaine forme aux données** (forme = liste de propriétés et méthodes).
- Peut être utilisée partout où l'on utilise un type, mais **aussi sur une classe** :

```
class SuperTodo implements Todo { ... }
```

- Ne peut pas être instanciée. Disparaît du code transpilé en JS —> Impossible de tester à l'exécution qu'une donnée est conforme à une interface donnée.
- À utiliser pour : settings de l'application, argument de méthode un peu complexe.

- **Classe**

- Définit la **forme** des données, mais aussi la **logique associée** (= le code de la classe).
- Peut être utilisée partout où l'on utilise un type.
- Peut être instanciée. Le type classe peut être “testé” dans le code transpilé en JS.
- À utiliser pour : modèles de données de l'application, logique applicative complexe.

Classes - Visibilité (ts)

- En TypeScript, tous les membres d'une classe sont **publiques par défaut** (comme si le mot-clé `public` avait été utilisé).
- Utilisez le mot-clé **`private`** pour qu'un membre ne soit pas accessible à l'extérieur de la classe, ou **`protected`** pour qu'il ne soit accessible que depuis la classe et ses descendants :

```
class Person {  
    private sayMyAge() {  
        console.log('Je ne veux pas dire mon âge...');  
    }  
}  
  
const p = new Person();  
p.sayMyAge(); // error
```

Classes

Propriété-paramètre (ts)

- Ce scénario est ultra-fréquent :

```
class Person {  
  
propriété → name: string;  
  
constructor(theName: string) {  
    this.name = theName;  
}  
}
```

paramètre

on affecte le paramètre à la propriété

- Syntaxe raccourcie :

```
class Person {  
    constructor(public name: string) {  
        // `this.name` est maintenant défini  
    }  
}
```

Classes - Erreur fréquente #1

- Accéder à un membre de la classe **sans passer par this** :

```
class Person {  
  
    name: string;  
  
    constructor(theName: string) {  
        this.name = theName;  
    }  
  
    sayMyName() {  
        console.log('Mon nom est ' + name);  
    }  
}
```

Classes - Erreur fréquente #2

- Assigner une **propriété non déclarée** :

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

Classes - Erreur fréquente #3

- Placer des instructions **en dehors** d'une méthode :

```
class Person {  
  
    age = 18;  
  
    if (age > 18) {  
        allowed = 'yes';  
    }  
  
    constructor(theName: string) {  
        this.name = theName;  
    }  
}
```

Décorateurs (ts)

- Un décorateur est une fonction qui **ajoute des métadonnées** à une classe, aux membres d'une classe (propriétés et méthodes) ou aux arguments d'une fonction.

```
@Decorateur()  
SymboleDécoré
```

- Un décorateur **commence par le symbole @** et se **termine par des parenthèses ()** (même s'il ne prend pas de paramètres). Exemple : `@Input()`
- Un décorateur doit être positionné juste **au-dessus ou à gauche** de l'élément qu'il décore.

Décorateurs

Où dans Angular ?

- Le framework Angular fournit de nombreux décorateurs.
- Ils servent à **transformer un simple symbole** du langage TypeScript en une entité spéciale reconnue par le framework :

```
@Component({  
  selector: 'my-app',  
  template: '<h1>App</h1>'  
})  
class AppComponent { }
```

```
@Injectable()  
class DataService { }
```

```
class AppComponent {  
  @Input() name: string;  
}
```

Modules (ES6)

- **Tout fichier de code ES6/TypeScript est un module⁽¹⁾.**
- Un module peut rendre les symboles qu'il déclare (classe, fonction, variable...) **accessibles aux autres modules** grâce au mot-clé **export** :

```
export class AppComponent { }
export const API_KEY = '-wy1LkhpeRz5TqEfI';
```

- Un module peut **utiliser les symboles** déclarés dans un autre module grâce au mot-clé **import {} from**⁽²⁾:

```
// Dans main.ts
import { AppComponent } from './app.component';
```

Modules - Imports

- On peut importer **plusieurs symboles du même fichier** en les séparant par des virgules :

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

- Il ne faut **pas mettre l'extension** du fichier duquel on importe :

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

- Car votre **IDE** et le **compilateur TS** résolvent les imports à partir des fichiers sources en **TypeScript**.
- Mais l'**application finale** résout les imports à partir des fichiers compilés en **JavaScript**.

Modules - Où dans Angular ?

- Quand un fichier doit utiliser un **symbole défini dans un autre fichier** (classe, fonction, variable...), c'est à dire **tout le temps**.

- Vous trouverez **2 types d'import** dans votre code :

- **Imports non relatifs** - Symboles **fournis par Angular⁽¹⁾**:
Commencent toujours par @angular

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

- **Imports relatifs** - Symboles **venant de votre propre code**:
Commencent toujours par au moins un ". "

```
import { AppModule } from './app/app.module';
import { User } from '../auth';
```

QUIZ #3

Avez-vous compris TypeScript/ES6 ?

<https://kahoot.it/>

SuperQuiz

Front-office, Back-office,
Modèle de données

Lecon
Copyright 2016-2019 - Toute reproduction interdite

SuperQuiz - Front-office

1

Liste des quizzes

Sélectionner un quiz :

- Quiz Angular 2 (général)
- Quiz TypeScript
- Quiz Module

2

Quiz Angular 2 (général)

Questions d'ordre général sur Angular 2.

Démarrer le quiz

3

Question : 1 / 3 - Score : 0

Suivant

5

Question

En quelle année AngularJS (première version) est-il sorti ?

2010

2011

2013

4

Soumettre

duction interdite

SuperQuiz - Back-office

Admin - Liste des quizzes

+ Nouveau Quiz

Titre

Quiz Angular 2 (général)

Quiz TypeScript

Quiz Module

Actions

Voir

Éditer

Supprimer

Voir

Éditer

Supprimer

Voir

Éditer

Supprimer

Questions

Gérer les questions

Gérer les questions

Gérer les questions

Modifier un quiz

Titre*

Quiz Angular 2 (général)

Description

Questions d'ordre général sur Angular 2.

Peut ré-essayer les questions

Enregistrer Annuler

Gérer les questions du quiz

[Revenir à la liste des quizzes](#)

+ Nouvelle question

Enregistrer

Position

Actions

Type

Question



multiple_choice

En quelle année AngularJS (première version) est-il sorti ?

- [X] 2010
- 2011
- 2012
- 2013

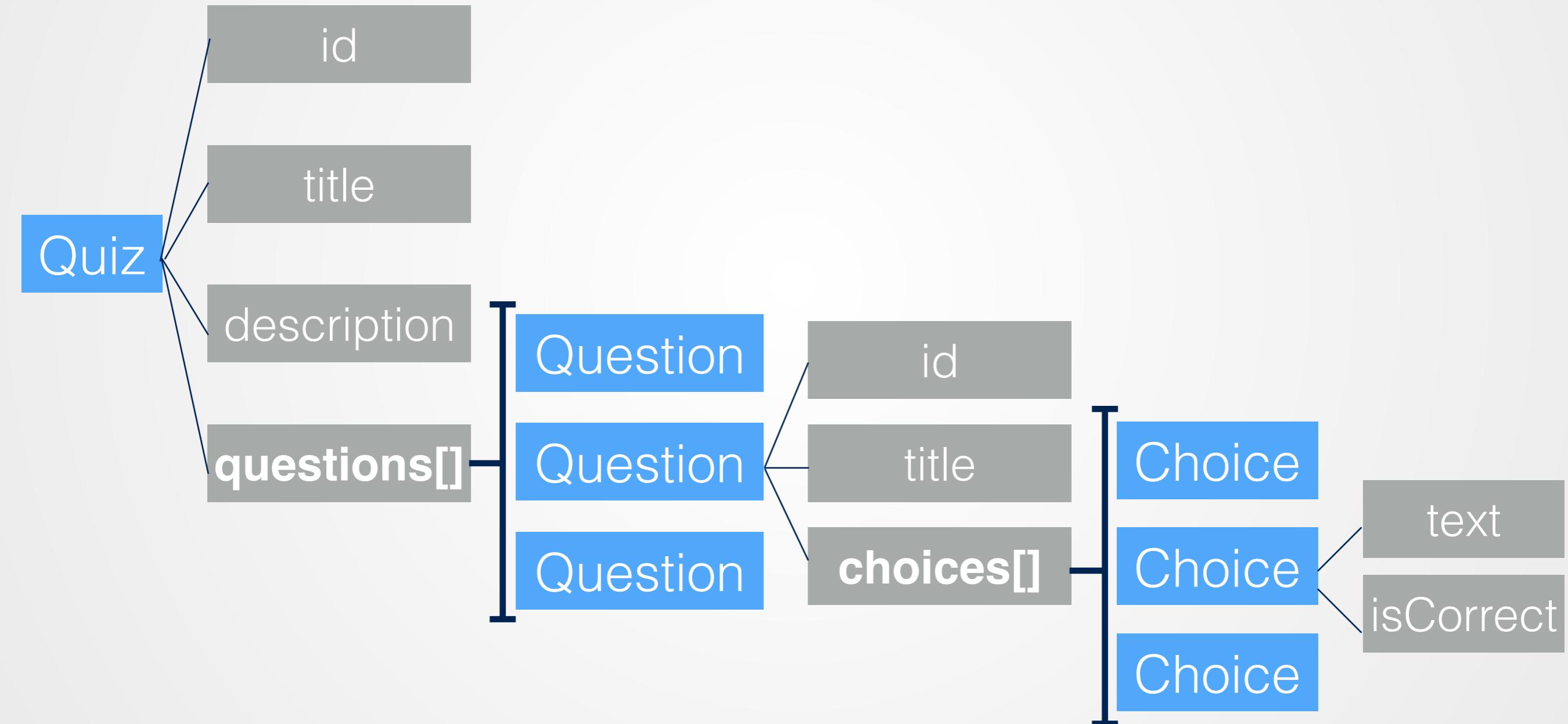


multiple_choice

Quels langages de programmation sont supportés par Angular ?

- [X] JavaScript
- [X] TypeScript
- VBScript
- PHP

SuperQuiz - Modèle de données



EXO 1

- Comprendre le modèle de données.

2. Composant I

Syntaxe, Arbre des composants,
Syntaxes de template

Module

Copyright 2016-2019 - Toute reproduction interdite

Composant Introduction

Composant - Définition

- Dans une application Angular, un composant représente un **bout de l'interface**.
- À ce titre, les composants sont responsables de l'**affichage** et de l'**interactivité** d'une application Angular.
- C'est au développeur de **découper** son interface en composants. Selon ses préférences, il peut donc y avoir **beaucoup de petits** composants, ou **quelques gros** composants.
- Une appli Angular consiste en **au moins un composant** (sans composant, il n'y aurait pas d'interface !). Mais dans la pratique, il est fréquent d'en avoir **plusieurs dizaines imbriqués**.

Composant - Imbrication

App

Navbar

Articles

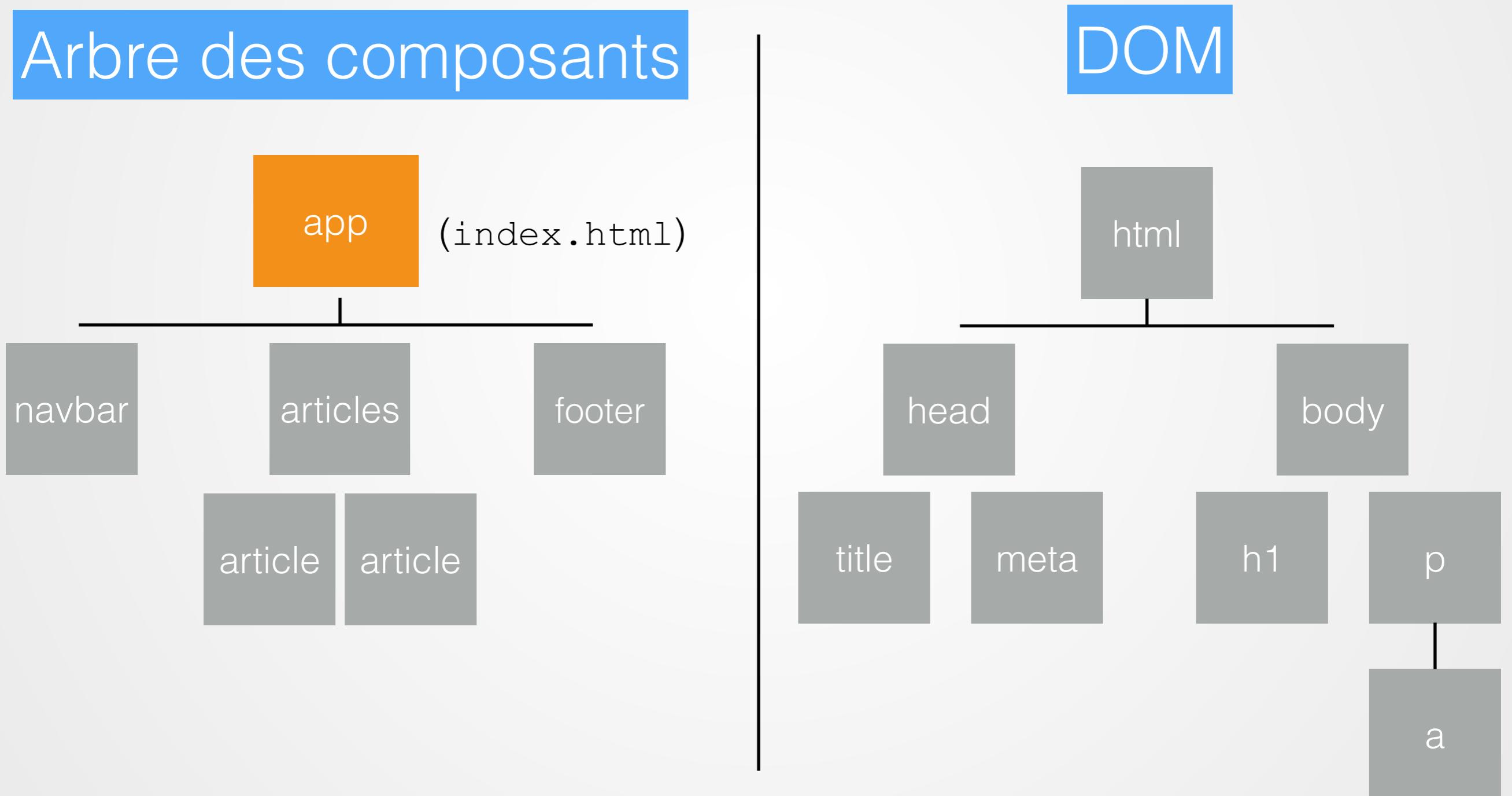
Article

Article

Article

Footer

L'arbre des composants



Exemple : FoodChéri.com

FOODCHÉRI

75011 Paris, France X

Aujourd'hui

JEU 30 VEN 31 DIM 02 LUN 03 MAR 04 De 18h30 à 19h00 ▼ 2

 Asian Salmon Bowl
Chef Caroline

- 2 + 11.90 €

Nos chefs se sont inspirés du fameux chirashi pour créer ce délicieux ... + Plus

 Fusilli pesto rosso
Chef Sébastien

- 0 + 8.90 €

Le pesto rosso se caractérise par sa couleur vibrante et son goût unique due à la présence de tomate séchée. ... + Plus





... interdite

Comment découper l'interface ?

- Quels critères utiliser pour déterminer la **granularité des composants** ? (i.e. **quelques gros composants** vs **plein de petits**)
- **Un “bout d’interface” mérite de (ou doit) avoir son propre composant si :**
 - Il va être **réutilisé** à plusieurs endroits (dans le même projet ou différents projets). Exemple : pagination, barre de navigation, image agrandissable...
 - Il contient du HTML ou des comportements complexes qu'il vaut mieux **encapsuler** (= invisibles depuis l'extérieur).
 - Il va être affiché par l'intermédiaire du **routeur**. Dans ce cas, l'utilisation d'un composant est obligatoire.

Composant Syntaxe & Utilisation

Composant - Syntaxe

Composant = Classe + Template

Classe Template

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

@Component({
  selector: 'meteo',
  templateUrl: './meteo.component.html'
})
export class MeteoComponent {
  weather = 'ensoleillé';
}
```

Classe décorée avec @Component
(selector et templateUrl obligatoires)

```
// meteo.component.html
<p>
  Le temps est {{weather}}.
</p>
```

Template
[\(HTML + Syntaxes Angular\)](#)

Le template est relié à la classe via la propriété **templateUrl** (template dans un fichier distinct). On peut aussi utiliser la propriété **template** pour un template en-ligne.

Composant - Utilisation

- On affiche un composant en écrivant son **selector comme une balise HTML**, dans le template d'un autre composant⁽¹⁾ :

```
<div>    ! balise fermante IDE
  <meteo></meteo>
  <p>Source: MétéoFrance</p>
</div>
```

Navigateur

```
<div>
  <meteo>
    <p>Le temps est ensoleillé.</p>
  </meteo>
  <p>Source: MétéoFrance</p>
</div>
```

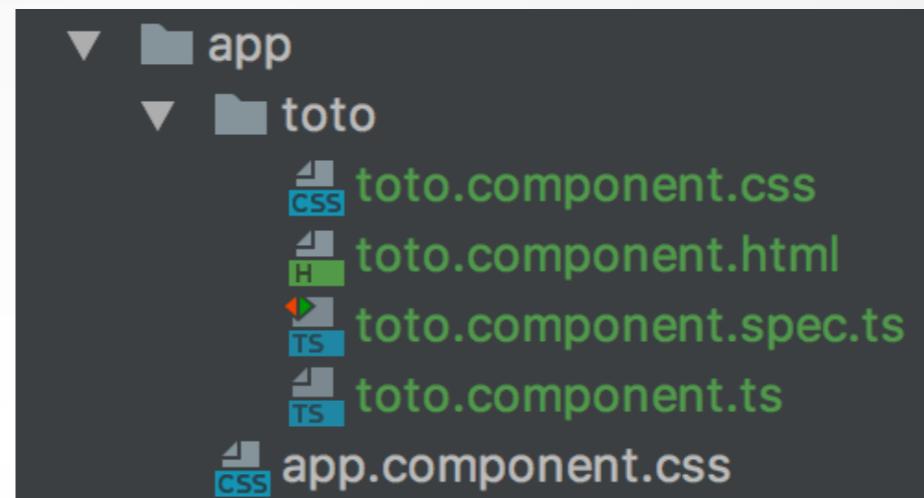
⚠ Pour être reconnu par Angular, un composant doit être **déclaré** dans AppModule, c. à d. que la classe du composant doit apparaître dans la propriété **@NgModule.declarations** :

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

Composant avec le CLI

- Générer un composant TotoComponent :

```
ng generate component toto
```



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

- Options :

ng g c toto	# Syntaxe raccourcie
ng g c foo/toto	# Crée le comp ds un sous-rép 'foo'
ng g c toto --flat	# Ne crée pas de répertoire dédié
ng g c toto --spec=false	# Pas de tests unitaires

Composant - CSS

- Les propriétés **styles** et **styleUrls** du décorateur @Component permettent d'associer des **styles CSS à un composant** :

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

@Component ({
  selector: 'meteo',
  template: '<p>Le temps est {{weather}}.</p>',
  styles: ['p { background: yellow; }'] ← tableau
})
export class MeteoComponent {}
```

- **Styles locaux.** Les styles définis ainsi n'affectent **que le template de ce composant**, même s'ils sont très généraux.
- **Styles globaux.** Pour définir des styles qui affectent **tous les composants**, placez-les dans le fichier **styles.css** du projet⁽¹⁾.

Relation Parent-Enfant

HTML

```
<div>
  <p>Hello !</p>
</div>
```

Angular - PAS OK

```
<app>
  <navbar></navbar>
  <articles></articles>
  <footer></footer>
</app>
```

<p> est un enfant de <div>

Ne fonctionne pas par défaut⁽¹⁾.
Les balises situées entre <app> et </app>
seront écrasées par le template du
composant AppComponent.

Angular - OK

```
<app></app>
```

```
@Component({
  selector: 'app',
  template:
    <navbar></navbar>
    <articles></articles>
    <footer></footer>
})
class AppComponent { }
```

Les balises enfant d'un composant Angular
doivent être placées **dans le template du
composant parent**.
Il faut donc examiner le code source du
parent pour savoir s'il contient des enfants.

Compilation des templates

- Le navigateur **ne voit jamais le code source** des templates de composants :
Template (HTML + Syntaxes Angular) —> JavaScript pur
- **Compilation des templates** Angular :
 - Les *syntaxes Angular* dans les templates sont exécutées : les interpolations sont valorisées, les bindings d'événement transformés en listeners...
 - Les *balises de composant* sont remplacées par leur template compilé.

```
@Component({
  selector: 'app-root',
  template: `
    <h1>Welcome to {{ title }}!</h1>
    <button (click)="hello()">Hello</button>`})
export class AppComponent {
  title = 'app';
  hello() {
    this.title = 'Hello';
  }
}
```

compilation

```
▼ <app-root ng-version="5.0.5">
  <h1>Welcome to app!</h1>
  <button>Hello</button>
</app-root>
```

EXO 2

- Vos premiers composants.

Syntaxes de template

1/2

Interpolation, Binding de propriété

Interpolation

- Permet d'**afficher du texte dynamiquement** dans le template d'un composant.
- L'expression interpolée peut être une **propriété** ou un **appel de méthode** de la classe, ou un **petit bout de code JavaScript** (par exemple un opérateur ternaire).
- L'interpolation est “**live**” : si la valeur de l'expression change côté classe, le template est mis à jour aussitôt.
- Syntaxe :

```
<p>Le temps est {{ expression }}.</p>
```

Interpolation - Exemples

Classe

```
export class MeteoComponent {  
  
    // Propriétés  
    weather = 'ensoleillé';  
    obj = {weather: 'ensoleillé'};  
  
    temp = 35;  
  
    // Méthode  
    getWeather() {  
        return 'ensoleillé';  
    }  
}
```

Template

```
<p>  
    Le temps est {{weather}}.  
    Le temps est {{obj.weather}}.  
</p>  
  
<p>  
    Le temps est {{getWeather()}}.  
</p>  
  
<p>  
    Le temps est  
    {{temp > 30 ? 'chaud' : 'ok'}}.  
</p>  
  
<!-- Affiche une chaîne vide -->  
<p>  
    Le temps est {{coucou}}.  
</p>
```

Binding de propriété

- Permet de **modifier les propriétés des balises HTML** qui se trouvent dans le template d'un composant.
- Utile pour modifier les **attributs** (``) ou le **formatage CSS** (`<p class="...">`) du HTML à partir de données définies dans la classe.
- Le binding est “**live**” : si la valeur de l'expression change côté classe, le template est mis à jour aussitôt.
- Syntaxe :



```
<balise [proprieteDOM]="expression"></balise>
```

Aparté : Propriétés DOM

- RAPPEL : Le DOM (*Document Object Model*) est une **représentation d'un document HTML sous forme d'arbre**, et permet de **manipuler ce document programmatiquement** (c. à d. de changer son contenu et ses styles).
- **NE PAS CONFONDRE attributs HTML et propriétés DOM :**
 - **HTML** : <input **type**=**"text"** **value**=**"hello"**>
 - **DOM** : HTMLInputElement avec les propriétés **type** et **value**.
- Chaque attribut HTML a une propriété DOM correspondante, mais **un élément DOM possède certaines propriétés qui ne correspondent à AUCUN attribut HTML** : innerHTML, textContent, hidden...
- Les propriétés du DOM sont sensibles à la casse :
~~textcontent~~ - ~~TEXTCONTENT~~ - **textContent** (OK)
- Liste de toutes les propriétés DOM existantes : https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model

Binding de propriété

Exemples

Classe

```
export class MeteoComponent {  
  form = new FormGroup(...);  
  
  // Propriétés  
  isReady = false;  
  weather = {  
    type: 'ensoleillé',  
    icon: 'images/soleil.jpg'  
  };  
  
  // Méthode  
  isValid(): boolean {  
    return this.form.valid;  
  }  
}
```

Template

```
<p [hidden]="isReady">  
  Chargement en cours...  
</p>  
  
<img [src]="weather.icon"> // OUI  
 // NON  
  
<button [disabled]="!isValid()">  
  Enregistrer  
</button>  
  
<!-- CSS -->  
<p [class.jaune]="weather.type==='ensoleillé'">  
  Voici le temps qu'il fait...  
</p>
```

EXO 3

- Utiliser l'interpolation et le binding de propriété.

Syntaxes de template

2/2

Directives structurelles,
Binding d'événement, Pipes

*ngIf

- *ngIf fait partie des directives structurelles, c. à d. les directives qui **modifient la structure du HTML** en ajoutant/retirant des balises. Elles commencent toutes par le caractère *****.
- ***ngIf** permet d'**insérer/retirer un fragment de HTML** du DOM selon qu'une expression vaut true ou false⁽¹⁾ :

```
<balise *ngIf="expression">Affiché si expr est true</balise>
```

- La syntaxe ***ngIf... else** permet d'**alterner entre deux fragments de HTML** :

```
<div *ngIf="isReady; else loading>Texte à afficher</div>
<ng-template #loading>Chargement en cours...</ng-template>
```

- La directive *ngIf est évaluée en **“live”** : si la valeur de l'expression change, le template est mis à jour aussitôt.

*ngFor

- ***ngFor** permet de **répéter un fragment de HTML** pour chaque élément d'une liste :

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

- En supposant que la propriété `items` contienne un tableau `['Banane', 'Pomme', 'Poire']`, le HTML final sera :

```
<ul>
  <li>Banane</li>
  <li>Pomme</li>
  <li>Poire</li>
</ul>
```

- La directive `*ngFor` est évaluée en “**live**” : si la valeur de l'expression change, le template est mis à jour aussitôt.

Directives structurelles

Exemples

Classe

```
export class MeteoComponent {  
  user; // undefined  
  villes: any[] = [  
    { name: 'Lille' },  
    { name: 'Paris' },  
    { name: 'Lyon' }  
  ];  
  
  ngOnInit() {  
    // user vient du backend  
    this.user = loadUserFromDb();  
  }  
}
```

Template

```
<ul>  
  <li *ngFor="let ville of villes">  
    {{ ville.name }}  
  </li>  
</ul>  
  
<p *ngIf="villes.length === 0">  
  Aucune ville trouvée.  
</p>  
  
<!-- Attend que user soit défini -->  
<p *ngIf="user">  
  {{ user.name }}  
</p>  
OU BIEN :  
<p>  
  {{ user?.name }}  
</p>
```

Binding d'événement

- Permet de **réagir aux actions de l'utilisateur.**
- Quand l'utilisateur manipule notre application, le navigateur **déclenche tout un tas d'événements** sur chaque balise HTML des templates : **click, keyup, mouseover...**
- Le binding d'événement permet de **réagir à ces événements en exécutant nos propres instructions.**
- Le binding est “**live**” : à chaque fois que l'événement se produit, l'instruction associée est ré-exécutée.
- Syntaxe :



```
<balise (evenementDOM) = "instruction"
```

Binding d'événement

Exemples

Classe

```
export class MeteoComponent {  
  villes: any[];  
  
  showDetails = false;  
  
  loadVilles() {  
    this.villes = [  
      { name: 'Lille' },  
      { name: 'Paris' },  
      { name: 'Lyon' }  
    ];  
  }  
  
  onChanged(ev) {  
    ev.preventDefault();  
    ev.stopPropagation();  
  }  
  
  onSp() {}  
}
```

Template

```
<button (click)="loadVilles()">  
  Charger les villes  
</button>  
<ul>  
  <li *ngFor="let v of villes">  
    {{v.name}}</li>  
</ul>  
  
<select (change)="onChanged($event)">  
  ...  
</select>  
!  
<textarea (keydown.space)="onSp()">  
  ...  
</textarea>  
  
<a (mouseover)="showDetails=true">  
  Voir détails  
</a>  
<p *ngIf="showDetails">  
  ... Détails ...  
</p>
```

Pipes

- Souvent, les données brutes n'ont **pas le bon format pour être affichées dans la vue**. On a envie de les transformer, les filtrer, les tronquer, etc.
- Les *pipes* permettent de **réaliser ces transformations directement dans le template** (dans AngularJS, les pipes s'appelaient les “filtres”).
- Syntaxe :

```
{ { expression | pipe } }  
{ { expression | pipe:'param1':'param2' } }
```

Pipes - Exemples

Template

```
<pre>
  { { villes | json } }
</pre>

{ { 10.6 | currency:'EUR':true } }

{ { birthday | date:'dd/MM/yyyy' } }

<p>
  { { "C'est **super**" | markdown } }
</p>
```

Navigateur

```
<pre>[
  { "name": "Lille" },
  { "name": "Paris" },
  { "name": "Lyon" }
]</pre>
```

€10.60

16/07/1986

```
<p>
  C'est <strong>super</strong>
</p>
```



! Les pipes orderBy et filter d'AngularJS n'existent plus.

EXO 4

- Utiliser les directives structurelles et le binding d'événement.

Templates - Résumé

Et points importants

Le “scope”

- À partir d'Angular 2, il n'y a plus de “scope”.
- Un template peut accéder à l'ensemble des **propriétés et méthodes publiques** de la classe associée. C'est ça le “scope”.

Classe

```
export class MeteoComponent {  
    // Propriété publique  
    weather = 'ensoleillé';  
  
    // Propriété privée  
    private user = {name: 'Vince'};  
  
    // Syntaxe raccourcie  
    constructor(public api: Api) {}  
  
    // Méthode publique  
    refresh() {  
        this.weather = 'pluvieux';  
    }  
}
```

Template

```
<p>  
    Le temps est {{weather}}.  
</p>  
  
<button (click)="refresh()">  
    Actualiser  
</button>  
  
<p>  
    Il fait {{ api.getCelsius() }}°.  
</p>  
  
<!-- PAS BIEN, car private -->  
    {{user.name}}
```

Résumé des syntaxes

Interpolation

```
<p>  
    Le temps est {{weather}}.  
</p>
```

Pipes

```
{{ birthday | date }}
```

Binding de propriété

```
<img [src]="weather.icon">  
<p [class.active]="isActive"></p>
```

Binding d'événement

```
<button (click)="refresh()">  
    Actualiser  
</button>
```

Directives structurelles

```
<ul>  
    <li *ngFor="let ville of villes">  
        {{ville.name}}  
    </li>  
</ul>
```

```
<p *ngIf="villes.length === 0">  
    Aucune ville trouvée.  
</p>
```

Directives attribut

```
<p [ngClass]="currentClasses"></p>
```

Tout est live !

- Dans tous les exemples qu'on vient de voir, **les bindings sont live**. La vue est une **projection en temps réel des données du composant**. Si ces données changent, la vue change immédiatement.
- **Qu'est-ce qui peut faire changer les données ?**
 - Une **action de l'utilisateur** (clic, saisie clavier...)
 - Le retour d'une **requête HTTP**.
 - Une émission de valeur par un **Observable**.
 - Un **timer**.
 - Un événement **web socket**.
- Angular surveille tous ces événements dans le cadre de ce qu'on appelle la "**détection de changement**".

QUIZ #4

Avez-vous compris les composants ?

<https://kahoot.it/>

Comprendre la classe Answer

Avant de faire l'exercice qui suit...

La classe Answer

- **RAPPEL.** Les différentes réponses possibles à une question s'appellent des **choix**. Ils sont stockés dans la propriété **question.choices** :

```
const q = new Question( options: {
  'title': 'Angular est vraiment trop canon.',
  'choices': [
    { 'text': 'Vrai', 'isCorrect': true },
    { 'text': 'Faux' }
  ]
});
```

- La classe **Answer** permet de stocker une “**réponse**” de l’utilisateur, c’est à dire le ou les **choix qu’il a sélectionné(s)** pour une question donnée :

```
const answer = new Answer ({
  questionId: 12
});
answer.addChoice(choice);
answer.removeChoice(choice);
```

Answer - Manipuler les choix

- Ensuite, la classe Answer expose des **méthodes/propriétés pour manipuler les choix** qu'elle gère :

```
// Sélectionner un choix
addChoice(choice: Choice) { ... }

// Dé-sélectionner un choix
removeChoice(choice: Choice) { ... }

// Tester qu'un choix est sélectionné
hasChoice(choice: Choice): boolean { ... }

// Renvoie true si la question courante a au moins un choix sélectionné
isAnswered(): boolean { ... }
```

- NB. Les méthodes ci-dessus gèrent automatiquement les **choix multiples**.

EXO 5

- Afficher le détail d'une question (+ réponses possibles) :

Titre de la question :

- Choix 1
- Choix 2
- Choix 3

Soumettre

Directives - Introduction

- On distingue 3 types de directives dans Angular :
 - **Composants** - Directives avec un template (e.g. **MeteoComponent**)
 - **Directives structurelles** - Directives qui modifient la structure du DOM (***ngIf**, ***ngFor**...)
 - **Directives attributs** - Changent l'*apparence* ou le *comportement* d'une balise ou d'un composant
 - Angular propose des **directives attribut natives** (**ngClass**, **ngStyle**...).
 - On peut aussi créer nos **directives attribut custom**.

Directives attribut natives

- La directive **ngClass** permet d'**ajouter ou d'enlever dynamiquement plusieurs classes CSS** sur une balise HTML :

```
<div [ngClass]="expression">J'ai la classe.</div>
```

- La directive **ngStyle** permet de **définir dynamiquement plusieurs styles CSS** sur une balise HTML :

```
<div [ngStyle]="expression">J'ai du style.</div>
```

- La directive **ngModel** permet de **faire une liaison de données bi-directionnelles** entre un champ de formulaire et une propriété de la classe⁽¹⁾ :

```
<input [(ngModel)]="currentUser.name">
```

Directives attribut natives

Classe

```
export class MeteoComponent {  
  
  currentClasses: {};  
  currentStyles: {};  
  user = {name: 'Bob'};  
  
  setCurrentClasses() {  
    this.currentClasses = {  
      'saveable': this.canSave,  
      'modified': !this.isUnchanged,  
      'special': this.isSpecial  
    };  
  }  
  
  setCurrentStyles() {  
    const s = this.isSp?'24px':'12px'  
    this.currentStyles = {  
      'font-style': 'italic',  
      'font-weight': 'bold',  
      'font-size': s  
    };  
  }  
}
```

Template

```
<div [ngClass]="currentClasses">  
  J'ai la classe.  
</div>  
  ou  
<div [class.special]="isSpecial">  
  ...  
</div>  
  
<div [ngStyle]="currentStyles">  
  J'ai du style.  
</div>  
  ou  
<div [style.fontSize]="isSpecial ? 'x-' :  
  ...  
</div>  
  
<input [(ngModel)]="user.name">
```

Directive attribut custom

Syntaxe

- Une directive est une classe décorée avec `@Directive` :

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

@Directive({
  selector: '[myHighlight]'!
```

⚠

```
)
```

```
export class HighlightDirective {
```

```
}
```

- Difference avec un composant :

- Une directive n'a **pas de template ni de CSS propres** puisqu'elle vient modifier un élément/composant existant. Elle n'a donc pas de propriétés `template[Url]` et `styles[Url]`.
- En conséquence, le `selector` d'une directive utilise plutôt la **syntaxe attribut** (crochets autour du nom) plutôt que la **syntaxe élément** (pas de crochets) réservée aux composants.

Directive attribut custom

Utilisation

- On utilise une directive en écrivant son **selector** comme si c'était **l'attribut d'une balise HTML ou d'un composant** :

```
<p myHighlight>Ce paragraphe va être surligné.</p>
```

- On peut utiliser **plusieurs directives attribut** sur le même élément/composant :

```
<p myHighlight otherDirective>...</p>
```

 **Pour être reconnue par Angular**, une directive attribut doit être déclarée dans AppModule, c. à d. que la classe de la directive doit apparaître dans la propriété `@NgModule.declarations` :

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

Directive attribut custom

3 techniques pour accéder à l'élément hôte

```
import { Directive, ElementRef }  
from '@angular/core';  
  
@Directive({  
  selector: '[myHighlight]'  
})  
export class HighlightDirective {  
  @HostBinding('class.valid') isValid;  
  @HostListener('click') onClick() {...}  
  constructor(el: ElementRef) {  
    el.nativeElement.style.background = 'red';  
  }  
}
```

```
<p myHighlight>  
  ...  
</p>
```

équivalent à

```
<p [class.valid]="isValid"  
  (click)="onClick()"  
  [style.background]="red">  
  ...  
</p>
```

- **@HostBinding()** binde une propriété de l'élément hôte à une propriété locale.
- **@Hostlistener()** binde un événement de l'élément hôte à une méthode locale.
- **ElementRef** injecté dans le constructeur de la directive récupère une référence à l'élément DOM de l'hôte.

Directive attribut custom

Exemple

Déclaration :

```
@Directive({
  selector: '[myHighlight]'
})
export class HighlightDirective {
  @Input() highlightColor: string; 3

  1 constructor(private el: ElementRef) { }

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

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

Utilisation :

```
<p myHighlight highlightColor="yellow">Highlighted in yellow</p>
```

Directives vs composants

	Composant	Directive
Définition	Affiche/génère un élément d'interface	Modifie un élément existant
Décorateur	@Component	@Directive
Sélecteur	selector: 'meteo'	selector: '[meteo]'
Template ?	OUI (+ styles CSS)	NON
Utilisation	<meteo></meteo>	<p meteo></p>

Les directives sont des composants sans template.

3. Composants II

Input/Output, Cycle de vie, Projection

Input/Output

Réutiliser les composants

- **PROBLÈME** : Les composants vus jusqu'à présent ne sont **pas très réutilisables** : ils affichent toujours les mêmes données.

```
<navbar></navbar>
```

- **SOLUTION :**

- Pouvoir **passer des données** à un composant → **INPUT**
- Pouvoir **réagir aux actions** effectuées dans un composant
→ **OUTPUT**

```
<navbar [items]="navItems" (logout)="doLogout () "></navbar>
```

- Syntaxiquement, les inputs/outputs ressemblent beaucoup au **binding de propriété** et **binding d'événement** :

```
<button [disabled]="true" (click)="openModal () >Voir</button>
```

Input - Définition & Syntaxe

- Un input permet de **passer des données** à un composant **au moment où on l'affiche** :

```
<meteo ville="Paris"></meteo>
```

- Dans cet exemple, le composant `<meteo>` possède un input `ville`. Les données transmises via l'input permettent de **personnaliser l'affichage** du composant, et de le rendre **ré-utilisable**.
- Pour créer l'input dans le composant `<meteo>`, on décore la propriété `ville` avec le décorateur `@Input()` :

```
@Component({  
  selector: 'meteo',  
  template: '<div>...</div>'  
})  
export class MeteoComponent {  
  @Input() ville: string;  
}
```

Input - 3 syntaxes pour passer des données

- On peut **passer des données** à l'input via **3 syntaxes** différentes :

```
<!-- Sans crochets --> Chaîne littérale -->
<meteo ville="Paris"></meteo>

<!-- Avec crochets --> Expression -->
<meteo [ville]="city"></meteo>

<!-- Interpolation --> Chaîne littérale -->
<meteo ville="{{city}}"></meteo>
```

- La syntaxe à utiliser dépend du type de données passées à l'input :
 - **Sans crochets** pour une **chaîne littérale**.
 - **Avec crochets** pour une **expression** (texte, objet, tableau...).
 - **Interpolation** - Cette syntaxe est déconseillée, car limitée : une interpolation produit toujours une **chaîne littérale**.

Output - Définition

- Un output permet à un composant enfant d'**émettre un événement** à son composant parent :

```
<meteo (alerteCanicule)="boireBeaucoup () "></meteo>
```
- Dans cet exemple, le composant **<meteo>** possède un output **alerteCanicule**. Le parent **écoute l'événement** avec la syntaxe d'event-binding — `(event) = "instruction"` — et **réagit** en appelant l'une de ses méthodes locales — ici, **boireBeaucoup ()**.
- Pour créer un output, il faut décorer une propriété de la classe enfant avec le décorateur **@Output()** (voir slide suivant).

Output - Syntaxe

- Composant **enfant** :

```
@Component ({  
    selector: 'meteo',  
    template: `<div>  
        <button (click)="declencher()">Déclencher</button>  
    </div>`  
})  
export class MeteoComponent {  
    temperature = 40;  
    1 @Output() alerteCanicule = new EventEmitter<number>();  
  
    declencher() {  
        this.alerteCanicule.emit(this.temperature);  
    }  
}
```

2 !

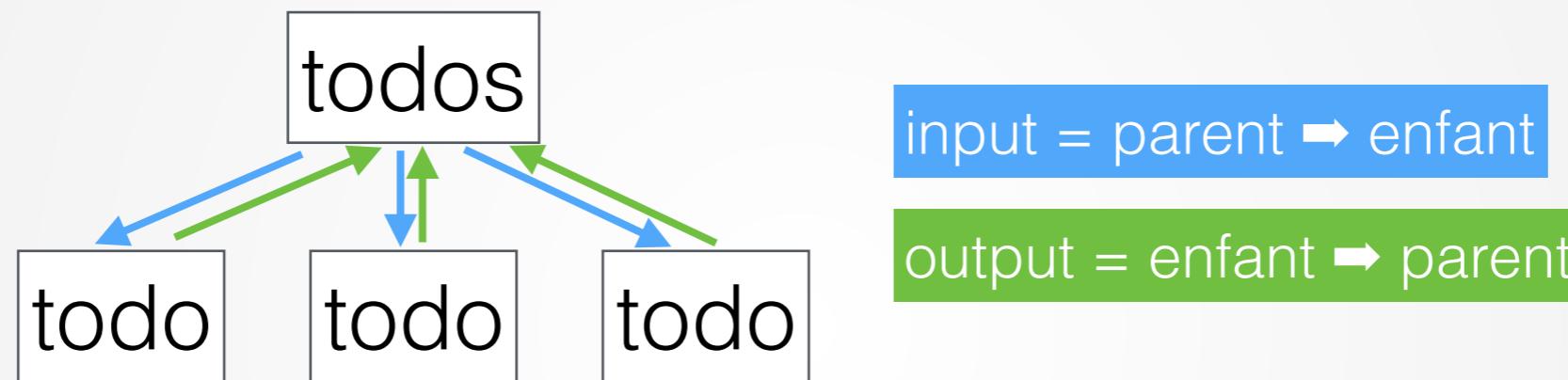
3 !

- Composant **parent** :

```
@Component ({  
    template: `<meteo (alerteCanicule)="boireBeaucoup($event)"></meteo>`  
})  
export class AppComponent {  
    1 boireBeaucoup(temperature: number) { ... }  
}
```

Input/Output - Bénéfices

- Permet à **des composants de communiquer** facilement*
(*à condition qu'ils soient parent / enfant ViewChild) :



- Permet de **rendre un composant hyper-réutilisable** (aka “pattern smart-dumb”) :
 - Car très facile de ré-utiliser un composant qui ne communique avec l’extérieur que via des inputs/outputs. Le contrat est clair.
 - Mais nécessite d’avoir (ou d’introduire) un composant parent qui gère les tâches “smart” (communication avec le backend, le routeur, des services...)

EXO 6

- Refactoriser la liste des quizzes.

Cycle de vie

Lecon

Copyright 2016-2019 - Toute reproduction interdite

Méthodes “Cycle de vie”

- Méthodes spéciales permettant d'exécuter des actions **à des moments-clé de la vie d'une directive / d'un composant⁽¹⁾**.
- Ces méthodes sont **reconnues** et **exécutées automatiquement** par Angular si elles existent et qu'elles portent le nom correct.
- Angular propose des **interfaces TypeScript** qui garantissent une implémentation correcte de ces méthodes.

ngOnInit

- La méthode **ngOnInit** est appelée **une fois après l'instanciation du composant**. Un composant est instancié lors qu'il est **affiché via le routeur** ou qu'il porte un ***ngIf qui passe à true**.
- À utiliser pour : initialiser le composant, récupérer les données du backend, faire qqchose avec les inputs du composant (pas encore évalués à l'exécution du constructor) .

```
@Component (...)  
export class MyComp implements OnInit {  
  @Input() isAdmin: boolean;  
  
  ngOnInit() {  
    this.loadData(); // récupère données du backend  
  
    if (this.isAdmin) { // utilise un Input  
      // do something  
    }  
  }  
}
```

ngOnDestroy

- La méthode **ngOnDestroy** est appelée **à la destruction du composant**. Un composant est détruit lors qu'on **navigue vers une autre page** via le routeur ou qu'il porte un ***ngIf qui passe à false**.
- À utiliser pour : faire des tâches de nettoyage bas niveau, par ex. se désabonner des observables ou des timeouts/intervalles.

```
@Component(...)  
export class MyComp implements OnDestroy {  
  private sub: Subscription;  
  private timeoutId: number;  
  
  ngOnDestroy() {  
    // Désabonnement d'observable  
    this.sub.unsubscribe();  
  
    // Désabonnement de timeout  
    clearTimeout(this.timeoutId);  
  }  
}
```

ngOnChanges

- La méthode **ngOnChanges** est appelée **à chaque fois qu'un input reçoit une nouvelle valeur.**
- À utiliser pour : recalculer l'état du composant basé sur la nouvelle valeur des inputs.

```
@Component (...)  
export class MyComp implements OnChanges {  
    @Input() isLoggedIn: boolean;  
  
    ngOnChanges (changes: SimpleChanges) {  
        // Ré-exécute des instructions à chaque  
        // fois que le statut utilisateur change  
        if (changes.isLoggedIn  
            && !changes.isLoggedIn.firstChange) {  
            // do something  
        }  
    }  
}
```

Projection

- La projection permet à un composant de **conserver et compiler son contenu HTML**.
- Imaginons un **composant custom** qui affiche du contenu sous forme d'onglets :

```
<tabset>
  <tab><p>Du contenu</p></tab>
  <tab><div>Autre chose</div></tab>
  <tab><strong>Bla bla bla</strong></tab>
</tabset>
```

- Pour que le composant TabComponent préserve et compile le contenu HTML de chaque <tab>, on utilise une directive spéciale <ng-content></ng-content> :

```
@Component ({
  selector: 'tab',
  template: `
    <div role="tabpanel" class="tab-pane">
      <!-- Sera remplacé par le contenu du tab -->
      <ng-content></ng-content>
    </div>`,
})
export class TabComponent { }
```

Passer des infos à un composant

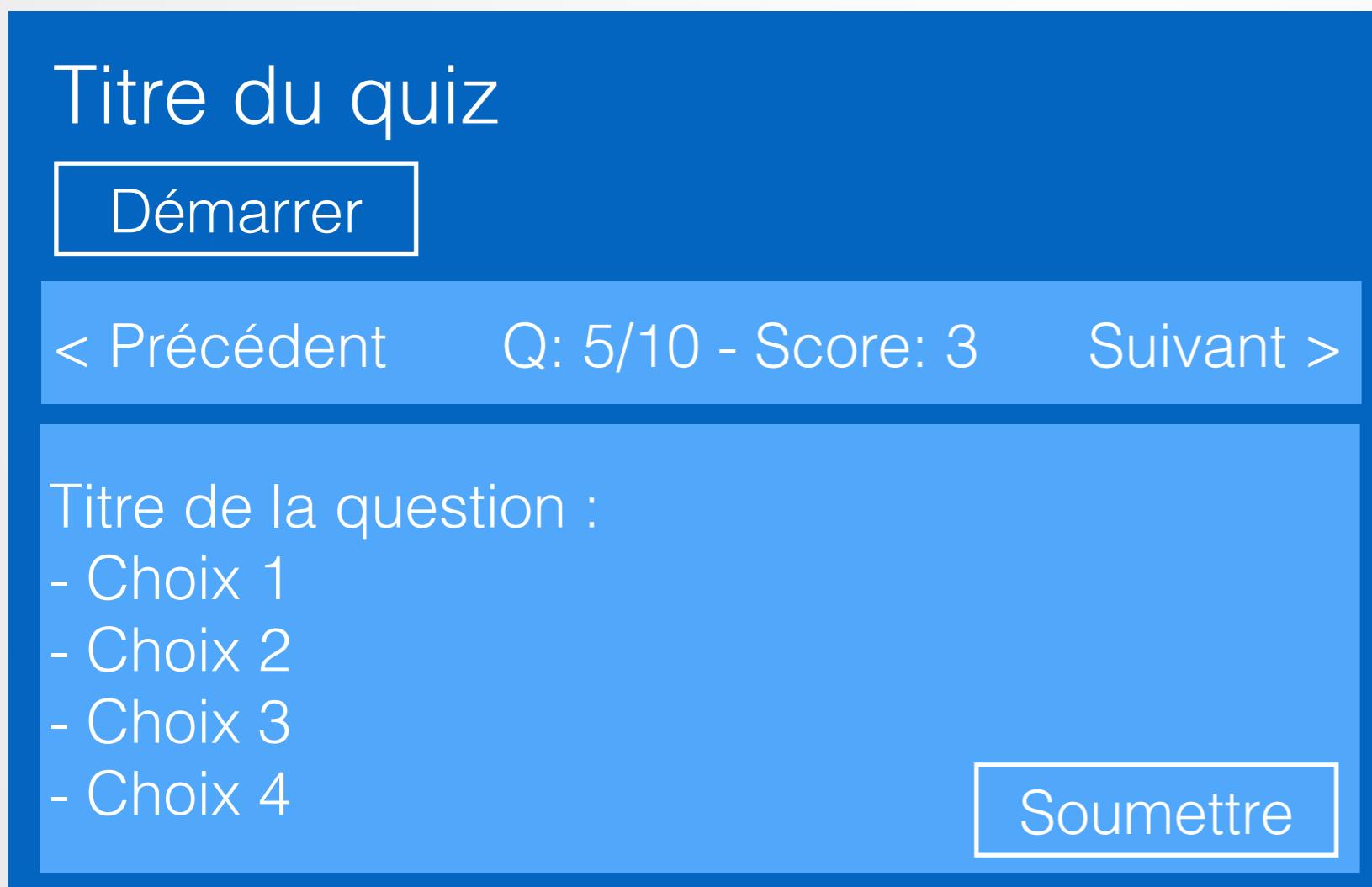
	Technique 1	Technique 2
Type d'info	Paramètres, données	Contenu (HTML + Angular)
Technique	Input	Projection
Implémentation dans le composant	<pre>@Component({ selector: 'meteo' }) export class MeteoComp { @Input() ville: string; }</pre>	<pre>@Component({ template: ` <div><ng-content> </ng-content></div>` }) export class AlertComp {}</pre>
Exemple d'utilisation	<pre><meteo ville="Paris"></pre>	<pre><alert> Etes-vous certain de vouloir supprimer ce contenu ? </alert></pre>

Le Quiz Player

Lecon
Copyright 2016-2019 - Toute reproduction interdite

Quiz Player 1/2

- Pour pouvoir naviguer dans le quiz, introduisons un **composant Quiz Player** contenant tous les éléments d'interface requis pour afficher un quiz complet : bouton **Démarrer**, **Score** + boutons **Précédent/Suivant**, et **question en cours**.

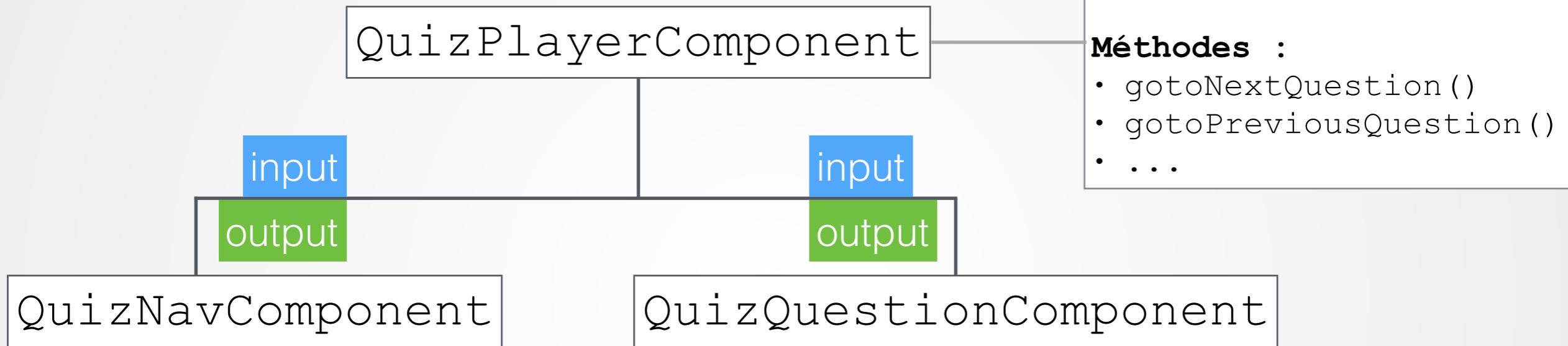


QuizPlayerComponent

QuizNavComponent

QuizQuestionComponent

Quiz Player 2/2



- **Le composant parent est “smart”.** Il est responsable de charger et d’initialiser les données, et de gérer la logique du quiz (navigation, score...).
- **Les composants enfants se contentent d’afficher** les données qu’on leur passe en input, et d’émettre de nouvelles données en output.
- Comment parent et enfants communiquent-ils ?

EXO 7

- Préparer la navigation dans un quiz.

4. Services & Injection de dépendances

Module

Copyright 2016-2019 - Toute reproduction interdite

Services

Définition,
Services natifs Angular, Créer un service

Services - Définition

- Un service peut contenir 2 choses :
 - **Logique applicative**
Vaste catégorie, car pratiquement **n'importe quel bout de code** peut être encapsulé dans un service.
ATTENTION. Ne pas mettre le code qui gère la logique d'affichage dans un service ; ce code doit se trouver dans un composant.
 - **Données**
Données représentant l'**état de l'application** (e.g. utilisateur en cours) ou **données partagées entre plusieurs composants** (e.g. n° de page en cours dans une pagination).
- Un service doit être **focalisé sur une tâche bien précise**, comme **l'authentification**, le **logging**, la **communication avec la base de données**...

Services vs Composants

- Architecture recommandée :
 - **SERVICE** : Contiennent tout le **code pur** (règles métier, traitements...). Pas liés au DOM.
 - **COMPOSANT** : Se procurent les **données** et gèrent la **logique d'affichage**. Utilisent les services grâce à l'**injection de dépendance**.
 - **CONCLUSION** : Les **composants** sont une couche fine, une “**glue**”, qui fait le médiateur entre les **vues** (rendues par les templates) et les **services**.

Services

D'où viennent-ils ?

- Il y a 3 moyens de se procurer un service :
 1. On l'importe depuis un module **Angular** :

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

2. On l'importe depuis une **librairie tierce-partie** :

```
import { NgxSpinnerService } from 'ngx-spinner';
```

3. On le crée **soi-même** ! 😊

- **IMPORTANT.** Contrairement aux composants, les services ne sont **pas obligatoires** : on peut coder une appli Angular sans service, mais pas sans composant.

Services fournis avec Angular

- Les modules d'Angular contiennent souvent des services :
 - HttpClientModule contient le service **HttpClient**.
 - RouterModule contient le service **Router**.
 - ReactiveFormsModule contient le service **FormBuilder**.
 - ...
- ⚠️ Les services Angular ne sont pas “activés” par défaut. Pour les activer, il faut **importer le module correspondant** dans son projet :

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

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

Créer son propre service

- Un service est une **simple classe** décorée avec `@Injectable` :

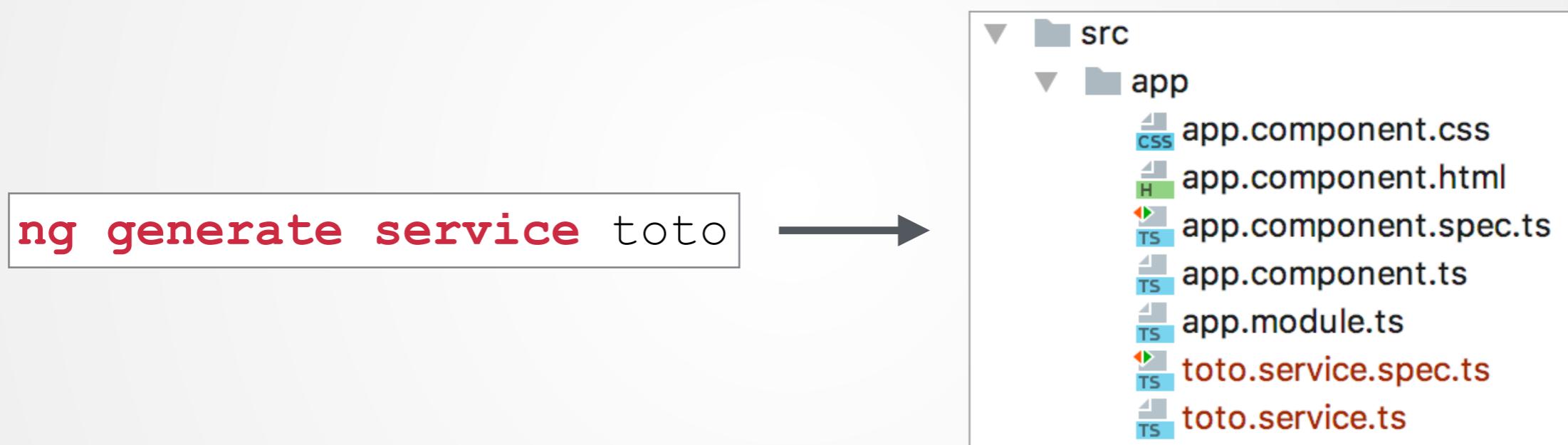
```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class UserService {
  loadUsers() {
    return [ {}, {}, {} ];
  }
}
```

- Un service est un **singleton**⁽¹⁾. Idéal pour **partager un état / des données entre plusieurs composants**.

Service avec le CLI

- Générer un service TotoService :



- Options :

```
ng g s toto          # Syntaxe raccourcie  
ng g s foo/toto      # Crée le service ds un sous-rép 'foo'  
ng g s toto --spec=false # Pas de tests unitaires
```

Injection de dépendance (DI)

Ou comment utiliser un service

DI - Qu'est-ce que c'est ?

- **Design pattern** bien connu, pas spécifique à Angular.
- Dans Angular, la DI est utilisée **pour se procurer et instancier un service**.
- Quand on veut utiliser un service, on dit à Angular “*Procure-moi une instance du service TotoService, peu importe comment*”, et c'est Angular qui fait tout le travail.
- Cette façon de procéder se nomme **l'inversion de contrôle**.

DI - Syntaxe détaillée

- L'injection de dépendance repose sur **2 points de syntaxe** :
 - Déclarer un **paramètre de constructeur** dans une classe Angular.
 - Typer ce paramètre avec un **type de “classe de service”**.

```
export class MyComp {  
  
    constructor(private authService: AuthService) {  
        // Ici, `authService` peut être utilisé.  
    }  
  
    loadData() {  
        if (this.authService.isLoggedIn()) { ... }  
    }  
}
```

- La DI fonctionne dans **toutes les classes Angular** : **composant, service, module, directive** (c. à d. toutes les classes avec un décorateur Angular).
- On peut injecter plusieurs dépendances dans le même constructeur :

```
constructor(param1: MyService,  
          param2: OtherService...)
```

DI - Sans et Avec

Sans DI

```
import { DataService }  
from '../services/data.service';  
  
@Component(...)  
export class AppComponent {  
  users: User[];  
  private dts;  
  
  constructor() {  
    this.dts = new DataService();  
  }  
  
  ngOnInit() {  
    this.data = this.dts.loadUsers();  
  }  
}
```

Avec DI

```
import { DataService }  
from '../services/data.service';  
  
@Component(...)  
export class AppComponent {  
  users: User[];  
  
  constructor(private dts: DataService)  
  { }  
  
  ngOnInit() {  
    this.data = this.dts.loadUsers();  
  }  
}
```

- Instanciation impérative / manuelle
- Chaque `new` → nouvelle instance
- Le service ne peut pas être “substitué” (tests unitaires, parties spécifique de l’appli)
- Le service ne peut pas dépendre d’autres services.

- Instanciation déclarative / automatique
- Singleton = une instance pour toute l’appli
- On peut substituer une autre version du service pour les tests unitaires ou des parties spécifique de l’appli.
- Le service peut dépendre d’autres services.

Scope d'un service

- Quand on crée un service, on peut choisir entre 2 syntaxes pour **déterminer sa portée**, i.e. où il sera injectable dans l'application :

	Syntaxe 1	Syntaxe 2
Syntaxe	@Injectable({providedIn: 'root'}) @Component.providers: [Service]	
Portée	GLOBALE. Le service est injectable partout dans l'application.	! LOCALE. Le service n'est injectable QUE dans ce composant et ses enfants.
Singleton ?	OUI. Instance unique pour toute l'application.	PAS VRAIMENT. Instance unique seult pour ce comp et ses enfants.
Exemple	<pre>@Injectable({ providedIn: 'root' }) export class AuthService { ... }</pre>	<pre>@Component({ selector: 'my-comp', template: '<p>Hello</p>', providers: [AuthService] }) export class MyComp { }</pre>

DI - Dépendances autres que “services”

- Neuf fois sur dix, la dépendance injectée avec la DI est un service.
- Mais il peut être pratique de pouvoir **injecter une simple valeur** avec la DI : paramètres de configuration de l'appli, URL du backend, etc.

```
@NgModule ({  
  providers: [  
    { provide: 'CLÉ', useValue: 'VALEUR' },  
    { provide: 'API_URL', useValue: 'https://api.site.com' },  
    { provide: 'IS_PROD', useValue: true },  
  ]  
})  
export class AppModule { }
```

Déclaration

```
import { Inject } from '@angular/core';  
  
constructor(@Inject('API_URL') private apiUrl: string) {  
  // ...  
}
```

Injection

QUIZ #5

Avez-vous compris les services ?

<https://kahoot.it/>

Le QuizService

Lecon
Copyright 2016-2019 - Toute reproduction interdite

QuizService

Présentation et Bénéfices

- Gère toutes les **interactions CRUD avec le backend** : charger un quiz, enregistrer un quiz, supprimer un quiz...
- Ajoute une **couche d'abstraction** : les parties de notre appli qui veulent manipuler des quizzes n'ont pas besoin de connaître les détails d'implémentation (le backend peut changer, l'implémentation peut changer...).
- Permet de **retravailler les données**. Souvent, les données n'ont pas le même format côté **back** et côté **front**⁽¹⁾. Il est donc utile ou nécessaire de les reformater avant usage.

EXO 8

- Créer un service d'accès aux données.

Le QuizStateManager

Lecon
Copyright 2016-2019 - Toute reproduction interdite

QuizStateManager

Présentation

- L'état du QuizPlayer consiste en **4 propriétés** :
 - Le **quiz** en cours (currentQuiz)
 - La **question/réponse** en cours (currentQuestion / currentAnswer)
 - Toutes les **réponses soumises** par l'utilisateur (currentAnswers)
- **BONNE PRATIQUE** : Encapsuler la logique métier qui gère l'état (question précédente/suivante, mémoriser une réponse...) dans un service : **QuizStateManager**.
- **ENSUITE** : On passe par les **APIs** du QuizStateManager pour manipuler l'état.

QuizStateManager

API à utiliser

- **Initialisation** : Définir le quiz en cours avec QuizStateManager.**setQuiz(currentQuiz)**
- **Démarrer** : Récupérer la première question/réponse avec QuizStateManager.**getFirstQA()**
- **Suivant** : Récupérer la question/réponse suivante avec QuizStateManager.**getNextQA()**
- **Précédent** : Récupérer la question/réponse précédente avec QuizStateManager.**getPreviousQA()**
- **Soumettre une réponse** : Enregistrer la réponse reçue avec QuizStateManager.**saveAnswer(answer)** + Récupérer la liste réactualisée de toutes les réponses avec QuizStateManager.**getAllAnswers()**

QuizStateManager

Les méthodes get ____ QA ()

- ATTENTION, les méthodes get ____ QA () renvoient **UN OBJET** contenant une **question** ET une **réponse**.

Utilisation :

```
const qa = qsm.getNextQA();  
  
// Option 1  
const question = qa.question;  
const answer = qa.answer;  
  
// Option 2  
const { question, answer } = qa;
```

Implémentation

QuizPlayerComponent - SMART

QuizStateManager

QuizNavComponent - DUMB

Titre du quiz

< Précédent Q : 5/10 - Score : 3

Suivant >

Ceci est la question :

- Choix 1
- Choix 2
- Choix 3

QuizQuestionComponent - DUMB ettre

- Le service QuizStateManager est injecté uniquement dans le composant parent “smart”, et la communication avec les enfants “dumb” se fait exclusivement via des inputs/outputs.

EXO 9

- Afficher un quiz entier (= toutes les questions).

5. Modules Angular

Définition, Syntaxe,
Root vs Feature Module

Module

Modules Angular

Bénéfices, Syntaxe & Modules natifs

NgModule - Pourquoi ?

- Une application Angular contient **de nombreuses briques de code**:
 - Composants
 - Directives
 - Pipes
 - Services
- Toutes ces briques ne peuvent pas “flotter” dans le vide. Elles doivent être **rangées dans des modules Angular**, ou **NgModules**.
- Un module est comme un **registre qui référence tout le code d'une application** Angular. Une brique de code ne peut pas faire partie de l'application sans être référencée dans un module.

NgModule - Bénéfices

- **Organisation du code.** Permet de **regrouper les fonctionnalités liées** de manière cohérente, notamment tous les composants, directives, pipes, et services liés à une fonctionnalité.
- **Réutilisation du code.** Puisqu'il **encapsule** tous les composants/directives/providers/modules dont il a besoin, un module est comme une **mini-application autonome** qui peut facilement être réutilisée d'un projet Angular à l'autre.
- **Amélioration des performances.** Un module peut être chargé via le routeur de manière **asynchrone** (aka *lazy-loading*), seulement au moment où l'utilisateur visite une page précise de l'application. Cela évite de charger inutilement du code pour tous les utilisateurs.

NgModules natifs Angular

- **BrowserModule** - Contient les fonctionnalités nécessaires à l'**exécution de l'application dans un navigateur**. À importer uniquement dans le module racine (AppModule). NB. Ce module inclut CommonModule.
- **CommonModule** - Contient les **directives de base d'Angular** comme ngIf et ngFor . A priori, TOUS VOS MODULES devront importer ce module (sinon, ils ne pourront pas utiliser les directives de base).
- **FormsModule** - Contient les fonctionnalités liées au **formulaires**.
- **HttpClientModule** - Contient les fonctionnalités liées aux **requêtes HTTP**.
- Pour utiliser les fonctionnalités ci-dessus, **listez les modules correspondants** dans la propriété **imports** vos propres modules :

```
@NgModule ({  
  imports: [ CommonModule, FormsModule, ... ],  
})
```

NgModule

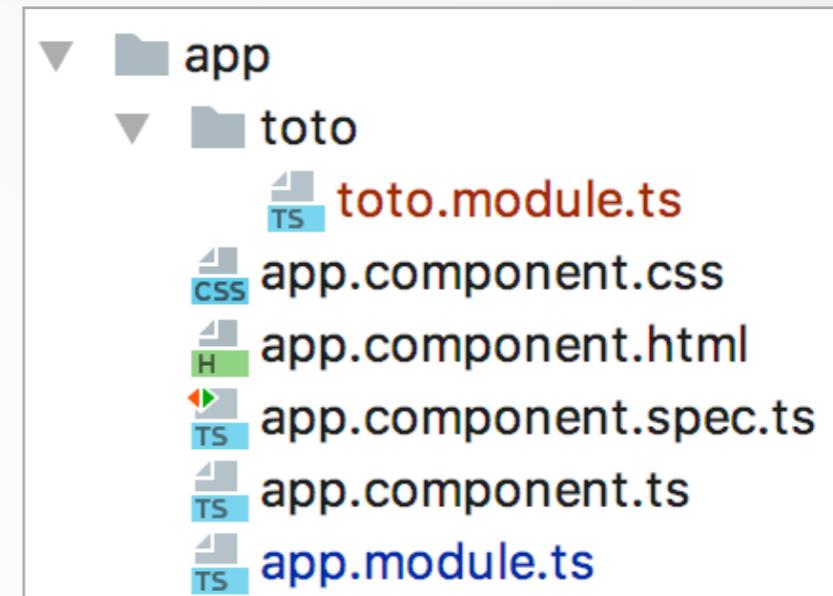
Créer son propre NgModule

```
@NgModule ({  
    // Rend les fonctionnalités fournies par d'autres modules  
    // utilisables dans ce module (ngIf, ngFor, form...) .  
    imports:      [ CommonModule, FormsModule ],  
  
    // Composants, directives et pipes utilisés dans ce module(1)  
    declarations: [ ContactComponent, HighlightDirective, AwesomePipe ],  
  
    // Rend certaines fonctionnalités de ce module  
    // utilisables par les modules qui importeront ce module.  
    exports:      [ ContactComponent ],  
  
    // Services injectables partout dans l'application(2)  
    providers:     [ ContactService, UserService ],  
  
    // Composant(s) à bootstrapper - Uniquement dans le module racine.  
    bootstrap:     [ AppComponent ]  
})  
export class ContactModule { }
```

NgModule avec le CLI

- Générer un NgModule TotoModule :

```
ng generate module toto
```



```
@NgModule({
  imports: [ CommonModule ],
  declarations: []
})
export class TotoModule { }
```

- Options :

ng g m toto	# Syntaxe raccourcie
ng g m foo/toto	# Crée le module ds un sous-rép 'foo'
ng g m toto --flat	# Ne crée pas de répertoire dédié
ng g m toto --routing	# Crée aussi un module de routing

NE PAS CONFONDRE :

Module JS vs Module Angular

	Module JavaScript	Module Angular (NgModule)
Définition	1 module = 1 fichier de code	1 module = 1 classe décorée avec <code>@NgModule</code>
Origine	Langage JavaScript ES6	Framework Angular (registre de code de l'appli)
Utilité	Encapsuler le code, Éviter le scope global	Encapsuler les fonctionnalités, Organiser le code de l'appli
Syntaxe d'export	Mot-clé <code>export</code>	Propriété <code>@NgModule.exports</code>
Syntaxe d'export	Mot-clé <code>import</code>	Propriété <code>@NgModule.imports</code>

Pourquoi la confusion ? À cause du terme “module”, et du fait que les deux types de modules ont une notion d’encapsulation, d’import et d’export.

Root Module vs Feature Module

Root vs Feature Modules

- Toute application Angular possède au moins un module : c'est le **module racine** (*root module*). Il est chargé automatiquement au démarrage de l'application, et il appelé **AppModule** par convention.
- On pourrait mettre tout le code de son application dans AppModule, mais il est recommandé d'organiser son code en **plusieurs modules fonctionnels** (*feature modules*) :
 - 1 module = **1 grande fonctionnalité de l'application** (authentification, back-office, front-office, rubrique d'un site web...).
 - 1 module = **code encapsulé et réutilisable**.
 - 1 module = **code qu'on peut charger à la demande**, quand on en a besoin.
- On peut créer **autant de feature modules** que l'on souhaite.

Charger un module

- Le **module racine AppModule** est **chargé automatiquement** au démarrage de l'application (pendant la phase de *bootstrap*) :

```
// Fichier main.ts
platformBrowserDynamic().bootstrapModule(AppModule);
```

- Tous les **autres modules** (modules Angular natifs, *feature modules...*) doivent être **chargés manuellement**. Pour cela, on les liste dans la propriété **imports** d'un module déjà chargé. Par exemple :

```
@NgModule({
  imports: [ CommonModule, MyModule ],
})
export class AppModule { }
```

- Les modules listés dans les `imports` sont **chargés immédiatement**, au démarrage de l'application. Nous verrons plus tard une autre syntaxe permettant de différer le chargement d'un module au moment où on en a besoin (*lazy-loading*).

Modules et Scope

Scope des declarations

- Dans la propriété `@NgModule.declarations`, on liste tous les **composants**, **directives** et **pipes** qui appartiennent à ce module :

```
@NgModule({  
    imports: [ CommonModule ],  
    declarations: [ MyComponent, MyDirective, MyPipe... ],  
}  
export class MyModule { }
```

- Par défaut, ces composants, directives et pipes ne sont **utilisables que dans le module où ils ont été déclarés**.
- Autrement dit : **les affichables sont scopés à leur module de déclaration**.

Réutiliser un “affichable” dans plusieurs NgModules

- Supposons qu'on ait déclaré TotoComponent dans **ModuleA**, et qu'on veuille aussi l'afficher dans **ModuleB**.
- La mauvaise idée serait de déclarer le même TotoComponent à la fois dans ModuleA et dans ModuleB : un composant donné ne peut être **déclaré que dans UN SEUL MODULE** :

```
// Déclaration initiale
@NgModule({
  declarations: [TotoComponent]
})
export class ModuleA { }
```

```
// NON, car déjà déclaré dans A
@NgModule({
  declarations: [TotoComponent]
})
export class ModuleB { }
```

- La solution est de lister TotoComponent dans les propriétés **declarations** ET **exports** de ModuleA. Puis le ModuleB doit importer le ModuleA :

```
@NgModule({
  declarations: [ TotoComponent ],
  exports: [ TotoComponent ]
})
export class ModuleA { }
```

```
@NgModule({
  imports: [ ModuleA ]
})
export class ModuleB { }
```

Scope des providers

- Dans la propriété `@NgModule.providers`, on liste tous les **providers injectables** (services, valeurs...) qui appartiennent à ce module :

```
@NgModule({  
    imports: [ CommonModule ],  
    providers: [ AuthService, DbService... ],  
})  
export class MyModule { }
```

- Ces providers sont **utilisables partout dans l'application Angular**, c'est à dire dans tous les modules.
- Autrement dit : **les providers ne sont PAS scopés à leur module de déclaration.**

En conséquence...

- Certains NgModules ne doivent être **importés qu'une fois dans toute l'appli**, car ils déclarent des **providers** qui deviennent disponibles dans toute l'application :

```
@NgModule({  
  ...,  
  imports: [ HttpClientModule ],  
  ...  
})  
export class MyModule { }
```

```
@NgModule({  
  ...,  
  providers: [ HttpClient... ],  
  ...  
})  
export class HttpClientModule { }
```

- D'autres NgModules doivent être **importés dans chaque module où on en a besoin**, car ils contiennent des **declarations** exportées :

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

```
@NgModule({  
  ...,  
  declarations: [ NgForm, ... ],  
  exports: [ NgForm, ... ]  
})  
export class FormsModule { }
```

QUIZ #6

Avez-vous compris les modules ?

<https://kahoot.it/>

EXO 10

- Créer un module dédié à l'affichage des quizzes.

6. Routeur

Utiliser, Syntaxes diverses,
Routes d'un *feature module*

Module

Utiliser le routeur

Préparer, Définir les routes, Naviguer

Routeur - Introduction

- Un routeur permet d'**associer une URL à un écran/état de l'application.**
- Le fait d'avoir différentes URLs permet de **bookmarker une page précise**, de l'**envoyer par e-mail**, et cela donne une meilleure expérience utilisateur en général.
- **IMPORTANT.** Le routeur d'Angular gère les routes **côté client**. Même si la barre l'URL donne l'impression que différentes URLs sont requêtées auprès du serveur, en réalité les changements d'URL sont reçus par Angular/index.html, et c'est Angular qui matche le chemin et la route correspondante :

```
http://exemple.com/  
http://exemple.com/accueil  
http://exemple.com/contact  
http://exemple.com/quiz/32  
...
```



```
http://exemple.com/index.html
```

Routeur - Tâche préparatoire

- Définir le **base href**⁽¹⁾ dans index.html :

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

- Le base href doit impérativement se trouver **juste après** la balise <head>.
- Remarque. Le base href est déjà défini si vous avez utilisé Angular CLI pour créer le projet.

Routeur

Déclarer les routes (1/3)

- **Déclarer les routes de l'application**, c. à d. un mapping entre des **chemins** et des **composants** à afficher :

```
@NgModule({  
  imports: [  
    RouterModule.forRoot([  
      { path: '', redirectTo: 'quizzes', pathMatch: 'full' },  
      { path: 'quizzes', component: QuizListComponent },  
      { path: 'quiz/:id', component: QuizDetailComponent },  
      { path: '**', component: PageNotFoundComponent }  
    ]  
  ]  
})  
export class AppModule {}
```



- On peut aussi déclarer les routes dans un **module dédié**, qu'on importe ensuite dans son module de rattachement pour l'activer⁽¹⁾.

Routeur - Syntaxe des routes

- **Pas de slash** au début des paths.
- La propriété **redirectTo** permet de rediriger vers une autre route.
- Le symbole **:id** est un **paramètre de route**. Il peut être utilisé par le composant associé à la route (`HeroDetailComponent`) pour afficher un héro particulier.
- Les ****** représentent le **joker**. Il sera matché si l'URL demandée ne matche aucun autre chemin déclaré. Utile pour implémenter une pseudo page 404. Doit apparaître dans la dernière route déclarée.
- La propriété **data** (pas utilisée dans l'exemple) permet d'associer des données arbitraires à un route (`title`, `breadcrumb`... READ ONLY).

Routeur : Où s'affichent les composants de route ? (2/3)

- Le routeur associe des URLs à des composants. Mais **où ces composants vont-ils s'afficher** dans la page quand l'utilisateur visitera les URLs correspondantes ?
- À l'endroit où vous placerez la directive **<router-outlet>** :

```
{  
  path: 'quizzes',  
  component: QuizListComponent  
}
```



```
<nav class="navbar">  
  <ul>...</ul>  
</nav>  
<router-outlet></router-outlet>  
<footer>  
  ...  
</footer>
```



```
http://example.com/quizzes  
  
<nav class="navbar">  
  <ul>...</ul>  
</nav>  
<router-outlet>  
  <quiz-list></quiz-list>  
</router-outlet>  
<footer>  
  ...  
</footer>
```

Routeur - Naviguer (3/3)

Question :

routerLink ou [routerLink] ?

- Faire des liens dans un **template** grâce à **RouterLink**:

```
<nav>
  <a routerLink="quizzes">Quizzes</a>
  <a [routerLink]=["['quiz', 34]"]>Quiz #34</a>
  !<a href="quizzes">Quizzes</a>
</nav>
<router-outlet></router-outlet>
```

- Naviguer **programmatiquement** grâce à **Router.navigate()** :

```
// <button (click)="goHome () ">Accueil</button>

export class ContactCmp {

  constructor(private router: Router) { }

  goHome () {
    this.router.navigate(['/home']); // Link Params Array
  }
}
```

Erreur fréquente : Oublier de déclarer les composants de route

- Le fait d'afficher un composant via le routeur **ne vous dispense pas de le déclarer** dans le module (dans la propriété @NgModule.declarations).
- Dans cet exemple, les composants affichés par le routeur sont **AUSSI déclarés** dans le module :

```
@NgModule({  
  imports: [  
    RouterModule.forRoot([  
      { path: 'heroes', component: HeroListComponent },  
      { path: 'hero/:id', component: HeroDetailComponent },  
    ])  
,  
  declarations: [ HeroListComponent, HeroDetailComponent ]  
})  
export class AppModule {}
```

Routes Syntaxes diverses

Ordre des routes, Liens absolus et relatifs,
Routes avec paramètre(s), Routes imbriquées

Routeur - Ordre des routes

- Les routes sont matchées dans l'ordre où elles sont déclarées. **La première route qui matche gagne.**
- **Erreur fréquente #1 :**

```
[  
  { path: 'quiz/:id', component: QuizDetailComponent },  
  { path: 'quiz/list', component: QuizListComponent },  
]
```

- **Erreur fréquente #2 :**

```
// app.module.ts  
imports: [  
  RouterModule.forRoot([  
    { path: 'quizzes', component: HeroListComponent },  
    { path: 'quiz/:id', component: HeroDetailComponent },  
    { path: '**', component: PageNotFoundComponent }  
  ]),  
  AdminModule // Contient aussi des routes  
]
```

Routeur

Liens absolus et relatifs

- **Liens absolus** : Partent de la racine de l'application. Doivent **commencer par un slash**, qu'on utilise la syntaxe *string* ou *link parameters array* :

```
// String  
<a routerLink="/heroes">Héros</a>  
// Link params array  
this.router.navigate(['/heroes']);
```

- **Liens relatifs** : Construits relativement au chemin du composant où ils apparaissent. Ne doivent **JAMAIS commencer par un slash**.

```
// Template de HeroComponent qui possède le chemin /heroes  
<a routerLink="38">GO</a> /heroes/38  
<a routerLink="38/edit">GO</a> /heroes/38/edit  
<a routerLink=". ">GO</a> /heroes  
<a routerLink="..">GO</a> / (racine du site)
```

- **Lien relatif dans le code :**

```
this.router.navigate(['..'], { id: crisisId }), { relativeTo: this.route });
```

Routeur

Route avec paramètre(s)

- Les paramètres permettent d'avoir une **page “dynamique”** : le contenu affiché par le composant peut être adapté en fonction de la valeur du(des) paramètre(s).
- **Déclarer** une route avec paramètres :

```
{  
  path: 'user/:userId/messages/: messageId',  
  component: UserMessageComponent  
}
```

- **Faire un lien** vers une route avec paramètres :

```
// Dans un template :  
<a [routerLink]=['user', user.id, 'messages', message.id]>  
  Voir le message  
</a>  
// Dans une classe :  
this.router.navigate(['user', user.id, 'messages', message.id]);
```

Routeur

Récupérer les params de route

- Récupérer les paramètres de route grâce à

`ActivatedRoute.snapshot.paramMap` :

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

export class UserMessageComponent {
  constructor(private route: ActivatedRoute) { }
  ngOnInit() {
    const params = this.route.snapshot.paramMap;
    const userId = params.get('userId');
    const messageId = params.get('messageId');
  }
}
```

- Version asynchrone.** Il existe un **observable** `ActivatedRoute.paramMap` pour quand le(s) paramètre(s) change(nt) mais pas l'URL. Exemple :

```
{
  path: 'photo/:photoId',
  component: PhotoComponent
}
```

http://example.com/photo/:photoId
http://example.com/photo/34
http://example.com/photo/35
...

Routeur - Routes imbriquées

- Une route peut avoir des **sous-routes**, c. à d. des composants qui s'afficheront **à l'intérieur** du composant de la route en cours.
- On déclare les sous-routes avec la propriété `children`. Chaque sous-route possède un `path` et un `component` :

```
{  
  path: 'heroes',  
  component: HeroComponent,  
  children: [  
    { path: 'list', component: HeroesListComponent },  
    { path: ':id', component: HeroDetailComponent }  
  ]  
}
```

- Le **chemin** d'une sous-route est la **concaténation du path parent + path enfant** (dans l'exemple : `heroes/:id`). Le **composant** enfant s'affiche dans le `<router-outlet></router-outlet>` du composant parent (`HeroComponent` dans l'exemple).

Routes enfant - Schéma

<http://example.com/heroes/list>

HeroComponent

<router-outlet>

HeroesListComponent

</router-outlet>

<http://example.com/heroes/34>

HeroComponent

<router-outlet>

HeroDetailComponent

</router-outlet>

Routeur - Lien en surbrillance

- La directive routerLinkActive permet d'**ajouter une ou plusieurs classe(s) CSS à un lien** lorsque sa route devient active :

```
<a routerLink="/user/bob" routerLinkActive="active-link">Bob</a>
```

- Dans l'exemple ci-dessus, **la classe active-link est ajoutée à la balise <a>** quand l'URL est /user ou /user/bob. Pour matcher l'intégralité de l'URL (vs une URL partielle), passer l'option {exact:true} :

```
<a routerLink="/user/bob" routerLinkActive="active-link" [routerLinkActiveOptions]="{exact:true}">Bob</a>
```

Routes d'un *feature module*

Déclaration, Activation

RFM - Déclaration

- Un *feature module* peut posséder **ses propres routes et sous-routes**.
- Ces routes sont déclarées de la **même manière que celles du module racine** à **une différence près**, **forChild** au lieu de **forRoot** :

```
@NgModule ({  
    imports: [  
        RouterModule.forChild([  
            { path: 'quizzes', component: QuizListComponent },  
            { path: 'quiz/:id', component: QuizDetailComponent }  
        ])  
    ]  
})  
export class QuizModule {}
```

- Ces routes seront affichées dans la directive <router-outlet> du module dans lequel elles sont importées.

RFM - Activation

- Par défaut, les routes d'un *feature module* viennent **s'ajouter aux routes du module principal** lorsque le *feature module* est importé - **PAS D'IMBRICATION⁽¹⁾** :

```
@NgModule({  
    imports: [ BrowserModule, routing, QuizModule ],  
    declarations: [ AppComponent ],  
    bootstrap: [ AppComponent ]  
})  
export class AppModule { }
```

- Alternativement**, dans les routes du module principal, on peut utiliser la propriété **loadChildren** pour : 1) venir **accrocher sous un chemin existant** les routes d'un *feature module*, et 2) **charger le feature module à la demande**, uniquement quand son path est requis⁽²⁾ :

```
{  
    path: 'path-to-quiz',  
    loadChildren: () => import('./quiz/quiz.module').then(m => m.QuizModule)  
}  
  
// Chemins finaux  
/path-to-quiz/quizzes  
/path-to-quiz/quiz/:id
```

RFM - Erreur fréquente

- Pour charger un *feature module*, vous avez donc le choix entre 2 options :
 - SOIT vous **importez** le feature module dans un autre. Le feature module est **chargé au démarrage de l'application** et les routes qu'il contient s'ajoutent à celles de l'application.
 - SOIT vous **lazy-loadez** le feature module en l'associant à une route d'un module existant. Le feature module ne sera chargé que **la première fois qu'une de ses routes sera consultée**, ses routes seront **préfixées** par le chemin de lazy-loading, et les fonctionnalités qu'il contient ne seront dispo qu'une fois le module chargé.
- C'est l'un ou l'autre. Soyez vigilant à ne pas faire les deux.

QUIZ #7

Avez-vous compris le routeur ?

<https://kahoot.it/>

EXO 11

- Créer les routes de l'application

7. HTTP

Module

Copyright 2016-2019 - Toute reproduction interdite

HTTP - Introduction

HTTP - Intro

- Une grande partie du développement d'applications web consiste à **envoyer/recevoir des données** vers/depuis un serveur grâce à des **requêtes HTTP**.
- Vous pouvez **utiliser la technologie de votre choix** pour faire ces requêtes : axios, XMLHttpRequest, ou la récente API fetch.
- Mais Angular fournit un **service HttpClient** avec plusieurs avantages :
 - Expose toutes les méthodes HTTP sous une **API facile à utiliser**.
 - Préconfiguré pour travailler avec des **données JSON** (assez répandues).
 - Les **réponses HTTP** sont renvoyées sous forme d'**observables**, parfaitement adaptés pour gérer l'asynchronicité et transformer les données.
 - Adapté aux **tests unitaires**, car permet de **bouchonner** le serveur, et de retourner des réponses prédéfinies.

HTTP - Tâche préalable

- Pour pouvoir utiliser le **service HttpClient** dans votre application, il faut importer le **module HttpClientModule** dans l'un de vos modules (en général, AppModule). Par exemple :

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

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

- ⚠ Puisque HttpClientModule contient un *provider* et que les providers sont GLOBAUX, le module HttpClientModule ne doit être **importé qu'une fois** pour que le provider soit utilisable **partout** dans l'application.

HTTP - Remarques

- Le service HttpClient réalise des **requêtes** avec XMLHttpRequest.
- HttpClient propose des **méthodes correspondant au verbes HTTP** courants :

```
http.get()          // SELECT
http.post()         // INSERT
http.put()          // UPDATE
http.delete()        // DELETE
...
// Méthode de base
http.request()
```

- ! Toutes ces méthodes retournent un **Observable**. Il faut s'y abonner pour en extraire la réponse :

// NON

```
const data = http.get('/api');
```

// OUI

```
http.get('/api').subscribe(data => {
    // Ici, utiliser data
});
```

Un mot sur les observables

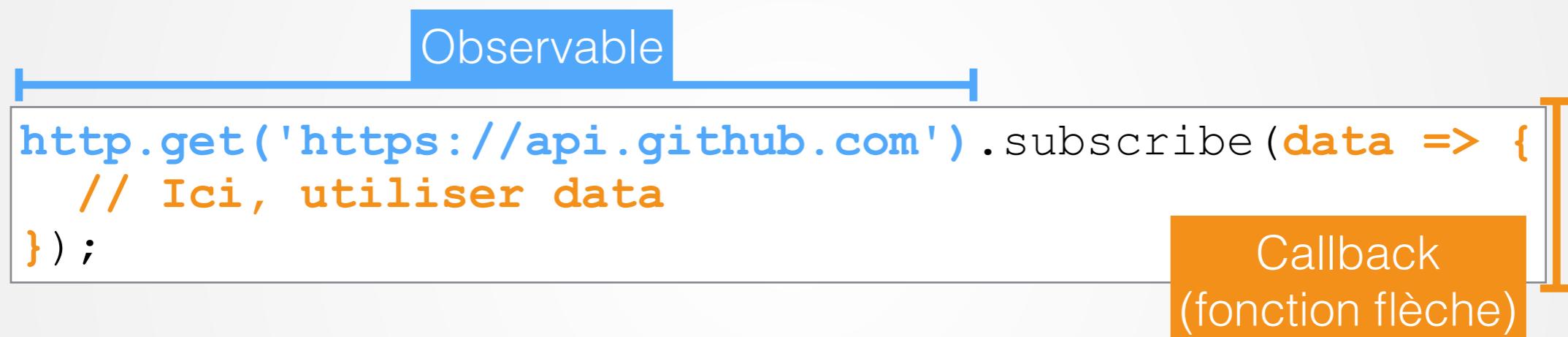
- Les observables représentent une **valeur ou une série de valeurs asynchrones**, c. à d. qui seront disponibles dans le futur mais ON NE SAIT PAS QUAND.
- Pour récupérer ces valeurs, on s'**abonne** à l'observable avec **Observable.subscribe(callback)** :

```
const obs = Observable.of(1, 2, 3);
obs.subscribe(val => console.log(val));
```

- La **fonction de callback** est appelée **pour chaque valeur** émise par l'observable et **reçoit la valeur courante** en argument. Dans l'exemple ci-dessus, la fonction est donc appelée trois fois, et reçoit respectivement les valeurs **1**, puis **2**, puis **3**.

Observables et HTTP

- Dans Angular, une requête HTTP est représentée par un observable qui ne renvoie qu'**une seule valeur**, les données de la réponse HTTP (en effet, UNE requête HTTP ne peut pas renvoyer PLUSIEURS réponses) :



- IMPORTANT.** On ne sait pas QUAND la fonction de callback sera appelée (ça dépend de la latence du serveur, de la connexion Internet...).
- On pourrait utiliser une écriture **plus explicite**, mais **moins compacte** :

```
const obs = http.get('https://api.github.com');  
variable intermédiaire  
  
obs.subscribe(function(data) {  
  // Ici, utiliser data  
});  
fonction flèche
```

HTTP GET

HTTP - Faire une requête GET et afficher les données reçues

avec subscribe()

```
@Component ({  
    template: `'  
        <ul>  
            <li *ngFor="let q of qList">  
                {{q.title}}  
            </li>  
        </ul>`  
    })  
export class QuizListComponent {  
    // ! undefined  
    qList: Quiz[]; 1  
  
    constructor(  
        private http: HttpClient 2  
    ) {}  
  
    ngOnInit() {  
        this.http.get('/api/quizzes')  
            .subscribe(data =>  
                this.qList = data; 3  
            ) 4  
    }  
}
```

HTTP - Faire une requête GET et afficher les données reçues

avec `async`

```
@Component ({  
    template: `  
        <ul>  
            <li *ngFor="let q of qList$ | async">  
                {{q.title}}  
            </li>  
        </ul>`  
    } )  
export class QuizListComponent {  
    // ! undefined  
    qList$: Observable<Quiz[]>; 1  
  
    constructor(  
        private http: HttpClient 2  
    ) {}  
  
    ngOnInit() {  
        this.qList$ = 3  
            this.http.get('/api/quizzes');  
    }  
}
```

4

1

2

3

HTTP - Syntaxes diverses

- **Récupérer des données JSON⁽¹⁾ :**

```
export class QuizListComponent {  
  constructor(http: HttpClient) {  
    http.get('/api/quizzes') .subscribe(data => { ... });  
  }  
}
```

- **Associer un type** aux données récupérées (**NE TRANSFORME PAS LES DONNÉES**) :

```
http.get<Quiz[]>('/api/quizzes')  
  .subscribe(data => // data est de type Quiz[] // );
```

- Récupérer la **réponse entière** :

```
http.get('/api/quizzes', {observe: 'response'})  
  .subscribe(resp => {});
```

- Gérer les **erreurs** :

```
http.get<Quiz[]>('/api/quizzes')  
  .subscribe(  
    // Le 1er callback est le callback de succès  
    data => { ... },  
    // Le 2e callback est le callback d'erreur  
    err => { console.log('Problème'); }  
  );
```

HTTP POST

HTTP - Requête POST

- **Envoyer des données** au serveur :

```
const quiz = {title: 'Quiz Angular'};  
  
http  
  .post('/api/quizzes/add', quiz)  
⚠️ .subscribe(...); // Ne pas oublier
```

- Configurer les **headers** :

```
http  
  .post('/api/quizzes/add', quiz, {  
    headers: new HttpHeaders().set('Authorization', 'my-auth-token')  
  })  
  .subscribe();
```

- Configurer les **paramètres d'URL** :

```
// Requête envoyée à /api/quizzes/add?id=3  
http  
  .post('/api/quizzes/add', quiz, {  
    params: new HttpParams().set('id', '3')  
  })  
  .subscribe();
```

HTTP

Remarques diverses

HTTP et Observables

- Un observable n'est exécuté **que si on s'y abonne** :

```
// Cette requête n'est JAMAIS exécutée  
this.http.get('api/quizzes');  
  
// Celle-là est bien exécutée  
this.http.get('api/quizzes') .subscribe();
```

- Un observable est exécuté **autant de fois qu'on s'y abonne** :

```
// Requête HTTP exécutée deux fois  
const obs = this.http.get('api/quizzes');  
obs .subscribe();  
obs .subscribe();
```

- C'est valable aussi pour le **pipe async**⁽¹⁾ :

```
<p>{ { (user$ async) ?.name } }</p>  
<p>{ { (user$ async) ?.email } }</p>
```

Solution pour async

- Le code suivant permet d'éviter de répéter le pipe async (il faut être dans un *ngIf) :

```
<ng-container *ngIf="user$ | async as user">
  <p>{{ user.name }}</p>
  <p>{{ user.email }}</p>
</ng-container>
```

HTTP - Transformer les données reçues

- **Tâche fréquente** : Transformer les données renvoyées par l'observable HTTP pour qu'elles aient le format attendu par l'application, grâce à la méthode `obs.pipe()`.
- **Exemple** : Récupérer uniquement la propriété `race.name` alors que le serveur renvoie un **tableau d'objets race entiers** :

```
import { map } from 'rxjs/operators'; (1)

http.get(`baseUrl/api/races`)
  .pipe(
    map(races: any[]) => races.map(race => race.name) )
  ) (2)
  .subscribe(names => {
    console.log(names);
  });
} );
```

- Puisque les requêtes HTTP renvoient un observable, on peut utiliser tous les **opérateurs de transformation** applicables aux observables.

HTTP - Utiliser un service !

- **Attention !** Pour faire une requête HTTP, depuis un composant, on peut être tenté d'injecter le `HttpClient` directement dans le composant :

```
@Component(...)
export class QuizListComponent {
  constructor(private http: HttpClient) { }
  ngOnInit() {
    this.http.get('api/quizzes')
      .subscribe(...)
  }
}
```

- Mais c'est mieux de passer par un **service intermédiaire**, qui **devra renvoyer des observables** (impossible de renvoyer les données) :

```
@Injectable(...)
export class QuizService {
  constructor(private http: HttpClient) { }
  loadQuizzes(): Observable<Quiz[]> {
    return this.http.get('api/quizzes');
  }
}
```

EXO 12

- Refactoriser le `QuizService` pour utiliser une API REST.

8. Formulaires

Créer et valider un formulaire

Module

Copyright 2016-2019 - Toute reproduction interdite

Formulaires - Intro

Activation, Présentation des 2 syntaxes
Points commun aux 2 syntaxes

Formulaires - Introduction

- Les formulaires sont **au cœur de toute application web ou mobile** :
 - **Moteur de recherche**.
 - **Récolter ou modifier des données** venant d'une base de données.
 - Permettre à l'utilisateur d'**interagir avec l'application** : déclencher des actions, changer des réglages...
- **Problématiques compliquées** :
 - **Valider** les saisies utilisateur ;
 - Afficher les **erreurs** ;
 - Formulaires **dynamiques** (champs répétables, champs qui dépendent d'un autre champ...) ;
 - **Tests unitaires** de formulaire...

Formulaires - 2 syntaxes

- Angular proposer **deux syntaxes** pour gérer les formulaires :
 - Formulaires classiques.** Syntaxe dite **piloté par le template** (*Template-driven Forms*) — Tout est dans le template, rien dans le code :

```
<input name="contactName" [(ngModel)]="contact.name">
```
 - Formulaires réactifs.** Syntaxe dite **piloté par le modèle** (*Model-driven Forms*) : La syntaxe est à la fois dans le template ET dans le code (voir plus loin).
- La syntaxe classique est utilisée uniquement pour les formulaires simples. Nous ne l'aborderons pas ici pour nous concentrer sur les formulaires réactifs, beaucoup plus puissants⁽¹⁾.

Formulaires - Tâche préalable

- Pour pouvoir utiliser les **syntaxes de formulaire** dans votre application, il faut **importer** le module **ReactiveFormsModule** dans l'un de vos modules (syntaxe “piloté par le modèle”⁽¹⁾).

Par exemple :

```
import { ReactiveFormsModule } from '@angular/forms';

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

- ⚠ Puisque `ReactiveFormsModule` contient des `@NgModule.declarations` qui sont exportées (propriété `@NgModule.exports`), ce module doit être **importé dans chaque module** où les syntaxes de formulaire vont être utilisées. En effet, les composants/directives ne sont disponibles que dans leur module de déclaration et les modules où ils ont été importés.

Formulaires réactifs

Template, Modèle,
Récupération des données

Lecon
Copyright 2016-2019 - Toute reproduction interdite

Formulaire réactif

Introduction

- La syntaxe des formulaires réactif est **plus verbeuse** que la syntaxe classique : il faut mettre du code à la fois dans la **classe** et dans le **template** pour que le formulaire fonctionne.
- Mais elle offre **plus de possibilités** :
 - Ajouter, modifier, retirer des **fonctions de validation à la volée**.
 - Générer et modifier le formulaire **dynamiquement**.
 - Tester la logique de validation grâce à des **tests unitaires**.

Formulaire réactif

Vue d'ensemble

```
@Component({
  template: `
    <!-- ICI, FORMULAIRE AVEC SYNTAXE ANGULAR -->
    <form>
      <input type="text" name="contactName">
      <button type="submit">Soumettre</button>
    </form>`
})
export class FormComponent {
  // Initialise les données.
  // Crée un modèle du formulaire.
  // Déclare les méthodes pour traiter le formulaire.
}
```

- Dans le **template**, on trouve les **formulaire HTML classique** (`<form>`, `<input>`...) avec des **syntaxes Angular** (non représentés ici) et du **code pour afficher les erreurs**.
- Dans la **classe**, on trouve les **données** utilisées dans le formulaire, le **modèle** du formulaire (ensemble d'objets `FormGroup` et `FormControl` permettant à Angular de se représenter le formulaire programmatiquement), et les **méthodes** nécessaires au fonctionnement du formulaire (par exemple pour enregistrer les données sur le backend).

Formulaire réactif - Template

1. Partir d'un **formulaire HTML classique**, puis ajouter les **syntaxes Angular** :
2. Ajouter la directive **formGroup** sur la balise <form>. Elle est bindée à une propriété de la classe et représente l'ensemble du formulaire.
3. Ajouter la directive **formControlName** sur chaque champ. Le nom donné à chaque champ devra se retrouver dans le modèle du formulaire.
4. La directive **(ngSubmit)** permet de **réagir à la soumission du formulaire** en appelant une méthode déclarée dans la classe.

1

2

4

```
<form [formGroup]="contactForm" (ngSubmit)="saveContact()">
    Prénom : <input formControlName="firstname">
    Nom : 3 <input formControlName="lastname">
    Rue : <textarea formControlName="street"></textarea>
    Ville : <input formControlName="city">
    <button type="submit">Submit</button>
</form>
```

Formulaire réactif - Classe 1/2

```
@Component({ ... })
export class FormComponent implements OnInit {

  contactForm: FormGroup;

  constructor(private fb: FormBuilder) { }

  ngOnInit() {
    this.contactForm = this.fb.group({
      firstname: ['James', Validators.required],
      lastname: ['Bond', Validators.required],
      street: [],
      city: []
    });
  }

  saveContact() {
    // Enregistre le contact sur le backend
  }
}
```

Formulaire réactif - Classe 2/2

- Le **modèle** du formulaire se composant de :
 - Une instance de **FormGroup** représentant l'**ensemble du formulaire**.
 - Une instance de **FormControl** pour chaque champ.
- La commande **FormBuilder.group()** offre une **syntaxe raccourcie** pour créer ce modèle :

```
myForm = FormBuilder.group({  
    champ1: [defaultValue1, Validateurs],  
    champ2: [defaultValue2, Validateurs],  
    ...  
});
```

- Les **chaînes** désignant les champs dans le modèle (**champ1**, **champ2**...) doivent **correspondre aux formControlNames** utilisés dans le template.

Formulaire réactif

Accéder aux champs individuels

- Dans le composant, on n'a qu'une seule variable qui représente l'**ensemble du formulaire** : `contactForm` (instance de FormGroup).
- Pourtant, il est nécessaire d'accéder aux champs individuels pour :
 - Lire leur valeur
 - Écrire leur valeur
 - Tester leur validité
- On accède à **un champ précis** (FormControl) avec la syntaxe `FORM_VAR.get('NOM_DU_CHAMP')`, par exemple :

```
// Instance de FormControl
this.contactForm.get('firstName');
```

- On peut ensuite accéder aux **différentes propriétés** du champ :

```
this.contactForm.get('firstName').value
this.contactForm.get('firstName').valid
```

Formulaire réactif

Récupérer les valeurs

- Récupérer LES valeurs de **tout le formulaire** d'un coup :

```
@Component(...)  
export class FormComponent {  
  // ...  
  saveContact() {  
    const allValues = this.contactForm.value;  
  }  
  // ...  
}
```

propriétés de l'objet
= champs du formulaire

```
{  
  firstname: 'James',  
  lastname: 'Bond',  
  street: 'Sesame St',  
  city: 'Washington'  
}
```

- Récupérer LA valeur d'**un champ précis** :

```
saveContact() {  
  const firstname = this.contactForm.get('firstname').value;  
}
```

'James'

Formulaire réactif

Modifier les valeurs

- **OPTION 1** - Définir les valeurs à l'**initialisation** du formulaire :

```
ngOnInit() {  
    this.contactForm = this.fb.group({  
        firstname: ['James', Validators.required],  
        lastname: ['Bond', Validators.required],  
        ...  
    });  
}
```

- **ATTENTION.** Si les valeurs initiales viennent du backend, elles seront obtenues de manière **asynchrone**. Le formulaire risque donc d'être affiché AVANT que le code d'initialisation n'ait pu être exécuté —> **ERREUR !**
- **OPTION 2** - Modifier les valeurs **a posteriori** :

```
this.contactForm = this.fb.group({  
    firstname: [],  
    lastname: []  
});  
this.contactForm.get('firstname').setValue('Bob');
```

EXO 13 - Partie I

- Créer le formulaire de création/édition d'un quiz.
- Vérifier que vous arrivez à définir les **valeurs initiales** du formulaire.
- Vérifier que vous arrivez à récupérer les **valeurs saisies par l'utilisateur** lors du clic sur Enregistrer.

Formulaire réactif

Valider les champs

- **PRÉREQUIS.** Pour valider les champs d'un formulaire, il faut leur avoir associé **un ou plusieurs validateurs** dans le modèle du formulaire :

```
this.contactForm = this.fb.group({  
    firstname: ['James', [Validators.required, Validators.maxLength(50)]],  
    lastname: ['Bond', Validators.required],  
    ...  
});
```

- **NB. Les validateurs n'ont aucun effet en eux-mêmes.** Il appartient au développeur de tester la présence d'erreurs de validation via l'API :

```
this.contactForm.get('firstName').invalid  
this.contactForm.get('firstName').hasError('maxlength')
```

- Voir d'**empêcher l'utilisateur de soumettre** le formulaire en cas d'erreur :

```
<form [formGroup]="contactForm" (ngSubmit)="saveContact()">  
    <button type="submit" [disabled]="contactForm.invalid">  
        Submit  
    </button>  
</form>
```

Exemple : Afficher les erreurs dans le template

- L'API de validation peut être utilisée **directement dans les templates**. La validation se fait **en temps réel** (= recalculée dès que l'utilisateur modifie un champ) et donc l'affichage des erreurs également.
- Dans l'exemple suivant, on affiche un message d'erreur dans le template si le champ **firstname** du formulaire **contactForm** a été modifié par l'utilisateur (**dirty**) et qu'il est invalide (**invalid**) et qu'il a l'erreur **required** :

```
<div *ngIf="contactForm.get('firstname').dirty  
          && contactForm.get('firstname').invalid">  
  <div *ngIf="contactForm.get('firstname').hasError('required')">  
    Le nom est obligatoire.  
  </div>  
</div>
```

- **Remarque** : L'API propose des options pour **différer la validation** à la perte de focus d'un champ (blur) ou à la soumission du formulaire (submit) :

```
const control = new FormControl('', { updateOn: 'blur' });
```

```
const control = new FormControl('', { updateOn: 'submit' });
```

Formulaire réactif - Validateurs

- Angular fournit nativement les **validateurs usuels** :
 - **Validators.required** - Champ doit être non vide.
 - **Validators.email()** - Champ doit contenir un e-mail valide.
 - **Validators.pattern(regex)** - Champ doit matcher l'expression régulière.
 - ...
- Les validateurs peuvent porter sur un FormControl (= un champ individuel) ou sur un FormGroup (= un formulaire / groupe de champs).
- Il est possible (et facile) de créer des **validateurs custom**.

Réagir à la soumission d'un formulaire

- **ATTENTION.** Contrairement aux formulaires HTML classiques, **aucune action par défaut** n'est associée aux formulaires Angular (ni HTTP GET, ni HTTP POST). Il appartient au développeur de réagir à l'événement **ngSubmit**.

```
// TEMPLATE
<form [formGroup]="contactForm" (ngSubmit)="saveContact()">...</form>
```

```
export class FormComponent {
  contactForm: FormGroup;

  constructor(private http: HttpClient) { }

  ...
  saveContact() {
    // Récupère toutes les valeurs
    const values = this.contactForm.value;
    // Enregistre le contact sur le backend
    this.http.post('https://api.mysite.com/', values)
      .subscribe();
  }
}
```

⚠ Exemple simplifié
HttpClient → service
raw data → modèle

FormGroup et FormControl

Propriétés

Les instances de FormGroup et FormControl (autrement dit : le formulaire et ses champs) exposent des **propriétés** très utiles :

	Description
value	Valeur courante du champ ou du groupe de champs (Si groupe, valeur de type {user: 'toto', pwd: '1234'})
valid invalid	true/false selon que le champ/groupe est valide ou pas.
pristine dirty	true/false selon que le champ/groupe est intact ou pas (i.e. que l'utilisateur a modifié sa valeur).
errors	Objet représentant les erreurs de validation du champ/groupe, par exemple : {required: true}

FormGroup et FormControl

Méthodes

Et aussi des **méthodes** qui permettent de **manipuler un formulaire programmatiquement** :

	Description
<code>get(path)</code>	Récupère un contrôle via son groupe parent : <code>let control = this.form.get('person.name');</code>
<code>setValue(val)</code>	Définit la valeur d'un champ/groupe. Si groupe, passer un objet dont les clés matchent STRICTEMENT la structure du groupe.
<code>patchValue(val)</code>	Modifie la valeur d'un champ/groupe. Si groupe, passer un objet dont les clés matchent PARTIELLEMENT la structure du groupe.
<code>reset(val?)</code>	Ré-initialise un champ/groupe. Valeur(s) passée(s) à null + Champ/groupe marqué pristine.
<code>addControl()</code> <code>removeControl()</code>	Ajoute ou retire un contrôle à un groupe. NB. Dispo uniquement sur l'objet FormGroup.

Formulaires réactifs

Résumé des syntaxes

Syntaxe “piloté par le modèle”

Formulaires réactifs

Module Angular
à importer

Syntaxe <form>

Syntaxe champ

Valeur
d'un champ

Validité
d'un champ

Adapté pour

ReactiveFormsModule

```
<form [formGroup]="contactForm">
```

```
    <input formControlName="firstName">
```

```
        contactForm  
        .get('firstName') .value
```

```
        contactForm  
        .get('firstName') .valid
```

Formulaire complexe
Validation complexe

QUIZ #8

Avez-vous compris les formulaires ?

<https://kahoot.it/>

EXO 13 - Partie II

- Ajouter des validateurs + affichage des erreurs
- Relier le formulaire au backend pour qu'il soit possible de créer ou modifier des quizzes.

9. Tests et déploiement

Module

Copyright 2016-2019 - Toute reproduction interdite

Tests - Intro

Tests - Introduction

- **Tests unitaires :**
 - **OBJECTIF** : Tester qu'une petite portion de code (un composant, un service, un pipe) fonctionne correctement en isolation, c'est à dire indépendamment de ses dépendances.
 - **MÉTHODE** : Exécuter chacune des méthodes d'un composant/service/pipe, et vérifier que les sorties sont celles attendues pour les entrées fournies.
- **Tests end-to-end (“de bout en bout”)** :
 - **Objectif** : Tester que l'application a le fonctionnement attendu en émulant une interaction utilisateur.
 - **MÉTHODE** : Démarrer une vraie instance de l'application, et piloter le navigateur pour saisir des valeurs dans les champs, cliquer sur les boutons, etc... On vérifie ensuite que la page affichée contient ce qui est attendu, que l'URL est correcte, etc.

Tests - Outils

- **Jasmine** - Fournit les **commandes pour écrire les tests**. Contient un *test runner* HTML qui exécute les tests dans le navigateur.
- **Utilitaires de test Angular** - Permettent de **configurer programmatiquement un environnement de test** dans lequel exécuter notre code et contrôler l'application en cours de test.
- **Karma** - Permet d'exécuter les tests pendant qu'on développe l'application. Peut être intégré à la chaîne de développement/déploiement.
- **Protractor** - Permet d'exécuter les tests end-to-end (e2e).

Tests - Mise en place

- La mise en place d'un environnement de test est **fastidieuse**.
- Votre meilleure option : **partir d'un environnement préconfiguré**.
 - Avec le [Quickstart officiel](#).
 - Avec [Angular CLI](#).
- Les deux installent les paquets npm, les fichiers, et les scripts nécessaires à l'écriture et l'exécution des tests.

Tests unitaires

Tester une petite portion de code

Tests unitaires

- Vérifient une petite portion de code en isolation.
- **Avantages :**
 - Très **rapides**.
 - Très **efficaces** pour tester (quasiment) l'intégralité du code.
- **Concept d'isolation** : pour éviter que le test soit biaisé par ses dépendances, on utilise généralement des **objets bouchonnés** (mock) comme dépendances. Ce sont des objets factices créés juste pour les besoins du test.
- **Outils** pour les tests unitaires :
 - **Jasmine** : bibliothèque pour écrire des tests.
 - **Karma** : permet d'exécuter les tests dans un ou plusieurs navigateurs.

Tests - Conventions

- Nommez vos fichiers de tests unitaires écrits en Jasmine avec l'**extension .spec.ts**.
- Placez chaque fichier de test **à côté du code qu'il teste**, en reprenant le même nom. Exemple : à la racine de app, créez un fichier **app.component.spec.ts** pour tester `app.component.ts`.
- **Respecter ces conventions garantit que :**
 - Vos tests seront **détectés automatiquement** par les *test runners* (même si la détection des tests est paramétrable).
 - Vous penserez à **mettre à jour le test** lorsque vous changerez le code testé.

Tests - Exécution

- Pour **exécuter vos tests** (c'est à dire le contenu de vos fichiers `.spec.ts`), exéutez la commande suivante dans le terminal⁽¹⁾ :

ng test
- Les tests sont exécutés **en mode watch** : à chaque changement du code, les tests sont ré-exécutés automatiquement.
- Les **résultats** des tests sont affichés à **deux endroits** :
 - Dans l'**instance de navigateur** lancée automatiquement par `ng test` (laissez-la ouverte).
 - Dans la **console** où vous avez exécuté `ng test`.

Tests - Syntaxe générale

```
// Définit un jeu de test
describe('Jeu de tests #1', () => {
    // Phase 1 - Configure l'environnement de test (setup)
    beforeEach(()) => {
        // Crée un module pour les tests
        // Crée un composant à tester
        // Récupère une instance de service injecté dans le composant
        // Fait des requêtes sur le template du composant
        // ...
    }) ;

    // Phase 2 - Tests proprement dit
    it('Test 1', () => {
        ...
    }) ;
    it('Test 2', () => {
        ...
    }) ;
}) ;
```

Tests - Setup pour tester le code simple

- Code simple = sans interaction avec Angular (services, pipes) :
- On crée des "tests unitaires isolés".
- Faciles à écrire, comme pour du code JavaScript "normal".

Tests - Setup pour tester le code complexe (composants...)

- **TestBed** - Utilitaire de test Angular :
 - **TestBed.configureTestingModuleTestingModule()** - Crée impérativement un **module** Angular (contient déjà les composants, directives et providers les plus courants).
 - **TestBed.createComponent(MyComponent)** - Crée impérativement un **composant** Angular.
- **Fixture** (valeur renvoyée par `createComponent()`) - Environnement de test qui wrappe le composant testé :
 - **fixture.componentInstance** - **CLASSE du composant testé** (permet d'accéder aux propriétés et méthodes du composant).
 - **fixture.debugElement** - **TEMPLATE du composant testé** (permet de tester le HTML).
 - Récupérer une référence à l'**élément DOM natif** d'une balise :
`fixture.debugElement.query(By.css('h1')).nativeElement`
 - Récupère une instance d'un **service injecté** :
`fixture.debugElement.injector.get(MyService)`

Tests - Écriture des tests proprement dit

- Chaque test est wrappé dans un **bloc it** qui contient des **assertions expect** :

```
it('Nom du test', () => {
    expect(el.textContent).toEqual('');
});
```

- **Le test est valide si l'assertion est valide.**
- **Exemples** d'assertion :

```
expect(userService.isLoggedIn).toBe(true);
expect(el.textContent).toEqual('');
expect(el.textContent).toContain('test title');
expect(el.textContent).not.toContain('Welcome', 'not welcomed');
expect(links.length).toBe(3, 'should have 3 links');
```

Tests - Détection de changement

- Dans une vraie appli, la détection de changement est **déclenchée automatiquement**.
- Dans les tests unitaires, chaque test doit la **déclencher explicitement** avec `fixture.detectChanges()`. Par exemple :

```
it('should display original title', () => {
  fixture.detectChanges();
  expect(el.textContent).toContain(comp.title);
});
```

- Il est toujours possible de déclencher la détection de changement automatiquement lors de la configuration du TestBed avec AutoDetect⁽¹⁾ :

```
TestBed.configureTestingModule({
  providers: [
    { provide: ComponentFixtureAutoDetect, useValue: true }
  ]
})
```

Tests - Stubs

- Les “stubs” ou “mocks” sont de **fausses versions** d’un service ou d’un composant utilisées pendant les tests.
- Les stubs permettent de **simplifier les tests**, de les rendre **plus prévisibles** et d'**éviter les effets de bord**. Exemple : stub simulant une fausse requête HTTP, stub simulant un login utilisateur...
- L’utilisation du stub se fait lors de la création du module de test :

```
// Service stub
StubService = {
    // Code du service stub
    // Typiquement : Propriétés et méthodes avec des valeurs en dur
};
// Lors de la création du module, on fait pointer le vrai service sur le stub
 TestBed.configureTestingModule({
    ...
    providers: [ provide: MyService, useValue: StubService } ],
    ...
});
```

Tests - Aller plus loin

- Les tests sont un vaste sujet et la nature des éléments testés influence la syntaxe des tests.
- Consultez la doc officielle testing (très bien faite) :
 - Tester un composant qui utilise un service async
 - Tester un composant qui possède un template externe
 - Tester un composant avec des inputs/outputs
 - Tester un composant associé au routeur
 - Tests unitaires isolés (services, pipes...)
 - Etc.

Tests end-to-end

Tester l'appli dans son ensemble

Tests e2e - Introduction

- Consistent à **lancer réellement l'appli dans un navigateur** et à **simuler l'interaction d'un utilisateur** (clic sur les boutons, saisie de formulaires, etc.).
- **INCONVÉNIENT** : Certes, ils permettent de tester l'application à fond mais sont **bien plus lents** (plusieurs secondes par test).
- Les tests e2e s'appuient sur un outil appelé **Protractor**. On écrit la suite de tests avec Jasmine comme pour un test unitaire, mais on utilise l'API de Protractor pour interagir avec l'application.

Test e2e - Exemple

```
describe('Home', () => {
  it('should display title, tagline and logo', () => {
    browser.get('/');
    expect(element.all(by.css('img')).count()).toEqual(1);
    expect($('h1').getText()).toContain('PonyRacer');
    expect($('small').getText()).toBe('Always a pleasure to bet on
ponies');
  });
});
```

EXO 17

- Tests unitaires

Déploiement

Vue d'ensemble, Prérequis serveur et prérequis Angular,
Builder son code, Compilation AOT

Préparer l'application à la mise en production = COMPLEXE !

- **Compilation Ahead-of-Time (AOT)** : Pré-compiler les **templates** de composant Angular.
- **Bundling** : **Concaténer** les modules JavaScript en un seul fichier (*bundle*).
- **Inlining** : Placer le **HTML et le CSS des composants** dans le code du composant (vs. dans des fichiers externes).
- **Minification** : **Réduire la taille des fichiers** en retirant les espaces, les retours chariot, les commentaires, les symboles optionnels.
- **Uglification** : **Ré-écrire le code** pour utiliser des noms de variables et de fonctions courts et opaques.
- **Élimination du code mort** : **Supprimer** les modules et le code non-utilisés.
- **Librairies allégées** : **Abandonner** les librairies non utilisées, ou **créer une version allégée** ne contenant que les fonctionnalités utilisées.

Déploiement

Vue d'ensemble

1. **Générer le build** de développement :
`ng build`
2. **Copier sur le serveur** tous les fichiers générés par ng build (ils sont dans le répertoire `dist/` par défaut).
3. Si votre appli est servie depuis un sous-répertoire,
définir un `base href` adapté :
`ng build --base-href=/my/app/`
4. Configurer le serveur pour **rediriger toutes les requêtes** pour des fichiers manquants vers `index.html`.

Environnements (dev, prod)

- Le CLI supporte plusieurs “environnements”, c. à. d. des paramètres différents pour le **dev**, la **prod**, etc.
- **Étape 1 - Définir les environnements :**
 - Paramètres de chaque environnement à définir dans les fichiers `src/environments/environment.NAME.ts`.
 - Dans le code, **TOUJOURS importer les paramètres depuis fichier de base** (`environments/environment.ts`).
- **Étape 2 - Utiliser l'environnement de son choix** pour servir ou builder le projet :

```
ng serve --env=prod  
ng build --env=prod
```

Optimiser le build pour la prod

- Il est possible de générer un build **optimisé pour la prod** avec la commande :

```
ng build --prod
```

Le flag **--prod** active les optimisations suivantes : compilation AOT + environnement de prod + bundling + minification + uglification + élimination du code mort.

- La compilation AOT (*Ahead of Time*) **pré-compile les composants et leurs templates lors du build** (vs lors de l'exécution, ce que fait la compilation par défaut appelée JIT, *Just In Time*).
 - Les composants **s'exécutent immédiatement**, ils ne doivent plus être compilés côté client.
 - Les templates sont embeddés dans le code du composant correspondant, **pas de requête HTTP supplémentaire**.
 - Le compilateur Angular ne doit pas être distribué** avec l'application (le client télécharge donc moins de code).
 - Le compilateur peut **supprimer les directives non utilisées**.

Redirection des requêtes

- Pour héberger l'application Angular (fichiers .html, .js et .css), un **serveur statique suffit**, puisque les pages sont générées **côté client**. Exemples : Amazon S3, Apache, IIS, CDN...
- Si votre appli utilise le routeur, votre serveur doit être configuré pour **systématiquement retourner la page hôte de l'application**⁽¹⁾, **index.html**. (Le serveur de développement le fait déjà pour nous !)
Exemple Nginx :

```
try_files $uri $uri/ /index.html;
```
- **Remarque sur CORS** : En production, les requêtes HTTP émises par l'appli peuvent produire des erreurs *cross-origin resource sharing* (ou CORS), car le serveur hébergeant l'API/le backend diffère du serveur hébergeant l'appli (.html, .js...). **Angular n'y peut rien**, **CORS s'active côté serveur** : enable-cors.org.

Déploiement - Backend

- **PEU IMPORTE le stack utilisé pour le back-end :**
 - **Langage** : Java, PHP, Python, Node.js...
 - **Base de données** : MongoDB, MySQL, SQL Server, Amazon SimpleDB...
- Les **2 gros points de contact entre le back-end et Angular** (front-end) sont :
 - **Authentification**.
 - **Endpoints** pour accéder aux données (API REST) ou déclencher un traitement.
- **Choisissez votre backend en fonction de :**
 - Technologies maîtrisées par VOTRE équipe.
 - Existence de librairies/helpers pour gérer l'authentification, exposer une BDD via une API REST, etc. Tous les principaux langages/frameworks back possèdent de telles librairies.
 - Des **plateformes cloud** peuvent aider : Kinvey, Firebase, Mlab, Back& (spécialisé Angular)...

Considérations déploiement

- **Le “starter” utilisé** pour démarrer le projet conditionne les **options disponibles pour le déploiement**. Gardez-le en tête.
 - angular-cli utilise **webpack** (déploiement clé-en-main, mais config inaccessible).
 - angular2-webpack-starter utilise **webpack** (déploiement plus manuel, mais config accessible).
 - angular2-seed utilise **SystemJS builder**.
- **Arbitrage** entre le côté prêt-à-l’emploi (angular-cli) et la flexibilité (angular2-webpack-starter).

EXO 19

- Démo Déploiement.

10. Observables & RxJS [OPTIONNEL]

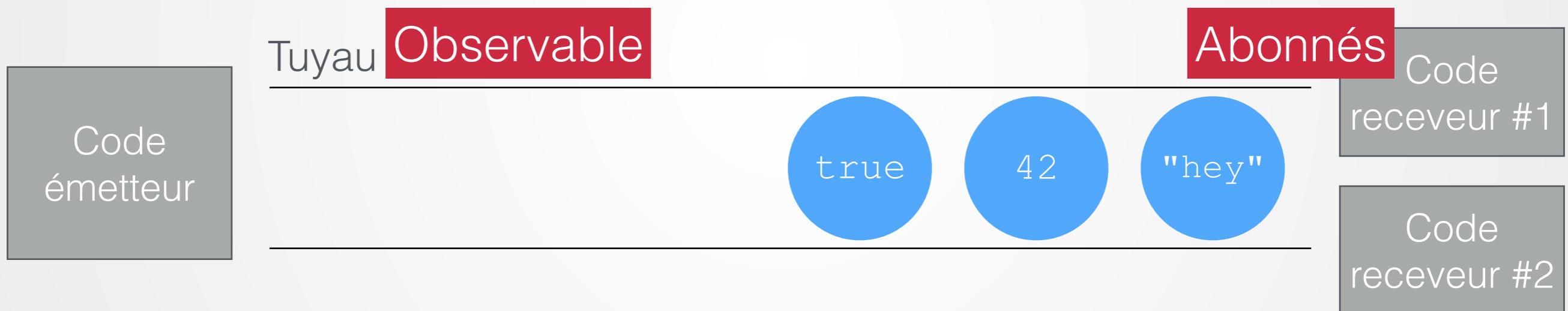
Programmation réactive,
Librairie RxJS, Opérateurs RxJS

Module

Programmation réactive

Programmation Réactive - Intro

- Consiste à construire son code à partir de **flux de données**.
- Un flux de données est un comme un **tuyau**. Typiquement, certaines parties du code **envoient** des données dans le tuyau, et d'autres **surveillent** les données qui circulent dans le tuyau :



- Le code émetteur peut envoyer un **nombre illimité de valeurs** dans le tuyau.
- Le tuyau peut avoir un **nombre illimité d'abonnés**. Le tuyau est actif tant qu'il a **au moins un abonné**.

Temps réel + Cross-applicatif

- Les tuyaux sont des flux **en temps réel** : dès qu'une valeur est poussée dans un tuyau, les abonnés reçoivent la valeur et peuvent réagir.
- **Analogie avec un binding** : au lieu de binder un template aux propriétés de la classe, *on binde une partie de l'application aux données émises par une autre partie de l'application.*
- Ces bindings **temps réel** et **cross-applicatifs** sont donc particulièrement **adaptés pour implémenter certaines problématiques typiques** des applis SPA :
 - **Attendre le résultat d'une opération asynchrone** (requête HTTP, action de l'utilisateur comme un clic sur modale de confirmation ou champ de formulaire).
 - **Rafraîchir une partie de l'interface** quand une action se produit **dans une autre partie** (ex : rafraîchir le nombre d'articles dans un panier quand le bouton “ajouter au panier” est cliqué).
 - **Rafraîchir l'interface** dès qu'une donnée est disponible sur le **serveur** (ex : message reçu dans un tchat).

Transformations

- En programmation réactive, les données qui circulent dans le tuyau peuvent être facilement **transformées** grâce à une **série d'opérations successives** (aka “opérateurs”) :



- On **déclare** les transformations successives à appliquer aux données du tuyau **une bonne fois pour toutes**.
- À chaque fois qu'une nouvelle donnée est envoyée dans le tuyau, **elle passe par toutes les transformations**, et les abonnés reçoivent toujours la **donnée transformée**.
- Ce fonctionnement **déclaratif** est pratique à utiliser et à débogguer.

RxJS

Librairie ReactiveX pour JavaScript
Observable, Observer, Subject

Lecon

Copyright 2016-2019 - Toute reproduction interdite

RxJS

- La programmation réactive est **formalisée par une API** : **ReactiveX** (<http://reactivex.io/>). On dit aussi parfois que ReactiveX est un “pattern” ou une “librairie”.
- Il existe des **implémentations de ReactiveX pour les principaux langages de programmation**. L’implémentation pour **JavaScript** s’appelle **RxJS**.
- **ATTENTION.** Angular utilise **RxJS 5**. Or, les recherches Google vous renvoient souvent sur la doc de RxJS 4...
- **Quelques URLs utiles :**
 - Doc de RxJS 5 - <http://reactivex.io/rxjs/>
 - Github de RxJS 5 - <https://github.com/ReactiveX/rxjs>
 - Migrating from RxJS 4 to 5 - <https://github.com/ReactiveX/rxjs/blob/master/MIGRATION.md> (il y a des différences d’API entre les 2 versions).

Observable - Création (1/3)

- Angular crée/renvoie des **observables à plusieurs endroits** :
 - **Requêtes HTTP** : Le résultat d'une requête `http.request()` est wrappé dans un observable.
 - **Changements de valeur d'un champ de formulaire** : Les valeurs successives du champ sont émises via un observable (propriété `monChamp.valueChanges`).
 - **Changements de valeur d'un paramètre d'URL** : Les valeurs successives du paramètre sont émises via un observable (propriété `ActivatedRoute paramMap`).
 - **@Output()** : Les événements émis avec `EventEmitter` utilisent les observables en coulisse.

Observable - Crédit (2/3)

- On peut aussi créer des observables grâce à des **opérateurs de création** tels que **from()**, **of()** ... :

```
// from() permet de convertir presque tout en observable  
const obs = Observable.from([10, 20, 30]);
```

```
// of() permet de passer une liste de valeurs à émettre  
const obs = Observable.of(1, 2, 3);
```

```
// fromEvent() permet de convertir un evenmt DOM en observable  
const inputElement = document.getElementById('my-input');  
const obs = Observable.fromEvent(inputElement, 'keyup')
```

- Liste des opérateurs de création : <http://reactivex.io/rxjs/manual/overview.html#creation-operators>

Observable - Cr ation (3/3)

- Enfin, la m thode **Observable.create()** permet de cr er son propre Observable **manuellement**.
- Elle prend en argument une fonction qui re oit un **observer** permettant d' mettre les 3 vnements pertinents dans la vie d'un observable : **next** (valeur suivante), **error** (erreur), et **complete** (terminaison).

```
const observable = Observable.create(observer => {  
    observer.next(1);  
    observer.next(2);  
    observer.next(3);  
    setTimeout(() => {  
        observer.next(4);  
        observer.complete();  
    }, 1000);  
    // observer.error('Oops, I did it again...');  
});
```

Observable - Abonnement

- Pour **récupérer les valeurs d'un observable**, on DOIT s'y abonner avec la méthode **Observable.subscribe()** :

```
console.log('just before subscribe'); S'abonner = Invoquer l'observable
observable.subscribe({
  next: x => console.log('got value ' + x),
  error: err => console.error('something wrong occurred: ' + err),
  complete: () => console.log('done'),
});
console.log('just after subscribe');
```

- Ce qui va afficher dans la console (pour l'observable du slide précédent) :

```
just before subscribe
got value 1
got value 2
got value 3
just after subscribe
got value 4
done
```

Observable - Exécution

- Le code à l'intérieur de Observable.create((observer) => {...}) représente l' **“exécution de l'Observable”**, c'est à dire le **traitement déclenché lazily et uniquement pour chaque Observer qui s'abonne.**
- Cette exécution renvoie **plusieurs valeurs sur une période de temps**, de manière synchrone ou asynchrone. On parle souvent de **flux** (feed) pour la désigner.
- L'exécution d'un Observable peut émettre **trois types de valeurs** :
 - **Notification “Next”** : une vraie valeur, telle qu'une chaîne, un nombre, un objet... Syntaxe : **observer.next(valeur)** .
 - **Notification “Error”** : une erreur JavaScript ou une exception. Syntaxe : **observer.error(error)** .
 - **Notification “Complete”** : un événement sans valeur, qui indique la terminaison de l'Observable. Syntaxe : **observer.complete()** .
- Les **notifications Next** sont les plus importantes ; elles représentent les **valeurs transmises à l'Observer**. Les **notifications Error** et **Complete** ne peuvent se produire qu'**une fois** lors de l'exécution (soit l'une, soit l'autre).

Observer

- L'Observer est le **consommateur des valeurs émises par l'Observable**.
- Un Observer est juste un **ensemble de callbacks**, un pour chaque type de notification renvoyée par l'Observable, **next**, **error** et **complete**⁽¹⁾ :

```
var observer = {  
  next: x => console.log('Observer got a next value: ' + x),  
  error: err => console.error('Observer got an error: ' + err),  
  complete: () => console.log('Observer got a complete notification'),  
};
```

- Pour utiliser l'Observer, on le passe à la méthode **subscribe**⁽²⁾ :

```
observable.subscribe(observer);
```

Observables & Composants

- Dans une appli Angular, il est fréquent de **s'abonner à un observable depuis un composant**. Exemple : quand on fait une requête HTTP pour récupérer des données à afficher dans la page.
- Les composants sont créés et détruits *automatiquement* par Angular, mais **pas les abonnements aux observables**, ce qui peut créer des **fuites mémoire**⁽¹⁾.
- Pour les éviter, pensez à vous **désabonner** de vos observables **à la destruction** du composant grâce au hook `ngOnDestroy()`⁽²⁾ :

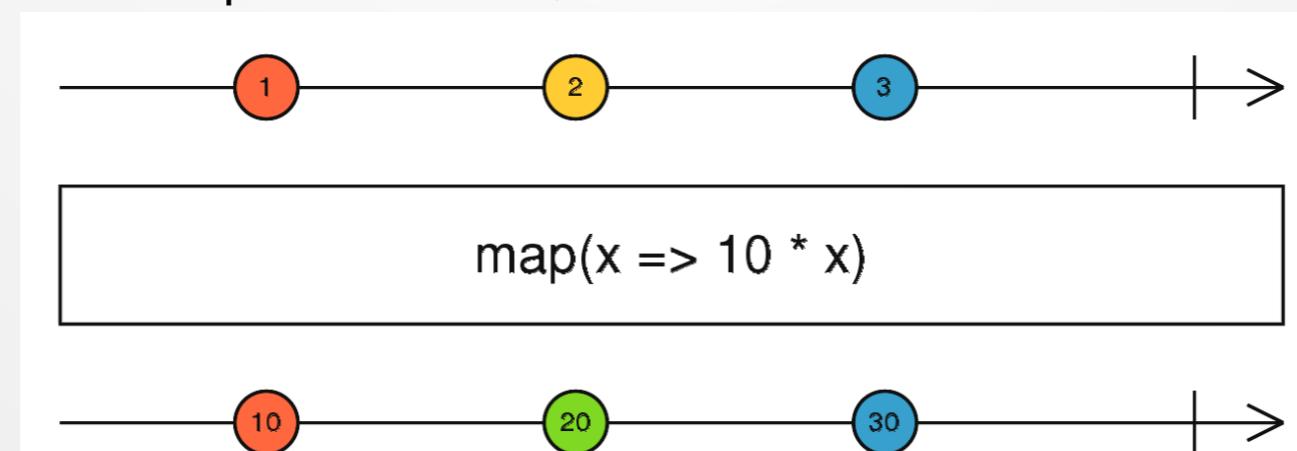
```
export class AppComponent {  
  ngOnInit() {  
    this.subscription = Observable.of([1,2,3]).subscribe();  
  }  
  ngOnDestroy() {  
    this.subscription.unsubscribe();  
  }  
}
```

Quelques opérateurs RxJS

map, mergeMap, reduce, filter...

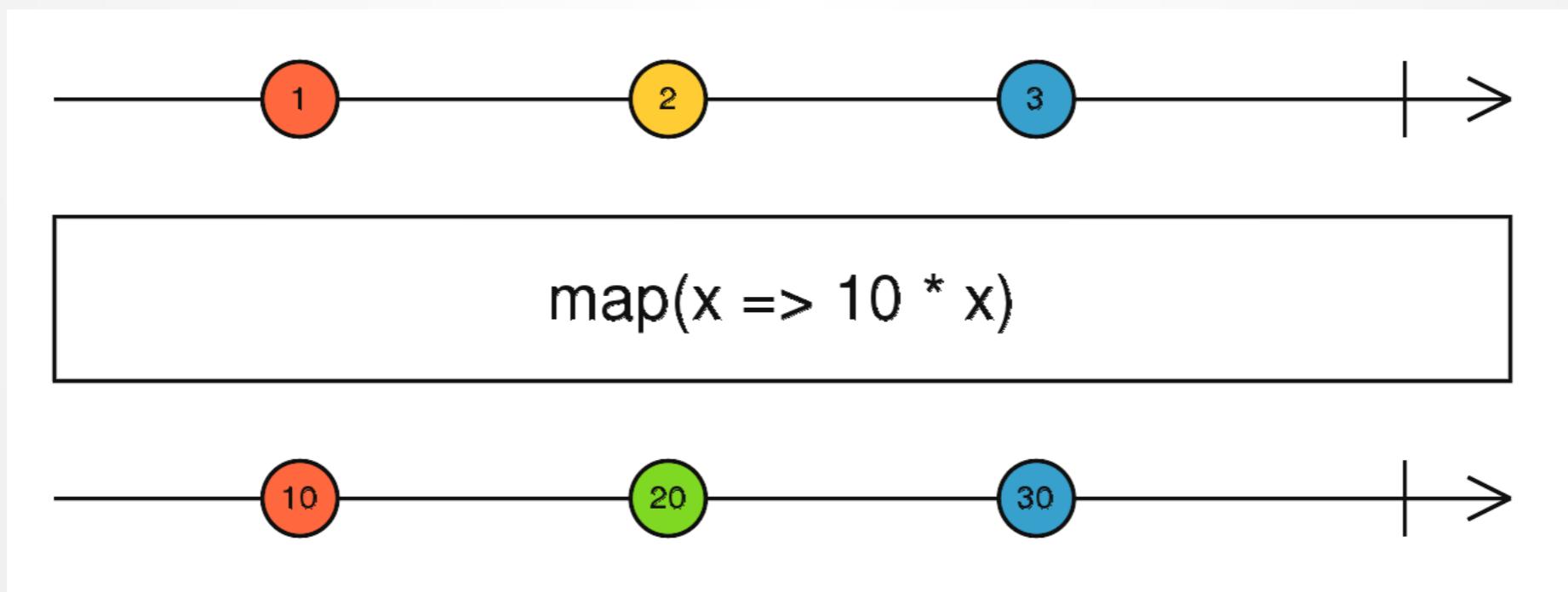
Opérateurs

- RxJS est surtout utile pour ses **opérateurs**.
- Les opérateurs sont des **méthodes sur l'objet** Observable telles que `map()`, `filter()`, `merge()` qui reçoivent l'Observable source et renvoient un nouvel Observable transformé (**NB. L'Observable source n'est pas modifié**).
- Les opérateurs ont **plusieurs utilités** : **créer** un Observable, **transformer** un Observable, **filtrer** un Observable, **combiner** des Observables, etc.
- Pour expliquer les opérateurs, la doc utilise des **marble diagrams**⁽¹⁾:



Opérateur - map

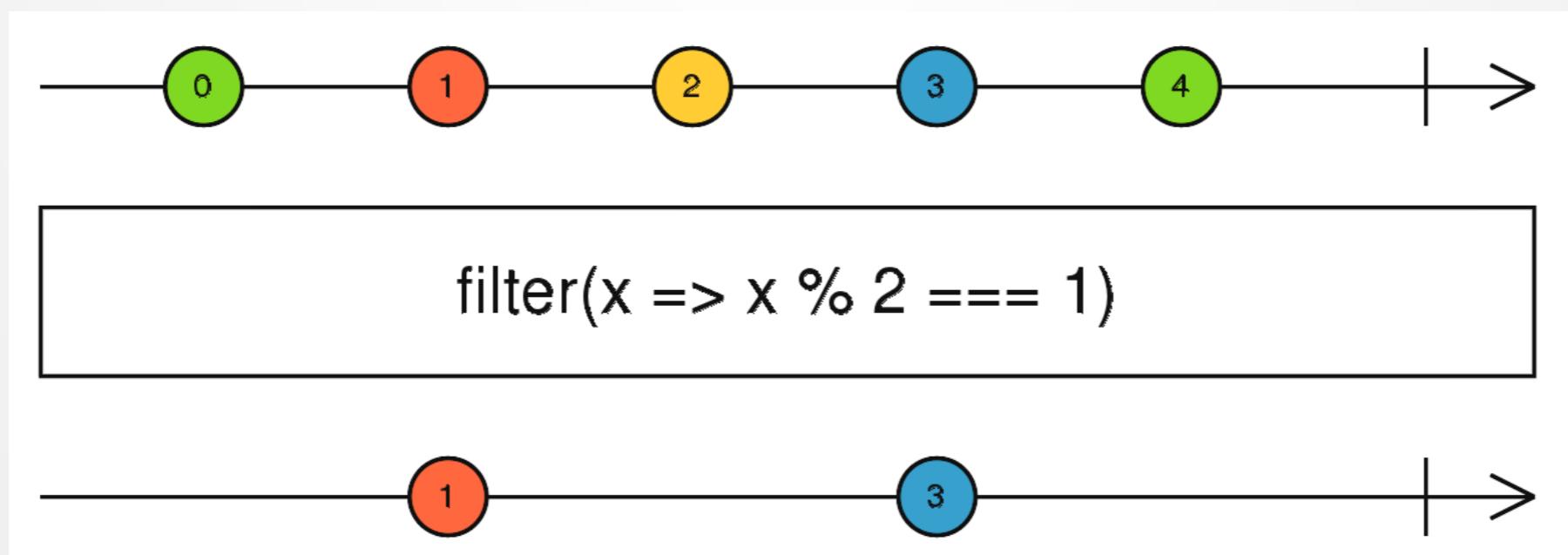
- Opérateur de **transformation**.
- Applique une fonction à chaque valeur émise par l'Observable source, et émet les valeurs transformées sous forme d'un nouvel Observable.



- Analogue à `Array.prototype.map`.

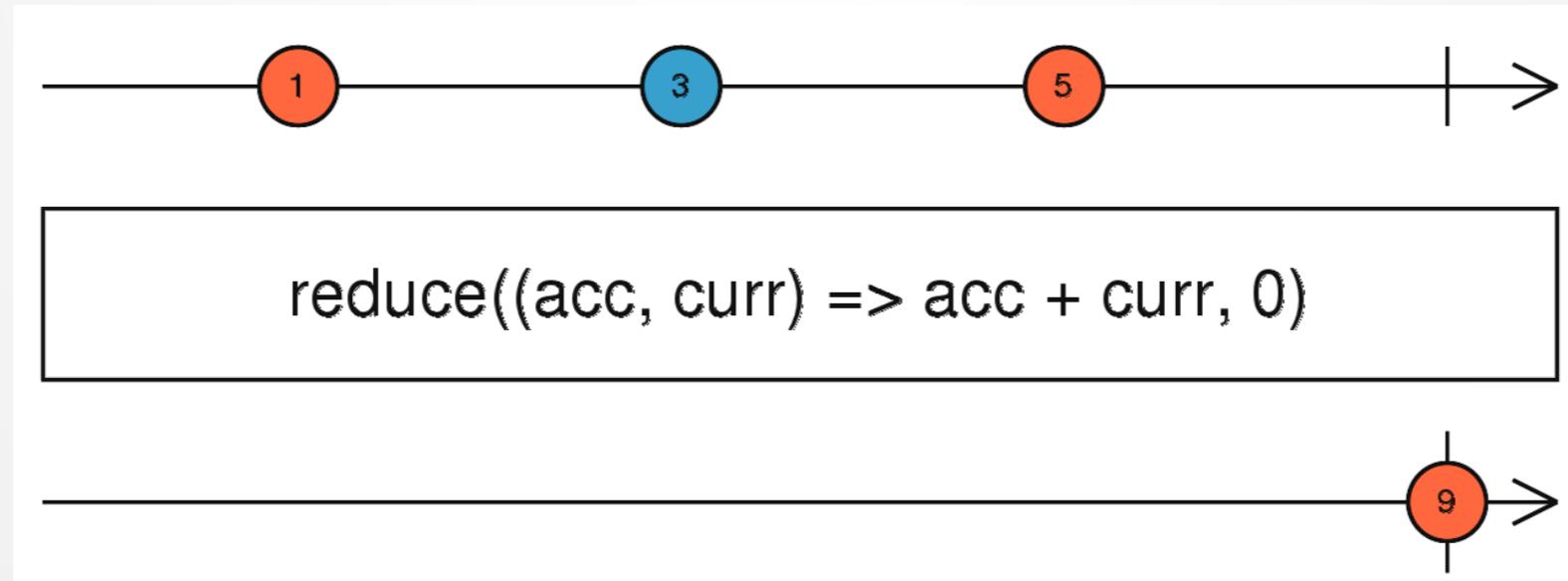
Opérateur - filter

- Opérateur de **filtrage**.
- Filtre les valeurs émises par l'Observable source en ne gardant que celles qui passent un certain critère.



Opérateur - reduce

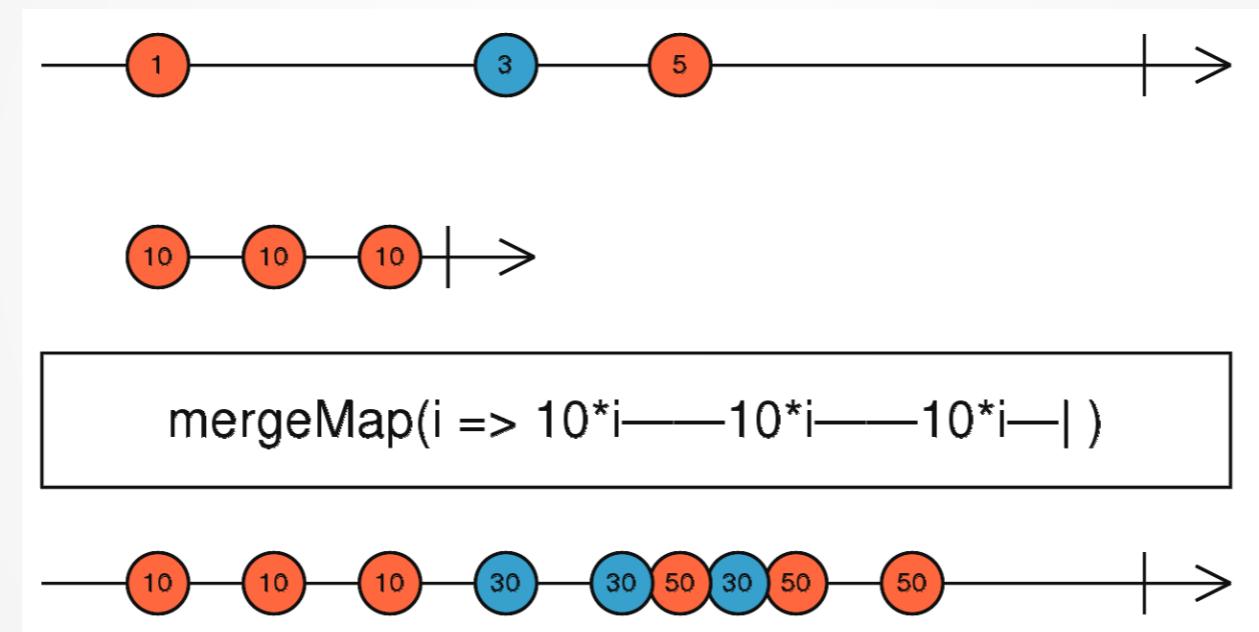
- Opérateur de **transformation**.
- Passe la première valeur de l'Observable source à une fonction d'accumulation, puis passe le résultat de cette fonction ainsi que la 2ème valeur à l'accumulateur, et ainsi de suite jusqu'à ce qu'il ne reste plus qu'une seule valeur. Seule la valeur finale est émise par l'Observable de sortie.



- L'opérateur scan est analogue, mais émet l'accumulation en cours à chaque fois que la source émet une valeur.

Opérateur - mergeMap

- Opérateur de **transformation**.
- Projette chaque valeur source dans un Observable qui est ensuite fusionné dans l'Observable de sortie.



- Cas d'usage typique : une API web renvoie une liste d'ids pour lesquels on veut requêter l'entité correspondante.
- Remarque : concatMap () est analogue mais fusionne les Observables projetés en préservant l'ordre des valeurs source.

Opérateur - Chaînage

- Puisque chaque opérateur renvoie un nouvel Observable, il est très fréquent d'appliquer plusieurs opérateurs à la suite **sous forme chaînée** :

```
Observable.range(1, 5)
    .map(x => x * 2)
    .filter(x => x > 5)
    .subscribe(
        x => console.log(x),
        error => console.log(error),
        () => console.log('fini')
    );
// Affichera : 6, 8, 10, fini
```

Opérateurs RxJS & Angular

- Angular expose une **version allégée de Observable** dans le module rxjs/Observable, dans laquelle de nombreux opérateurs sont **absents**.
- Ces opérateurs peuvent être importés **un par un**, manuellement :

```
// Opérateurs de création (statiques)
import 'rxjs/add/observable/of';
import 'rxjs/add/observable/from';

// Opérateurs de transformation
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/filter';
import 'rxjs/add/operator/toPromise';
```

- Ou ils peuvent être ajoutés **tous ensemble**, d'un coup, mais **attention à la taille du bundle final lors du déploiement** :

```
import 'rxjs/Rx';
```

EXO 14

- Comprendre les Observables.

11. Fonctionnalités avancées [OPTIONNEL]

Routeur avancé, Formulaires avancés,
Affichage avancé

Routeur avancé

Empêcher d'accéder à ou de quitter une route,
Précharger les données

Gardes - Intro

- Dans une application Angular, par défaut, **tout le monde peut accéder à tous les chemins de l'application.**
- Ce n'est **pas toujours souhaitable** :
 - Peut-être l'utilisateur doit-il **être identifié** pour accéder au composant cible ?
 - Peut-être faut-il **récupérer certaines données** avant d'afficher le composant cible ?
 - Peut-être faut-il **sauvegarder les changements en cours** avant de quitter un composant ?
 - Ou demander à l'utilisateur si on peut abandonner les changements en cours plutôt que de les sauvegarder ?

Gardes - Fonctionnement

- Un “garde” permet de **contrôler le comportement du routeur** :
 - S'il renvoie **true**, la navigation **se poursuit**.
 - S'il renvoie **false**, la navigation **est interrompue** (l'utilisateur reste sur la même page).
 - Il peut aussi **rediriger l'utilisateur** vers une autre page.
- La valeur renvoyée par le garde est **très souvent asynchrone** :
 - Attente que l'utilisateur réponde à une question.
 - Attente que le serveur renvoie des données.
 - Attente que les changements soient enregistrés sur le serveur.
- Le garde peut donc renvoyer un booléen, un **Observable<boolean>** ou une **Promise<boolean>**, et le routeur attendra que ces derniers soient dénoués à true ou false.

CanActivate

Empêcher d'accéder à une route

1. **Créer un service** qui implémente l'interface CanActivate :

```
canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot  
): Observable<boolean> | Promise<boolean> | boolean {  
    // Utilise route.params.id (par exemple)  
    // Renvoie true ou false  
}
```

2. Ajouter ce service à la propriété canActivate de la route à protéger (cette propriété contient un **tableau** !) :

```
[  
  { path: 'team/:id', component: TeamCmp, canActivate: [canActivateTeam] }  
]
```

CanDeactivate

Empêcher de quitter une route

1. **Créer un service** qui implémente l'interface CanDeactivate :

```
canDeactivate(component: TeamComponent, route: ActivatedRouteSnapshot,  
state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {  
  // Utilise route.params.id (par exemple)  
  // Renvoie true ou false  
}
```

2. Ajouter ce service à la propriété canDeactivate de la route à protéger (cette propriété contient un **tableau** !) :

```
[  
  {path: 'team/:id', component: TeamCmp, canDeactivate: [CanDeactivateTeam]}  
]
```

Resolve

Précharger des données (1/2)

1. **Créer un service** qui implémente l'interface Resolve :

```
resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot  
): Observable<any> | Promise<any> | any {  
    // Utilise route.params.id (par exemple)  
    // Renvoie true ou false  
}
```

2. Ajouter ce service à la propriété `resolve` de la route à protéger (cette propriété contient un **objet dont la clé permettra de récupérer la valeur du resolve**) :

```
{  
    path: 'team/:id',  
    component: TeamCmp,  
    resolve: {  
        team: TeamResolver  
    }  
}
```

Resolve

Précharger des données (2/2)

- Ajouter ce service aux providers du module adéquat :

```
@NgModule ({  
    providers: [ TeamResolver ]  
})  
export class MyModule { }
```

- Dans le composant associé à la route, récupérer la valeur du resolve dans ActivatedRoute.data :

```
constructor(private route: ActivatedRoute) {}  
  
ngOnInit() {  
    this.route.data.subscribe(data => {  
        this.team = data.team;  
    });  
}
```

EXO 15

- Protéger les routes de l'application et précharger les données.

Formulaires avancés

Champ répété,
Validateur custom, Observer les changements

Champ répété

Côté composant (1/2)

- Un champ répété est modélisé comme un **FormArray contenant une série de FormGroupS**.
- Imaginons un **formulaire de contact qui peut contenir plusieurs adresses** :
 - **1) Initialisation du FormArray :**

```
this.contactForm = this.fb.group({  
    name: '',  
    addresses: this.fb.array([ this.initAddress() ])  
})  
// this.initAddress() renvoie un FormGroup  
// contenant 2 champs : street et postcode
```

- **2) Manipulation du FormArray :**

- Ajouter un élément : FormArray.push()

```
this.contactForm.get('addresses').push(...)
```

- Retirer un élément : FormArray.removeAt()

```
this.contactForm.get('addresses').removeAt(...)
```

Champ répété

Côté template (1/2)

```
<div formArrayName="addresses">
  <div *ngFor="let address of contactForm.get('addresses').controls;
let i=index">
    Address {{i + 1}}
    <span *ngIf="contactForm.get('addresses').controls.length > 1"
      (click)="removeAddress(i)">
      X
    </span>
    <div [formGroupName]="i">
      <input type="text" formControlName="street">
      <input type="text" formControlName="postcode">
    </div>
  </div>
</div>
```

Validateur custom (1/2)

- Lorsque les validateurs natifs d'Angular ne suffisent pas, on peut **créer ses propres validateurs et appliquer ses propres règles métier**.
- **Exemples** : Vérifier qu'un nom d'utilisateur est encore disponible ; Vérifier qu'un numéro de commande a le bon format...
- **Syntaxe** : Un validateur custom est une simple fonction qui reçoit un **Control/ControlGroup** en paramètre, et qui renvoie **null** si la valeur est valide, ou un objet dont les clés représentent les identifiants d'erreur.
- Angular distingue les validateurs **synchrones** et **asynchrones** :

```
const control = new FormControl(formState, validator, asyncValidator);
```

Validateur custom (2/2)

```
// Validateur custom pour un champ
const control = fb.control('', [Validators.required, isOldEnough]);

function isOldEnough(control: Control) {
  let birthDatePlus18 = new Date(control.value);
  birthDatePlus18.setFullYear(birthDatePlus18.getFullYear() + 18);
  return birthDatePlus18 < new Date() ? null : {tooYoung: true};
}
```

```
// Validateur custom pour un groupe de champs
const form = new FormGroup({
  password: new FormControl('', Validators.minLength(2)),
  passwordConfirm: new FormControl('', Validators.minLength(2)),
}, passwordMatchValidator);

// form = this.fb.group({...}, {validator: passwordMatchValidator});

function passwordMatchValidator(g: FormGroup) {
  return g.get('password').value === g.get('passwordConfirm').value
    ? null : {'mismatch': true};
}
```

Observer les changements de champs

- Il peut être utile de **réagir en direct aux changements de valeur d'un champ**, par exemple pour afficher des infos complémentaires à l'utilisateur.
- **Exemples** : Afficher le niveau de sécurité d'un mot de passe alors que l'utilisateur le tape ; Afficher des suggestions de nom d'utilisateur basé sur le prénom entré dans un autre champ (vincent99, vincent_01...), etc.
- **Syntaxe** : S'abonner à l'observable `valueChanges`, qui est une propriété de chaque `Control/ControlGroup` :

```
const password = fb.control('', Validators.required);

// On s'abonne aux changements du champ password
password.valueChanges.subscribe((newValue) => {
  this.passwordStrength = newValue.length;
});
```

EXO 16

- Formulaires avancés

Affichage avancé

Référencer ou modifier l'élément hôte

Référencer les éléments enfant

Modifier le DOM de l'élément hôte

- L'élément hôte est l'**élément HTML auquel une directive est appliquée**. Parfois, la directive doit **attacher du markup** ou un **modifier le comportement** de son hôte.
- **Syntaxe** - Pour accéder à l'élément DOM natif auquel une directive est attachée :
 - Injecter la classe **ElementRef** dans le constructeur de la directive.
 - Utiliser la propriété **nativeElement** de cette classe.
- Exemple - Supposons une directive popup qui s'utilise de la manière suivante :

```
<div class="alert alert-info" popup
```

- Cette directive peut accéder à son hôte — la balise `<div class="alert">...</div>` — de la manière suivante :

```
@Directive({  
  selector: '[popup]'  
})  
class Popup {  
  constructor(elementRef: ElementRef) {  
    console.log(elementRef.nativeElement);  
  }  
}
```

Modifier les propriétés/événements de l'élément hôte

- Il peut aussi être utile de **changer les attributs et les comportements de l'élément hôte.**
- **Syntaxe** : Utiliser les décorateurs
@HostBinding (property) pour binder aux propriétés et
@HostListener (event) pour binder aux événement de l'élément hôte.

```
import { Directive, HostBinding, HostListener } from '@angular/core';
@Directive({
  selector: '[myValidator]'
})
export class ValidatorDirective {
  @HostBinding('attr.role') role = 'button';
  @HostListener('mouseenter') onMouseEnter() {
    // do work
  }
}
```

Référencer les éléments enfants (1/2)

- Imaginons qu'on veuille créer un composant custom pour **afficher du contenu sous forme d'onglets**. Il pourrait s'utiliser ainsi :

```
<tab>
  <pane id="1">Contenu</pane>
  <pane id="2">Contenu</pane>
  <pane id="3" *ngIf="shouldShow">Contenu</pane>
</tab>
```

- Une telle fonctionnalité serait implémentée avec 2 composants : un composant **parent** pour le `<tab>`, et un composant **enfant** pour les `<pane>`.
- Le décorateur **@ContentChildren()** va permettre au parent `<tab>` de **récupérer une référence à tous ses enfants** `<pane>` :

```
@Component({selector: 'tab'})
export class Tab {
  @ContentChildren(Pane) panes: QueryList<Pane>;
  get serializedPanes(): string {
    return this.panes ? this.panes.map(p => p.id).join(', ') : '';
}
```

Référencer les éléments enfants (2/2)

- `@ContentChildren()` permet de **requêter le DOM** en lui passant un type d'élément ou de directive à trouver.
- La requête renvoie une `QueryList`, qui représente le **résultat live** de la requête.
- Ainsi, dès qu'un élément enfant est **ajouté, retiré, ou déplacé**, **le résultat est mis à jour**, et l'observable exposé par `QueryList` émettra une nouvelle valeur (`QueryList.changes`).
- L'objet `QueryList` est directement itérable (par exemple : `*ngFor="let i of myList"`), mais il expose aussi des méthodes facilitant sa manipulation : `QueryList.map()` , `QueryList.forEach()` ...).

Conclusion

Module

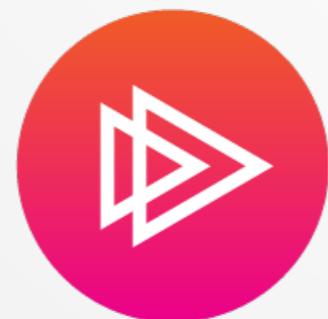
Copyright 2016-2019 - Toute reproduction interdite

Aller plus loin avec Angular

- **Livre recommandé :**
“Deviens un ninja avec Angular”.



- **Cours vidéos recommandés :** tous les cours “Angular” de Pluralsight (en anglais, mais sous-titrés en anglais et souvent en français).



PLURALSIGHT

Vos impressions ?

- **Vos impressions en quelques mots :**
 - Rythme ?
 - Contenu ?
 - Vous sentez-vous capable d'utiliser Angular dans vos projets ?

Merci !