

# Formation Angular 5.x

Février 2018

Animée par Vincent Caillierez

AngularFrance

# 1. Introduction

Présentations, Angular,  
TypeScript/ES6, Quickstart

# Présentations

Participants, Formateur, Formation

Leçon

[AngularFrance.com](https://angularfrance.com)

Copyright 2016-2018 - Toute reproduction interdite

# Vincent Caillierez

- **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...

# Le programme

- **Module #1** - Introduction
- **Module #2** - Composants I
- **Module #3** - Composants II
- **Module #4** - Services & Injection de dépendance
- **Module #5** - Modules Angular
- **Module #6** - Routeur
- **Module #7** - HTTP
- **Module #8** - Formulaires
- **Module #9** - Observables & RxJS
- **Module #10** - Fonctionnalités avancées (routeur, formulaires, affichage, HTTP)
- **Module #11** - Tests
- **Module #12** - Outillage, internationalisation et déploiement
- **Module #13** - Migration de AngularJS à Angular 2+

# QUIZ #1

Connaissez-vous Angular ?

<https://kahoot.it/>

Quiz

# Introduction à Angular

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

Leçon



# 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...
- Le **back-end** peut utiliser la technologie de VOTRE choix : Spring MVC (Java), ASP.NET, Symfony (PHP)...
- **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**.

- 1) La partie “front-end” de l'application est responsable d'afficher les données, parfois en temps réel, et de gérer les actions utilisateur (modifier les données, déclencher une action backend)  
Toutes les données et les traitements lourds seront gérés côté backend. Angular sert essentiellement à créer l'interface de l'application qui permettra de **visualiser** ou de **modifier des données**, et de **déclencher des actions** qui seront exécutées côté serveur.

Où trouver des références Angular+ ? Pour l'instant, vu son jeune âge, il est surtout utilisé en interne à Google (Google Adwords, Google Fiber...). Le site suivant contient une liste de référence : <http://builtwithangular2.com/>

# Principales caractéristiques

- **Plusieurs langages supportés.** ES5, ES6/TypeScript et Dart.
- **Complet.** Inclut **toutes les briques nécessaires** à la création d'une appli professionnelle. Routeur, requête 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<sup>(1)</sup>.
- **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...

(1) Gains en rapidité grâce à plusieurs améliorations :

- **Amélioration du compilateur** Angular (qui convertit les scripts et templates en JavaScript optimisé) ;
- **Réduction de la taille de la librairie** Angular (aujourd’hui 45 Ko alors que Angular 1 fait 56 Ko) ;
- **Lazy-loading** au niveau de l’injecteur de dépendance et du routeur.

# 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...

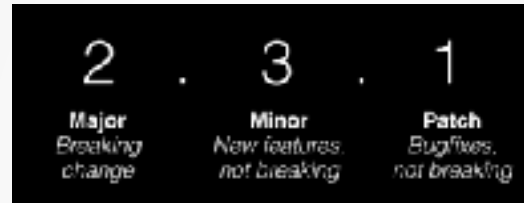
Leçon

[AngularFrance.com](http://AngularFrance.com)

Copyright 2016-2018 - Toute reproduction interdite

# Angular 2 ou 4 ?

- 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 **totalemt 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 5.1.0** du 6-DEC-2017.

# Quickstart

Outils, Stack technique, Appli minimale

Leçon

[AngularFrance.com](https://angularfrance.com)

Copyright 2016-2018 - Toute reproduction interdite



# Outils de développement



node & npm  
(installer les librairies)



IDE  
(coder avec TypeScript et Angular)



Navigateur  
(débugueur)

[AngularFrance.com](https://angularfrance.com)

17

Copyright 2016-2017 - Toute reproduction interdite

## node et npm (<https://nodejs.org/>)

Permettent d'installer et gérer les librairies utilisées notre application. Outil incontournable dans la boîte à outils du développeur front-end (peut également fournir un serveur web local, une API REST locale, des outils de test...).

Versions minimum à installer : node v4.x.x et npm 3.x.x.

Vous pouvez afficher les versions de node et npm installées sur votre système en exécutant les commandes suivantes dans la console / le terminal :

```
node -v
```

```
npm -v
```

## IDE

Tous les éditeurs habituels conviennent, mais mieux vaut utiliser un IDE qui supporte TypeScript (et Angular).

Les IDE suivants sont souvent utilisés avec Angular :

- WebStorm 2017 (30j gratuits) - **recommandé** - <https://www.jetbrains.com/webstorm/>
- Visual Studio Code (gratuit) - <https://code.visualstudio.com/>
- Atom avec le plugin TypeScript (gratuit)
- Sublime Text avec Typescript-Sublime-Plugin



- **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<sup>(1)</sup>:

```
PROJECT_ROOT/.angular-cli.json
```

(1) On peut paramétrer le type de fichiers générés par Angular-CLI, la verbosité, les librairies tierce-partie ou assets à inclure dans le build, etc.

# CLI - Principales commandes

- Installer Angular-CLI :

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

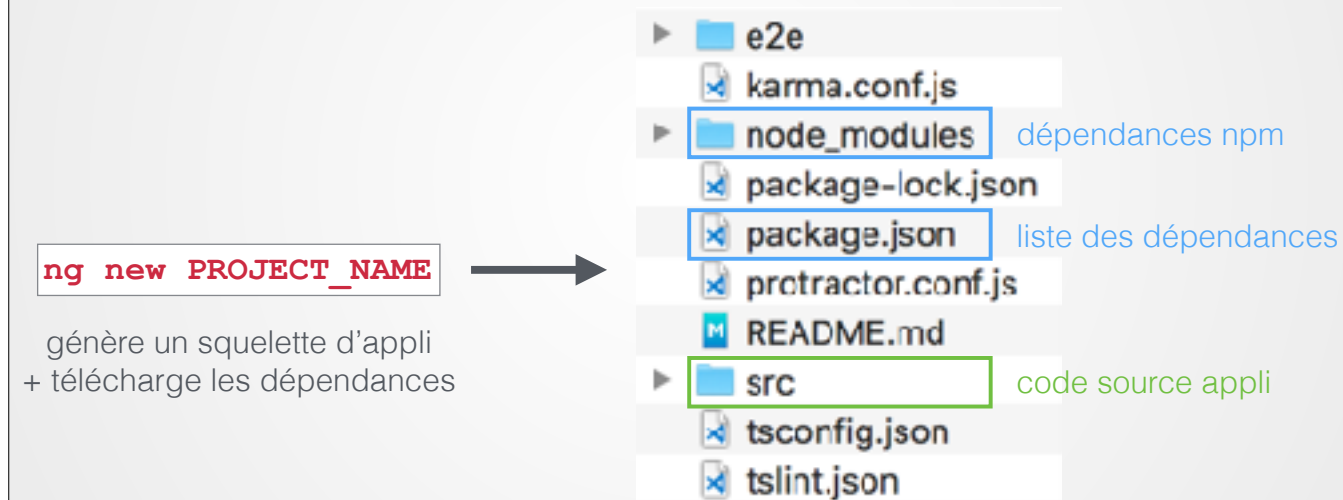
- Générer une nouvelle appli Angular + Démarrer serveur de dev :

```
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
```

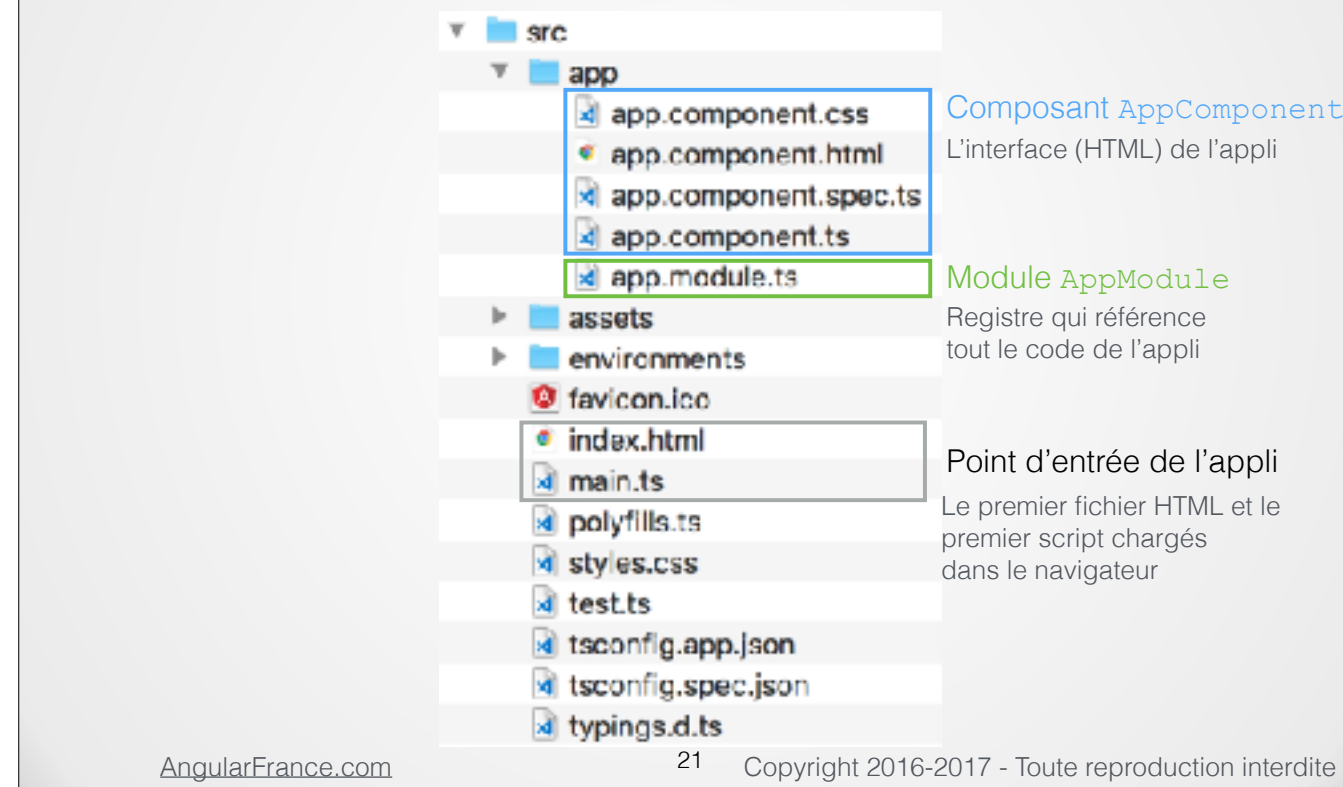
# CLI - Fichiers importants



Cette capture représente les fichiers créés après avoir lancé la commande `ng new PROJECT_NAME` qui génère une nouvelle application Angular.

- `package.json` - Contient la liste des paquets npm (on pourrait dire "dépendances") nécessaires à l'exécution de notre application.
- `node_modules` - Contient les paquets npm téléchargés en local.
- `src` - Code source de l'application Angular.

# CLI - Appli Angular



L'appli Angular générée par le CLI contient les fichiers suivants :

- Un composant `AppComponent` qui représente l'interface de l'application, c'est-à-dire le HTML affiché dans le navigateur. (Ce composant correspond aux 4 fichiers `app.component.XX`.) Pour l'instant, il n'y a qu'un seul composant, mais d'autres seront ajoutés au fil du développement.
- Un module `AppModule` qui est un registre qui référence tout le code de l'appli.
- `index.html` et `main.ts` - Point d'entrée de l'application. Le fichier `index.html` est le premier (et le seul) fichier HTML qui sera chargé par le navigateur ; `main.ts` est le premier fichier de l'application Angular à être exécuté ; c'est lui qui démarre ("bootstrap") l'application.

# 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

- Première application Angular.

# QUIZ #2

Connaissez-vous vraiment JavaScript ?

<https://kahoot.it/>

Quiz



# Révisions JS

Variables, Objets, Tableaux, Fonctions

Leçon

[AngularFrance.com](https://angularfrance.com)

Copyright 2016-2018 - Toute reproduction interdite

# 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**.

Les variables déclarées avec `let` et `const` sont scopées à leur bloc de déclaration. Avec `const`, les variables ne peuvent pas être ré-assignées (avec `let`, si). Quant à `var`, il faut l'oublier. Ses règles de scoping sont différentes et quasiment jamais souhaitables.

## DOC MDN:

- `let` - <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let>
- `const` - <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const>
- `var` - <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/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;
}
```

[AngularFrance.com](https://angularfrance.com)

dite

DOC opérateurs : <https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Op%C3%A9rateurs>

# 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"
```

Valeurs primitives : string, number, boolean, null, undefined.

# 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.** Exemple “TODO List”. On récupère les données saisies dans le formulaire (**sous forme d’objet**) qu’on ajoute à une liste de TODOs. Quand on saisit un 2ème item dans le formulaire, le texte du 1er item (ds la liste) continue à être modifié...

# 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);
```

**Syntaxe 2**

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

**Syntaxe 3**

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

Les trois syntaxes sont équivalentes :

1. La fonction est créée grâce à une **déclaration de fonction**. Elle est nommée et déclarée à un endroit différent de son invocation (elle pourrait être déclarée dans un tout autre fichier, puis importée dans le fichier où elle est utilisée).
2. La fonction est déclarée “en ligne”, à l’endroit où elle est utilisée. Son nom a disparu, car on n’a plus besoin de la référencer par son nom.
3. Idem que 2 mais avec la syntaxe fonction flèche de ES6.

Remarque : Si vous nommez la fonction explicitement (syntaxe 1), peu importe comment vous l’appellez.

# Fonction - Erreur fréquente #1

- Confondre le fait de **référencer** une fonction et le fait d'**invoker** 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);
});
```

Dans une fonction de callback, c'est la **position des paramètres** qui est chargée de sens, pas le nom des paramètres.

Par exemple, la fonction passée à `Array.prototype.forEach` reçoit, **dans cet ordre**, la valeur de l'itération en cours, puis la position en cours. Peu importe comment vous nommez les paramètres qui reçoivent ces valeurs.

# Introduction à TypeScript

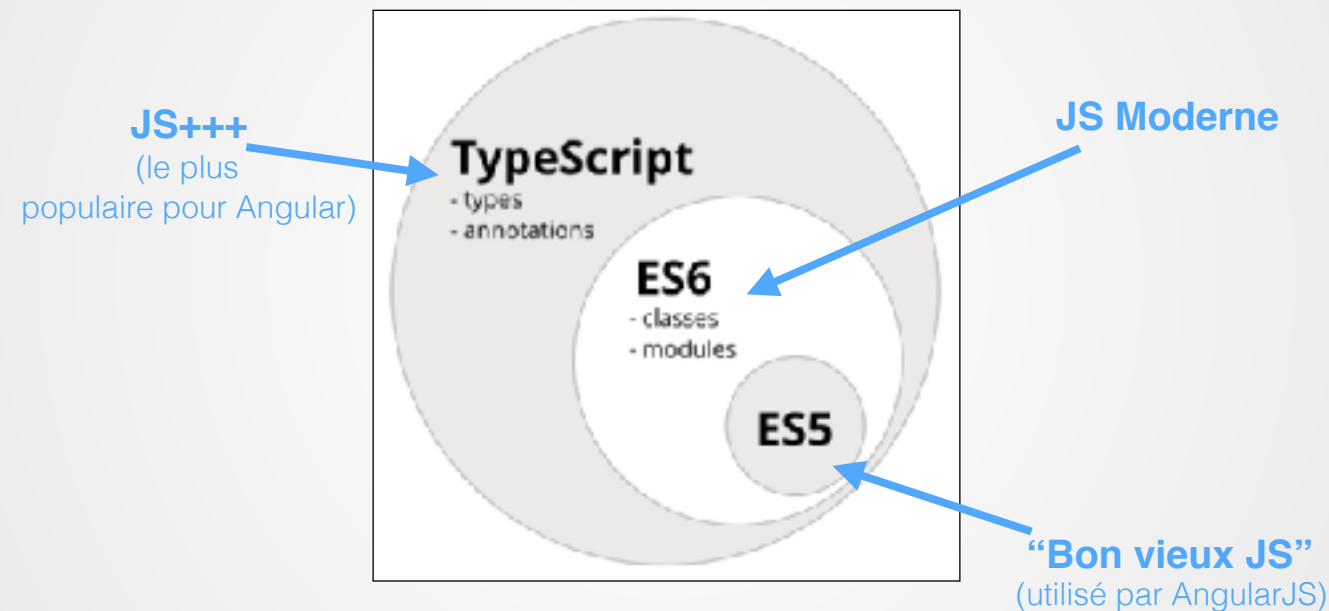
Types facultatifs, Compilation obligatoire

Leçon

[AngularFrance.com](https://angularfrance.com)

Copyright 2016-2018 - Toute reproduction interdite

# Langages supportés par Angular



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

Angular supporte 4 langages différents : ES5, ES6, TypeScript, et Dart.

**ES5.** Le "bon vieux JavaScript" tel que vous le connaissez. Supporté par tous les navigateurs modernes. Langage de facto pour les applis Angular 1.

**ES6/ES2015.** Dernière version du standard ECMAScript, aka "JavaScript moderne", avec de nouvelles fonctionnalités comme les modules (import/export), les classes, les fonctions flèche... Pas (encore) supporté nativement par les navigateurs, doit donc être compilé en ES5 avant d'être exécuté.

**TypeScript.** Reprend toutes les fonctionnalités de ES6 et lui en ajoute quelques autres, comme les types et les interfaces. Comme ES6, TypeScript doit être compilé en ES5 pour que les navigateurs puissent l'exécuter.

**Dart.** Remplacement de JavaScript à la sauce Google. Taux d'adoption faible. Doit également être transpilé en JavaScript.

# 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 prenom = ['Pierre', 'Paul', 'Jo'];

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

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

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

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

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

TypeScript un sur-ensemble de JavaScript, c'est-à-dire que tout code JavaScript syntaxiquement correct est du TypeScript valide.

Pour commencer, il suffit de renommer les fichiers .js en .ts et d'introduire le compilateur TypeScript. Ensuite, on peut introduire progressivement les syntaxes ES6 et TypeScript pour tirer partie d'un JavaScript plus puissant.



# TypeScript - Compilation

TypeScript



```
const API_KEY = '466fgjH$DIT0';

class Person {
  name: string;

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

Syntaxes TypeScript :

- const
- class
- string

JavaScript (ES5)



```
var API_KEY = '466fgjH$DIT0';
var Person = (function () {
  function Person(theName) {
    this.name = theName;
  }
  return Person;
})();
```

Les syntaxes TypeScript ont disparu.

ng serve / tsc

On code en TypeScript dans l'IDE. Mais le navigateur exécute du JavaScript. Entre les deux, la compilation intervient.

En phase de développement, ce processus de compilation est transparent : les fichiers `.ts` sont recompilés automatiquement à chaque changement par la commande `ng serve` du CLI qui tourne en tâche de fond.

On peut aussi utiliser le compilateur TypeScript officiel `tsc`.

# Syntaxes TypeScript/ES6

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

Leçon

# Types - 1/2 (ts)

- Les **types** sont au coeur 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];
```

Comme on le voit, on peut déclarer le type d'une variable ET lui assigner une valeur en même temps. C'est le cas avec la variable `name` dans l'exemple ci-dessus.

TypeScript supporte les **types de base** (`string`, `number`, `boolean`...) et les **types "complexes"** définis par votre application (nous verrons un exemple plus loin).

Rappelez-vous que les types sont **facultatifs** : on peut commencer à écrire du code sans information de type, ce qui donnera aux variables le type par défaut `any`, et ajouter plus tard les informations de type quand le code est stabilisé.

DOC : <http://www.typescriptlang.org/docs/handbook/basic-types.html>

# 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**<sup>(1)</sup> :

```
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");  
}
```

AngularFrance.com

44

Copyright 2016-2017 - toute reproduction interdite

(1) Dans cet exemple, on a typé à la fois le paramètre et la valeur de retour de la fonction. La fonction `isAdmin` accepte un paramètre `username` qui doit être une chaîne de caractères et elle doit renvoyer un booléen (`true` ou `false`).

Si votre code tente d'utiliser d'autres types que ceux attendus, le compilateur TypeScript générera une erreur (et votre IDE aussi, s'il supporte TypeScript).

# 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**<sup>(1)</sup> :

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

- **Meilleur type commun**<sup>(2)</sup> :

```
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`
```

- 1) Ce type d'inférence est utilisé lors de l'initialisation de variables ou de membres de classe, pour les valeurs par défaut des paramètres, et les valeurs de retour de fonction.
- 2) Ce type d'inférence est utilisé lorsque le type est issu de plusieurs expressions. Le type de chaque expression est considéré et un algorithme tente de déterminer le "meilleur type commun".

DOC : <http://www.typescriptlang.org/docs/handbook/type-inference.html>

# 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

Ce code dit que la variable `myTodo` est de type `Todo`, et qu'à ce titre elle doit être un objet qui possède **au minimum** une propriété `text` qui est une chaîne de caractères, et une propriété `done` qui est un booléen.

DOC : <http://www.typescriptlang.org/docs/handbook/interfaces.html>

# 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.

# Interfaces - Où dans Angular ?

- Quand vous souhaitez **imposer une certaine forme aux modèles de données** de VOTRE application.
- Exemple : Si votre application manipule des livres, vous pourriez créer l'interface Book suivante :

Cette propriété référence  
une autre interface

```
interface Book {  
  id: number;  
  title: string;  
  summary: string;  
  authors: Author[];  
  published: {  
    day: number;  
    month: number;  
    year: number;  
  }  
}
```

Objet dans un objet

Dans la pratique, on utilise plutôt les interfaces :

- Pour les objets simples qui ne doivent pas être instanciés, par exemple les paramètres passés à une fonction/méthode
- Pour imposer la présence de certaines propriétés/méthodes sur une classe (`class Foo implements InterfaceX`).



# Classes - Syntaxe (ES6/ts)

Déclaration :

mot-clé	<b>class</b>	Person {
propriété	<b>name</b> :	string;
constructeur	<b>constructor</b> (theName: string) {	<b>this.name</b> = theName;
		}
méthode	<b>sayMyName</b> () {	console.log('Mon nom est ' + <b>this.name</b> );
		}
		}

Instanciation :

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

[AngularFrance.com](http://AngularFrance.com)

49

Copyright 2016-2017 - Toute reproduction interdite

On déclare une classe avec le mot-clé `class` suivi du nom de la classe et d'une paire d'accolades `{ }`.

Notre classe `Person` possède 3 **membres** :

- une **propriété** `name` ;
- un **constructeur**, qui est une méthode spéciale exécutée au moment où l'on fait `new Person()`.
- et une **méthode** `sayMyName`.

Une classe peut posséder **autant de propriétés et de méthodes que l'on souhaite** (mais un seul constructeur). Ces dernières peuvent être déclarées **dans n'importe quel ordre**.

À l'intérieur de la classe, on accède aux membres de la classe via le préfixe `this` (par exemple, `this.name` ou `this.sayMyName()`).

**IMPORTANT.** Déclarer des propriétés dans une classe (dans notre exemple, la propriété `name`) n'est pas une fonctionnalité standard d'ES6, c'est seulement possible en TypeScript.

# Classes - Visibilité (ts)

- En TypeScript, tous les membres d'une classe sont **publics 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  
  }  
}
```

Dans le scénario initial, on veut transformer un **paramètre** du constructeur (`theName`) en **propriété** de la classe (`this.name`). C'est très fréquent.

Dans la syntaxe raccourcie, le paramètre `name` est automatiquement transformé en propriété `this.name` grâce au mot-clé `public`. Ça marche aussi avec les autres modifieurs d'accessibilité (`private` et `protected`).

# 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;  
    }  
  
}
```

# Classes - Où dans Angular ?

- **Toutes les “briques” Angular** sont des classes :
  - Module Angular
  - Composant Angular
  - Directive Angular
  - Service Angular
  - Pipe Angular
  - Etc.

# 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.

`@Décorateur()`  
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;  
}
```

Certains décorateurs portent sur une classe (@Component), d'autre sur une propriété de classe (@Input).

Certains décorateurs reçoivent un objet de configuration en paramètre (@Component). D'autres pas (@Input).

IMPORTANT. Même si un décorateur ne prend pas d'objet de configuration, **son nom doit TOUJOURS être suivi de parenthèses** et précédé du symbole @.

# Modules (ES6)

- **Tout fichier de code ES6/TypeScript est un module**<sup>(1)</sup>.
- 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**<sup>(2)</sup>:

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

1) Ce n'est pas tout à fait exact. C'est la présence d'au moins un mot-clé `import` ou `export` dans le code qui transforme un simple fichier en module.

2) N'oubliez pas les **accolades** autour des éléments importés.

# 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**<sup>(1)</sup>:  
*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';
```

- 1) Dans un vrai projet, vous trouverez également des imports depuis des **librairies tierce-partie**. Par exemple, si vous utilisez ng-bootstrap , vous pourriez trouver la ligne suivante :

```
import { NgbDateStruct, NgbTimeStruct } from '@ng-bootstrap/ng-bootstrap';
```

Notez que le chemin d'import ne commence pas par un point, mais par le symbole @ comme pour les imports Angular.

# 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 **{ }**<sup>(1)</sup>:

```
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<sup>(2)</sup>.
  - Le **return** est implicite en l'absence d'accolades.

1. Autrement dit :
  - Les **parenthèses** à GAUCHE de la flèche ne sont obligatoires QUE SI la fonction ne prend pas d'argument ou si elle prend plus d'un argument.
  - Les **accolades** à DROITE de la flèche ne sont obligatoires QUE SI le corps de la fonction contient plusieurs instructions. Si la fonction ne contient qu'une expression, alors on peut oublier les accolades et **le return est implicite**.
2. Jusqu'à l'apparition des fonctions fléchées, chaque nouvelle fonction définissait son propre `this`. Les fonctions fléchées, elles, ne créent pas de nouveau contexte, elles capturent la valeur `this` de leur contexte.

# 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}`;
```

Note. Cette fonctionnalité s'appelle "littéraux de gabarits" (*template literals*) dans la doc. Voir [https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Litt%C3%A9raux\\_gabarits](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Litt%C3%A9raux_gabarits).

# QUIZ #3

Avez-vous compris TypeScript/ES6 ?

<https://kahoot.it/>



# SuperQuiz

Front-office, Back-office,  
Modèle de données

Leçon

[AngularFrance.com](http://AngularFrance.com)

Copyright 2016-2018 - Toute reproduction interdite

# SuperQuiz - Front-office



# SuperQuiz - Back-office

## Admin - Liste des quizzes

+ Nouveau Quiz

Titre	Actions	Questions
Quiz Angular 2 (général)	<a href="#">Voir</a> <a href="#">Éditer</a> <a href="#">Supprimer</a>	<a href="#">Gérer les questions</a>
Quiz TypeScript	<a href="#">Voir</a> <a href="#">Éditer</a> <a href="#">Supprimer</a>	<a href="#">Gérer les questions</a>
Quiz Module	<a href="#">Voir</a> <a href="#">Éditer</a> <a href="#">Supprimer</a>	<a href="#">Gérer les questions</a>

## Modifier un quiz

Titre\*

Quiz Angular 2 (général)

Description

Questions d'introduction général sur Angular 2

Peut ré-essayer les questions ☐

[Enregistrer](#) [Annuler](#)

## Gérer les questions du quiz

[Retour à la liste des quizzes](#)

[Enregistrer](#)

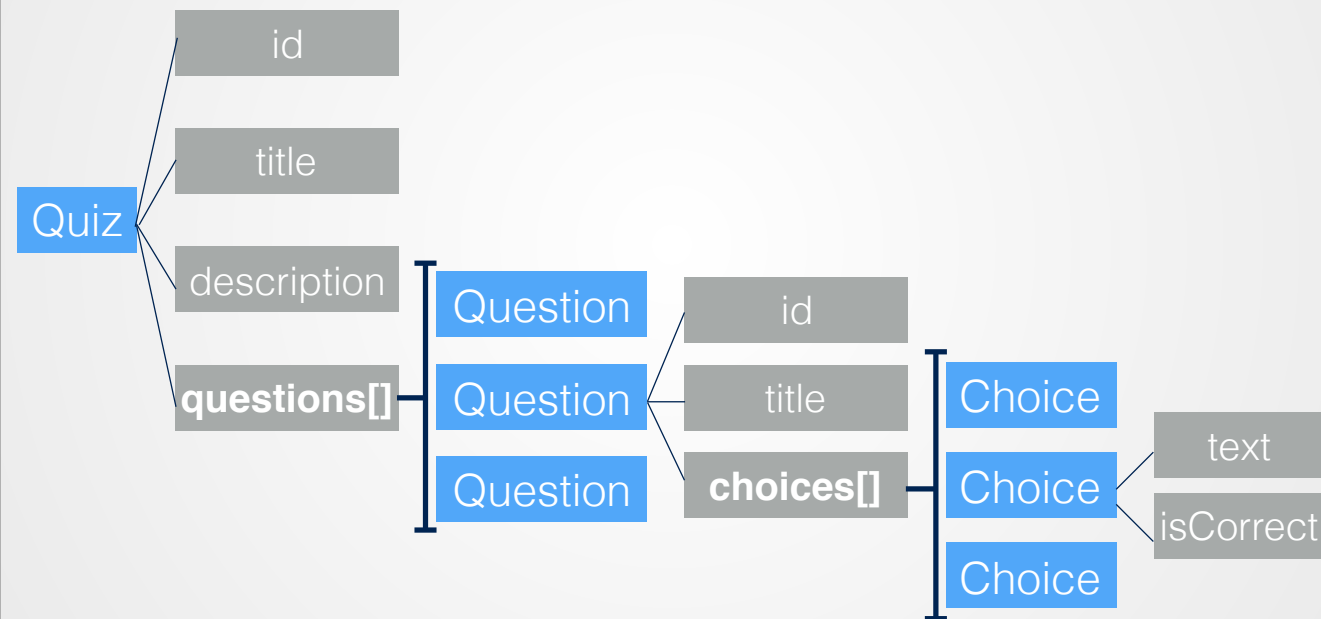
+ Nouvelle question

Position	Actions	Type	Question
<a href="#">↑</a> <a href="#">↓</a>	<a href="#">G</a> <a href="#">S</a>	multiple_choice	En quelle année AngularJS (première version) est-il sorti ? <ul style="list-style-type: none"><li><input type="checkbox"/> 2010</li><li><input checked="" type="checkbox"/> 2011</li><li><input type="checkbox"/> 2012</li><li><input type="checkbox"/> 2013</li></ul>
<a href="#">↑</a> <a href="#">↓</a>	<a href="#">G</a> <a href="#">S</a>	multiple_choice	Quels langages de programmation sont supportés par Angular ? <ul style="list-style-type: none"><li><input checked="" type="checkbox"/> JavaScript</li><li><input checked="" type="checkbox"/> TypeScript</li><li><input type="checkbox"/> VBScript</li><li><input type="checkbox"/> PHP</li></ul>

AngularFrance.com

ite

# SuperQuiz - Modèle de données



AngularFrance.com

Copyright 2016-2017 - Toute reproduction interdite

- Un `Quiz` possède un titre et un nombre illimité de questions.
- Une `Question` possède un titre (libellé de la question) et plusieurs choix possibles, c. à d. réponses possibles.
- Un choix (`Choice`) est composé d'un texte et d'un booléen indiquant si le choix est correct. Une question peut donc avoir plusieurs choix corrects.

# EXO 1

- Comprendre le modèle de données.

# 2. Composant I

Syntaxe, Arbre des composants,  
Syntaxes de template

# Composant - Introduction

Définition, Syntaxe, Utilisation

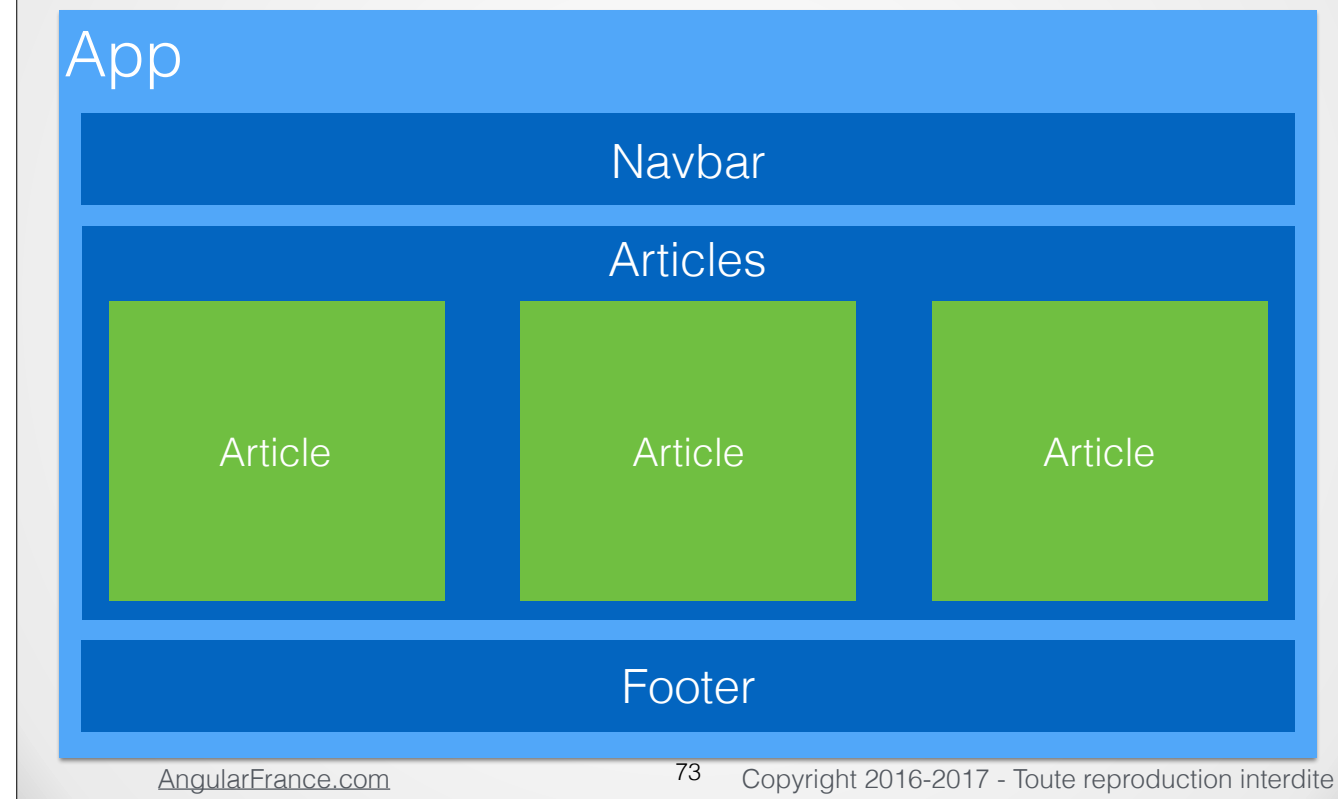
Leçon

# 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.
- Toute application Angular doit contenir **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**.



# Composant - Imbrication



Dans une appli Angular, l'interface se compose de plusieurs composants imbriqués comme des poupées gigognes. On part toujours d'un composant "racine", qui représente l'ensemble de l'application et qui contient tous les autres.

# Composant - Syntaxe

Composant = Classe + Template

Classe Template

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

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

Classe décorée avec @Component  
(selector et templateUrl obligatoires)

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

Template  
(HTML + Syntaxes Angular)

La classe et le template sont reliés via la propriété `template` (template en-ligne) ou `templateUrl` (template dans un fichier distinct).

Un composant, c'est une classe décorée avec `@Component` et un template.

La **classe** initialise les données et contient la logique applicative. C'est l'équivalent du "scope" dans AngularJS. On pourrait aussi parler de **contrôleur** (pattern MVC) ou de **ViewModel** (pattern MVVM).

Le **template** est compilé par Angular et peut utiliser toutes les propriétés et méthodes définies dans la classe via des syntaxes spécifiques à Angular (interpolation, *property binding*...). Le template une fois compilé se nomme la **vue**.

Le template peut se trouver dans un **fichier distinct** référencé par la propriété `templateUrl`, mais s'il est relativement court, on peut directement mettre le template dans le décorateur en utilisant la propriété `template`.

# Composant

## Compilation des templates

- Le navigateur **ne voit jamais le code source** des composants :  
TypeScript —> JavaScript —> JavaScript avec templates compilés
- **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>
```

# 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>Beau temps</p>',
  styles: ['p { background: yellow; }']
})
export class MeteoComponent {}
```



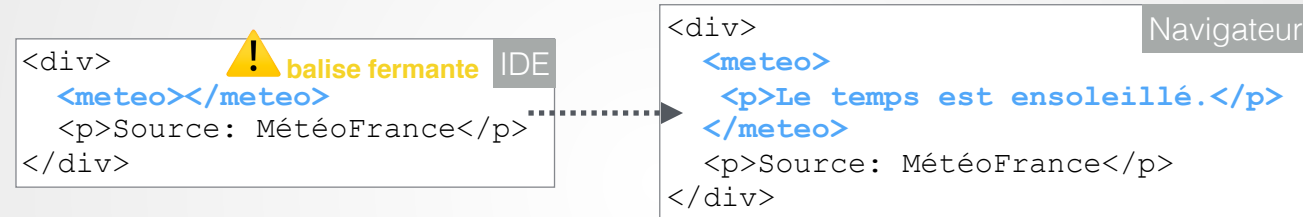
tableau

- **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 composants**, placez-les dans le fichier `styles.css` du projet<sup>(1)</sup>.

(1) Vous pouvez charger d'autres fichiers CSS globaux avec le CLI, en les déclarant dans la propriété `styles` du fichier `.angular-cli.json`. Le fichier `styles.css` est le fichier généré par défaut par la commande `ng new`.

# Composant - Utilisation

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



**!** 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 { }
```

Par défaut, un composant n'est affichable que dans les autres composants du module où il est déclaré.

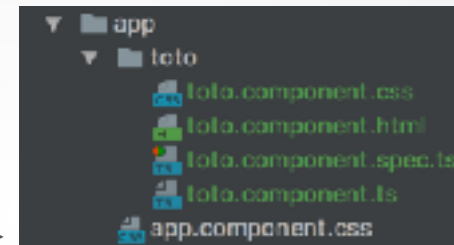
Dans notre exemple, `MeteoComponent` n'est donc affichable que dans le template de `AppComponent`, puisque c'est le seul autre composant déclaré dans `AppModule`.

Nous verrons plus loin comment afficher le même composant dans plusieurs modules.

# 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
```

AngularFrance.com

78

Copyright 2016-2017 - Toute reproduction interdite

Par défaut, le CLI génère un composant dans son propre répertoire, avec les fichiers suivants :

- `toto.component.ts` : Classe du composant
- `toto.component.html` : Template du composant
- `toto.component.css` : CSS du composant
- `toto.component.spec.ts` : Tests unitaires du composant

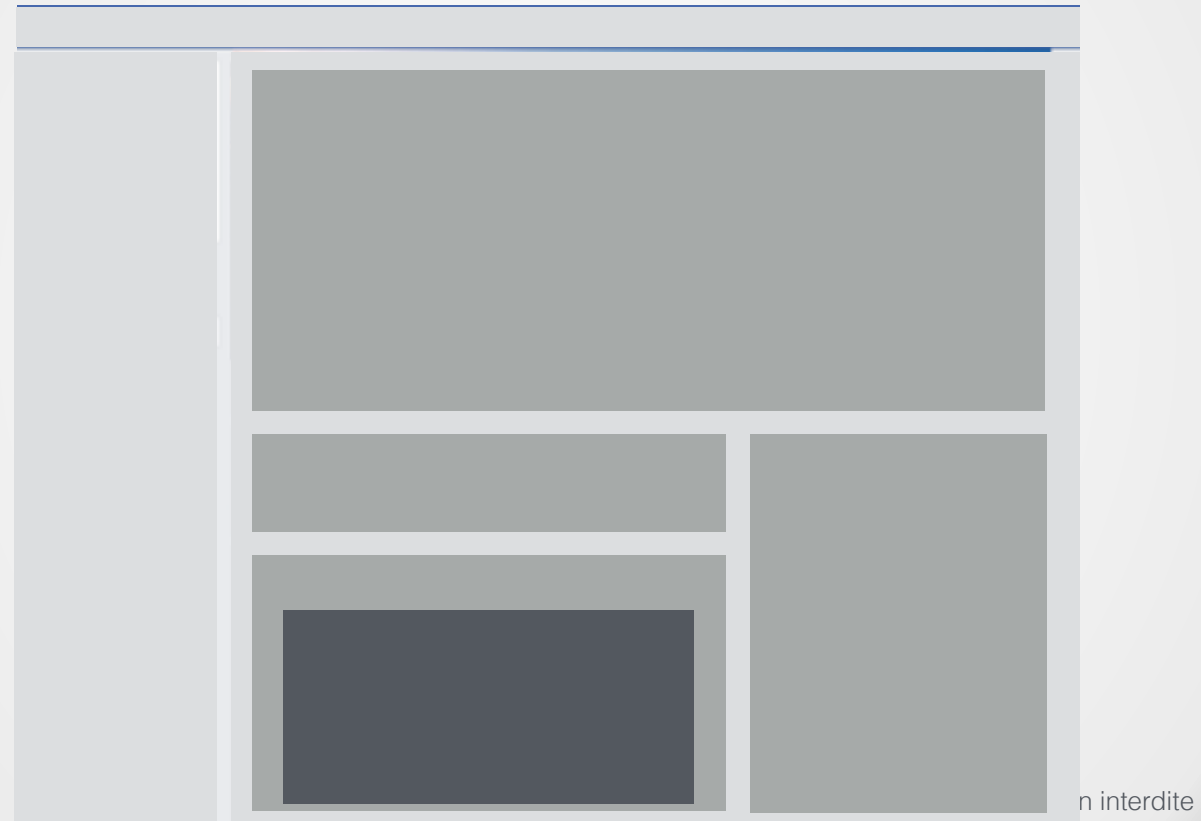
DOC: <https://github.com/angular/angular-cli/wiki/generate-component>

# L'arbre des composants

Découpage de l'UI,  
Relation parent-enfant

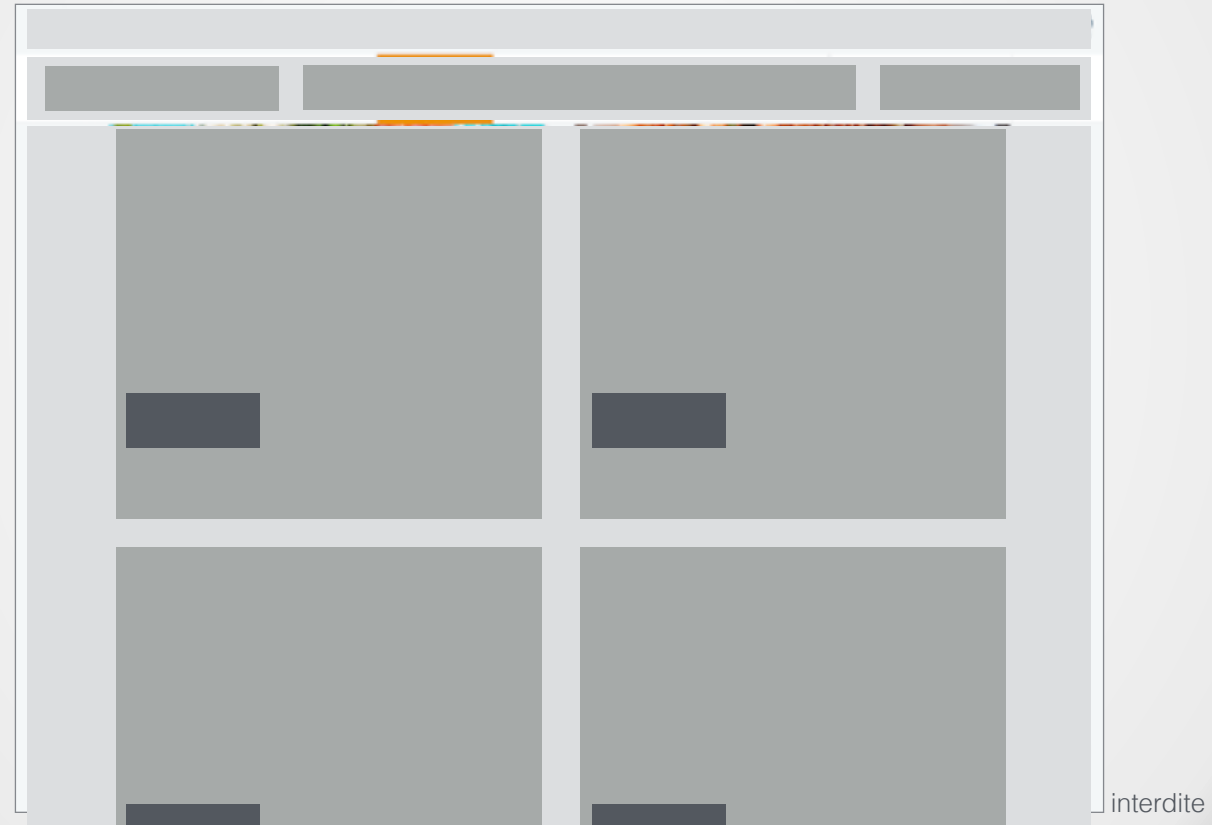
Leçon

# Exemple #1 : Facebook





# Exemple #2 : FoodChéri

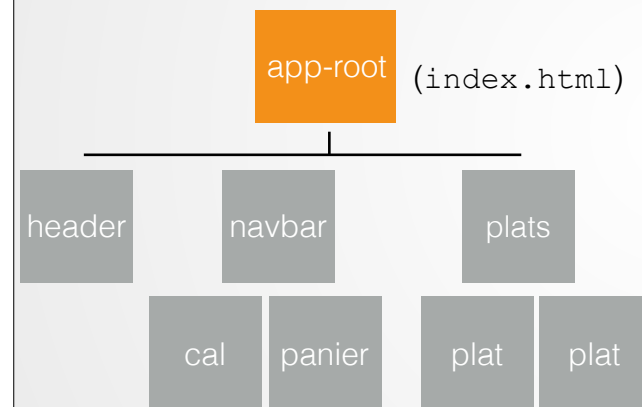


# Comment découper l'interface ?

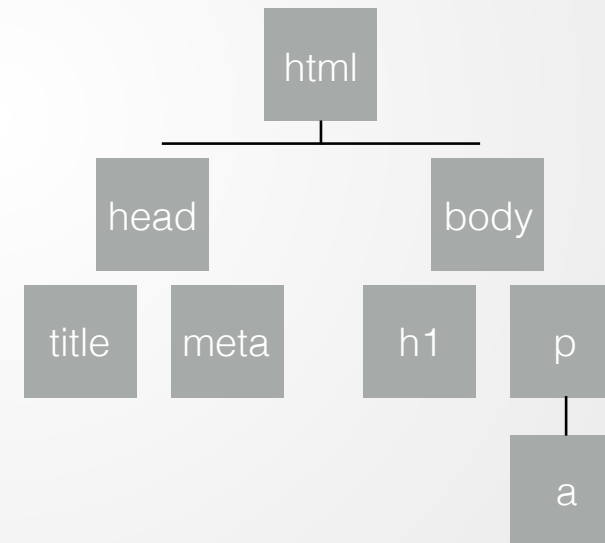
- Quels critères utiliser pour déterminer la **granularité des composants** ? (i.e. **quelques gros composants** vs **plein de petits**)
- Un “bout d'interface” mérite de (ou doit) avoir son propre composant si :
  - Il va être **réutilisé** à plusieurs endroits (dans le même projet ou différents projets). 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.

# L'arbre des composants

## Arbre des composants



## DOM



L'interface d'une application Angular correspond à un arbre de composants. Seul le composant racine est affiché dans `index.html`. Il contient des sous-composants, qui contiennent à leur tour des sous-composants, comme des **poupées gigognes**.

Cet arbre de composants est à une application Angular **ce que le DOM est à une page HTML** : une représentation arborescente des éléments qui composent la page / l'interface.

# Relation Parent-Enfant

Angular

HTML

ViewChild<sup>(1)</sup>

```
<plats></plats>
```

```
@Component({  
  selector: 'plats',  
  template: `  
    <plat></plat>  
    <plat></plat>  
  `,  
})  
class PlatsComponent { }
```

```
<div class="desc">  
  <p>Hello !</p>  
</div>
```

<p> est un enfant de <div>

En HTML, une balise qui est encadrée par une autre devient l'enfant de l'autre balise.

Dans Angular, il y a 2 manières de créer une relation parent-enfant entre deux composants :

- Un composant affiché dans le template d'un autre composant est un **enfant de vue**, ou *ViewChild*, du premier composant. Sur le slide, les 2 composants `<plat>` sont des enfants de vue du composant `<plats>`.
- Un composant qui est simplement encadré par un autre, comme en HTML, est un **enfant de contenu**, ou *ContentChild*. Sur le slide, `<cal>` et `<panier>` sont des enfants de contenu du composant `<navbar>`. Pour que ces enfants ne soient pas écrasés par le template du parent, il faut impérativement placer la balise `<ng-content></ng-content>` dans le template du parent.

# ViewChild ou ContentChild?

- Utiliser un **ViewChild**
  - Quand 2 composants sont **intimement liés**. Exemple : une liste d'items ; on utilise un composant parent pour afficher la liste, et dans le template du parent, on répète un composant enfant pour chaque item.
  - Pour **encapsuler** une imbrication complexe de composants.
  - Pour **empêcher** l'utilisateur d'un composant de modifier son affichage.
- Utiliser un **ContentChild**
  - Pour **permettre** à l'utilisateur du composant de **contrôler une partie de son affichage**, en lui passant du HTML ou d'autres composants.

# EXO 2

- Vos premiers composants.

# Syntaxes de template

## 1/2

Interpolation, Binding de propriété

Leçon

# 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>

**ATTENTION. On binde aux propriétés DOM, pas aux attributs HTML.** Parfois, la propriété DOM est identique à l'attribut HTML correspondant. Exemple : `src` est à la fois un attribut de la balise `<img>` et une propriété de l'élément DOM `HTMLImageElement`.

Parfois, la propriété DOM diffère ou n'a pas d'attribut HTML correspondant. Exemples : `innerHTML`, `hidden`...

Liste de toutes les propriétés DOM existantes : [https://developer.mozilla.org/en-US/docs/Web/API/Document\\_Object\\_Model](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model)

**IMPORTANT.** La partie droite du binding dans `[proprieteDOM]="expression"` est évaluée comme une expression (c. à. d. comme du code JavaScript). **DONC** : il ne faut surtout pas mettre de `{{ ... }}` autour de l'expression.

# 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 [style.display]="isReady ? 'none': 'block'">
  Chargement en cours...
</p>
<p [class.jaune]="weather.type==='ensoleillé'"
  Voici le temps qu'il fait...
</p>
```

Ce n'est pas une bonne idée d'écrire ``. En effet, le navigateur va essayer de récupérer l'image dès qu'il lira l'attribut `src`, et réalisera une requête HTTP sur l'URL `{{weather.icon}}`, qui n'est pas une URL valide...

Dans AngularJS 1.x, on utilisait d'ailleurs la directive `ng-src` pour résoudre ce problème : ``. L'attribut `ng-src` était ignoré par le navigateur (car non-standard) et remplacé après compilation par Angular par un attribut `src` standard.

# EXO 3

- Utiliser l'interpolation et le binding de propriété.

# Syntaxes de template

## 2/2

Directives structurelles,  
Binding d'événement, Pipes

Leçon

# Directives structurelles

- **Modifient la structure du HTML** en ajoutant/retirant des balises. Commencent toutes par le caractère **\***.
- **\*ngIf** permet d'**insérer/retirer un fragment de HTML** du DOM selon qu'une expression vaut `true` ou `false`<sup>(1)</sup> :

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

- **\*ngFor** permet de **répéter un fragment de HTML** pour chaque élément d'une collection<sup>(2)</sup> :

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

- Ces directives sont évaluées en **"live"** : si la valeur de l'expression change côté classe, le template est mis à jour aussitôt.

(1) `*ngIf` ne se contente pas d'afficher/cacher. Elle insère ou retire la balise portant le `*ngIf` dans le DOM. Si la balise est retirée, le HTML qu'elle contient n'est même pas compilé. Mais il est recompilé à chaque fois que la balise est ré-insérée... (attention aux performances). Cela dit, il est fréquent d'utiliser `*ngIf` pour afficher/cacher un petit fragment de HTML.

(2) Dans cet exemple, le HTML final contiendra autant de `<li>...</li>` qu'il y a d'éléments dans `items`.

Il existe aussi `*ngSwitch`, non présenté ici. Voir un exemple sur <https://angular.io/docs/ts/latest/guide/structural-directives.html#ngSwitch>.

# 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 :  
{{ user?.name }}
```

# 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"></balise>`

**ATTENTION.** On binde aux événements DOM natifs, pas à des événements spécifiques à Angular. Les wrappers `ng-click`, `ng-keyup`... d'AngularJS n'existent plus.

Liste de tous les événements DOM : <https://developer.mozilla.org/en-US/docs/Web/Events>



# Binding d'événement Exemples

## Classe

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

## Template

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

⚠ L'événement DOM standard est `keydown` mais Angular fournit une **syntaxe raccourcie** permettant d'intercepter des combinaisons de touches : `keydown.space`, `keydown.alt.space`...

# 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.

Liste des pipes natifs d'Angular : <https://angular.io/guide/pipes>

On peut créer son propre pipe — comme `markdown` sur ce slide — en créant une classe annotée avec le décorateur `@Pipe` et qui implémente l'interface `PipeTransform`. Voir <https://angular.io/guide/pipes#custom-pipes>.

# EXO 4

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

# Templates - Résumé

Et points importants

Leçon

[AngularFrance.com](https://angularfrance.com)

Copyright 2016-2018 - Toute reproduction interdite

# 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 structurales

```
<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/>

Quiz

# Comprendre la classe Answer

Avant de faire l'exercice qui suit...

Leçon

# Answer - Créer une réponse

- 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.
- Il y a **2 manières de créer une réponse** avec le(s) choix qu'elle contient<sup>(1)</sup> :

- **IMPÉRATIF**. On instancie une réponse vierge puis on ajoute les choix avec `answer.addChoice()` :

```
const answer = new Answer({  
  questionId: 12  
});  
answer.addChoice(new Choice({  
  text: 'Faux'  
}));
```

- **DÉCLARATIF**. On instancie une réponse avec des choix prédéfinis :

```
const answer = new Answer({  
  questionId: 12,  
  choices: [{ text: 'Faux' }]  
});
```

(1) On utilisera plutôt la manière **déclarative** pour instancier les questions chargées depuis le backend, et la manière **impérative** pour réagir aux actions de l'utilisateur dans l'interface.

# Answer - Manipuler les choix

- Ensuite, la classe Answer expose **4 méthodes 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 permettent de gérer les **choix multiples**.
- Le statut **correct/incorrect** de la réponse est mis à jour en temps réel sur la base des choix sélectionnés :

```
// true si réponse correcte  
answer.isCorrect
```

# EXO 5

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

Titre de la question :

- Choix 1
- Choix 2
- Choix 3

Soumettre

# Directives attribut

Natives et Custom

Leçon

# 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<sup>(1)</sup> :

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

(1) Bi-directionnel = si la propriété change côté classe, le template est mis à jour ; et si elle change côté template, la classe est mise à jour. C'était très populaire dans AngularJS, mais plus très utilisé dans Angular 2 et +.

# Directives attribut natives

## Exemples

### 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.font-size]="isSpecial ? 'x-1'  
  ...  
</div>  
  
<input [(ngModel)]="user.name">
```

- (1) `currentClasses` peut prendre plusieurs valeurs. Ça peut être une chaîne contenant des classes CSS séparées par un espace. Ou un objet dont les propriétés sont des noms de classes CSS, et les valeurs sont des booléens `true/false` indiquant si la classe doit être ajoutée ou enlevée.
- (2) `currentStyles` peut prendre plusieurs valeurs. C'est typiquement un objet dont les propriétés sont des propriétés CSS (par ex, `backgroundColor`), et les valeurs sont les réglages correspondant à ces propriétés.



# Directives attribut custom

- 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 ([ngClass](#), [ngStyle](#)...)
- Dans cette partie, nous allons voir comment créer une **directive attribut custom**.

# 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 { }
```

Par défaut, une directive n'est utilisable que dans les autres composants du module où elle est déclarée.

Dans notre exemple, HighlightDirective n'est donc affichable que dans le template de AppComponent, puisque c'est le seul autre composant déclaré dans AppModule. Nous verrons plus loin comment utiliser la même directive dans plusieurs modules.

# 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.

Une directive dispose de 3 techniques pour accéder à / modifier l'élément sur lequel elle porte (aka son "hôte").

# Directive attribut custom

## Exemple

Déclaration :

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

  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>
```

AngularFrance.com

117

Copyright 2016-2017 - Toute reproduction interdite

- 1) `ElementRef` est injecté pour accéder à l'élément DOM qui accueille la directive (et modifier ses propriétés).
- 2) Le décorateur `@HostListener` permet d'écouter les événements DOM `mouseenter` et `mouseleave` de l'élément qui accueille la directive.
- 3) Le `@Input()` est utilisé pour transmettre des paramètres à la directive, ici la couleur à appliquer sur le background de l'élément hôte

# 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 <b>meteo</b> ></p>

Les directives sont des composants sans template.

# 3. Composants II

Input/Output, Cycle de vie, Projection

# Input/Output

Leçon

[AngularFrance.com](https://angularfrance.com)

Copyright 2016-2018 - Toute reproduction interdite



# Input - Définition & Syntaxe

- Un input permet de **passer des données** à un composant ou une directive, via le template où il/elle est utilisé(e) :

```
<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 à une **chaîne littérale**.

# [ ] - Ne pas confondre

Sans [ ... ]

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

Je passe une **chaîne littérale**  
à l'input `ville` de mon  
composant custom.

```

```

Je passe une **chaîne littérale**  
à l'attribut HTML `src`  
de la balise `<img>`.

Pas de [...] -> **STRING**

Avec [ ... ]

```
<meteo [ville]="city"></meteo>
```

Je passe une **expression JS**  
à l'input `ville` de mon  
composant custom.

```
<img [src]="imageUrl">
```

Je passe une **expression JS**  
à la propriété DOM `src`  
de la balise `<img>`.

[...] -> **EXPRESSION**

# 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);
  }
}
```

- Composant **parent** :

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

(1) Le composant enfant doit décorer la propriété qui sert à émettre l'événement avec le décorateur `@Output`.

(2) Et obligatoirement lui assigner une instance de `EventEmitter`.

⚠ - Le type déclaré dans `new EventEmitter<TYPE>()` doit correspondre au type de données qui transiteront dans l'événement. Ici, c'est `number` mais on pourrait mettre ce qu'on veut — `any` pour être permissif, `void` si aucune donnée ne transite...

(3) Enfin, il faut émettre l'événement au moment opportun. Ici quand on clique sur le bouton "Déclencher", on appelle une méthode locale `declencher()` qui émet l'événement.

Parent

- Transmet les données à l'enfant via l'input `[todo]`.
- Écoute le retour émis par l'enfant via l'output `(done)` et exécute une méthode locale `onDone()`.

```
@Component({
  selector: 'todos',
  template: `
    <div *ngFor="let td of todos">
      <todo [todo]="td" (done)="onDone($event)"></todo>
    </div>`
})
export class TodosComponent {
  todos = [
    {text: 'Refactoriser le code', done: false},
    {text: 'Créer un nouveau quiz', done: false},
    {text: 'Acheter un bounty', done: false}
  ];
  onDone(todo: Todo) {
    todo.done = true;
  }
}
```

input

output

méthodes locales

Enfant

- Déclare les propriétés `@Input` et `@Output`.
- Ne fait rien d'autre qu'afficher des données et émettre un output via sa méthode locale `markAsDone()`.

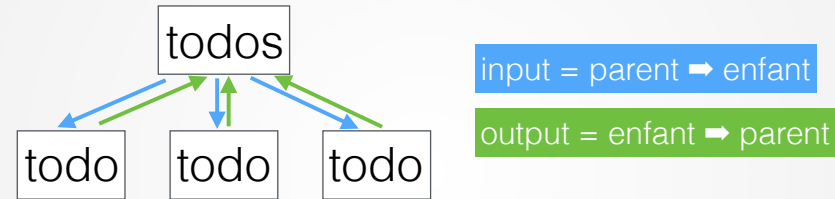
```
@Component({
  selector: 'todo',
  template: `
    <h2 [style.text-decoration]="todo.done ? 'line-through' : 'none'">
      {{ todo.text }}
    </h2>
    <button (click)="markAsDone()">Fait</button>`
})
export class TodoComponent {
  @Input() todo: Todo;
  @Output() done = new EventEmitter<Todo>();

  markAsDone() {
    this.done.emit(this.todo);
  }
}
```

Exemple Input + Output

# 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

Leçon

[AngularFrance.com](https://angularfrance.com)

Copyright 2016-2018 - 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**<sup>(1)</sup>.

Exemples :

- La méthode **ngOnInit** est appelée **une seule fois après l'instanciation du composant**. Parfaite pour un **travail d'initialisation**. Peut aussi être utilisée pour récupérer les inputs d'une directive/composant, qui ne sont pas encore évalués à l'exécution du `constructor()`.
- La phase **ngOnChanges** est appelée **à chaque fois que la valeur d'un input est modifiée**.
- Ces méthodes sont **reconnues** et **exécutées automatiquement** par Angular si elles sont implémentées.
- OPTIONNEL. Le framework propose des **interfaces TypeScript** qui garantissent une implémentation correcte de ces méthodes (ex : `OnInit`).

[AngularFrance.com](http://AngularFrance.com)

130

Copyright 2016-2017 - Toute reproduction interdite

Récap des méthodes “cycle de vie” importantes :

- `ngOnChanges` - Appelé quand les inputs changent, et reçoit en paramètre la dernière valeur des inputs.
- `ngOnInit` - Appelé une fois à l'initialisation du composant (ou de la directive), juste après le premier appel à `ngOnChanges`.
- `ngDoCheck` - Appelé à chaque détection de changement.
- `ngOnDestroy` - Appelé juste avant la destruction d'un composant ou d'une directive.

Voir la liste complète : <https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html>

# Exemple : `ngOnInit`

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

@Directive({
  selector: '[initDirective]'
})
export class OnInitDirective implements OnInit {

  @Input() pony: string;

  ngOnInit() {
    console.log(`inputs are ${this.pony}`);
  }
}
```

Notons que l'interface `OnInit` est totalement optionnelle. Elle permet de garantir que la méthode `ngOnInit()` est bien implémentée sur la classe, mais ce n'est pas grâce à elle que la méthode `ngOnInit()` est appelée.

# Projection

Leçon

[AngularFrance.com](https://angularfrance.com)

Copyright 2016-2018 - Toute reproduction interdite

# 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 { }
```

Angular France.com

Copyright 2016-2017 - toute reproduction interdite

NB. Le contenu projeté est compilé par Angular.

Par exemple, on pourrait avoir le markup suivant dans un <tab> :

```
<tab>
  <p *ngIf="isLoggedIn">{{ userName }}</p>
  <meteo ville="Paris"></meteo>
</tab>
```

L'instruction `*ngIf` et le composant `</meteo>` seront compilés et exécutés par Angular

# Passer des infos à un composant

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

En résumé, on a 2 techniques pour passer des infos à un composant :

- Input
- Projection

# Le Quiz Player

Leçon

[AngularFrance.com](https://angularfrance.com)

Copyright 2016-2018 - 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**.

Titre du quiz

Démarrer

QuizPlayerComponent

< Précédent

Q: 5/10 - Score: 3

Suivant >

QuizNavComponent

Titre de la question :

- Choix 1

- Choix 2

- Choix 3

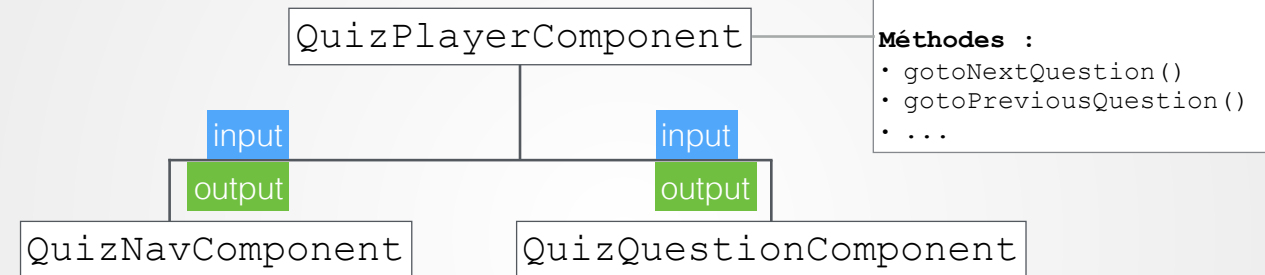
- Choix 4

QuizQuestionComponent

Soumettre



# 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

# Services

Définition,  
Utiliser un service, Créer un service

Leçon

# Services - Définition

- Un service peut contenir deux 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).

# Services - Bonnes pratiques

- Un service doit être **focalisé sur une tâche bien précise**, comme l'**authentification**, le **logging**, la **communication avec la base de données**...
- **Service vs Composant** - 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**.

# Utiliser un service

- Angular propose nativement plusieurs services : **Http**, **Router**...
- On les utilise grâce à l'**injection de dépendances**, en **deux étapes** :
  1. **Déclarer** le service à l'injecteur.
  2. **Injecter** le service dans le constructeur du composant/service qui en a besoin.
- Voici un **exemple d'utilisation du service **Title**** qui permet de **changer dynamiquement le titre de la page** :

```
import { Component } from '@angular/core';
import { Title } from '@angular/platform-browser';
@Component({
  selector: 'my-app',
  providers: [Title],
  template: `<h1>My App</h1>`
})
export class AppComponent {
  constructor(title: Title) {
    title.setTitle('Une super appli');
  }
}
```

AngularF

production interdite

# Créer son propre service

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

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

@Injectable()
class RacesService {
  list() {
    return [{name: 'London' }];
  }
}
```

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

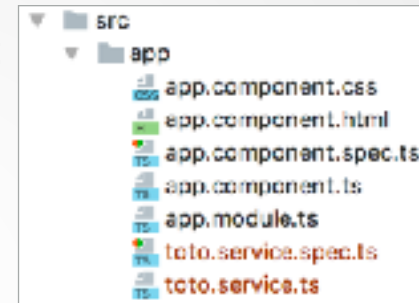
(1) Un service singleton est un service qui n'existe qu'**en un seul exemplaire**. C'est pratique lorsque le service contient des données qui ne doivent exister qu'en un seul exemplaire pour être partagées dans toute l'application, par exemple l'utilisateur courant, le contenu d'un panier, etc.  
ATTENTION. Les services **peuvent être déclarés à l'injection à plusieurs endroits** (dans les `providers` d'un module, dans les `providers` d'un composant, etc.). Si vous déclarez le même service à plusieurs endroits, vous risquez d'obtenir des instances différentes lors de l'injection, et donc de ne pas avoir de véritable singleton.



# Service avec le CLI

- Générer un service TotoService :

```
ng generate service toto
```



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

- Options :

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

AngularFrance.com

145

Copyright 2016-2017 - Toute reproduction interdite

Par défaut, le CLI génère un service avec les fichiers suivants :

- toto.service.ts : Classe du service
- toto.service.spec.ts : Tests unitaires du service

DOC: <https://github.com/angular/angular-cli/wiki/generate-service>

# Injection de dépendances (DI)

Définition, Déclarer une dépendance (globale vs locale),  
Utiliser une dépendance

Leçon

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

- **Design pattern bien connu**, pas spécifique à Angular.
- Supposons un **composant** qui a besoin de faire appel à des **fonctionnalités définies ailleurs dans l'application**, typiquement **dans un service**. C'est ce qu'on appelle une **dépendance** : le composant dépend du service.
- Au lieu de laisser au composant la responsabilité de mettre la main sur le service et de l'instancier, **c'est le framework qui va localiser et instancier le service, et le fournir au composant** qui en a besoin. Cette façon de procéder se nomme **l'inversion de contrôle**.

# DI - Bénéfices

- **Développement simplifié.** On exprime juste ce que l'on veut, où on le veut, et le framework se charge du reste.
- **Tests simplifiés.** Car on peut remplacer les dépendances par des versions bouchonnées.
- **Configuration simplifiée.** On peut permuter facilement différentes implémentations.

# DI - Déclarer et Injecter

- L'injection de dépendances dans Angular se fait **en deux temps**. Il faut :
  1. **DÉCLARER les dépendances qu'on va utiliser** dans l'application, pour les rendre disponibles à l'injection dans d'autres composants/services.
  2. **INJECTER chaque dépendance dans le composant ou service** qui en a besoin.
- **Le framework se charge du reste** : quand on injecte une dépendance dans un composant ou un service, Angular la cherche dans son registre de dépendances, récupère l'instance existante ou en crée une nouvelle, puis réalise l'injection.

# DI - DÉCLARER une dépendance (1/2)

- Pour rendre une dépendance injectable, il faut d'abord la déclarer. Cette déclaration peut se faire **à 2 endroits** :

	Option 1	Option 2
Où ?	Dans un NgModule	Dans un composant
Portée ?	GLOBALE. La dépendance est injectable partout dans l'application.	LOCALE. La dépendance 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 de déclaration	<pre>@NgModule({   imports: [ CommonModule ],   providers: [ AuthService ] }) export class MyModule { }</pre>	<pre>@Component({   selector: 'my-comp',   template: '&lt;p&gt;Hello&lt;/p&gt;',   providers: [ AuthService ] }) export class MyComp { }</pre>

Notez que les injectables - les “providers” - sont déclarés sous forme de **tableau**. On peut en déclarer autant que souhaité dans ces tableaux :

```
@Component ({
  ...
  providers: [Http, AuthService],
  ...
})
```

# DI - INJECTER

## une dépendance (2/2)

- Injecter une dépendance permet de **recupérer une instance prête à être utilisée**.
- L'injection se fait **dans la fonction `constructor()`** d'un **composant**, d'une **directive** ou d'un autre **service** :

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

- Syntaxe : `constructor(param1: MyService, param2: OtherService)`

Ici, le composant `MyComp` a pour dépendance le service `AuthService`. Il pourrait bien sûr y avoir **autant de dépendances que souhaité**.

La convention est de nommer le paramètre qui contient le service avec le même nom que le service, **en commençant par une minuscule**.

ATTENTION. Le mot-clé `private` ne fait pas partie de l'injection de dépendance. Il permet juste d'utiliser la fonctionnalité des **propriétés-paramètres de classe** de TypeScript. En ajoutant un modifieur de visibilité devant les arguments du `constructor` (`public`, `private`...), on en fait automatiquement des propriétés de la classe. En d'autres mots, `authService` est affecté à `this.authService`, et il devient utilisable dans toutes les méthodes de la classe, pas seulement dans le `constructor()`.

# DI - Quelles “dépendances” ?

- **Que peut-on utiliser comme “dépendance” ?**

- Un **service** (service natif Angular ou service custom), dans 90% des cas.
- Une **valeur** (par ex, une constante, des paramètres de configuration...).
- Une **classe**, une **fonction factory** (permet d’avoir une dépendance générée à la volée).

- La **syntaxe de la déclaration** change en fonction du type de valeur<sup>(1)</sup> :

```
// Service
providers: [
  LoggerService, UserContextService, UserService
]
// Valeur
providers: [
  { provide: 'TITLE', useValue: 'Hero of the Month' }
]
// Classe
providers: [
  { provide: LoggerService, useClass: DateLoggerService }
]
```

Angular

Injection interdite

Pour injecter une factory, utiliser :

```
{ provide: XXX, useFactory: FactoryName}.
```

1) La syntaxe de l’injection des dépendances peut aussi changer légèrement. Pour injecter une classe, c’est la syntaxe habituelle. Pour injecter un token comme 'TITLE', on utilise la syntaxe:

```
constructor(@Inject('TITLE') private title: string) { }
```



# DI - Injecteurs hiérarchiques

- En réalité, il y a **plusieurs injecteurs de dépendance** dans une application Angular :
  - Il y a un **injecteur global**, qui regroupe tous les services déclarés dans les **@NgModule.providers** (autrement dit : **PAS DE SCOPE PAR MODULE**).
  - Il y a un **injecteur par composant**, qui contient les services déclarés dans **@Component.providers**. Cet injecteur hérite de l'injecteur de son composant parent, qui hérite également de son parent, etc.
- **Conclusions :**
  - Les dépendances **déclarées dans l'injecteur global** sont **utilisables PARTOUT** dans l'application.
  - Quand une dépendance est **injectée dans un composant précis**, Angular la cherche dans les providers de ce composant. S'il ne la trouve pas, il **remonte l'arborescence des composants**, et **renvoie la première correspondance trouvée**. **Ou une erreur** s'il atteint la racine des composants sans rien avoir trouvé.
  - **Attention aux conflits**, au cas où vous **déclarez plusieurs fois la même dépendance à différents niveaux de la hiérarchie**<sup>(1)</sup>.

- 1) Si vous déclarez la même dépendance à la fois dans `@NgModule.providers` ET dans l'attribut `providers` d'un décorateur `@Component`, ce sont deux instances différentes qui seront créées et utilisées par votre application !
- Dans le composant avec l'attribut `providers`, et dans ses composants enfant, vous obtiendrez une certaine instance de la dépendance. Et dans tout le reste de l'application, vous obtiendrez une autre instance.
- Cela peut être souhaitable mais souvent, on préfère n'avoir qu'un seul exemplaire du service pour toute l'application. **Soyez donc vigilant à ne pas déclarer plusieurs fois le même service.**

# QUIZ #5

Avez-vous compris les services ?

<https://kahoot.it/>

Quiz

# Le QuizService

Leçon

[AngularFrance.com](https://angularfrance.com)

Copyright 2016-2018 - 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**<sup>(1)</sup>. Il est donc utile ou nécessaire de les reformater avant usage.

(1) Côté back, les données ont un format optimisé pour le **stockage** et le **requêtage**. Côté front, elles ont un format optimisé pour l'**affichage**.

# EXO 8

- Créer un service d'accès aux données.

# Le QuizStateManager

Leçon

[AngularFrance.com](http://AngularFrance.com)

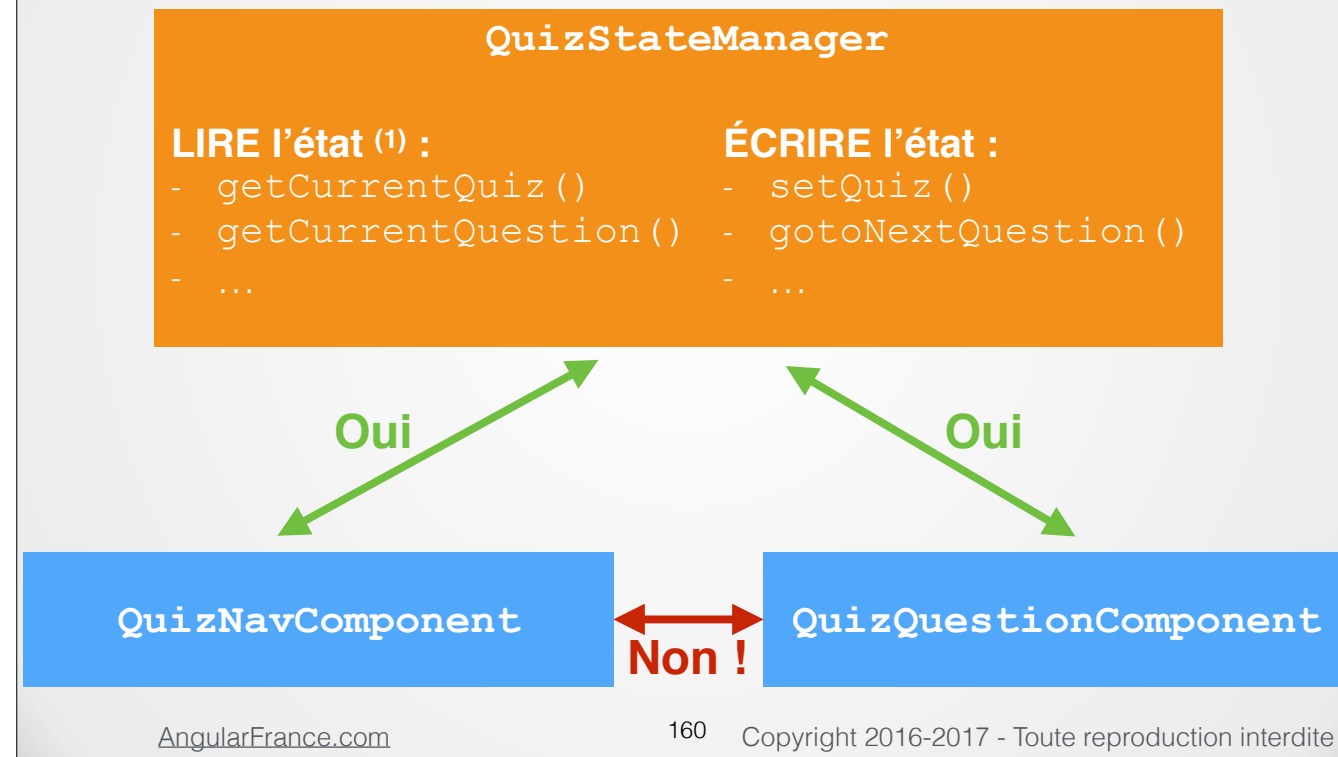
Copyright 2016-2018 - Toute reproduction interdite

# QuizStateManager

## Présentation

- Nous allons utiliser un service **QuizStateManager** pour **gérer l'état du QuizPlayer**, qui est relativement complexe.
- Ce service contient **3 propriétés** :
  - Le **quiz en cours**
  - La **question en cours**
  - Toutes les **réponses soumises** par l'utilisateur
- Et **plusieurs méthodes** :
  - Pour **recupérer** l'état: `getCurrentQuiz()`, `getCurrentQuestion()` ...
  - Pour **modifier** l'état : `setQuiz()`, `setCurrentQuestion()`, `gotoNextQuestion()`, `gotoPreviousQuestion()` ...

# Workflow avec le QuizStateManager



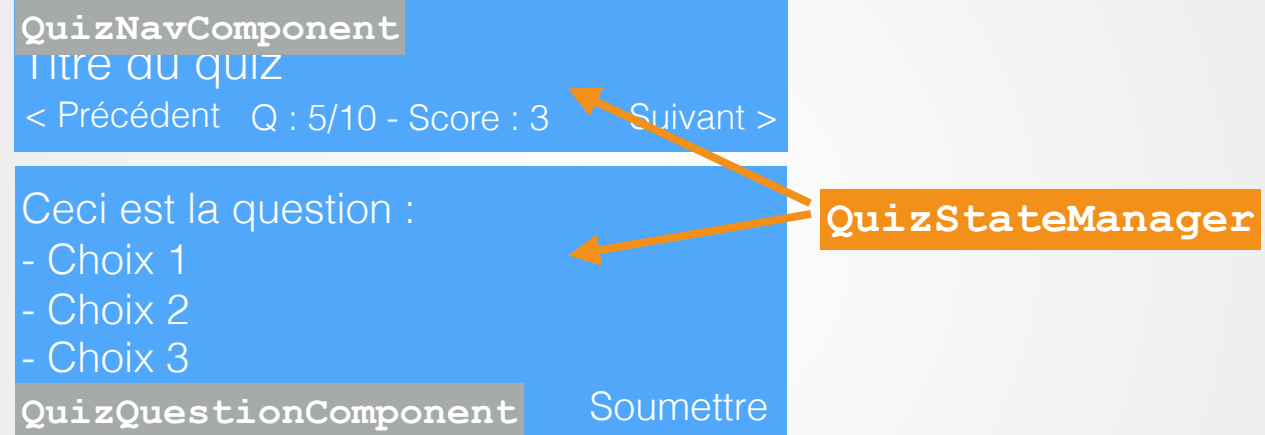
Toutes les communications passent par le service `QuizStateManager`.

Le `QuizStateManager` représente la **source de vérité**.

Les composants ne se parlent jamais directement.

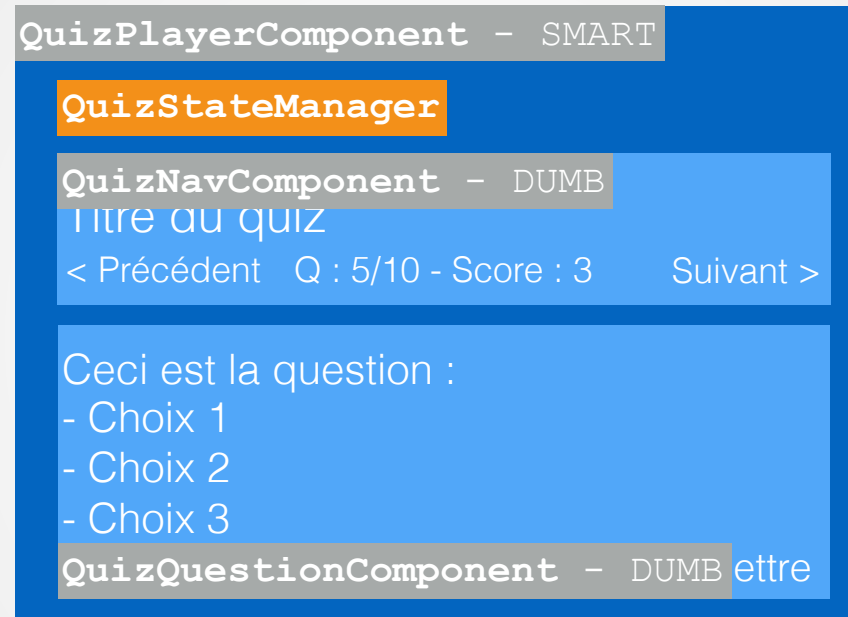


# Implémentation #1



- Le service `QuizStateManager` est injecté dans les 2 composants.

# Implémentation #2



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

# QuizStateManager

## Des observables dans tous les coins

- Toutes les valeurs renvoyées par le QuizStateManager sont des **observables**.
- Pourquoi ? Un observable permet de créer une “**valeur live**” : quand une partie de l’application modifie une valeur wrappée dans un observable, toutes les parties de l’application qui utilisent cette valeur sont averties.

### Sans observable

```
// Dans QuizQuestionComponent
const q = QSM.getCurrQuestion();

// Puis, dans QuizNavComponent
QSM.gotoNextQuestion();

// Dans QuizQuestionComponent
// `q` n'a pas changé
```

### Avec observable

```
// Dans QuizQuestionComponent
const q$ = QSM.getCurrentQuestion();

// Puis, dans QuizNavComponent
QSM.gotoNextQuestion();

// Dans QuizQuestionComponent
q$.subscribe(quest => {
  console.log('New question', quest);
})
```

# Observables dans un template

## Le *pipe* `async`

- Dans un template, le *pipe* `async` permet d'**extraire en temps réel la valeur wrappée dans un observable** :

```
<p>
  {{ currentQuestion | async }}
</p>

<!-- Parenthèses pour accéder à une propriété -->
<p>
  {{ (currentQuestion | async).title }}
</p>
```

- En coulisse, le *pipe* `async` :
  - **S'abonne** à l'observable avec `subscribe()`, et renvoie la ou les valeur(s) émise(s).
  - **Se désabonne** lorsque le composant correspond est détruit. Bien pour éviter les fuites mémoire.

DOC async pipe : <https://angular.io/docs/ts/latest/guide/pipes.html#async-pipe>

(1) En effet, avec Angular, une requête HTTP est wrappée dans un observable.

# 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

Leçon

[AngularFrance.com](https://angularfrance.com)

Copyright 2016-2018 - Toute reproduction interdite

# 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 **formulaire**.
- **HttpModule** - 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 { }
```

Un module Angular est une classe décorée avec le décorateur `@NgModule`. Voici ses propriétés :

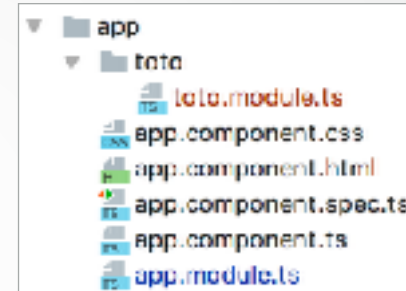
- `declarations` : Liste des classes de composants, de directives et de pipes *appartenant* à ce module.
- `providers` : Liste des services ou valeurs injectables dans toute l'application.
- `imports` : Permet d'utiliser les fonctionnalités d'un autre module dans ce module.
- `exports` : Permet de rendre des composants/directives/pipes de ce module accessibles aux modules qui importeront ce module.(propriété).

- (1) Les composants, directives et pipes ne peuvent appartenir qu'à UN SEUL module. NE JAMAIS RE-DÉCLARER des classes qui appartiennent à un autre module.
- (2) Les services déclarés dans `@NgModule.providers` sont disponibles PARTOUT dans l'application, **quel que soit le module où ils sont déclarés**. Autrement dit, il n'y a PAS de cloisonnement des services par module.

# 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

Par défaut, le CLI génère le module dans un répertoire `toto` avec le fichier suivant :

- `toto.module.ts` : Classe du module

DOC: <https://github.com/angular/angular-cli/wiki/generate-module>

# 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'import	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

Leçon

# 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 est 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

Leçon

[AngularFrance.com](https://angularfrance.com)

Copyright 2016-2018 - Toute reproduction interdite

# 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*



# Utiliser le routeur

Préparer, Définir les routes, Naviguer

Leçon

# 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
```

Pour info, le routeur d'Angular est fortement inspiré d'`ui-router`, un module très souvent utilisé pour remplacer le routeur natif et trop limité d'AngularJS 1.x (`ngRoute`).

# Routeur - Tâche préparatoire

- Définir le **base href**<sup>(1)</sup> 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.


(1) Le base href représente l'**URL de base à partir de laquelle toutes les URLs intra-app seront construites**. Ce paramètre doit être défini pour que les URLs de style HTML5 fonctionnent correctement. (Il s'agit d'URLs qui ressemblent à de véritables URLs serveur, sans caractère #, par exemple `http://localhost:3000/mes-contacts`).  
Si le répertoire `app` représente la racine de votre application, alors le base href est un simple slash `/`.

# 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<sup>(1)</sup>.

Les routes sont un **tableau d'objets**, chacun d'eux ayant au minimum les propriétés suivantes :

- **path** : Chemin sous lequel la route est exposée. L'URL finale ressemblera à `http://exemple.com/path`.
- **component** : Le composant à afficher quand le chemin correspondant est visité par l'utilisateur. ATTENTION, c'est classe du composant qu'on affiche ici (pas son selector) ; il faut donc importer la classe en TypeScript, par exemple : `import { QuizListComponent } from './quiz-list/quiz-list.component';`

(1) On peut effectivement définir les routes dans un module Angular distinct, appelé "module de routing". Il faut taper un peu plus de code, mais cela présente plusieurs avantages :

- Séparation des responsabilités (le module de routing ne contient que des routes, pas de déclarations de composant et autre...).
- Possibilité de remplacer les routes lors des tests.
- Facile d'identifier où les routes ont été déclarées.

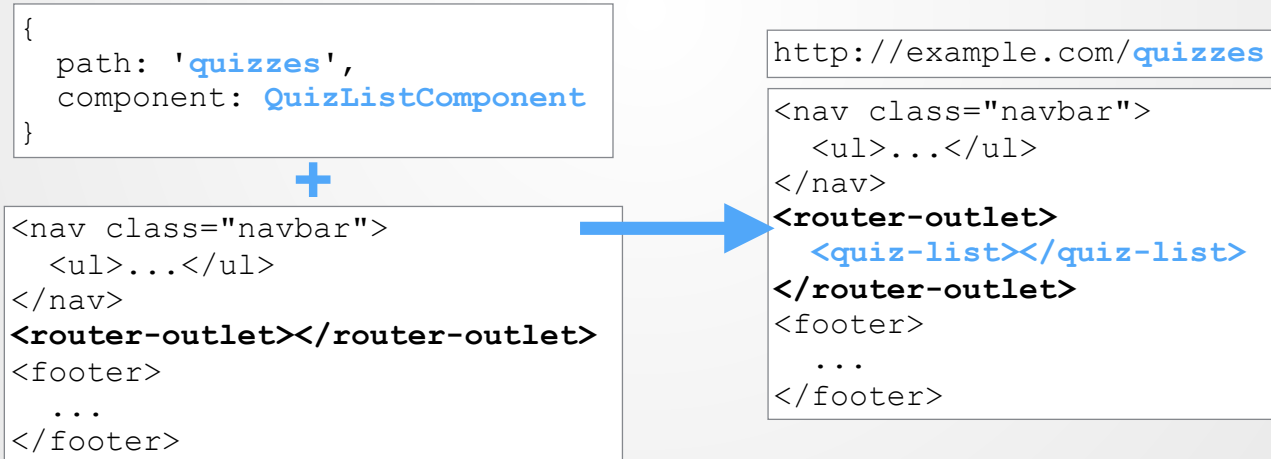
Voir la syntaxe détaillée sur <https://angular.io/guide/router#routing-module>.

# 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 héros 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 une 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>** :



Pour que le composant associé à chaque route soit affiché, il faut insérer un élément spécial dans le template d'un des composants du module : `<router-outlet>`. Cet élément est fourni par une directive Angular, `RouterOutlet`, dont le seul rôle est de marquer l'emplacement du template du composant de la route en cours.

# 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 **programmatically** grâce à **Router.navigate()** :

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

export class ContactCmp {

  constructor(private router: Router) {}

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

AngularFrance.com

191

Copyright 2016-2017 - Toute reproduction interdite

Il y a 2 syntaxes pour écrire une route vers laquelle on fait un lien :

- Chaîne littérale : `'quizzes'`
- Tableau de liens paramètres (*link parameters array*) : `['quizzes']`

La syntaxe tableau est obligatoire avec `Router.navigate()`, ou si la route contient des paramètres.

# 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

Leçon

# 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 });
```

DOC liens relatifs : <https://angular.io/docs/ts/latest/guide/router.html#relative-navigation>

Pour faire un lien relatif avec `Router.navigate()`, il faut explicitement passer la route en cours via le paramètre `relativeTo`.

# 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]);
```

Les paramètres peuvent s'appeler comme on veut. Il suffit qu'ils commencent par le symbole `:` et qu'ils ne contiennent pas de caractères spéciaux.

# Routeur

## Récupérer les params de route

- Récupérer les paramètres de route grâce à `ActivatedRoute.paramMap` :

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

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

- **Pourquoi asynchrone ?** Pour gérer le scénario où le paramètre change, mais l'URL ne change pas :

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

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

⚠ La récupération des paramètres doit se faire dans le composant associé à la route paramétrée.

On peut aussi récupérer les paramètres de manière synchrone avec `ActivatedRoute.snapshot.paramMap` :

```
const userId = this.route.snapshot.paramMap.get('userId');
```

# 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>`

- ATTENTION. Il n'existe que 2 paths en tout, pas 3 !<sup>(1)</sup>

(1) Plus précisément, le path `/heroes` tout court n'existe pas. Pour que ce path soit valide, il faudrait définir un troisième enfant avec un path vide `""`.

# 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>
```

Dernière remarque, on peut appliquer la directive `routerLinkActive` à un ancêtre du `routerLink`. C'est pratique quand on utilise un framework CSS qui impose de placer la classe ailleurs que sur le lien. Exemple avec Bootstrap CSS :

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

DOC : <https://angular.io/docs/ts/latest/api/router/index/RouterLinkActive-directive.html>



# Routes d'un *feature module*

Déclaration, Activation

Leçon

# 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**<sup>(1)</sup> :

```
@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<sup>(2)</sup> :

```
{ path: 'path-to-quiz', loadChildren: 'app/quiz/quiz.module#QuizModule' }

// Chemins finaux
/path-to-quiz/quizzes
/path-to-quiz/quiz/:id
```

AngularFrance.com

203

Copyright 2016-2017 - Toute reproduction interdite

- (1) Il est important de comprendre que les routes d'un *feature module* viennent s'ajouter **au même niveau** que les routes du module principal. Il n'y a pas de notion d'imbrication. Autrement dit, il n'y a rien qui permette de différencier les routes qui viennent du module principal de celles qui viennent d'un *feature module*. **Toutes ces routes s'affichent d'ailleurs dans le même <router-outlet>.**
- (2) Avec la syntaxe `loadChildren`, il NE FAUT PAS lister le *feature module* dans les imports du module principal. Cela irait à l'encontre du *lazy-loading* du module géré par le routeur.

# QUIZ #7

Avez-vous compris le routeur ?

<https://kahoot.it/>

Quiz

# EXO 11

- Créer les routes de l'application

# 7. HTTP

Module

# HTTP - Introduction

Leçon

[AngularFrance.com](https://angularfrance.com)

Copyright 2016-2018 - Toute reproduction interdite

# 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 - 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 resp = http.get('/api');
```

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

# HTTP - Tâche préalable

- Pour pouvoir utiliser le **service** `HttpClient` dans votre application, il faut importer le **module** `HttpClientModule` dans l'un de vos modules. 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 GET

Leçon

# HTTP - Faire une requête GET et afficher les données reçues

Afficher avec `subscribe()`

```
@Component({
  template: `
    <ul>
      <li *ngFor="let q of qList">
        {{q.title}}
      </li>
    </ul>`
})
export class QuizListComponent {
  // ⚠ undefined
  qList: Quiz[];

  constructor(
    private http: HttpClient
  ) {}

  ngOnInit() {
    this.http.get('/api/quizzes')
      .subscribe(data =>
        this.qList = data);
  }
}
```

AngularFrance.com

Afficher avec `async`

```
@Component({
  template: `
    <ul>
      <li *ngFor="let q of qList$ | async">
        {{q.title}}
      </li>
    </ul>`
})
export class QuizListComponent {
  // ⚠ undefined
  qList$: Observable<Quiz[]>;

  constructor(
    private http: HttpClient
  ) {}

  ngOnInit() {
    this.qList$ =
      this.http.get('/api/quizzes');
  }
}
```

212 Copyright 2016-2017 - Toute reproduction interdite

Ce slide présente deux manières d'afficher les données renvoyées par une requête HTTP : l'une utilise un `subscribe` explicite, l'autre utilise le pipe `async`.

- 1) On déclare une propriété de classe destinée à recevoir les données renvoyées par la requête HTTP. Attention, cette propriété a un type TypeScript, mais pas de valeur ; elle vaut donc `undefined`.
- 2) On injecte le service `HttpClient` avec la syntaxe habituelle d'injection de dépendance.
- 3) À gauche, on fait la requête et on s'abonne à l'observable renvoyé avec `subscribe()`. On récupère les données dans le callback passé à `subscribe`, et on les affecte à une propriété de classe. À droite, on fait la requête et on affecte l'observable renvoyé directement à une propriété de classe.
- 4) À gauche, on peut afficher la propriété de classe telle quelle, puisqu'elle contient des données. À droite, on doit passer la propriété de classe dans le pipe `async`, puisqu'elle contient un observable auquel il faut s'abonner.

# HTTP - Syntaxes diverses

- Récupérer des données JSON<sup>(1)</sup> :

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

- Associer un type aux données récupérées :

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

- Récupérer la **réponse entière** :

```
http.get<MyJsonData>('/api/json-data', {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'); }  
  );
```

AngularFrance.com

213

Copyright 2016-2017 - Toute reproduction interdite

- (1) Pour les données non-JSON, il faut passer le paramètre `responseType` :

```
http.get('/textfile.txt', {responseType: 'text'})
```

# HTTP POST

Leçon

# 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

Leçon



# 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**<sup>(1)</sup> :

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

(1) Le pipe `async` (permettant de récupérer la dernière valeur émise par un observable directement depuis un template) crée un nouveau `.subscribe()` à chaque fois qu'il est utilisé, même s'il est utilisé sur le même observable. Si l'observable en question wrappe une requête HTTP, elle sera exécutée de multiples fois.

# HTTP - Transformer les données reçues

- **Tâche fréquente** : Transformer les données renvoyées par le serveur pour qu'elles aient le format attendu par l'application.
- **Exemple** : Récupérer uniquement la propriété `race.name` alors que le serveur renvoie un **tableau d'objets** `race` entiers :

```
import 'rxjs/add/operator/map'; (1)

http.get(`${baseUrl}/api/races`)
  .map((races: Array<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.

(1) Importe l'opérateur `map` de RxJS. Il faut importer explicitement les opérateurs dont on a besoin.

(2) Transforme le tableau de `[races]` en tableau de `[names]`. Ne pas confondre l'opérateur RxJS `Observable.map()` avec `Array.prototype.map()`. Ici, les deux sont utilisés sur la même ligne.

# HTTP - Bonne pratique

- Séparer le code qui **requête/transforme** des données du code qui **consomme** ces données.

## 1. Requêter et transformer les données dans un service

```
class QuizService {  
  // Renvoie un observable  
  getQuizzes(): Observable<any> {  
    return this.http.get(`/quizzes`)  
      .map(data => otherData)  
      .mergeMap(...)  
      .reduce(...);  
  }  
}
```

## 2. Consommer les données finales en s'abonnant à l'observable renvoyé par le service

```
this.quizService.getQuizzes()  
  .subscribe(finalData => {  
    console.log(finalData);  
  });
```

- **BÉNÉFICE** : Complexité masquée + tous les utilisateurs du service reçoivent les mêmes données, au bon format.

# EXO 12

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

# 8. Formulaires

Créer et valider un formulaire

# Formulaires - Intro

Activation, Présentation des 2 syntaxes  
Points commun aux 2 syntaxes

Leçon

# Formulaires - Introduction

- Les formulaires sont **au coeur 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<sup>(1)</sup>.

(1) Reportez-vous à la doc pour voir la syntaxe des formulaires classiques : <https://angular.io/guide/forms>



# 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"<sup>(1)</sup>).

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.

(1) Pour la syntaxe pilotée par le template, le module à importer est `FormsModule`.

# Formulaires réactifs

Template, Modèle,  
Récupération des données

Leçon

[AngularFrance.com](https://angularfrance.com)

Copyright 2016-2018 - Toute reproduction interdite

MDF = *Model-driven Forms*, aka *Reactive Forms*.

# 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 qui 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.

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

(1) Toutes les syntaxes d'un formulaire HTML classique sont autorisées.

(2) La directive `formGroup` permet de relier le formulaire au modèle déclaré dans le composant. Elle est entre `[...]` parce qu'elle est bindée à une variable. Mais maintenant, vous savez bien pourquoi on met des crochets. 🤔

(3) La directive `formControlName` permet de relier chaque champ HTML au contrôle correspondant dans le modèle. Pas de `[...]`, car cette directive est bindée à une chaîne littérale.

(4) La directive `ngSubmit` permet d'attacher un *event listener* à l'événement "soumission du formulaire".

Il existe un événement DOM natif appelé `submit`. On pourrait donc en théorie utiliser la syntaxe `<form (submit)="doSomething()">`. Le problème est que si le *submit handler* contient une erreur et plante, une requête HTTP POST sera envoyée au serveur. La directive `ngSubmit` permet d'éviter ce comportement non souhaitable.

# Formulaire réactif - Classe 1/2

```
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  
  }  
}
```

Syntaxe équivalente sans passer par le FormBuilder :

```
this.contactForm = new FormGroup({  
  firstname: new FormControl('', Validators.required),  
  lastname: new FormControl('', Validators.required),  
  address: new FormGroup({  
    street: new FormControl(),  
    city: new FormControl()  
  })  
});
```

# 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 - Validateurs

- Pour valider les champs, on leur associe des **validateurs**. Un validateur retourne une **map des erreurs**, ou **null** si aucune n'a été détectée.
- **Quelques validateurs** fournis par Angular :
  - **Validators.required** - Valeur doit être non vide.
  - **Validators.email()** - Valeur doit être un e-mail valide.
  - **Validators.pattern(regex)** - Valeur doit matcher la regex.
- Plusieurs validateurs peuvent être **appliqués au même champ** en passant un **tableau de validateurs** au FormBuilder :

```
firstname: ['', [Validators.required, Validators.maxLength(20)]]
```

- Les validateurs peuvent porter sur un `FormControl` (= un champ individuel) ou sur un `FormGroup` (= un groupe de champs).



# Formulaire réactif

## Accéder au champ

- On **accède** au champ avec la syntaxe **FORM.get(CHAMP)** :

```
// `firstName` contient une instance de FormControl  
const firstName = this.contactForm.get('firstName');
```

- On peut alors lire la **valeur** du champ :

```
const val = this.contactForm.get('firstName').value;  
  
// Ou bien, si on a déjà récupéré le FormControl  
const val = firstName.value;
```

- On peut aussi accéder à la **validité** du champ :

```
// true ou false selon la validité  
this.contactForm.get('firstName').valid  
  
// Erreurs associées au champ  
this.contactForm.get('firstName').errors
```

# Formulaire réactif

## Afficher les erreurs

- L'affichage des erreurs se fait **dans le template**, et s'appuie **sur le modèle** :

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

- En effet, les règles de validation ont été définies **dans le modèle**.
- **Désactiver le bouton Submit** en cas d'erreur<sup>(1)</sup> :

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

(1) On voit que le `FormGroup` a lui aussi une propriété `invalid`, exactement comme les instances de `FormControl`.

# Formulaire réactif

## Récupérer les données du form

- Dans la **classe associée au formulaire**, on peut récupérer les données de la manière suivante :

- **De l'ensemble du formulaire :**

```
saveContact() {  
  // `contactForm` est déjà une propriété de la classe  
  // On n'a rien à faire pour le récupérer.  
  const formValue = this.contactForm.value;  
}
```

- **D'un champ précis :**

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

# FormGroup et FormControl

## Propriétés

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

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

# FormGroup et FormControl

## Méthodes

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

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

# Formulaires

## Résumé des 2 syntaxes

Leçon

# Syntaxe modèle vs template

	Formulaires classiques	Formulaires réactifs
Module Angular à importer	FormsModule	ReactiveFormsModule
Syntaxe <form>	<pre>&lt;form #contactForm="ngForm"&gt;</pre>	<pre>&lt;form [formGroup]="contactForm"&gt;</pre>
Syntaxe champ	<pre>&lt;input name="contactName" [ (ngModel) ]="contact.name" #contactName&gt;</pre>	<pre>&lt;input formControlName="firstName"&gt;</pre>
Valeur d'un champ	contact.name	contactForm.get('firstName').value
Validité d'un champ	contactName.valid	contactForm.get('firstName').valid
Adapté pour	Formulaire simple Validation standard	Formulaire complexe Validation complexe

# QUIZ #8

Avez-vous compris les formulaires ?

<https://kahoot.it/>



# EXO 13

- Créer le formulaire de création/édition d'un quiz.

# 9. Observables & RxJS

Programmation réactive,  
Librairie RxJS, Opérateurs RxJS

# Programmation réactive

Leçon

[AngularFrance.com](https://angularfrance.com)

Copyright 2016-2018 - Toute reproduction interdite

# 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 déboguer.

# RxJS

Librairie ReactiveX pour JavaScript  
Observable, Observer, Subject

# Leçon

[AngularFrance.com](https://angularfrance.com)

Copyright 2016-2018 - 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 “bibliothèque”.
- 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.

Avant de créer nos propres observables, on peut manipuler ceux qui sont renvoyés par Angular.

# Observable - Création (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 - Création (3/3)

- Enfin, la méthode **Observable.create()** permet de créer son propre Observable **manuellement**.
- Elle prend en argument une fonction qui reçoit un **observer** permettant d'émettre les 3 événements 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');  
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');
```

**S'abonner = Invoquer l'observable**

- 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
```

S'abonner à un observable est analogue à invoquer une fonction.

Si personne ne s'abonne à un observable, il n'est pas exécuté.

# 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** <sup>(1)</sup> :

```
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** <sup>(2)</sup> :

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

(1) Rien n'oblige à définir les 3 callbacks. Vous pouvez créer un Observer qui implémente uniquement l'un des callbacks, ou deux, ou trois...

(2) Une syntaxe alternative consiste à passer directement les trois callbacks à la méthode `subscribe` :

```
observable.subscribe(  
  callbackNext,  
  callbackError,  
  callbackComplete  
);
```

# 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**<sup>(1)</sup>.
- Pour les éviter, pensez à vous **désabonner** de vos observables **à la destruction** du composant grâce au hook `ngOnDestroy()`<sup>(2)</sup> :

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

AngularFrance.com

200

Copyright 2016-2017 - toute reproduction interdite

(1) Les observables qui se terminent (événement “complete”) sont désabonnés automatiquement. C’est le cas par exemple de l’observable renvoyé pour une requête HTTP : après que la requête a renvoyé ses données, l’observable se termine et le désabonnement est automatique. La remarque ci-dessus s’applique **uniquement aux observables qui ne se terminent pas**, par exemple un *timer*.

(2) On voit dans ce code que `Observable.subscribe()` renvoie un objet `Subscription` qui expose une méthode `.unsubscribe()`.

DOC `ngOnDestroy` : <https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html#ondestroy>

# Quelques opérateurs RxJS

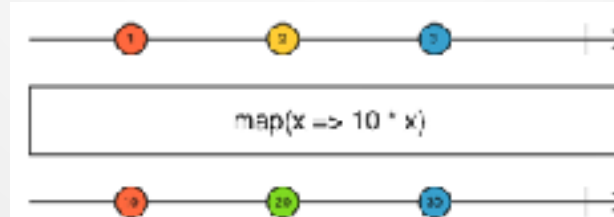
map, mergeMap, reduce, filter...

Leçon



# 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**<sup>(1)</sup>:



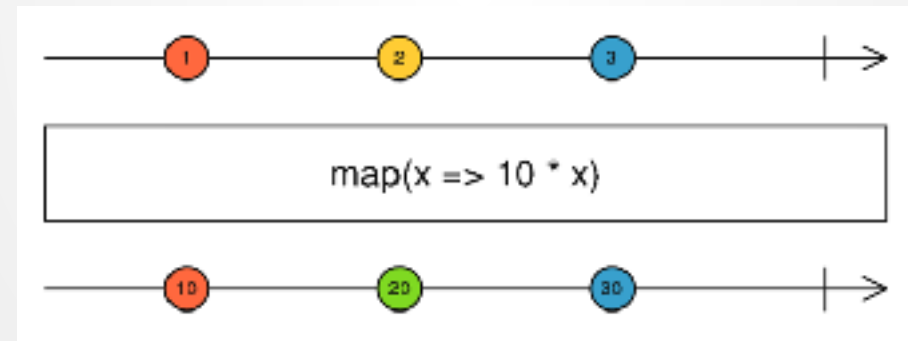
AngularFrance.cc

Toute reproduction interdite

(1) Le *marble diagram* représente l'Observable en entrée sur la flèche du haut et l'Observable en sortie (transformé) sur la flèche du bas. La flèche représente le temps qui passe (de gauche à droite) et les petits ronds les valeurs émises au fil du temps. Le rectangle central représente l'opérateur qui permet de passer de l'entrée à la sortie.

# Opérateur - map

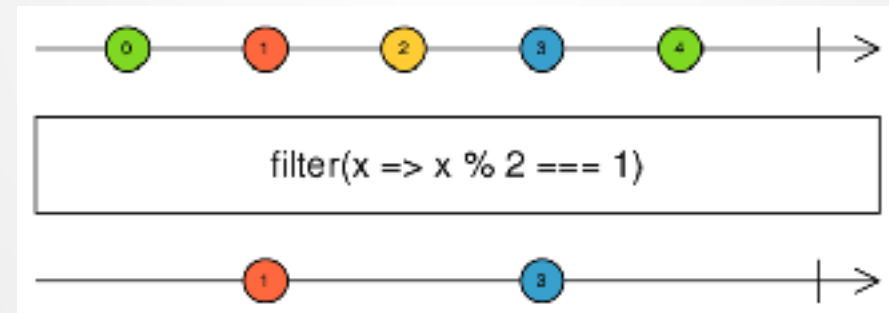
- Opérateur de **transformation**.
- Applique une fonction à chaque valeur émise par l'Observable source, et émet les valeurs transformées sous forme d'un nouvel Observable.



- Analogue à `Array.prototype.map`.

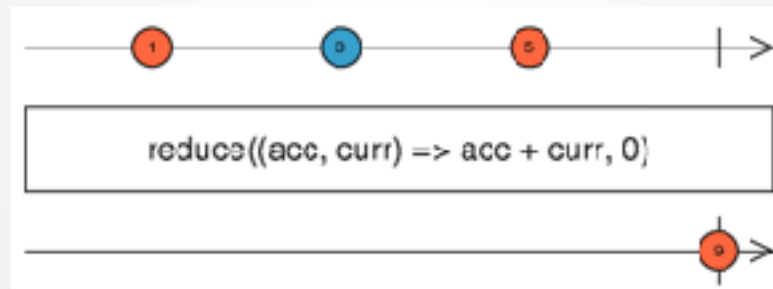
# Opérateur - filter

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



# Opérateur - reduce

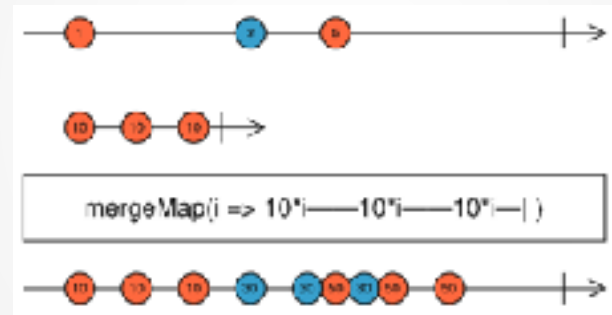
- Opérateur de **transformation**.
- Passe la première valeur de l'Observable source à une fonction d'accumulation, puis passe le résultat de cette fonction ainsi que la 2ème valeur à l'accumulateur, et ainsi de suite jusqu'à ce qu'il ne reste plus qu'une seule valeur. Seule la valeur finale est émise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



# 10. Fonctionnalités avancées

Routeur avancé, Formulaires avancés,  
Affichage avancé, HTTP avancé

# Routeur avancé

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

Leçon

# 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 **recupérer 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] }  
]
```

3. Ajouter ce service aux providers du module adéquat :

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

# 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]}  
]
```

3. Ajouter ce service aux providers du module adéquat :

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

# 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)

3. Ajouter ce service aux providers du module adéquat :

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

4. 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,  
Valdateur custom, Observer les changements

Leçon

# 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>
```

# Valideur 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};
}
```

Dans le premier exemple, on a un contrôle `birthdate` ("date de naissance") avec deux validateurs combinés. Le premier est le validateur natif `required`, et le second est le validateur custom `isOldEnough`.

Ensuite, dans le template, on peut afficher l'erreur associée au validateur custom de la manière habituelle :

```
<div>
  <label>Date de naissance</label>
  <input type="date" ngControl="birthdate">
  <div *ngIf="birthdate.dirty">
    <div *ngIf="birthdate.hasError('required')">La date de naissance est obligatoire.</div>
    <div *ngIf="birthdate.hasError('tooYoung')">Vous êtes trop jeune.</div>
  </div>
</div>
```

Notez que la clé de l'erreur testée - `tooYoung` - correspond à la clé renvoyée par le validateur custom.

# 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;
});
```

Le code de `RegisterFormCmp` a été abrégé pour se concentrer sur les éléments essentiels : l'abonnement aux changements du champ `password` via l'observable `valueChanges`.

Bien entendu, le code fourni ici est trop simple : la "force" du mot de passe est uniquement basée sur sa longueur. Mais il serait facile de modifier ce code pour calculer un "score de force" plus réaliste.

La force du mot de passe est exposée dans une variable `passwordStrength`, qui peut être affichée dans le template.

# EXO 16

- Formulaires avancés



# Affichage avancé

Référencer ou modifier l'élément hôte  
Référencer les éléments enfant

Leçon

# 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>Salut toi !</div>`
- 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
  }
}
```

- (1) Cette directive va modifier de 2 manières son élément hôte (c. à d. la balise HTML à laquelle elle sera appliquée) :
- (1) Elle ajoute à son élément hôte un gestionnaire d'événement qui appellera la méthode `displayMessage` quand l'élément hôte sera cliqué. Notez la présence des parenthèses autour de `(click)`, comme quand on ajoute un gestionnaire d'événement directement sur une balise.
  - (2) Elle ajoute la classe CSS `alert-error` à l'élément hôte.

# 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()...`.

# HTTP avancé

Leçon

[AngularFrance.com](https://angularfrance.com)

Copyright 2016-2018 - Toute reproduction interdite

# Modifier toutes les requêtes HTTP

- Créer un service qui hérite de **BaseRequestOptions** :

```
@Injectable()
export class AppRequestOptions extends BaseRequestOptions {
  constructor() {
    super();
    // Ici, accéder et modifier les propriétés via this.XXX
  }
}
```

- Faire de ce service le nouveau **RequestOptions** :

```
{ provide: RequestOptions, useClass: AppRequestOptions }
```

# 11. Tests

Tests unitaires et end-to-end



# Tests - Intro

Leçon

# 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

Leçon

# 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`), exécutez la commande suivante dans le terminal<sup>(1)</sup> :

**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`.

1) Cette commande est spécifique à Angular CLI. La commande npm conventionnelle d'exécution des tests est `npm test`.

Pour exécuter les tests une seule fois (pas en mode watch) :

```
ng test --watch=false
```

Pour exécuter les tests d'intégration (*end-to-end*) :

```
ng e2e
```



# 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', () => {
    ...
  });
});
```

- `describe()` déclare un jeu de tests (un groupe de tests).
- `beforeEach()` permet d'initialiser un contexte avec chaque test. Elle est exécutée avant chaque bloc `it()`.
- `it()` déclare un test individuel.

# 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.configureTestingModule()** - 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)`

Examinons ensemble le code de `app/common/home.component.spec.ts` pour un exemple concret.

# 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`<sup>(1)</sup> :

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

terdite

1) Attention. `AutoDetect` ne détecte que les activités asynchrones (résolution de promesses, timers, événements DOM), mais pas les changements directs de propriété. Cette technique n'est pas recommandée par la doc officielle.

# 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 } ],
  ...
});
```

Pour un stub de composant, il suffit de déclarer dans la propriété `@NgModule.declarations` du module de test un composant “bidon”, spécialement créé pour le test, qui possède le **même selector** que le composant original.

# 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

Leçon



# 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');
  });
});
```

Protractor fournit un objet `browser`, avec quelques méthodes utilitaires comme `get()` pour aller à une URL.

Puis on a `element.all()` pour sélectionner tous les éléments répondant à un prédicat donné. Ce prédicat s'appuie souvent sur `by` et ses méthodes variées (`by.css()` pour faire une requête CSS, `by.id()` pour récupérer un élément par son identifiant, etc...). `element.all()` retourne une promesse, avec une méthode spéciale `count()` utilisée dans le test ci-dessus.

`$('h1')` est un raccourci, équivalent de `element(by.css('h1'))`. Il récupère le premier élément correspondant à la requête CSS.

# EXO 17

- Tests unitaires

# 12. Outillage, i18n et déploiement

Outils divers, Internationalisation,  
Déploiement

# Outillage Angular

TypeScript, Tests, Angular CLI, Augury, Frameworks UI

Leçon

[AngularFrance.com](https://angularfrance.com)

Copyright 2016-2018 - Toute reproduction interdite

Si la syntaxe et l'architecture des applis Angular est plus simple qu'AngularJS 1.x, l'outillage est un peu plus complexe. C'est notamment lié à la compilation TypeScript → JavaScript.

# TypeScript

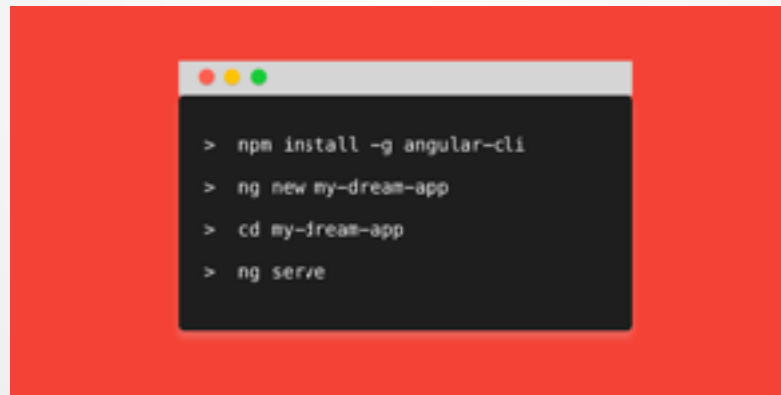
- Langage créé par Microsoft en 2012, open-source, qui **transpile vers JavaScript**.
- **Surensemble d'ES6** (aka ES2015).  
**Tout JavaScript est donc du TypeScript valide.**
- **Principales caractéristiques**<sup>(1)</sup> : types, interfaces, classes, décorateurs, modules, fonctions fléchées, templates chaîne.
- Supporté par de nombreuses **bibliothèques JavaScript tierce-partie**<sup>(2)</sup>.
- **Supporté par plusieurs IDE** : WebStorm/IntelliJ Idea, Visual Studio Code, Sublime Text, etc.
- **Langage le plus populaire pour Angular**. En train de s'imposer comme le langage officiel.

# Tests

- Angular embarque un **module de test** avec toutes les fonctionnalités support et les objets bouchonnés (*mocks*) permettant la mise en place des tests.
- Les tests unitaires sont écrits avec **Jasmine** (<http://jasmine.github.io/>).
- Les suites de tests sont exécutées avec **Karma** (<http://karma-runner.github.io/>) qui permet notamment d'exécuter les tests dans plusieurs navigateurs.
- Les tests d'intégration (*end-to-end*) sont exécutés avec le framework **Protractor** (<http://www.protractortest.org/>).

# Angular CLI

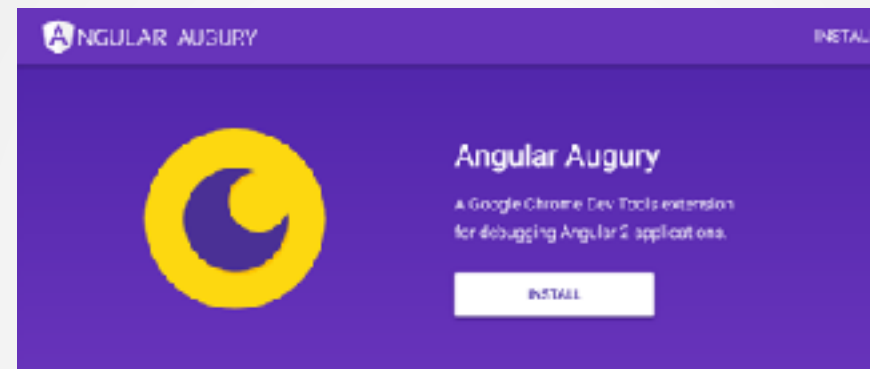
- Outil en ligne de commande (en cours de développement) pour **simplifier les tâches de développement avec Angular**.
- **Fonctionnalités** : génération initiale d'un projet, génération de composants, exécution des tests, déploiement en production...
- <https://github.com/angular/angular-cli>

A terminal window with a dark background and light text, set against a red rectangular background. The terminal shows four commands entered one by one, each preceded by a prompt character '>'. The commands are: 'npm install -g angular-cli', 'ng new my-dream-app', 'cd my-dream-app', and 'ng serve'.

```
> npm install -g angular-cli
> ng new my-dream-app
> cd my-dream-app
> ng serve
```



# Angular Augury



- **Extension Chrome Dev Tools** pour débbugger les applications Angular, et aider les développeurs à comprendre le fonctionnement de leurs applications — <https://augury.angular.io/>.
- **Fonctionnalités** : Comprendre les relations entre composants et leur hiérarchie, obtenir des infos sur chaque composant et modifier leurs attributs à la volée, etc.
- **NOTE.** On peut aussi débbugger avec **Chrome Dev Tools**. Les **source maps** permettent de débbugger le code TypeScript alors que le navigateur exécute du JavaScript.

Angular Augury s'appelait autrefois Batarangle.

# Frameworks UI

- **ng-bootstrap** (<https://github.com/ng-bootstrap/core>) - Ré-écriture en Angular des composants UI de Bootstrap CSS (v4).
- **Angular Material** (<https://material.angular.io/>) - Librairie de composants UI développés par Google spécifiquement pour Angular. Actuellement en **early alpha**, mais développement assez actif.
- **PrimeNG** (<http://www.primefaces.org/primeng/>) - Collection de composants UI pour Angular par les créateurs de PrimeFaces (une librairie populaire utilisée avec le framework JavaServer Faces).
- **Wijmo 5** (<http://wijmo.com/products/wijmo-5/>) - Librairie payante de composants UI pour Angular. Achat de licence nécessaire.
- **Polymer** (<https://www.polymer-project.org/>) - Librairie de “Web Components” extensibles par Google. L’intégration avec Angular est réputée par évidente.
- **NG-Lightning** (<http://ng-lightning.github.io/ng-lightning/>) - Librairie de composants et directives Angular écrits directement en TypeScript sur la base du framework CSS Lightning Design System.

Plusieurs librairies de composants UI sont déjà disponibles pour Angular.

# Internationalisation

Traduire et localiser son application

Leçon

[AngularFrance.com](https://angularfrance.com)

Copyright 2016-2018 - Toute reproduction interdite

# Internationalisation - Intro

- L'internationalisation est complexe et revêt plusieurs aspects, notamment la **traduction des textes** et la **localisation de certaines données** (dates, montants...).
- Dans cette section, nous aborderons **uniquement la traduction du texte** situé dans les **templates** de composant.
- Notez enfin que l'internationalisation n'a été intégrée dans Angular que tout récemment et qu'elle n'offre **pas encore une réponse complète** à la problématique.

# Traduction du texte des templates

1. **Marquer les chaînes à traduire** dans les templates de composant.
2. Utiliser l'outil i18n d'Angular pour **extraire les chaînes à traduire** dans un fichier de traduction source (formats standard supportés, non spécifiques à Angular).
3. Grâce à un outil standard, on peut éditer le fichier de traduction et y **enregistrer les traductions**.
4. Le **compilateur Angular** importe les fichiers de traduction finalisés, remplace les messages originaux par le texte traduit, et génère une “nouvelle” version de l'application dans le langage cible.

# 1. Marquer les chaînes à traduire

- Pour cela, on pose l'attribut spécial `i18n` sur les balises contenant du texte à traduire :

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

- On peut ajouter une **description** pour aider le traducteur :

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

- On peut aussi ajouter un **contexte** :

```
<h1 i18n="User welcome|An introduction header for this sample">Hello i18n!</h1>
```

## 2. Extraire les chaînes à traduire

- Pour cela, on utilise l'outil `ng-xi18n`.
- Cet outil fait partie d'Angular CLI dans le package `compiler-cli` :

```
npm install @angular/compiler-cli @angular/platform-server --save
```

- Ouvrir une fenêtre de terminal à la racine du projet et exécuter la commande `ng-xi18n` :

```
ng xi18n --output-path src/locale
```

- Cela va générer un fichier `messages.xlf` au format XLIFF.

## 3. Traduire le texte

- Grâce aux outils standard de traduction, on peut **produire plusieurs versions traduites** du fichier de traduction source :
  - `messages.fr.xlf`
  - `messages.es.xlf`
  - ...
- Il est recommandé de placer tous ces fichiers dans un **répertoire dédié**, par exemple **locale** à la racine du projet.



## 4. Compiler une version traduite de l'application

- En gros, il faut indiquer à Angular :
  - la **locale** à utiliser (fr, en-US...).
  - le **fichier** de traduction
  - le **format** de ce fichier

- **VERSION AOT:**

```
ng serve --aot --locale fr --i18n-format xlf --i18n-file src/locale/messages.fr.xlf
```

- **VERSION JIT<sup>(1)</sup>:** C'est lors du **bootstrap** (fichier app/main.ts) qu'on passe tous ces paramètres à l'application pour la **démarrer dans une langue précise** :

```
import { getTranslationProviders } from './i18n-providers';

getTranslationProviders().then(providers => {
  const options = { providers };
  platformBrowserDynamic().bootstrapModule(AppModule, options);
});
```

(1) Pour le code détaillé de la version JIT, voir <https://angular.io/guide/i18n>

# Internationalisation

## Limites de la solution actuelle

- Impossible de traduire les **chaînes utilisées dans le code**, par exemple dans la classe d'un composant (par opposition aux chaînes du template) :

```
export class AdminQuizFormComponent implements OnInit {  
  pageTitle = "Update a quiz"; // Non traduisible  
  // ...  
}
```

- Impossible de gérer les chaînes avec des **fragments dynamiques** :

```
<p>The quiz {{ quiz.title }} has been saved.</p>
```

# EXO 18

- Internationalisation

# Déploiement

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

Leçon

[AngularFrance.com](https://angularfrance.com)

Copyright 2016-2018 - Toute reproduction interdite

# Préparer l'application à la mise en production

- **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 avec angular-cli

- Le CLI **simplifie énormément le déploiement**, avec une commande qui crée la version prête à déployer de l'application :

`ng build`

- Ce build peut être **customisé dans une certaine mesure** via des flags (passés à la commande ng build ou au fichier `.angular-cli.json`) :

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

# 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
```

# Déploiement - Compilation AOT

- 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*).
- **Bénéfices :**
  - 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**.
- Avec Angular CLI (expérimental) : `ng build --aot`



# Déploiement - webpack

- **Webpack 2** est un outil populaire pour gérer les problématiques de *inlining*, *bundling*, *minification*, et *uglification*.
- Angular CLI utilise webpack en coulisse.
- Malheureusement, à ce jour, **la configuration de webpack n'est pas directement exposée** aux utilisateurs de Angular CLI.
- **Seuls quelques paramètres** sont exposés via des **flags Angular-CLI** ou via le **fichier angular-cli.json**, mais on n'a pas un contrôle aussi fin qu'avec un déploiement webpack "natif".

# Déploiement - Config Angular

- **Modifier le <base href>.** Sur le serveur de développement, l'appli est typiquement servie à la racine du serveur (<http://localhost:4200/>). En production, l'appli sera peut-être servie depuis un sous-répertoire (<http://mysite.com/my/app/>).

```
ng build --base-href /my/app/
```

- **Activer le mode production<sup>(1)</sup> :**

```
// main.ts
import { enableProdMode } from '@angular/core';
if (!/localhost/.test(document.location.host)) {
  enableProdMode();
}
```

(1) À ce jour, la principale différence entre le mode développement et le mode production est qu'en mode développement, Angular exécute **2 passes de détection de changement à la suite** (au lieu d'une seule en production) pour voir si les valeurs ont changé entre les 2 passes. Cela permet d'identifier un bug où le seul fait de lire/récupérer une valeur produit un effet de bord en ne renvoyant pas la même valeur à chaque fois.

# Déploiement - Config Serveur

- Pour héberger l'application Angular (fichiers `.html`, `.js` et `.css`), un **serveur statique suffit**, puisque les pages sont générées sur le 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**<sup>(1)</sup>, **`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](https://enable-cors.org).

(1) Sans cette configuration, le serveur est incapable d'interpréter une URL comme <http://www.mysite.com/heroes/42>, car pour le serveur, le chemin `/heroes/42` n'existe pas. Il faut donc configurer le serveur pour renvoyer `index.html`, qui va exécuter l'application Angular, qui va activer le chemin `/heroes/42`. La nécessité de cette configuration n'est apparente que lorsqu'on essaie d'accéder **directement** à une page "profonde" du site (via un bookmark, un lien partagé par e-mail, ou en rafraîchissant la page). En effet, si l'utilisateur accède aux pages profondes uniquement en navigant depuis `index.html`, c'est Angular qui a la main depuis le début et le serveur n'entend jamais parler du chemin `/heroes/42`.

La doc contient des exemples de configuration pour plusieurs serveurs (Apache, Nginx, IIS, Github Pages...) : <https://angular.io/docs/ts/latest/guide/deployment.html#production-servers>

# 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 bibliothèques/helpers pour gérer l'authentification, exposer une BDD via une API REST, etc. Tous les principaux langages/frameworks back possèdent de telles bibliothèques.
- 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.

# Conclusion

# 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).





# Vos impressions ?

- **Vos impressions en quelques mots :**
  - Rythme ?
  - Contenu ?
  - Vous sentez-vous capable d'utiliser Angular dans vos projets ?
- Pour toute **question** :
  - <https://twitter.com/angularfrance>
  - [hello@angularfrance.com](mailto:hello@angularfrance.com)

# Merci !