





Angular 9/11

David THIBAU – 2021

david.thibau@gmail.com



Agenda

- **Introduction**
 - Présentation, Outils, Démarrage
- **Langages**
 - Rappels Javascript, ES6 et TypeScript
- **Introduction aux composants**
 - Syntaxe, Arbre des composants, syntaxe des templates, les directives
- **Composants suite**
 - Input/Output, Cycle de vie, Projection
- **Services et injection**
 - Définition, Injection de dépendances, HTTP
- **Modules**
- **Routes**
 - Le routeur, Syntaxe des routes, routes d'un feature module
- **Formulaires**
- **Programmation réactive**
- **Fonctionnalités avancées**
- **Les tests**
 - Tests unitaires
 - Tests e2e



Introduction

Présentations, Outils, Démarrage



Introduction

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**.



Principaux atouts du framework Angular

Typescript : Permet de bénéficier d'un langage de programmation typée permettant une meilleur maintenabilité

Approche Composant/Module : Permet la réutilisation et l'utilisation de bibliothèques fournies par des éditeurs tierces. (Material, ng-bootstrap, PrimeNG). Met en œuvre une architecture MV*

Angular CLI : Un outil permettant de démarrer un projet, de créer des assets (composant, service, modèle,...), de builder en mode dév ou prod, d'utiliser LiveReload, de déployer

Pattern IoC et Injection de dépendances : Faciliter pour mocker des services, // des développements front-end et back-end

Outils de tests : Tests unitaires et end2end

Documentations : Bcp de ressources sur le web



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.

Releases fréquentes. Les releases majeures sont supportées 18 mois (6 mois active puis LTS). Les upgrades de releases doivent se faire release majeure par release majeure.

Navigateurs récents. Les utilisateurs doivent mettre à jour régulièrement leur navigateur
<https://angular.io/guide/browser-support>



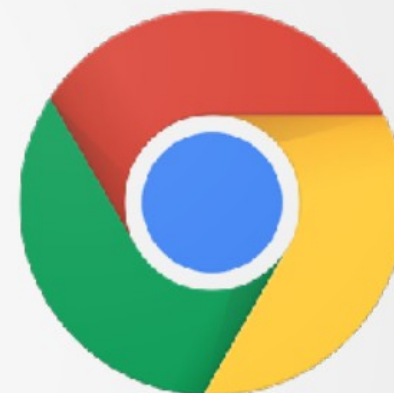
Outils de développement



node & npm
(installer les librairies)



IDE
(coder avec TypeScript et Angular)



Navigateur
(débugueur)



Angular CLI

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 :

- *PROJECT_ROOT/.angular-cli.json*



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
```

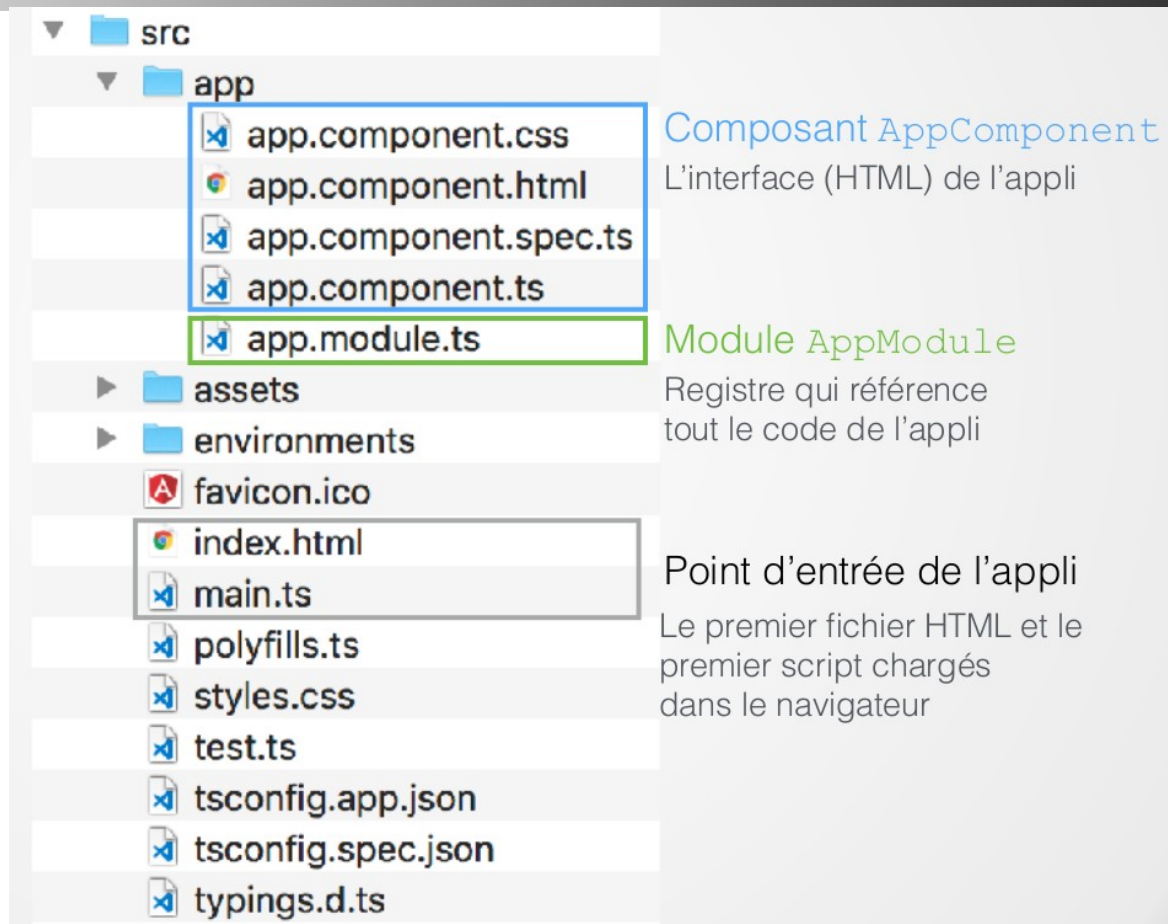
Fichiers importants

`ng new PROJECT_NAME`

génère un squelette d'appli
+ télécharge les dépendances



Application



Composant AppComponent
L'interface (HTML) de l'appli

Module AppModule
Registre qui référence tout le code de l'appli

Point d'entrée de l'appli
Le premier fichier HTML et le premier script chargés dans le navigateur



Workflow de dévpt.

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.



Langages

Rappels JavaScript
ES6 et Typescript



Rappels Javascript

Variables, Objets, Tableaux, Fonctions



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>';
```

var ne s'utilise plus



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;
}
```



Objets

Objet JS = 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
```

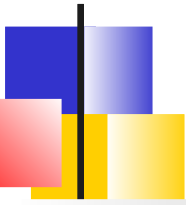
```
contact.name = "Pierre";
```

```
const name = contact.name;
```

```
// Syntaxe crochet
```

```
contact["name"] = "Pierre";
```

```
const name = contact["name"];
```



Passage par Valeur / 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 :

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



Importance de *Object.assign*

```
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.

Très fréquent dans Angular.



Les 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);  
});
```

// Ou

```
contacts.forEach( contact => 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);
```



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'];
```

```
//SYNTAXE « lambda » avec return implicite
```

```
const formalContacts = contacts.map(contact => 'Monsieur ' +  
contact);
```



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);
```



Erreur fréquente !

Confondre le fait de référencer une fonction et le fait d'invoquer une fonction :

```
function hey() {  
    alert('coucou');  
}  
  
// CORRECT (référence)  
setTimeout(hey, 500);  
  
// INCORRECT (invocation)  
setTimeout(hey(), 500);
```




Erreur fréquente !

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);  
});
```



Points à creuser ...

Les call-backs peuvent être remplacés par des Promesses

```
getUser(login)
  .then(user => getRights(user))
  .then(rights => updateMenu(rights)) ;
```

Javascript permet également de définir des classes (on en parle + tard),
ES2015 propose les classes *Set* et *Map*

```
const cedric = { id: 1, name: 'Cedric' };
const users = new Map();
users.set(cedric.id, cedric); // adds a user
```

Javascript supporte l'interpolation de String (backquote)

```
const fullname = 'Miss ' + firstname + ' ' + lastname;
// Equivalent à
const fullname = `Miss ${firstname} ${lastname}`;
```

Les modules permettent de définir des espaces de noms (on en reparle + tard)

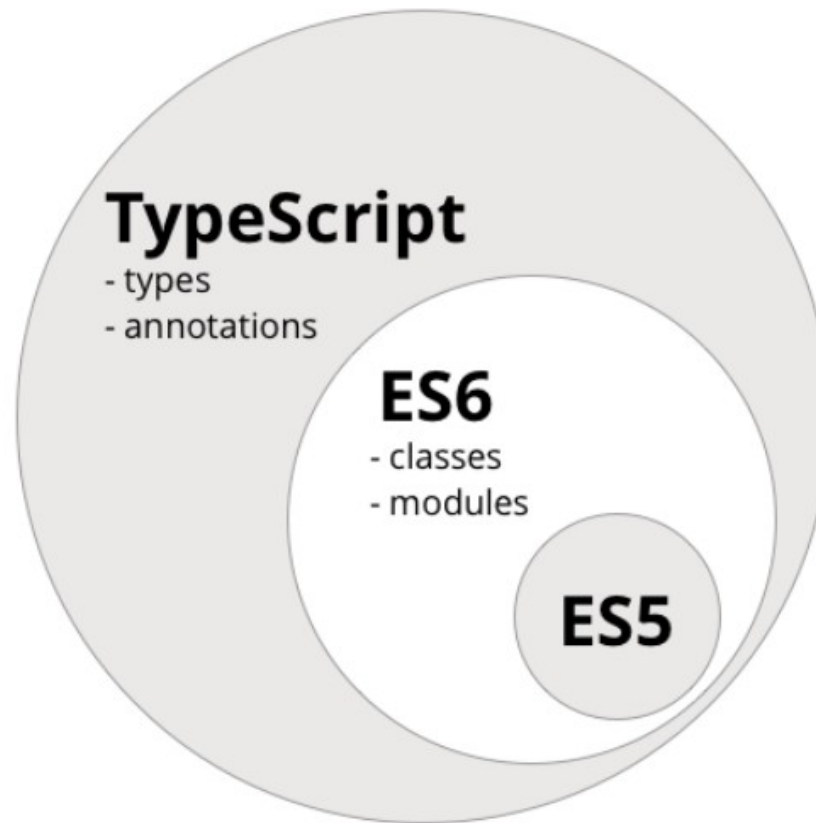


TypeScript

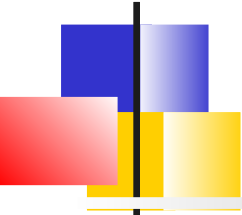
Types facultatifs, Compilation obligatoire



Apports de Typescript



ES5, ES6, and TypeScript



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/>



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 paramètres 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.



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) => 'Monsieur ' + prenom);
```

// Ajout des syntaxes TypeScript

```
const prenom: string[] = ['Pierre', 'Paul', 'Jo'];  
prenom.map((prenom: string) => {  
    return 'Monsieur ' + prenom;  
});
```

Transpilation

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



Types 1/2

Les types sont au cœur de TypeScript (ils ont donné son nom au langage), mais ils sont facultatifs.

Ils permettent de spécifier le type d'une variable en l'annotant avec la syntaxe `variable: type`.

Exemples :

```
let age: number;
let is_customer: boolean;
// Typage + Affectation de valeur
let name: string = 'Vince';
// Tableau - Syntaxe 1
let list: number[] = [1, 2, 3];
// Tableau - Syntaxe 2
let list: Array<number> = [1, 2, 3];
```



Types 2/2

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 :

```
function isAdmin(username: string): boolean {  
    return true;  
}
```

// Si la fonction ne renvoie rien :

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



Inférence de type

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 :

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

En conséquence, le code suivant passe en JavaScript mais pas en TypeScript :

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



Valeurs énumérées

TypeScript propose des valeurs énumérées :

```
enum RaceStatus {  
  Ready,  
  Started,  
  Done  
}  
const race = new Race();  
race.status = RaceStatus.Ready;
```

En réalité, *enum* est une valeur numérique commençant par 0 par défaut



Interface (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.
- Une interface définit des propriétés ou fonctions et leurs types

Exemple - Utilisation d'une interface *Todo* pour décrire un objet :

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



Propriétés facultatives (ts)

On peut marquer certaines propriétés de l'interface comme facultatives en plaçant un **?** juste après leur nom¹ :

```
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.

1. On peut utiliser ? pour déclarer un argument optionnel d'une fonction



Interfaces dans Angular

Utiles lorsque l'on souhaite imposer une certaine forme aux modèles de données de l'application.

Par exemple : Si l'application manipule des livres, on peut créer l'interface *Book* suivante :

```
interface Book {  
  id: number;  
  title: string;  
  summary: string;  
  authors: Author[]; // Cette propriété référence une autre interface  
  published: { // Objet dans un objet  
    day: number;  
    month: number;  
    year: number;  
  }  
}
```



Définition de contrat

Comme dans d'autres langages, les interfaces servent à définir un contrat à implémenter.

```
interface ClockInterface {  
    currentTime: Date;  
    setTime(d: Date);  
}
```

```
class Clock implements ClockInterface {  
    currentTime: Date;  
    setTime(d: Date) {  
        this.currentTime = d;  
    }  
    constructor(h: number, m: number) { }  
}
```




Classes - Syntaxe (ES6/ts)

Déclaration :

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

Instanciación :

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

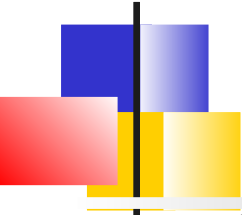


Classes - Visibilité (ts)

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 {  
  name: string; // propriété  
  constructor(theName: string) {  
    this.name = theName; // on affecte le paramètre à la propriété  
  }  
}
```

Syntaxe raccourcie :

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



Erreur fréquente

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 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 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 {  
  @Input() name: string;  
  ..  
}
```

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



Modules ES6

Les différents fichiers de code ES6/TypeScript sont organisés en modules.

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** :

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




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';
```

L' IDE et le compilateur TS résolvent les imports à partir des fichiers sources en TypeScript.

L'application finale résout les imports à partir des fichiers compilés en JavaScript.



Modules dans Angular

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

On trouve 2 types d'import dans le code :

- Imports non relatifs - Symboles fournis par Angular :
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';
```



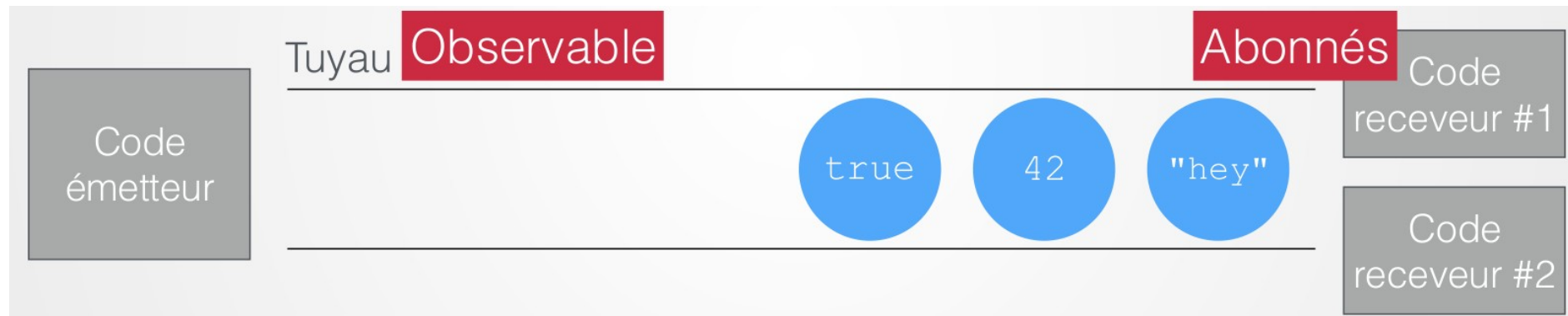
Observables & RxJS

Programmation réactive,
Librairie RxJS, Opérateurs RxJS

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.



Flux Temps réel, Analogie avec le binding

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.



Usage de la programmation réactive

Ces bindings temps réel sont donc particulièrement adaptés pour implémenter des problématiques typiques des applis SPA :

- Attendre le résultat d'une opération asynchrone (requête HTTP, Attendre une action de l'utilisateur (un clic, la saisie d'un caractère, etc.).
- 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 chat).



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.





Pattern et *ReactiveX*

La programmation réactive se base sur le pattern ***Observable*** qui est une combinaison des patterns *Observer* et *Iterator*¹

Elle utilise en plus la programmation fonctionnelle permettant de définir facilement des opérateurs

Elle est formalisée par l'API ***ReactiveX*** (<http://reactivex.io>) et de nombreuses implémentations existent pour différent langages (*RxJava*, *Rx.NET*, ...)

L'implémentation pour JavaScript s'appelle ***RxJS***.

1. On peut boucler (*Iterator*) mais les éléments arrivent au fur et à mesure (*Observer*)



API

Un **observable** représente un flux d'évènements ou source de données.

On peut créer un observable à partir d'un tableau, d'une ressource REST, d'évènements utilisateur, d'une requête NoSQL

Les **subscriptions** ou **abonnements** sont les objets qui déclenchent le flux. Sans abonné pas d'évènements !

Les abonnés peuvent gérer le débit des évènements (back-pressure)

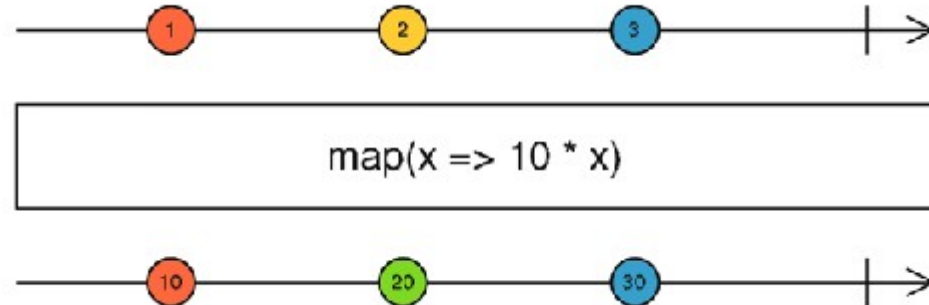
Les **opérateurs** offrent un moyen de manipuler les valeurs d'une source, en renvoyant une observable des valeurs transformées. Ils peuvent être combinés.



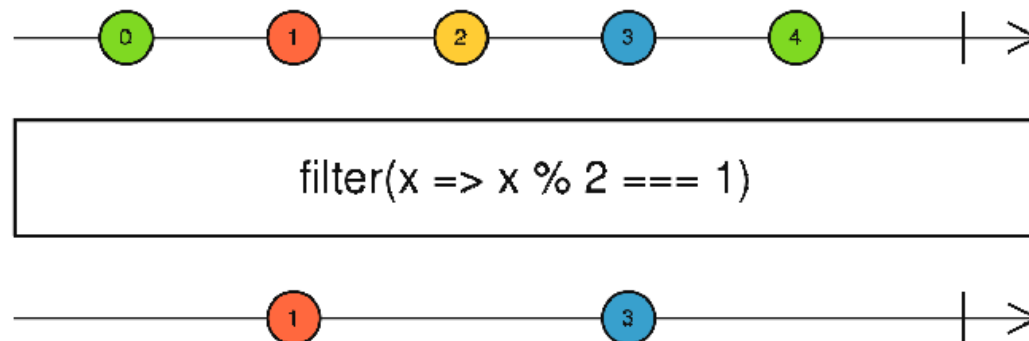
Marble diagrams

Pour expliquer les opérateurs, la doc utilise des ***marble diagrams***

Exemple *map* :



Exemple *filter*





Exemple RxJS

```
// L'appel de cette méthode est non-bloquant
// Opérateur map utilisé
find(id: number): Observable<Tweet> {
    return this.http.get<Tweet>(`${this.resourceUrl}/${id}`,
        { observe: 'response' })
        .map((res: Tweet) => this.convertResponse(res));
}

-----
load(id) {
    this.tweetService.find(id)
        .subscribe(
            // Fonction exécutée lors de la réception de l'évt
            (tweetResponse: HttpResponse<Tweet>) => {
                this.tweet = tweetResponse.body;
            }
        );
}
```



Observable - Création

On peut créer des observables via RxJS 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);
```

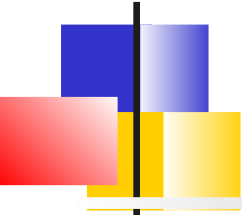
// fromEvent() permet de convertir un événement DOM en observable

```
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>



Opérateur - Chaînage

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

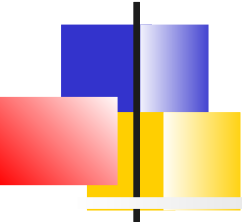
```
Observable.range(1, 5)
  .map(x => x * 2)
  .filter(x => x > 5)
  .subscribe(
    x => console.log(x),
    error => console.log(error),
    () => console.log('fini')
  );
// Affichera : 6, 8, 10, fini
```



Chaînage d'opérateur avec RxJS

```
// observable, pipe chaine les opérateurs
```

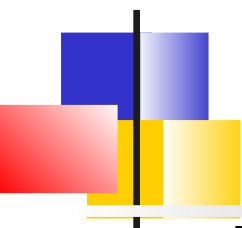
```
inputValue
  .pipe(
    // Attente de 200ms
    debounceTime(200),
    // Si la valeur est identique, ignore
    distinctUntilChanged(),
    // si une nouvelle valeur arrive pendant que le traitement est actif
    // annule la demande précédente et «passe» à un nouvel observable
    switchMap(searchTerm => typeaheadApi.search(term))
  )
// Crée l'abonnement et déclenche le flux
  .subscribe(results => {
    // Mise à jour du DOM
  });
```



Création d'Observable Angular

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 *valueChanges* d'un *formControl*).
- Changements de valeur d'un paramètre d'URL : Les valeurs successives du paramètre sont émises via un observable (propriété *ActivatedRoute.paramMap*).
- @Output() : Les événements émis avec *EventEmitter* utilisent les observables en coulisse.



Observable - Abonnement

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

On peut préciser 3 méthodes de call-back :

- *next* : Méthode appelée sur chaque évènement
- *error* (optionnelle) : Méthode sur un évènement erreur
- *complete* (optionnelle): Méthode sur l'évènement terminal

```
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');
```




Introduction aux composants

Définition et arbre de composants
Syntaxe des templates
Directives



Définition

Dans une application Angular, un composant représente **une partie de l'interface**.

À ce titre, les composants sont responsables **de l'affichage et de l'interactivité** d'une application Angular.

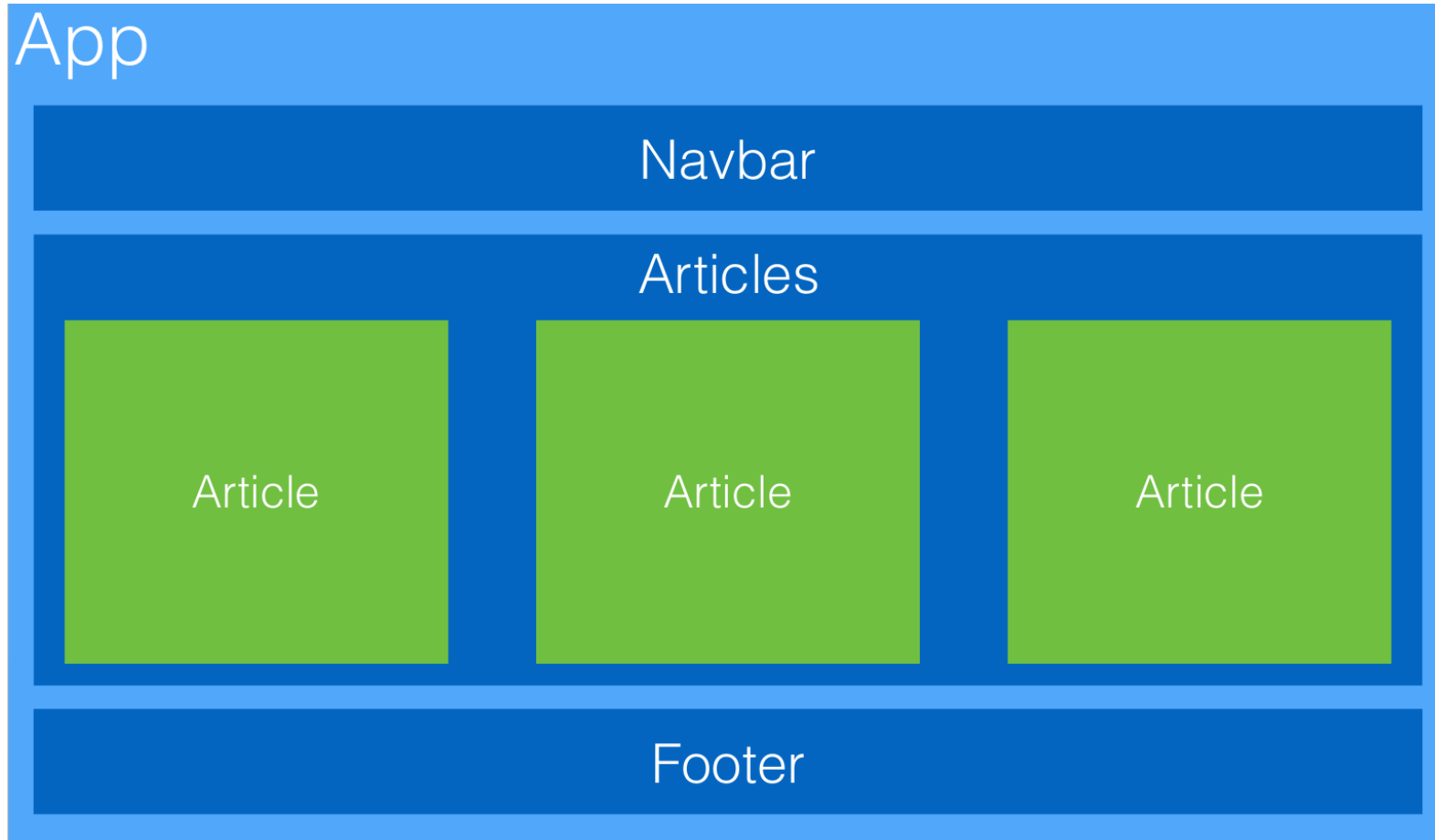
C'est au développeur de découper son interface en composants.

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.



Arbre de composants





Composants Syntaxe

Composant = Classe + Template

Classe

```
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)

Template

```
// 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).



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 et 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; }'] // tableau
})
export class MeteoComponent {}
```

Styles locaux. Les styles définis ainsi n'affectent que le template de ce composant, même s'ils sont très généraux.

Styles globaux. Pour définir des styles qui affectent tous composants, placez-les dans le fichier **styles.css** du projet

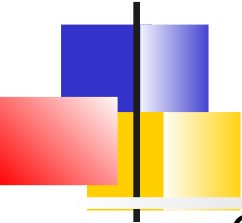


:host

Au rendu, le composant angular génère une balise correspondant au sélecteur.

Pour définir le style des classes rattachés à cette balise, il faut utiliser ***:host***

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



Composant - Utilisation

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

IDE

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

Navigateur

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

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

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




Composant avec le CLI

ng generate component toto

- Produit les fichiers *toto.component.css*, *toto.component.html*, *toto.component.ts*, *toto.component.spec.ts*
- Déclare le composant dans le module applicatif

Options :

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



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).

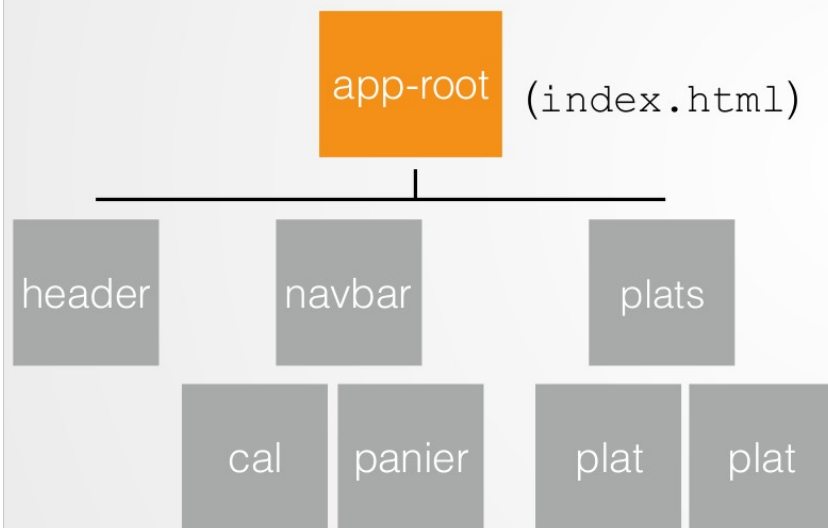
Exemple : pagination, barre de navigation, image redimensionnable...

- 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 (navigation dans l'application).

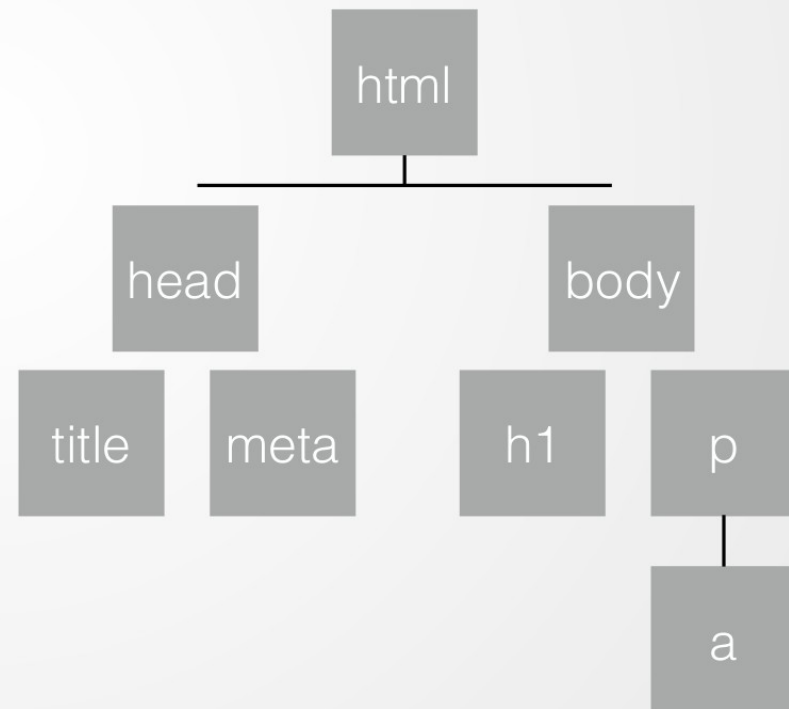
Dans ce cas, l'utilisation d'un composant est obligatoire.

L'arbre des composants

Arbre des composants



DOM





Relations Parent-Enfant

ANGULAR

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

```
@Component({  
  selector: 'plats',  
  template: `  
    <plat></plat>  
    <plat></plat>`  
})  
class PlatsComponent {  
  // PlatComponent est un  
  // enfant de PlatsComponent  
}
```

=> API : **viewChild, viewChildren**
Permet de récupérer des composants enfants Angular via des sélecteurs (Type, Variable, ...)

DOM

```
<div class="desc">
```

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

```
</div>
```

```
<!--
```

```
<p> est un enfant de <div>
```

```
-->
```

=> API : **contentChild, contentChildren**

Permet de récupérer des éléments Angular ou du contenu DOM Angular via des sélecteurs (Type, Variable, ...)



Syntaxe de templates

Interpolation, Binding de propriété, Directives structurelles,
Binding d'événement, Variables locales, Pipes, Résumé



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>`

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>
```



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* :

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

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

```
<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.



Directives d'attribut Angular

ngClass permet d'ajouter/d'enlever dynamiquement des classes CSS :

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

ngStyle permet de définir des styles CSS :

```
<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 :

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

```
<!-- <input name="username" [ngModel]="user.username"
      (ngModelChange)="user.username = $event"> -->
```

Attention, nécessite d'avoir importé le module FormsModule d'Angular

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



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>
```



Exemples

Classe

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

  onChanged(ev) {
    ev.preventDefault();
    ev.stopPropagation();
  }
  onSp() {}
}
```

Template

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



Variables locales

Les variables locales sont des variables déclarées dynamiquement dans le template avec la notation **#**. Elles peuvent ensuite être accédées n'importe où dans le template (interpolation, binding de propriété, ...)

Exemple :

```
<input type="text" #name>
{{ name.value }}
```

```
<button (click)="name.focus()">Focus the input</button>
```



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”).

Angular propose de nombreux pipes (date, json, ...) et on peut implémenter ses propres pipes.

Syntaxe :

```
{{ expression | pipe }}
```

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




Exemples

Classe

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

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

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

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

Template

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

€10.60

16/07/1986

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



Règle d'accessibilité

Un template peut accéder à l'ensemble des propriétés et méthodes publiques de la classe associée, ainsi qu'aux variables locales.



Exemples

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 -->  
<del>{{ user.name }}</del>
```



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".



Composants suite

Input/Output
Cycle de vie
Projection



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) :

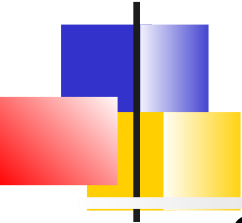
```
<meteo ville="Paris"></meteo>
```

– Dans cet exemple, le composant `<meteo>` possède un input `ville`.

Les données transmises par le template permettent de personnaliser l’affichage du composant, et de le rendre ré-utilisable.

La propriété du composant est décoré avec : **@Input()** :

```
@Component({  
  Selector: 'app-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 **.

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 **.

[...] → EXPRESSION



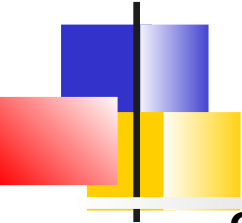
Output - Définition

Un **output** permet 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

Pour créer un output, il faut décorer une propriété de la classe enfant avec le décorateur **@Output()** (voir slide suivant).



Output - Syntax

Composant enfant :

```
@Component({
  selector: 'meteo',
  template: `<div>
    <button (click)="declencher()">Déclencher</button> </div>`
})
export class MeteoComponent {
  temperature = 40;
  @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) { ... }
}
```



Exemple (1)

Le composant 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;
  }
}
```

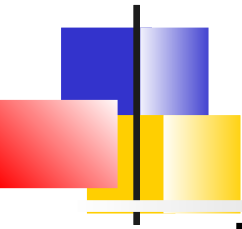


Exemple (2)

Le composant 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);
  }
}
```



Input/Output - Bénéfices

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

Permet de rendre un composant réutilisable :

- 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...)



Cycles de vie



Méthodes “Cycle de vie”

Méthodes spéciales permettant d'exécuter des actions à des moments-clé de la vie d'une directive / d'un composant .

Ces méthodes sont reconnues et exécutées automatiquement par Angular si elles sont implémentées.

Si l'on utilise TypeScript, il est recommandé d'explicitement implémenter l'interface

Exemple :

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().



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}`);
  }
}
```



Hooks complets

ngOnChanges(): Lorsqu'Angular positionne des propriétés d'entrée. La méthode reçoit un objet *SimpleChanges* contenant l'ancienne valeur et la courante.

ngOnInit() : Initialise la directive ou composant après qu'Angular ait positionné les valeurs d'entrée. Appelée une seule fois.

ngDoCheck() : Détecte et réagit à des changements qu'Angular ne peut pas détecter. Appelée fréquemment

ngAfterContentInit() : Après qu'Angular ait projeté du contenu dans la vue du composant ou la vue contenant la directive. Appelée une seule fois après le premier *ngDoCheck()*.

ngAfterContentChecked() : Après qu'Angular ait vérifié le contenu projeté. Appelée après *ngAfterContentInit()* et tous les autres *ngDoCheck*.

ngAfterViewInit() : Après qu'Angular ait initialisé les vues du composants (et les vues enfants). Appelée une fois après le premier *ngAfterContentChecked*.

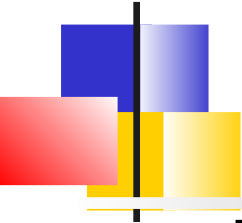
ngAfterViewChecked() : Après qu'Angular ait vérifié les vues du composant (incluant les vues enfants). Appelée après *ngAfterViewInit* et les *ngAfterContentChecked*.

ngOnDestroy() : Avant qu'Angular détruise le composant ou directive.



Services et injection

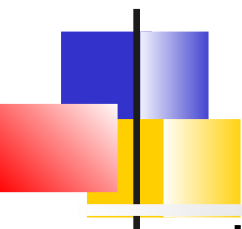
Définition de services
Injection de dépendances
HTTP



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.
MAIS 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.

=> 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 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');
  }
}
```



Créer son propre service

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

```
import {Injectable} from '@angular/core';
@Injectable({
  providedIn: 'root' // Le service est injectable partout
})
class RacesService {
  list() {
    return [{name: 'London' }];
  }
}
```

Un service est un singleton.

=> Idéal pour partager un état / des données entre plusieurs composants.



Service avec le CLI

ng generate service toto

Génère les 2 fichiers *toto.service.ts* et *toto.service.spec.ts*

Si désiré on peut le rajouter dans les providers du module applicatif

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
```




Injection de dépendances (DI)



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 d'instancier le service, c'est le framework qui va localiser et instancier le service, le fournir au composant qui en a besoin et le conserver dans un registre.

Cette façon de procéder se nomme **l'inversion de contrôle (IoC)**.



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 via son constructeur.

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 - 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



DI - DÉCLARER une dépendance (1/2)

Pour rendre une dépendance injectable, il faut d'abord la déclarer. Cette déclaration peut se faire à différents 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	Angular 6+ <pre>@Injectable({ providedIn: 'root' }) export class ApiService { Avant : @NgModule({ imports: [CommonModule], providers: [AuthService] }) export class MyModule { }</pre>	<pre>@Component({ selector: 'my-comp', template: '<p>Hello</p>', providers: [AuthService] }) export class MyComp { }</pre>



DI - INJECTER une dépendance (2/2)

Injecter une dépendance permet de récupérer 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(private param1: MyService, private param2: OtherService)



DI - Quelles “dépendances” ?

Que peut-on utiliser comme “dépendance” ?

- Un **service natif** Angular comme http, router.
- Une **valeur** (par ex, une constante, des paramètres de configuration...).
- Un service applicatif
 - En dur
 - Ou instancié dynamiquement via une classe factory (permet d’avoir des injections différentes selon une configuration).

La syntaxe de la déclaration est différente dans chaque cas.



Syntaxes de déclaration

```
// Classe Service
providers: [
  LoggerService, UserContextService, UserService
]

// Valeur (TITLE est un InjectionToken)
providers: [
  { provide: TITLE, useValue: 'Hero of the Month' }
]

// Classe implémentant une classe abstraite
providers: [
  - { provide: LoggerService, useClass: DateLoggerService }
]

// Via une factory
providers: [
  {provide: RaceService,
useFactory: () => IS_PROD ? new RaceService() : new FakeRaceService()
}]

// Via une factory avec paramètres
Providers: [
  ApiService,
  {provide: RaceService,
useFactory: (apiService: ApiService) => IS_PROD ? new RaceService(apiService) : new FakeRaceService(),
  deps : [ApiService]
}]
```



Apport d'une couche service

Gère toutes les interactions avec le backend

Ajoute une couche d'abstraction : les parties de notre appli qui veulent manipuler le modèle métier n'ont pas besoin de connaître les détails de l'implémentation

Permet de retravailler les données. Souvent, les données n'ont pas le même format côté back et côté front. Il est donc utile ou nécessaire de les reformater avant usage



Utilisation de librairies tierces

Exemple de *ng-bootstrap*



Introduction

Dans la pratique, il est utile de baser sa charte graphique sur une bibliothèque des composants d'un éditeur:

- **Material** : <https://material.angular.io/>
Bibliothèques de composants, sans technologie de layout
- **ng-bootstrap** : <https://ng-bootstrap.github.io/>
Bibliothèques de composants minimaliste basée sur bootstrap.css
- **PrimeNG** : <https://primefaces.org/primeng/>

La mise en place s'effectue via *npm* puis l'importation des composants dans le module Angular



Routes

Le routeur
Syntaxes des routes



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 d'URL donne l'impression que différentes URLs sont requêtées auprès du serveur, en réalité les changements d'URL sont reçus par `Angular/index.html`, et c'est Angular qui matche le chemin et la route correspondante :

`http://exemple.com/`

`http://exemple.com/accueil`

`http://exemple.com/contact`

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

...

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



Routeur - Tâche préparatoire

Définir le base *href* 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.

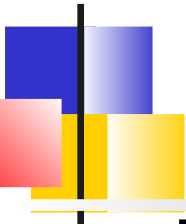


Déclarer les routes (1/3)

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

```
@NgModule({  
  Imports: [ RouterModule,  
    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



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.

Les ******** représentent le joker. Il sera matché si l'URL demandée ne matche aucun autre chemin déclaré. Utile pour implémenter une pseudo page 404. Doit apparaître dans la dernière route déclarée.

La propriété ***data*** (pas utilisée dans l'exemple) permet d'associer des données arbitraires à un route (*title*, *breadcrumb*... *READ ONLY*).



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

Le routeur associe des URLs à des composants et les composants s'affichent à l'endroit où est placée la directive **<router-outlet>** :

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

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



<http://example.com/quizzes>

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



Routeur - Naviguer (3/3)

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

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

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

```
// <button (click)="goHome()">Accueil</button>
export class ContactCmp {
  constructor(private router: Router) {}
  goHome() {
    this.router.navigate(['/home']); // Link Params Array
  }
}
```



Erreur fréquente : Oublier de déclarer les composants de route

Le fait d'afficher un composant via le routeur ne vous dispense pas de le déclarer dans le module (dans la propriété *@NgModule.declarations*).

Dans cet exemple, les composants affichés par le routeur sont AUSSI déclarés dans le module

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



Routes

Syntaxes diverses

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



Routeur - Ordre des routes

Les routes sont matchées dans l'ordre où elles sont déclarées. La première route qui matche gagne.

Erreur fréquente #1 :

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

Erreur fréquente #2 :

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



Liens absolus

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">G0</a> /heroes/38
<a [routerLink]="38/edit">G0</a> /heroes/38/edit
<a [routerLink]=".">G0</a> /heroes
<a [routerLink]="..">G0</a> / (racine du site)
```

Lien relatif dans le code :

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




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]);
```



Récupérer les paramètres 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
...
```



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).



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>
```



Formulaire



Introduction

Problématiques des formulaires :

- 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



Tâche préalable

Pour pouvoir utiliser les syntaxes de formulaire dans votre module, il faut importer le module ***ReactiveFormsModule***.

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.



Formulaires réactifs : Introduction

La syntaxe des formulaires réactifs 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

Dans le template, on trouve les formulaire HTML classique (`<form>`, `<input>`...) avec des syntaxes Angular 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).



FormControl et FormGroup

Du côté TypeScript, le composant doit créer :

- Des objets de type ***FormControl*** qui représente des éléments de saisie
- Les *FormControl* sont regroupés dans des ***FormGroup*** qui représente un sous-ensemble d'un formulaire qui a des règles de validation communes



Attributs d'un *FormControl*

- ***valid*** : si le champ est valide, au regard des contraintes et des validations qui lui sont appliquées.
- ***invalid*** : l'opposé de *valid*
- ***errors*** : un objet contenant les erreurs du champ.
- ***dirty*** : *false* jusqu'à ce que l'utilisateur modifie la valeur du champ.
- ***pristine*** : l'opposé de *dirty*.
- ***touched*** : *false* jusqu'à ce que l'utilisateur soit entré dans le champ.
- ***untouched*** : l'opposé de *touched*.
- ***value*** : la valeur du champ.
- ***valueChanges*** : un Observable qui émet à chaque changement sur le champ.



Attributs d'un *FormGroup*

- **valid** : si tous les champs sont valides, alors le groupe est valide.
- **invalid** : si l'un des champs est invalide, alors le groupe est invalide.
- **errors** : un objet contenant les erreurs du groupe, ou null si le groupe est entièrement valide.
Chaque erreur en constitue la clé, et la valeur associée est un tableau contenant chaque contrôle affecté par cette erreur.
- **dirty** : false jusqu'à ce qu'un des contrôles devienne "dirty".
- **pristine** : l'opposé de dirty.
- **touched** : false jusqu'à ce qu'un des contrôles devienne "touched".
- **untouched** : l'opposé de touched.
- **value** : la valeur du groupe. Pour être plus précis, c'est un objet dont les clé/valeurs sont les contrôles et leur valeur respective.
- **valueChanges** : un Observable qui émet à chaque changement sur un contrôle du groupe.



FormBuilder

Angular fournit **FormBuilder** facilitant la création des groupes de formulaires. Il permet facilement de

- Créer des groupes
- Y Déclarer les contrôles, leurs valeurs par défaut et leurs validateurs

```
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: [],  
    });  
  }  
}
```



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).



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

- Les validateurs asynchrones ne sont pas appelés que lorsque les validateurs synchrones sont passés.
- Exemple Asynchrone : Appel serveur pour validation

```
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};
}
```



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
```



Association au gabarit

Le formulaire HTML du template doit être relié à l'objet *FormGroup* du composant via la directive ***formGroup***.

Chaque champ de saisie est relié à un *FormControl*, grâce à la directive ***formControlName***

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>  
<button type="submit">Submit</button>  
</form>
```



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 :

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



Récupérer les données du formulaire

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;  
}
```



HTTP



Introduction

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.
- 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.



Remarques

Le service *HttpClient* réalise des requêtes avec *XMLHttpRequest*.

HttpClient propose des méthodes correspondant aux 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');
```

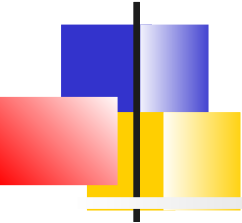
```
// OUI
http.get('/api').subscribe(resp => {
  // Ici, utiliser resp
});
```




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 { }
```



Requête GET - *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);
  }
}
```



Requête GET - *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');
  }
}
```

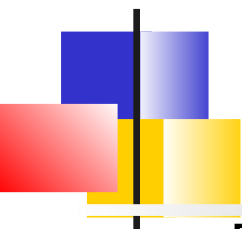


Le pipe async

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.



GET : Syntaxes diverses (1)

Récupérer des données JSON :

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

Associer un type aux données récupérées

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



GET : Syntaxes diverses (2)

Récupérer la réponse entière

```
http.get<Response>('/api/quizzes')  
.subscribe(resp => {  
... // resp est de type Response from '@angular/http'  
    // Permet d'accéder au statut, entêtes, etc..  
});
```

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'); }  
);
```



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 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* :

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




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'; // Reactive API
http.get(`${baseUrl}/api/races`)
  .map((races: Array<any>) => races.map(race => race.name))
  .subscribe(names => {
    console.log(names);
  });
```

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

(dans l'exemple *map* appliquant une fonction sur chaque événement et retournant le résultat).



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.



Modifier toutes les requêtes HTTP

Les intercepteurs HTTP permettent de modifier toutes les requêtes et réponses de l'application.

Ils sont souvent utilisés pour ajouter un token d'authentification .

La classe doit implémenter ***HttpInterceptor*** et elle doit être ajoutée au tableau ***HTTP_INTERCEPTORS*** par injection :

```
providers: [  
  { provide: HTTP_INTERCEPTORS, useClass:  
    GithubAPIInterceptor, multi: true }  
]
```



Tests

Tests unitaires et end-to-end



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.



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).



Mise en place

La mise en place d'un environnement de test est fastidieuse.

La 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



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.



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 tests runners (même si la détection des tests est paramétrable).
- Vous penserez à mettre à jour le test lorsque vous changerez le code testé.



Exécution

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

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*.



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




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".



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)`



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');
```




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*** :

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



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

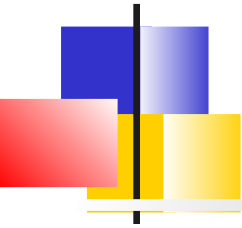


Aller plus loin

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



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.

Les tests se lancent par ***ng e2e***



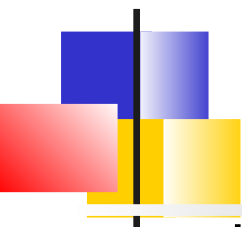
Example

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



Outillage, i18n et déploiement

Outils divers, Internationalisation,
Déploiement



TypeScript

Langage créé par Microsoft en 2012, open-source, qui transpile vers JavaScript.

Sur-ensemble d'ES6 (aka ES2015).

=> Tout JavaScript est donc du TypeScript valide.

Principales caractéristiques : types, interfaces, classes, décorateurs, modules, fonctions fléchées, templates chaîne.

Supporté par de nombreuses bibliothèques JavaScript tiers

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>



Angular Augury

Extension Chrome Dev Tools pour déboguer 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 déboguer avec Chrome Dev Tools. Les source maps permettent de déboguer le code TypeScript alors que le navigateur exécute du JavaScript.



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 pas 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.



Internationalisation

Traduire et localiser son application



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 :

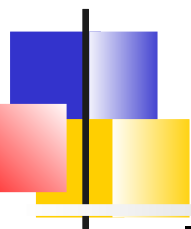
- la locale à utiliser (fr, en-US...).
- le fichier de traduction
- le format de ce fichier

VERSION AOT:

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

VERSION JIT : 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);  
});
```



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 i18n>The quiz {{ quiz.title }} has been  
  saved.</p>
```



Déploiement

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



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).

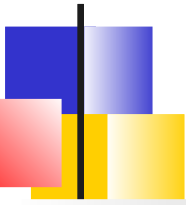
Inlining : Placer le HTML et le CSS des composants dans le code du composant (vs. dans des fichiers externes).

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
```



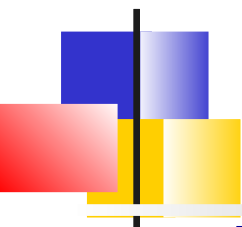

Compilation AOT

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`



Webpack

Webpack 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".



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 :

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



Distribution

Lorsque l'on construit l'application pour la production, Angular CLI place les fichiers générés dans le répertoire ***dist*** du projet.

Il suffit de déployer le contenu de ce répertoire sur un serveur web



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, *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*.



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*).



Ejection Angular-CLI

Si à un moment donné d'un projet démarré avec Angular-CLI, on veuille accéder à la configuration webpack ; il est possible d' « éjecter » Angular CLI :

ng eject

Cela génère le fichier de configuration webpack (*webpack.config.js*) et après une commande *npm install*, il est possible de :

- builder, lancer le serveur, exécuter les tests unitaires et e2e avec *npm* .



Merci!!!

❖ MERCI DE VOTRE ATTENTION



Ressources utiles

Documentation :

- Site officiel Angular (doc, API...) : <https://angular.io/>
NE PAS CONFONDRE AVEC AngularJS : <https://angularjs.org/>

- Livres

Ninja Squad “Deviens un Ninja avec Angular” (FR)
The ng-book : The complete book on Angular5/6

Communauté :

- News (EN) :
<https://blog.angular.io/>
- Questions (EN) :
<http://stackoverflow.com/questions/tagged/angular>



Annexes



Directives attribut

Natives et Custom



Directives attribut natives

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

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

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

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

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

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



Exemples

Classe

```
export class MeteoComponent {
  currentClasses: {};
  currentStyles: {};
  user = {name: 'Bob'};
  // true ajout de la classe
  // false suppression
  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-l
  ...
</div>
<input [(ngModel)]="user.name">
```



Directives Angular

On distingue 3 types de directives dans Angular :

- **Composants** - Directives associés à 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*.



Syntaxe

Une directive est une classe décorée avec **@Directive** :

```
import {Directive} from '@angular/core';
@Directive({
  selector: '[myHighlight]'
})
export class HighlightDirective {
}
```

Différence 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.



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 { }
```



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.



Example

```
@Directive({
  selector: '[myHighlight]'
})
export class HighlightDirective {
  @Input() highlightColor: string;
  constructor(private el: ElementRef) { }
  @HostListener('mouseenter') onMouseEnter() {
    this.highlight(this.highlightColor);
  }
  @HostListener('mouseleave') onMouseLeave() {
    this.highlight(null);
  }
  private highlight(color: string) {
    this.el.nativeElement.style.backgroundColor = color;
  }
}
```

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



Directives vs composants

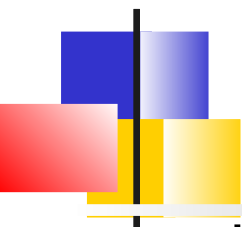
	Composant	Directive
Définition	Affiche/génère un élément d'interface	Modifie un élément existant
Décorateur	@Component	@Directive
Sélecteur	selector: 'meteo'	selector: '[meteo]'
Template ?	OUI (+ styles CSS)	NON
Utilisation	<meteo></meteo>	<p meteo></p>

Les directives sont des composants sans template.

TP : Directive attribut



Modules Angular



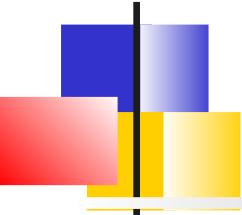
NgModule - Pourquoi ?

Une application Angular contient différentes briques de code :

- Composants
- Directives
- Pipes
- Services

Toutes ces briques doivent être rangées dans des modules Angular, ou **NgModules**.

Un module est comme un registre qui référence tout le code d'une application Angular. Une brique de code ne peut pas faire partie de l'application sans être référencée dans un module.



NgModule - Bénéfices

Organisation du code. Permet de regrouper les fonctionnalités liées de manière cohérente, notamment tous les composants, directives, pipes, et services liés à une fonctionnalité.

Réutilisation du code. Puisqu'il encapsule tous les composants/directives/providers/modules dont il a besoin, un module est comme une mini-application autonome qui peut facilement être réutilisée d'un projet Angular à l'autre.

Amélioration des performances. Un module peut être chargé via le routeur de manière asynchrone (aka lazy-loading), seulement au moment où l'utilisateur visite une page précise de l'application. Cela évite de charger inutilement du code pour tous les utilisateurs.



NgModules natifs Angular

BrowserModule - Contient les fonctionnalités nécessaires à l'exécution de l'application dans un navigateur.

- À importer uniquement dans le module racine (*AppModule*).
- NB. Ce module inclut *CommonModule*.

CommonModule - Contient les directives de base d'Angular comme *ngIf* et *ngFor*. A priori, TOUS VOS MODULES devront importer ce module (sinon, ils ne pourront pas utiliser les directives de base).

FormsModule - Contient les fonctionnalités liées au formulaires.

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, ... ],  
})
```




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  
  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  
  providers:[ ContactService, UserService ],  
  // Composant(s) à bootstrapper - Uniquement dans le module racine.  
  bootstrap:[ AppComponent ]  
})  
export class ContactModule { }
```



NgModule avec le CLI

ng generate module toto

Génère le fichier *toto/toto.module.ts*

Modifie *app.module.ts*

Options

```
ng g m toto # Raccourci  
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
```



Module JS vs Module Angular

	Module JavaScript	Module Angular (NgModule)
Définition	1 module = 1 fichier de code	1 module = 1 classe décorée avec @NgModule
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 export	Propriété @NgModule.exports
Syntaxe d'import	Mot-clé import	Propriété @NgModule.imports

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 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 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. Une autre syntaxe permettant de différer le chargement d'un module au moment où on en a besoin (lazy-loading).



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 ne peut être déclaré que dans UN SEUL MODULE :

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 déclarations exportées :

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

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



Routes d'un feature module Déclaration, Activation



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.



Activation et lazy-loading

Par défaut, les routes d'un feature module viennent s'ajouter aux routes du module principal lorsque le feature module est importé.

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,
- 2) charger le feature module à la demande, uniquement quand son path est requis

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

```
// Chemins finaux
```

```
/path-to-quiz/quizzes
```

```
/path-to-quiz/quiz/:id
```



Fonctionnalités avancées

Routeur avancé, Formulaires avancés,
Affichage avancé

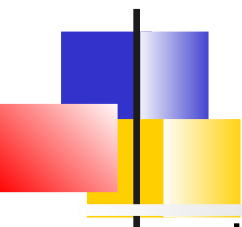


Gardes - Intro

Dans une application Angular, par défaut, tout le monde peut accéder à tous les chemins de l'application.

Ce n'est pas toujours souhaitable :

- Peut-être l'utilisateur doit-il être identifié pour accéder au composant cible ?
- Peut-être faut-il récupérer certaines données avant d'afficher le composant cible ?
- Peut-être faut-il sauvegarder les changements en cours avant de quitter un composant ?
- Ou demander à l'utilisateur si on peut abandonner les changements en cours plutôt que de les sauvegarder ?



Gardes - Fonctionnement

Un “**garde**” permet de contrôler le comportement du routeur :

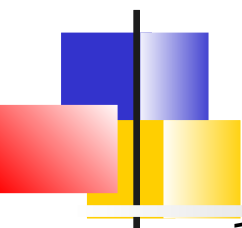
- S’il renvoie *true*, la navigation se poursuit.
- S’il renvoie *false*, la navigation est interrompue (l’utilisateur reste sur la même page).
- Il peut aussi rediriger l’utilisateur vers une autre page.

La valeur renvoyée par le garde est très souvent asynchrone :

- Attente que l’utilisateur réponde à une question.
- Attente que le serveur renvoie des données.
- Attente que les changements soient enregistrés sur le serveur.

Le garde peut donc renvoyer un booléen, un

Observable<boolean> ou une *Promise<boolean>*, et le routeur attendra que ces derniers soient dénoués à *true* ou *false*.



CanActivate

Empêcher d'accéder à une route

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

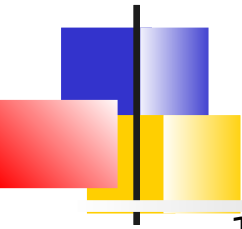
```
canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):  
  Observable<boolean>|Promise<boolean>|boolean {  
    // Utilise route.params.id (par exemple)  
    // Renvoie true ou false  
  }
```

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

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

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)

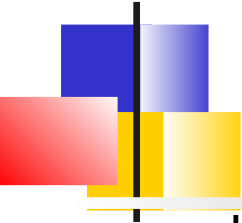
3. Ajouter ce service aux providers du module adéquat :

```
@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;  
  });  
}
```



Champ répété Côté composant (1/2)

Un champ répété est modélisé comme un *FormArray* contenant une série de *FormGroups*.

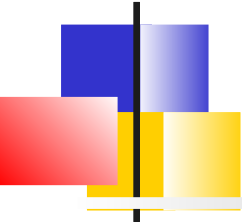
Imaginons un formulaire de contact qui peut contenir plusieurs adresses :

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
```

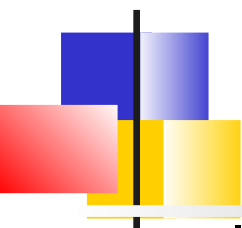
2) Manipulation du *FormArray* :

```
// 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>
```



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;  
});
```



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
  }
}
```




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>;
  getSerializedPanes(): 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()...*.



Exemple

```
@Injectable()
export class GithubAPIInterceptor implements HttpInterceptor {
  intercept(req: HttpRequest<any>, next: HttpHandler):
    Observable<HttpEvent<any>> {
    if (req.url.includes('api.github.com')) {
      const clone = req.clone({ setHeaders:{ 'Authorization': `token
${OAUTH_TOKEN}` } });
      return next.handle(clone);
    }
    return next.handle(req);
  }
}
```