

---

# Angular pour Débutants

Formation Complète

De zéro à la production



# Objectifs de la Formation



## Architecture Modulaire

Comprendre comment Angular organise le code en modules, composants et services pour des applications évolutives.



## Maîtrise de TypeScript

Utiliser le typage fort et les fonctionnalités modernes d'ES6+ pour écrire un code plus robuste et maintenable.



## Outils & Écosystème

Maîtriser la CLI Angular, le routage, les formulaires et la communication HTTP avec les APIs.



## Déploiement Production

Savoir compiler, optimiser et déployer votre application Angular sur un serveur web.

# Qu'est-ce qu'Angular ?

Composants

Routage

## ✓ Framework Complet

Une plateforme tout-en-un développée et maintenue par Google pour créer des applications web performantes.



## ✓ Basé sur TypeScript

Utilise TypeScript par défaut pour offrir un typage fort, une meilleure autocomplétion et moins de bugs.

Formulaires

Client HTTP

## ✓ Architecture SPA

Crée des "Single Page Applications" où la navigation est fluide sans rechargement complet de la page.

# Pourquoi Choisir Angular ?



## Tout Inclus

Contrairement à d'autres librairies, Angular fournit tout ce dont vous avez besoin dès le départ.

- ✓ Routage puissant
- ✓ Gestion de formulaires
- ✓ Client HTTP
- ✓ Outils de test



## Productivité

Des outils conçus pour accélérer le développement et maintenir la qualité du code.

- ✓ Angular CLI
- ✓ Typage fort (TypeScript)
- ✓ Autocomplétion intelligente
- ✓ Refactoring facile



## Entreprise Ready

Une plateforme stable et maintenue par Google, idéale pour les projets à grande échelle.

- ✓ Support Long Terme
- ✓ Architecture modulaire
- ✓ Communauté immense
- ✓ Mises à jour régulières

# Angular vs React vs Vue



## Angular

Type

Framework Complet

Langage

TypeScript

Philosophie

Tout inclus (Opinionated)



## React

Type

Bibliothèque UI

Langage

JavaScript / JSX

Philosophie

Flexible (Minimaliste)



## Vue

Type

Framework Progressif

Langage

HTML / JS

Philosophie

Approachable

# Prérequis pour cette Formation

## Connaissances Requises



### HTML & CSS

Structure de page et styles de base.



### JavaScript (ES6+)

Variables, fonctions, objets, tableaux.



### Ligne de Commande

Navigation basique (cd, ls, mkdir).

## Outils Nécessaires



### Node.js & npm

Environnement d'exécution (LTS recommandé).



### Éditeur de Code

VS Code (recommandé) ou WebStorm.



### Navigateur Moderne

Chrome, Firefox ou Edge avec DevTools.

# Node.js & npm

L'infrastructure technique de votre projet Angular



## Node.js

Environnement d'exécution JavaScript construit sur le moteur V8 de Chrome. Il permet d'exécuter les outils de développement Angular (CLI, compilateur) en dehors du navigateur.

RUNTIME ENVIRONMENT

## npm

Le gestionnaire de paquets par défaut pour Node.js. Il est utilisé pour installer Angular, ses dépendances, et lancer les scripts de commande (start, build, test).

PACKAGE MANAGER

# Installation de Node.js

## 01 Télécharger

Rendez-vous sur [nodejs.org](#) et téléchargez la version LTS (Long Term Support) recommandée pour la plupart des utilisateurs.

## 02 Installer

Lancez l'installateur et suivez les instructions. Les options par défaut conviennent parfaitement pour commencer.

## 03 Vérifier

Ouvrez votre terminal ou invite de commande et tapez les commandes ci-contre pour confirmer l'installation.

```
user@pc:~$node -v
v18.16.0

user@pc:~$npm -v
9.5.1

user@pc:~$
```

# Angular CLI



Command Line Interface

Bien plus qu'un simple outil, la CLI est le moteur qui propulse tout le cycle de développement Angular.



## Automatisation

Générez des composants, services et modules complets en une seule commande, sans erreur de boilerplate.



## Standardisation

Assure que tous les développeurs suivent les mêmes conventions de structure et de style (Style Guide officiel).



## Optimisation

Gère la compilation complexe, le tree-shaking et la minification pour des performances maximales en production.

# Installation de la CLI

```
user@pc:~$ npm install -g @angular/cli
```

## npm

Le gestionnaire de paquets de Node.js qui va télécharger l'outil.

## install

La commande standard pour ajouter un nouveau paquet à votre environnement.

## -g

Option **Global**. Indispensable pour accéder à la commande ng depuis n'importe quel dossier.

## @angular/cli

Le nom officiel du paquet contenant tous les outils de la ligne de commande Angular.

# Créer un Projet Angular

```
user@dev:~$ ng new mon-app
? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use? CSS
CREATE mon-app/package.json (1024 bytes)
CREATE mon-app/src/app/app.component.ts
✓ Packages installed successfully.

user@dev:~$ cd mon-app

user@dev: ~/mon-app $ ng serve -o
✓ Browser application bundle generated complete.
** Angular Live Development Server is listening on localhost:4200, open your browser on http://localhost:4200/ **
```

## 1. Génération

ng new

Crée le dossier du projet, génère les fichiers de configuration et installe toutes les dépendances npm nécessaires.

## 2. Navigation

cd

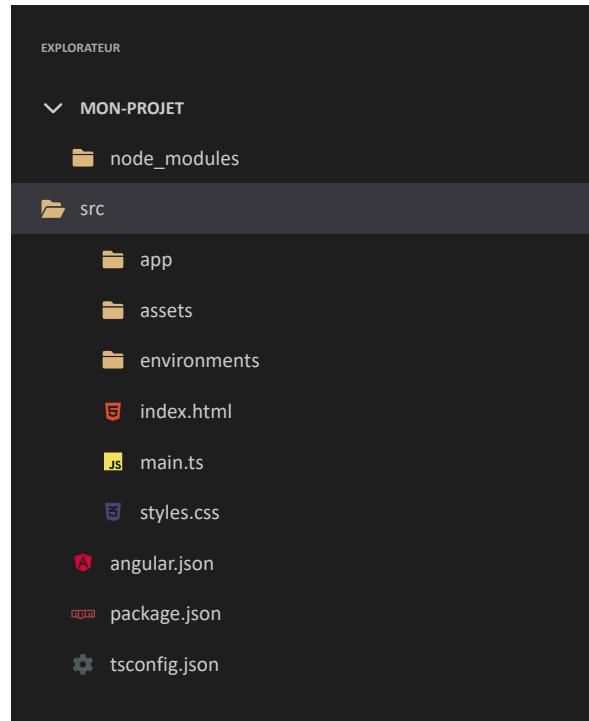
Déplacez-vous dans le répertoire nouvellement créé pour exécuter les commandes dans le contexte du projet.

## 3. Exécution

ng serve

Compile l'application en mémoire, lance un serveur local et ouvre automatiquement le navigateur (option -o).

# Structure du Projet



## src/

RACINE

Le dossier principal où vous passerez 99% de votre temps. Il contient tout le code source de votre application.

## src/app/

LOGIQUE

Contient les **composants**, modules, et services. C'est ici que la logique de votre application réside.

## src/assets/

STATIQUE

Pour les ressources statiques comme les images, les polices de caractères ou les fichiers de traduction.

## Fichiers de Configuration

`angular.json` configure le workspace CLI.

`package.json` gère les dépendances npm.

`tsconfig.json` configure le compilateur TypeScript.

# Fichiers Clés



## angular.json

Configuration globale du workspace, des projets, et des outils de build/test.



## package.json

Liste des dépendances npm et scripts de commande (start, build, test).



## src/main.ts

Point d'entrée de l'application. C'est ici que le module racine est bootstrapé.



## src/index.html

La page HTML principale qui héberge l'application via la balise <app-root>.



## tsconfig.json

# ES6+ : Modules

Les modules permettent de découper le code en fichiers réutilisables et maintenables. Chaque fichier est un module avec son propre scope.

math.js (Export)

```
// Exportation nommée  
  
export const PI = 3.14;  
  
// Exportation de fonction  
  
export function add (a, b){  
    return a + b;  
}
```

app.js (Import)

```
// Importation sélective  
  
import { PI , add } from './math.js' ;  
  
console .log (PI );  
// 3.14
```



## Encapsulation

Les variables définies dans un module ne sont pas accessibles globalement, sauf si elles sont explicitement exportées.

## Réutilisabilité

Un module peut être importé dans n'importe quel autre fichier, favorisant le principe DRY (Don't Repeat Yourself).

## Organisation

Permet de structurer l'application en fonctionnalités logiques plutôt qu'en un seul fichier monolithique.

# ES6+ : Classes



## Sucré Syntaxique

Une syntaxe plus claire et orientée objet pour créer des objets et gérer l'héritage, basée sur les prototypes.



## Structure Familière

Rapproche JavaScript des langages comme Java ou C#, facilitant la transition pour les développeurs backend.



## Fondation d'Angular

Les composants, services et directives Angular sont tous définis comme des classes.

```
class Developpeur { constructor(nom, langage) { this.nom = nom; this.langage =  
langage; } coder() { return `${this.nom} code en ${this.langage}`; } } //  
Instanciation const dev = new Developpeur('Alice', 'Angular'); console.log(dev.coder());
```

# Fonctions Fléchées

ES5 (Classique)

```
var double = function(x) {  
    return x * 2;  
};
```



ES6 (Moderne)

```
const double = (x) => x * 2;
```



## Syntaxe Concise

Plus besoin des mots-clés `function` et `return` pour les expressions simples. Le code devient plus lisible et direct.



## Contexte "this" Lexical

La fonction fléchée ne crée pas son propre contexte `this`. Elle hérite du contexte parent, résolvant le problème classique du `var self = this`.



## Usage dans Angular

Omniprésentes dans Angular, notamment pour les callbacks, les Observables RxJS et les méthodes de tableau (`map`, `filter`).

# Templates Littéraux

66

## Syntaxe Backticks

Utilise les accents graves `` au lieu des guillemets simples ou doubles pour définir les chaînes.

</>

## Interpolation

Permet d'insérer des expressions directement dans la chaîne avec  `${expression}`.

☰

## Multi-lignes

Supporte nativement les sauts de ligne sans avoir besoin de caractères d'échappement `\n`.

```
var message='Bonjour '+nom+'\n'+  
'Vous avez '+age+' ans.';
```

```
const message = `Bonjour ${ nom },  
Vous avez ${ age }ans.`;
```

# ES6+ : Destructuration

## Objets

Extrait les propriétés d'un objet directement dans des variables portant le même nom.

ES6

```
const user = { nom: 'Alice', age: 25 };

// Extraction directe
const { nom, age } = user;

console.log(nom); // 'Alice'
```

## Tableaux

Extrait les valeurs d'un tableau dans des variables selon leur position (index).

ES6

```
const coords = [10, 20];

// Extraction par position
const [x, y] = coords;

console.log(x); // 10
```

# Spread Operator

L'opérateur `...` permet d'étendre un itérable (tableau, chaîne) ou un objet en éléments individuels. C'est l'outil clé pour l'immutabilité.

## Tableaux

```
// Fusionner des tableaux  
  
const parts = [ 1, 2];  
const total = [... parts , 3, 4];  
// Résultat: [1, 2, 3, 4]
```

```
// Cloner un tableau  
  
const clone = [... parts ];
```

## Objets

```
// Fusionner / Étendre  
  
const base = { nom : 'A'};  
const user = { ... base , age : 30 };  
// { nom: 'A', age: 30 }
```

```
// Mise à jour immuable  
  
const updated = { ... user , age : 31 };
```

# Atelier 1 : Configuration

Préparation de l'environnement de développement

## Objectif

Installer et configurer les outils nécessaires pour commencer à développer avec Angular : Node.js et Angular CLI.

### 1 Node.js

Téléchargez et installez la version **LTS** (Long Term Support) de Node.js depuis le site officiel.

### 2 Angular CLI

Installez l'outil en ligne de commande Angular globalement sur votre machine via npm.

### 3 Vérification

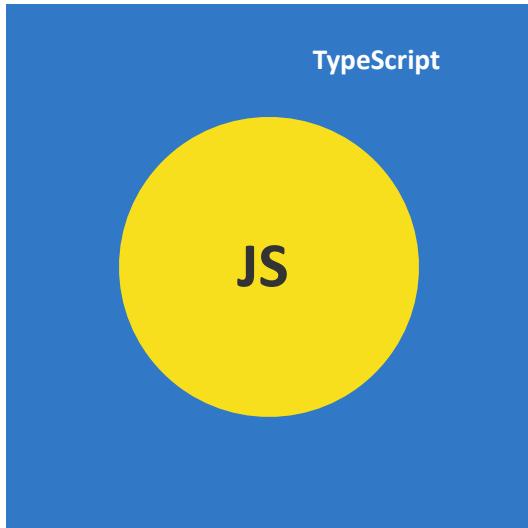
Assurez-vous que tout est correctement installé en vérifiant la version de la CLI.

```
# Vérifier après install  
node-v  
npm-v
```

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

```
ng version
```

# TypeScript : Introduction



## Le Superset Typé

TypeScript n'est pas un nouveau langage, c'est une [surcouche](#) de JavaScript. Tout code JavaScript valide est aussi du code TypeScript valide.



## Sécurité du Typage

Le typage statique permet de détecter les erreurs [pendant le développement](#) plutôt qu'à l'exécution chez le client.



## Outilage Supérieur

Grâce aux types, les éditeurs (VS Code) offrent une autocomplétion intelligente, une navigation précise et un refactoring sûr.

# Types Primitifs

TypeScript ajoute un typage statique optionnel à JavaScript. Les trois types les plus fondamentaux correspondent aux primitives JavaScript.

A

**string**

Représente des données textuelles. Peut utiliser des guillemets simples, doubles ou des backticks.

```
let framework: string='Angular';
```

#

**number**

Tous les nombres en TypeScript sont des valeurs à virgule flottante. Il n'y a pas de distinction int/float.

```
let version: number=16.2;
```



**boolean**

Le type le plus simple, ne pouvant prendre que deux valeurs : true ou false.

```
let isReady: boolean=true;
```

# Types Complexes

## Tableaux (Arrays)

Collection d'éléments du même type.

```
letskills: string[] = ['Angular', 'RxJS'];
// Syntaxe alternative (Générique)
letscores: Array<number> = [10, 20];
```

## Tuples

Tableau à longueur fixe et types hétérogènes.

```
// [Nom, Age, Actif]
letuser: [string, number, boolean];
user = ['Alice', 25, true];
```

## Énumérations (Enums)

Ensemble de constantes nommées.

```
enumRole { Admin, User, Guest }
letmyRole: Role = Role.Admin;
```

## Union Types

Une valeur pouvant être de plusieurs types.

```
letid: string | number;
id = 101; // Valide
id = 'A-101'; // Valide
```

# Interfaces



## Le Contrat

Une interface définit la **forme** (shape) qu'un objet doit avoir.

C'est un contrat que votre code s'engage à respecter.

TypeScript



## Propriétés Optionnelles

Le symbole ? permet de définir des propriétés qui ne sont pas obligatoires dans l'objet final.



## Zéro Runtime

Les interfaces existent uniquement pour le compilateur TypeScript. Elles disparaissent totalement dans le code JavaScript généré.

```
interface Utilisateur {  
    nom : string ;  
    age : number ;  
    email ?: string ; // Optionnel  
}  
  
// Utilisation valide  
const admin : Utilisateur = {  
    nom : 'Alice' ,  
    age : 30  
};  
  
// Erreur : propriété manquante  
const guest : Utilisateur = {  
    nom : 'Bob'  
}; // Error: Property 'age' is missing
```

# Classes TypeScript

TypeScript

```
class Utilisateur {  
    // 1. Propriétés typées  
    nom : string ;  
    age : number ;  
  
    // 2. Constructeur  
    constructor (n: string , a: number ) {  
        this.nom = n;  
        this.age = a;  
    }  
  
    // 3. Méthode avec retour  
    sePresenter (): string {  
        return `Je suis ${this.nom}` ;  
    }  
}
```

## 1 Déclaration des Propriétés

Contrairement à ES6, les propriétés doivent être déclarées et typées [avant](#) le constructeur. Cela définit la "forme" de l'objet.

## 2 Constructeur Typé

Les arguments du constructeur sont typés pour garantir que l'objet est initialisé avec les bonnes données.

## 3 Retour de Méthode

On spécifie explicitement le type de retour de la méthode (ici : string) pour éviter de retourner accidentellement une autre valeur.

# Modificateurs d'Accès

```
classEmploye { publicnom: string; protectedrole: string;  
privatesalaire: number; constructor() { this.nom = 'Alice'; //  
OKthis.role = 'Dev'; // OKthis.salaire = 5000; // OK } }
```



## public

Tout le monde

Le niveau par défaut. La propriété est accessible depuis l'intérieur de la classe, les classes enfants, et depuis l'extérieur (instances).



## protected

Héritage

Accessible dans la classe où elle est définie et dans toutes les classes qui en héritent (extends). Invisible depuis l'extérieur.



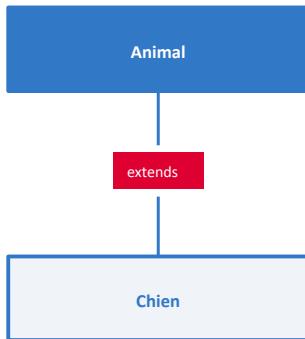
## private

Interne

Le niveau le plus restrictif. Accessible **uniquement** à l'intérieur de la classe elle-même. C'est la base de l'encapsulation.

# Héritage de Classes

L'héritage permet de créer une nouvelle classe qui réutilise, étend ou modifie le comportement d'une classe parente.



```
class Animal { constructor(public nom: string) {} bouger() { console.log(`${this.nom} bouge.`); } } // Héritage avec 'extends'  
class Chien extends Animal { aboyer() { console.log('Wouf! Wouf!'); } } const rex = new Chien('Rex'); rex.bouger(); // Méthode héritée rex.aboyer(); // Méthode propre
```

**Note :** Si la classe enfant a un constructeur, elle *doit* appeler `super()`.

# Génériques



## Variables de Type

Les génériques permettent de créer des composants réutilisables qui fonctionnent avec **plusieurs types** tout en conservant les informations de type (contrairement à any).

### Fonction Générique

```
// T capture le type passé par l'utilisateur
function identite <T>( arg : T): T{
    return arg ;
}

// Utilisation explicite
let output1 = identite <string>("Angular");
// output1 est de type 'string'

// Inférence de type
let output2 = identite (42);
// output2 est de type 'number'
```

# Décorateurs



## Méta-programmation

Un décorateur est une fonction spéciale qui permet d'ajouter des métadonnées ou de modifier le comportement d'une classe, d'une méthode ou d'une propriété.

### Component

```
@Component({ selector: 'app-root', templateUrl: './app.component.html' })  
export class AppComponent { @Input() title: string; constructor() {} }
```



De Classe



De Méthode



De Propriété



De Paramètre



*En Angular, les décorateurs sont essentiels pour définir le rôle de chaque classe.*

# Décorateurs de Classe

```
// Définition du décorateurfunction Sceau(constructor: Function)  
{ Object.seal(constructor); constructor.prototype.log = function()  
{ console.log("Classe scellée"); } } // Application@Sceau class Utilisateur  
{ constructor(public nom: string) {} }
```



## Cible : Le Constructeur

Un décorateur de classe reçoit le **constructeur** de la classe comme argument unique. Il peut l'inspecter ou le modifier.



## Usage dans Angular

Angular utilise intensivement ces décorateurs (@Component, @NgModule) pour attacher des **métadonnées** à vos classes.



## Transformation

Ils permettent de transformer une simple classe TypeScript en un élément complexe du framework (Composant, Service, etc.).

# Décorateurs de Propriété

## Définition

Ils sont déclarés juste avant une propriété de classe. Ils permettent d'ajouter des métadonnées ou de modifier la façon dont la propriété est gérée par le framework.

## Exemples Angular

- @Input()
- @Output()
- @HostBinding()
- @ContentChild()

Component.ts

```
export class ProductCard {  
  
    // Propriété reçue du parent  
    @Input()  
    productTitle : string = '';  
  
    // Événement émis vers le parent  
    @Output()  
    onBuy = new EventEmitter<void>();  
  
    // Liaison à une propriété du DOM hôte  
    @HostBinding('class.active')  
    isActive : boolean = false;  
}
```

# Décorateurs de Méthode

Ils permettent d'intercepter, de modifier ou de remplacer la définition d'une méthode. Ils reçoivent 3 arguments :

## Arguments

<b>target</b>	Le prototype de la classe.
<b>key</b>	Le nom de la méthode.
<b>descriptor</b>	L'objet PropertyDescriptor.

## Le Descriptor

Contient la propriété value qui est la fonction elle-même. En la modifiant, on change le comportement de la méthode.

```
functionLog(target, key, descriptor) { // 1. Sauvegarder la méthode originaleconstoriginal =
  descriptor.value; // 2. Remplacer par une nouvelle fonctiondescriptor.value = function(...args)
  { console.log(`Appel de ${key} avec:`, args); // 3. Exécuter l'originalereturnoriginal.apply(this,
    args); }; } classCalculatrice { @Logadd(a, b) { returna + b; } }
```

# Union Types



L'opérateur pipe | permet à une variable d'accepter **plusieurs types** définis.

C'est une alternative sécurisée au type any lorsque les possibilités sont connues.

```
let identifiant : string | number ;
```

// Cas 1 : Nombre

```
identifiant = 101 ; // Valide
```

// Cas 2 : Chaîne

```
identifiant = "EMP-101" ; // Valide
```

// Cas 3 : Booléen

```
identifiant = true ; // Erreur !
```

TypeScript

# Types Littéraux

## Valeurs Exactes

Un type littéral restreint une variable non pas à un "type" de données (comme string), mais à un ensemble précis de [valeurs spécifiques](#).

Valeurs Autorisées :

'start'

'center'

'end'

'middle'

TypeScript

```
// Définition du type
type Alignement = 'start' | 'center' | 'end';

let position : Alignement;

position = 'center'; // OK

// Erreur de compilation !
position = 'middle';
// Error: Type "'middle'" is not assignable to type
// 'Alignement'.
```

# Types Avancés

## Intersection

Combine plusieurs types en un seul. L'objet doit avoir toutes les propriétés.

```
typeAdmin = { role: 'admin' };
typeUser = { name: string };

// Combine les deux
typeSuperUser = Admin & User;

const: SuperUser = {
  role: 'admin',
  name: 'Alice'
};
```

## Conditionnels

Sélectionne un type en fonction d'une condition (comme un ternaire).

```
typeIsString<T> =
  T extends string ?
    true : false;

typeA = IsString<"Hello">; // true
typeB = IsString<number>; // false
```

## Mappés

Crée un nouveau type en itérant sur les propriétés d'un type existant.

```
// Rend tout optionnel
typePartial<T> = {
  [P in keyof T]?: T[P];
};

interface Todo { id: number }
typeOptionalTodo = Partial<Todo>;
```

# Configuration TypeScript

```
tsconfig.json

{
  "compilerOptions": {
    "target": "es2022",
    "module": "es2022",
    "strict": true,
    "outDir": "./dist", // Vital pour Angular
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true
  },
  "include": ["src/**/*"]
}
```



## Version Cible **target**

Définit la version de JavaScript générée (ex: ES5 pour les vieux navigateurs, ES2022 pour les modernes).



## Mode Strict **strict**

Active toutes les vérifications de type strictes. Recommandé pour éviter les erreurs silencieuses (ex: noImplicitAny).



## Décorateurs **experimentalDecorators**

Indispensable pour Angular. Active le support expérimental des décorateurs (@Component, etc.).

# Atelier 2 : TypeScript

Gestion de Catalogue Produits

## 1 L'Interface

Créez une interface `IProduit` avec les propriétés nom (`string`) et prix (`number`).

## 2 La Classe

Créez une classe `Smartphone` qui implémente cette interface. Ajoutez une méthode `afficherDetails()`.

## 3 Le Générique

Créez une classe générique `Catalogue<T>` capable de stocker une liste d'éléments de type `T`.

## 4 Utilisation

## Structure de départ

```
// 1. Interface
interface IProduit { ... }

// 2.
class Smartphone implements IProduit {
    constructor(...){}

    // ...
}

// 3.

// 4.
```

# Architecture Angular



## Modules (NgModules)

Le contexte de compilation. Ils regroupent les composants, directives, pipes et services associés.



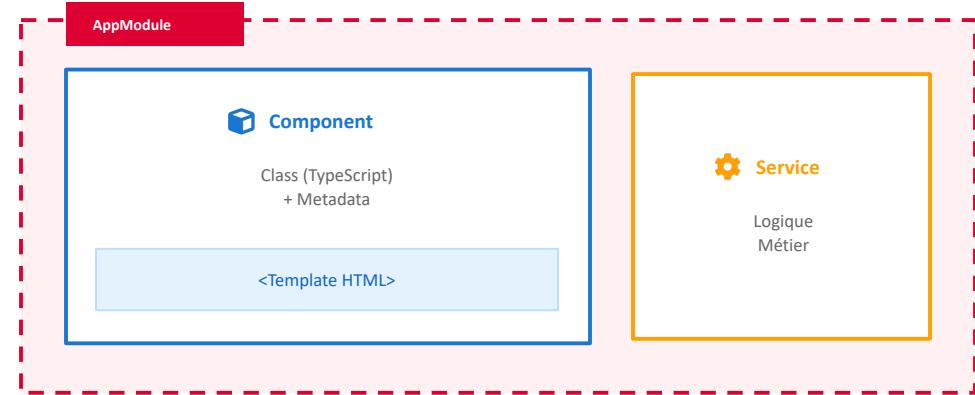
## Composants

Les blocs de construction de l'interface utilisateur. Ils contrôlent une portion de l'écran (la Vue).



## Services

La logique métier et les données. Ils sont injectés dans les composants pour partager des fonctionnalités.



# Composants

## La Brique Fondamentale

Le composant est l'élément de base d'une application Angular. Il contrôle une partie de l'écran appelée **la vue**.



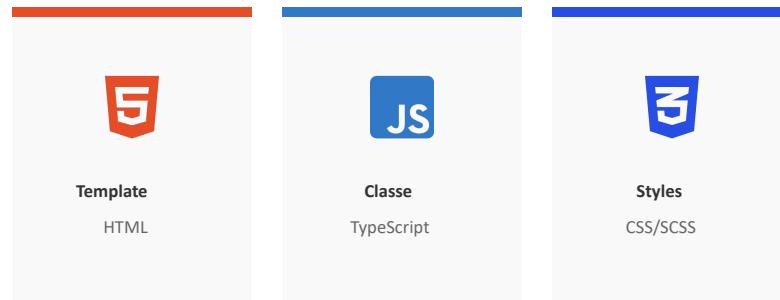
**Autonome** : Encapsule sa propre logique et son style.



**Réutilisable** : Peut être utilisé plusieurs fois.



**Hiérarchique** : Organisé en arbre (Parent/Enfant).



**Composant**

# Structure d'un Composant

```
// 1. Importationsimport { Component } from '@angular/core'; // 2.  
Métadonnées (Décorateur)@Component({ selector: 'app-root', templateUrl:  
'./app.component.html', styleUrls: ['./app.component.css'] }) // 3. Classe  
(Logique)export class AppComponent { title = 'Mon Application'; direBonjour()  
{ console.log('Bonjour !'); } }
```

## 1 Importations

On importe le décorateur Component depuis le cœur d'Angular pour pouvoir l'utiliser.

## 2 Métadonnées

Le décorateur @Component configure le composant : son nom dans le HTML (selector), son template et ses styles.

## 3 Classe (Logique)

Une classe TypeScript standard qui contient les données (propriétés) et le comportement (méthodes) de la vue.

# Templates

## 5 La Vue

Le template définit l'interface utilisateur du composant.  
C'est du HTML standard enrichi par la syntaxe Angular.

- ✓ Structure HTML
- ✓ Directives Angular
- ✓ Liaison de données (Binding)

```
@Component({ selector: 'app-user', templateUrl: './user.component.html', styleUrls: ['./user.component.css'] }) export classUserComponent {}
```

```
@Component({ selector: 'app-hello', template: `<h1>Bonjour {{ name }}</h1> <button (click)="sayHi()">Click</button>` }) export classHelloComponent {}
```

# Interpolation {{ }}

## Liaison Unidirectionnelle

L'interpolation permet d'afficher des données dynamiques du composant vers la vue (HTML). Angular évalue l'expression et insère le résultat sous forme de chaîne.

### Ce qu'on peut faire :

- ✓ Afficher une propriété
- ✓ Calculs arithmétiques ( $1 + 1$ )
- ✓ Appeler une méthode (qui retourne une valeur)
- ✗ Pas d'effets de bord (`=, new, ++, ;`)

```
Component.ts
```

```
export class AppComponent {  
  title = 'Mon App';  
  count = 10;  
}
```



```
Template.html
```

```
<h1>{{ title }}</h1>  
<p>Total: {{ count * 2 }}</p>
```



```
Navigateur
```

Mon App

Total: 20

# Property Binding [property]

## Liaison de Propriété

Permet de définir la valeur d'une propriété d'un élément DOM (ou d'une directive) à partir d'une propriété du composant.

Syntaxe : **[propriété]="expression"**



### Attributs HTML

src, href, disabled, hidden...



### Propriétés de Composant

@Input() d'un composant enfant

```
export class AppComponent {  
  imageUrl = 'assets/logo.png';  
  isButtonDisabled = true;  
}
```

Component.ts



<!-- Liaison à l'attribut src -->

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

Template.html

<!-- Liaison à la propriété disabled -->

```
<button [disabled] = "isButtonDisabled" >
```

Click me

```
</button >
```

# Event Binding (event)

## Réagir aux Actions

Permet d'écouter les événements du DOM (clic, frappe, survol) et d'exécuter une méthode du composant.



**Syntaxe :** Les parenthèses () entourent le nom de l'événement.



**Direction :** De la Vue (HTML) vers le Composant (TS).



**\$event :** Variable spéciale contenant les données de l'événement (ex: valeur d'un input).

Template

```
<button (click) = "sauvegarder()">  
Enregistrer  
</button>  
  
<input (input) = "onType($event)" />
```



Déclenche

Component

```
export class AppComponent {  
  
    sauvegarder () {  
        console .log ('Sauvegardé !');  
    }  
  
    onType (event : any ) {  
        console .log (event .target .value);  
    }  
}
```

# Two-Way Binding [(ngModel)]

## Synchronisation Totale

Permet de synchroniser les données dans les deux sens : du composant vers la vue, et de la vue vers le composant (ex: champs de formulaire).

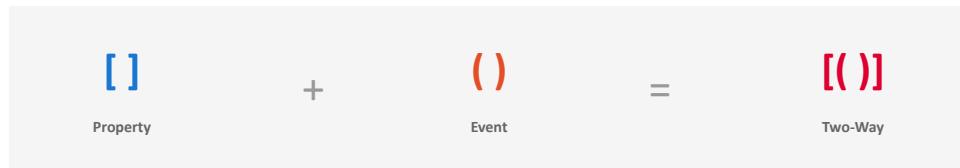


Banana in a Box

[( )]

La banane (event) est dans la boîte (property).

Nécessite l'import de  
FormsModule dans votre  
AppModule.



### Template HTML

```
<input type="text" [(ngModel)]="username"> <p>Bonjour {{ username }} !</p>
```

### AppModule (Configuration requise)

```
import { FormsModule } from '@angular/forms'; @NgModule({ imports: [ BrowserModule,  
FormsModule// Indispensable ! ] })
```

# Modules (NgModule)

## L'Organisateur

Un module est un conteneur qui regroupe un ensemble cohérent de fonctionnalités. Il définit le contexte de compilation pour les composants.



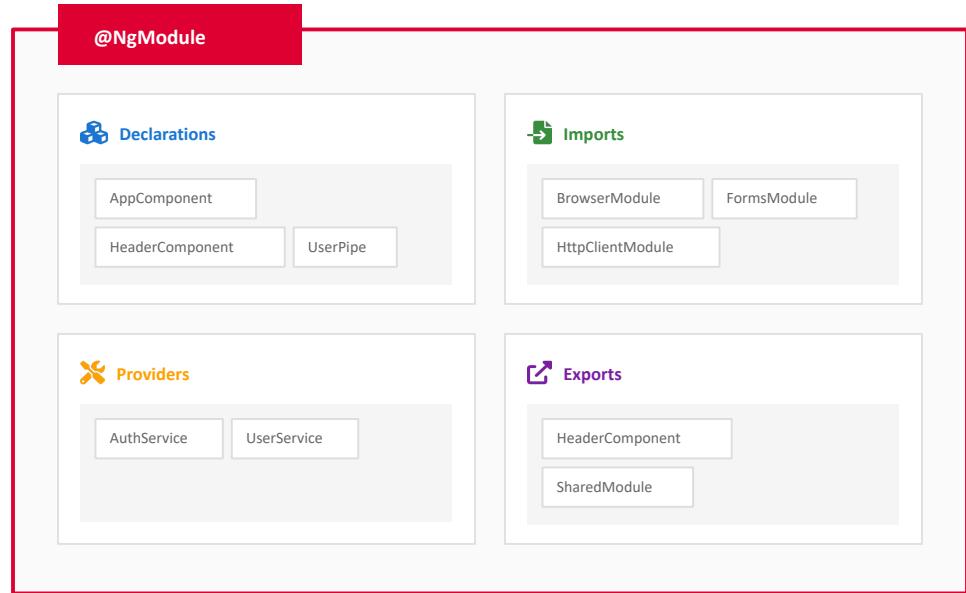
Regroupe composants, directives et pipes



Gère les dépendances (Imports)



Expose des fonctionnalités (Exports)



# Décorateur @NgModule

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

## declarations

Liste des composants, directives et pipes qui appartiennent à ce module. Ils peuvent s'utiliser entre eux.

## imports

Autres modules dont ce module a besoin pour fonctionner (ex: BrowserModule pour le navigateur).

## providers

Services à injecter. (Note : Depuis Angular 6, on préfère souvent providedIn: 'root' dans le service lui-même).

## bootstrap

Le composant racine qui est chargé au démarrage de l'application. Utilisé uniquement dans le module racine (AppModule).

# Décorateur @Component

```
@Component({ selector: 'app-profile', templateUrl: './profile.component.html', styleUrls: ['./profile.component.css'] }) export classProfileComponent { // Logique du composant... }
```

## selector

Définit le nom de la balise HTML personnalisée pour utiliser ce composant.

Exemple : <app-profile></app-profile>

## </> templateUrl

Chemin vers le fichier HTML contenant la vue. On peut aussi utiliser template pour du HTML inline.

## styleUrls

Tableau de chemins vers les fichiers CSS/SCSS. Les styles sont encapsulés et ne s'appliquent qu'à ce composant.

# Décorateur @Injectable

## Le Marqueur de Service

Ce décorateur marque une classe comme participant au système d'injection de dépendances. Il permet à Angular d'injecter ce service dans des composants ou d'autres services.



### Injectable

Indique que la classe peut recevoir des dépendances.



### providedIn: 'root'

Crée un singleton disponible dans toute l'application.

```
import { Injectable } from '@angular/core'; @Injectable({ providedIn: 'root' })  
export class AuthService { constructor() {} login() { // Logique de connexion } }
```



### Tree Shaking

L'utilisation de providedIn: 'root' permet à Angular de supprimer le service du bundle final s'il n'est jamais utilisé.

# Cycle de Vie

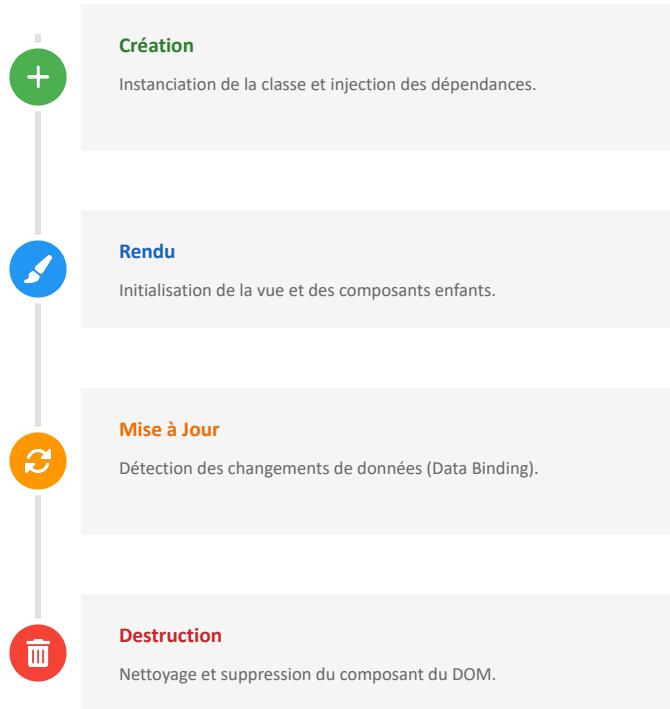
## Une Vie Gérée

Chaque composant a un cycle de vie géré par Angular : création, rendu, détection de changements, et destruction.

### Lifecycle Hooks

Angular offre des méthodes spéciales (hooks) pour nous permettre d'intervenir à des moments précis de ce cycle.

ⓘ Ces méthodes sont définies dans des interfaces (ex:  
OnInit ).



# Hooks du Cycle de Vie

## ngOnChanges

Appelé quand une propriété liée par @Input() change de valeur. C'est le tout premier hook exécuté.

## ngOnInit

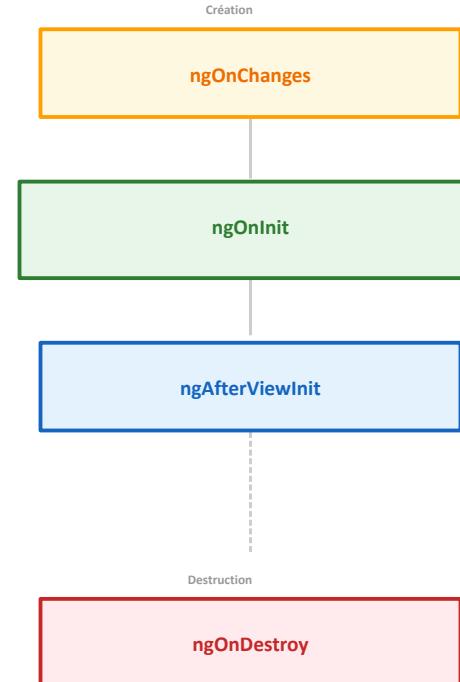
Appelé une seule fois après le premier affichage des propriétés. Idéal pour l'initialisation (appels API).

## ngAfterViewInit

Appelé après que la vue du composant (et ses enfants) a été initialisée.

## ngOnDestroy

Appelé juste avant la destruction du composant. Utilisé pour le nettoyage (unsubscribe, timers).



# ngOnInit

## L'Initialisation

Appelé une seule fois, juste après la création du composant et l'initialisation des propriétés liées (@Input).

C'est l'endroit idéal pour charger des données (API) ou configurer le composant.

### Constructor vs ngOnInit

**Constructor** Injection de dépendances uniquement. Pas de logique métier lourde.

**ngOnInit** Logique d'initialisation, appels HTTP, abonnements.

✓ Bonne Pratique : Implémenter l'interface OnInit

TypeScript

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

export class UserComponent implements OnInit {
  users : User [] = [];

  // Injection de dépendances
  constructor (private userService : UserService) {}

  // Logique d'initialisation
  ngOnInit (): void {
    this.userService.getUsers ()
      .subscribe (data => this.users = data );
  }
}
```

# ngOnDestroy

## Le Nettoyage

Appelé juste avant qu'Angular ne détruise le composant. C'est votre dernière chance pour nettoyer les ressources et éviter les fuites de mémoire.

### ⚠ À nettoyer impérativement :

- ✗ Souscriptions aux Observables (RxJS)
- ✗ Timers (setTimeout, setInterval)
- ✗ Écouteurs d'événements DOM manuels

⚠ Attention aux Memory Leaks !

TypeScript

```
export class ChatComponent implements OnInit, OnDestroy {  
  private sub : Subscription;  
  
  ngOnInit() {  
    // On stocke la souscription  
    this.sub = this.chatService.getMessages()  
      .subscribe(...);  
  }  
  
  ngOnDestroy() {  
    // On se désabonne pour libérer la mémoire  
    if( this.sub ){  
      this.sub.unsubscribe();  
    }  
  }  
}
```

# @Input

## Communication Descendante

Le décorateur @Input() permet à un composant enfant de recevoir des données depuis son composant parent.

### Parent vers Enfant

Flux de données unidirectionnel.

### Propriété Personnalisée

Crée un attribut HTML utilisable dans le template du parent.



# @Output & EventEmitter

## Communication Vers le Haut

Permet à un composant enfant d'envoyer des données ou de signaler une action à son composant parent.



### EventEmitter

Classe utilisée pour créer et émettre des événements personnalisés.



### Flux Ascendant

Contrairement à @Input, les données remontent l'arbre des composants.

```
@Output() deleteRequest = new EventEmitter<number>();  
  
onDelete() {  
  this.deleteRequest.emit(123);  
}
```

Enfant (Émetteur)

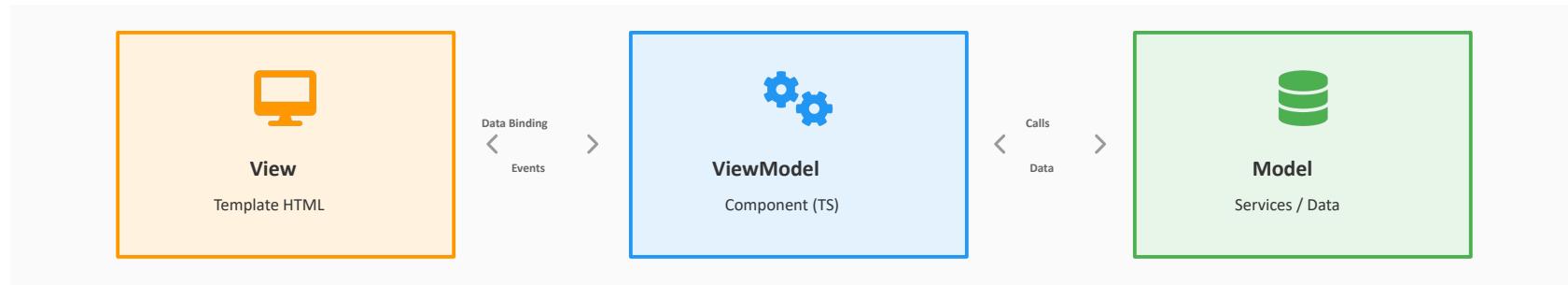


```
<app-item (deleteRequest)="supprimerItem($event)">  
</app-item>
```

Parent (Écouteur)

```
supprimerItem(id: number) {  
  console.log('ID reçu : ' + id);  
}
```

# Pattern MVVM



## View (La Vue)

C'est ce que l'utilisateur voit. Elle est passive et ne contient aucune logique métier. Elle affiche les données fournies par le ViewModel.

## ViewModel (Le Médiateur)

C'est le cerveau de la vue. Il contient l'état de l'interface et les méthodes pour réagir aux actions utilisateur. Il ne manipule pas le DOM directement.

## Model (Les Données)

Représente les données métier et la logique d'accès aux données (API). Il est indépendant de l'interface utilisateur.

# One-Way Data Binding



## A Interpolation

Utilisé pour insérer du texte dynamique dans le HTML.

```
<h1>{{ title }}</h1>
<p>Score: {{ 10 + 2 }}</p>
```

## Property Binding

Utilisé pour définir la valeur d'une propriété d'un élément ou d'une directive.

```
<img[src]="user.avatar">
<button[disabled]="isBusy">...</button>
```

# Two-Way Data Binding



Banana in a Box

Combinaison de Property Binding [] et Event Binding () .

[ ( ngModel ) ]



```
<input [(ngModel)] = "username" />  
  
<p> Bonjour {{ username }} ! </p>
```

Template HTML



# Variables de Template

## Référence Locale

Permet de créer une référence à un élément du DOM, à un composant ou à une directive directement dans le template.

#nom

Déclare une variable locale utilisable partout dans le template HTML.

**Utilité :** Accéder aux propriétés d'un élément (comme value, valid) sans passer par le code TypeScript.

### Exemple Pratique

```
<!-- Déclaration de la variable #phone -->  
<input type = "text" #phone placeholder = "Votre numéro" >  
  
<!-- Utilisation de la référence -->  
<button (click) = "call(phone.value)" >  
Appeler  
</button >
```



Ici, phone fait référence à l'objet DOM de l'input. On peut donc accéder à phone.value directement.

# @ViewChild

## Accès Direct au DOM

Permet d'accéder à un élément du DOM, une directive ou un composant enfant depuis la classe du composant parent.

### Cas d'usage :

- ✓ Donner le focus à un input
- ✓ Appeler une méthode d'un enfant
- ✓ Lire la taille d'un élément



Disponible uniquement après  
ngAfterViewInit()

Template HTML

```
<input #myInput type="text" />
<button (click)="focusInput()" >Focus</button>
```



Component Class

```
@ViewChild('myInput') inputRef : ElementRef;

ngAfterViewInit() {
  // L'élément est accessible ici
}

focusInput() {
  this.inputRef.nativeElement.focus();
}
```

# Atelier 3 : Créer un Composant

## ☰ Objectifs

### 1 Générer le composant

Créez un nouveau composant nommé user-profile via Angular CLI.

Terminal

```
ng generate component user-profile  
# ou en version courte :  
ng g c user-profile
```

### 2 Définir les données

Dans le fichier TypeScript, ajoutez des propriétés : name, job, et avatar.

Résultat attendu (App Component)

```
<div class="container">  
<h1>Mon Application</h1>  
<app-user-profile></app-user-profile>  
</div>
```

### 3 Créez le template

Affichez ces informations dans le HTML du composant en utilisant l'interpolation {{ }}.

### 4 Intégrer

Ajoutez le sélecteur du composant dans app.component.html pour l'afficher.



Astuce : Vérifiez que le serveur de développement est lancé avec ng serve.

# Change Detection

## La Synchronisation

Mécanisme interne qui permet à Angular de savoir quand l'état de l'application a changé pour mettre à jour la vue (DOM).

### Events DOM

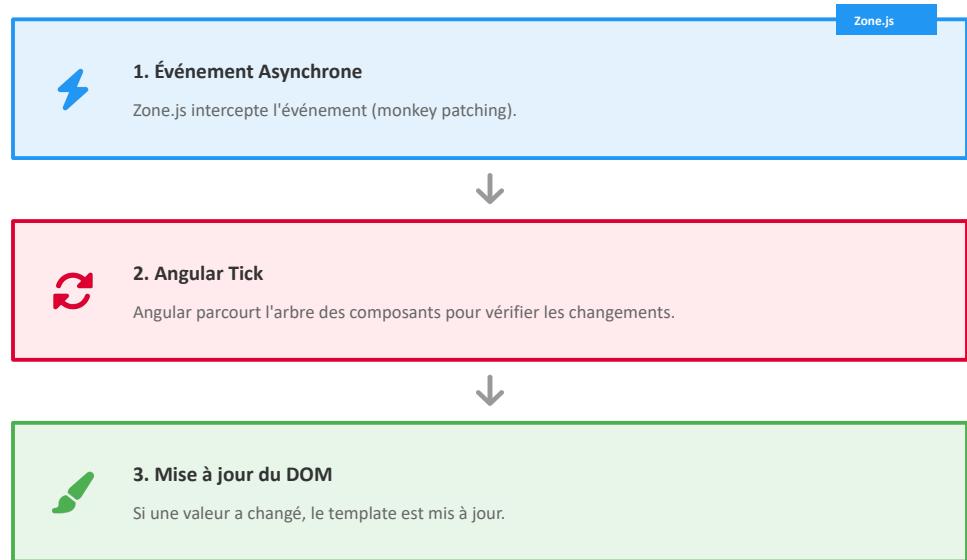
Click, Submit, Input...

### Timers

setTimeout, setInterval

### Requêtes HTTP

XHR, Fetch



# Stratégies de Détection



## Default

Angular vérifie le composant et tous ses enfants à chaque cycle de détection. Sûr mais coûteux.

## </> Implémentation

```
@Component({  
  selector: 'app-item',  
  templateUrl: './item.component.html',  
  changeDetection: ChangeDetectionStrategy.OnPush  
})  
export class ItemComponent { ... }
```



## OnPush

Angular ne vérifie le composant que si ses entrées (@Input) changent ou si un événement interne survient.

## ✓ Déclencheurs OnPush

➡ **Changement de référence @Input**  
L'objet doit être immuable (nouvelle instance).

➔ **Événement interne**  
Click, Submit, etc. déclenché dans le composant.

☰ **Pipe Async**  
Émission d'une nouvelle valeur par un Observable.

# Directives Structurelles

## Manipulation du DOM

Elles modifient la structure du DOM en ajoutant, supprimant ou manipulant des éléments HTML.



### Ajout / Suppression

Les éléments sont physiquement créés ou détruits.



### Impact Performance

Coûteux si mal utilisé (re-rendu complet).



### Syntaxe Astérisque

Toujours précédées par un \*.



L'astérisque est du "sucre syntaxique".

Angular le transforme en <ng-template>.

\*ngIf

Conditionnel

\*ngFor

Boucle

\*ngSwitch

Cas multiples



### Définition avec [hidden]

[hidden] masque l'élément via CSS (display: none), mais il reste dans le DOM.

\*ngIf retire complètement l'élément du DOM (libère la mémoire).

# ngIf

## Affichage Conditionnel

Ajoute ou supprime un élément du DOM en fonction d'une expression booléenne.

### Comportement :

- ✓ True : L'élément est créé et inséré.
- ✗ False : L'élément est détruit et retiré.



L'astérisque \* indique qu'il s'agit d'une directive structurelle qui modifie la structure du DOM.

```
<div *ngIf = "isLoggedIn" >  
  Bienvenue, utilisateur !  
</div>
```

Usage Simple

```
<div *ngIf = "isLoggedIn; else loginTemplate" >  
  Bienvenue !  
</div>  
  
<ng-template #loginTemplate >  
  <button>Se connecter</button>  
</ng-template>
```

Avec Bloc Else

# ngFor

## Boucles d'Affichage

La directive structurelle \*ngFor permet de répéter un élément HTML pour chaque entrée d'une liste (tableau).

### Variables Locales Exportées

index	Position (0-based)
first	Est le premier ? (bool)
last	Est le dernier ? (bool)
even/odd	Pair / Impair (bool)

```
<!-- Syntaxe de base -->  
  
<li *ngFor = "let user of users ">  
  {{ user.name }}  
</li>  
  
<!-- Avec index -->  
  
<div *ngFor = "let u of users ; let i=index" >  
  {{ i + 1 }} - {{ u.name }}  
</div >
```

Résultat Visuel

0 Alice Dupont

1 Bob Martin

2 Charlie Durand

# ngSwitch

## Choix Multiple

Affiche un élément parmi plusieurs possibles, en fonction d'une condition de correspondance (comme le switch en JavaScript).



### [ngSwitch]

Le conteneur qui évalue l'expression.



### \*ngSwitchCase

Affiche l'élément si la valeur correspond.

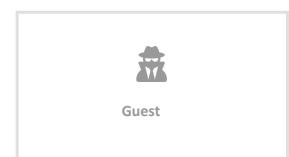
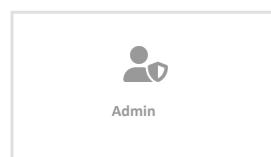


### \*ngSwitchDefault

Affiche si aucune correspondance n'est trouvée.

Exemple : Gestion des Rôles

```
<div [ngSwitch] = "user.role" >  
  
  <app-admin *ngSwitchCase = "'admin'" >  
    </app-admin>  
  
  <app-user *ngSwitchCase = "'user'" >  
    </app-user>  
  
  <app-guest *ngSwitchDefault >  
    </app-guest>  
  
</div>
```



# Directives d'Attributs

## Apparence & Comportement

Elles modifient l'apparence ou le comportement d'un élément DOM, d'un composant ou d'une autre directive.



### Style Dynamique

Changer les classes CSS ou les styles inline.



### Propriétés

Modifier les propriétés natives (disabled, hidden...).



### Syntaxe

Utilisées comme des attributs HTML classiques :

[directive] .



**ngClass**

Ajoute/supprime des classes CSS.

```
<div[ngClass]="{active: isActive}">
```

**ngStyle**

Modifie les styles inline.

```
<div[ngStyle]="{color: 'red'}">
```

# ngClass

## Classes Dynamiques

Permet d'ajouter ou de supprimer plusieurs classes CSS sur un élément HTML de manière dynamique.

### Cas d'usage fréquents :

- ✓ Marquer un élément comme actif
- ! Indiquer une erreur de validation
- ⌚ Changer de thème (clair/sombre)

### 1. Syntaxe Chaîne (Simple)

```
<div[ngClass]="'btn btn-primary'">...</div>
```

### 2. Syntaxe Tableau (Liste)

```
<div[ngClass]=["'btn', 'btn-primary'"]>...</div>
```

### 3. Syntaxe Objet (Conditionnelle)

```
<div[ngClass]="{ 'active': isActive, 'error': hasError }">  
État Dynamique  
</div>
```



Résultat si isActive = true :

Active

Résultat si hasError = true :

Error

# ngStyle

## Styles Dynamiques

Permet de définir plusieurs styles CSS inline de manière dynamique via un objet JavaScript.

 Couleurs calculées (ex: rouge si erreur)

 Dimensions variables (ex: barres de progression)

 Images de fond dynamiques

 **Attention aux unités :**  
N'oubliez pas d'ajouter 'px', '%', 'em' si nécessaire.

<!-- Syntaxe Objet -->

```
<div [ngStyle] = "{ 'background-color': user.color, 'font-size.px': 24, 'font-weight': isBold ? 'bold' : 'normal' }" >  
Texte Stylisé  
</div >
```

<!-- Exemple Barre de Progression -->

```
<div [ngStyle] = "{ 'width': progress + '%' }" >  
</div >
```

Résultat Visuel

Texte Stylisé

Progress: 75%

A horizontal progress bar consisting of a green segment on the left and a grey segment on the right. The green segment is approximately three-quarters of the way across, with the number '75%' centered below it.

75%

# Atelier 4 : Directives

## ☰ Objectifs

### 1 Préparer les données

Dans le composant, créez un tableau users contenant des objets avec name et role ('admin' ou 'user').

### 2 Afficher la liste

Utilisez \*ngFor pour afficher chaque utilisateur dans une liste <ul>.

### 3 Style conditionnel

Utilisez [ngClass] pour mettre en gras les utilisateurs ayant le rôle 'admin'.

### 4 Élément conditionnel

Ajoutez un bouton "Supprimer" qui n'apparaît que si l'utilisateur est un 'admin' (avec \*ngIf).

#### Données de test (TypeScript)

```
this.users = [  
  { name : 'Alice' , role : 'admin' },  
  { name : 'Bob' , role : 'user' },  
  { name : 'Charlie' , role : 'user' }  
];
```

#### 💡 Rappel Syntaxe

```
*ngFor="let u of users"  
[ngClass]="{{bold-text: u.role === 'admin'}}"  
*ngIf="u.role === 'admin'"
```

# Atelier 5 : TodoList

## Objectifs

### 1 Structure de Données

Définissez une interface Todo avec id, label (texte) et done (booléen).

### 2 Affichage

Affichez la liste des tâches avec \*ngFor. Barrez le texte si la tâche est terminée ([class.done]).

### 3 Interaction

Ajoutez un champ input et un bouton pour créer une nouvelle tâche.

### 4 Suppression

Ajoutez un bouton "Supprimer" sur chaque ligne pour retirer la tâche de la liste.

#### Modèle de Données

```
export interface Todo {  
  id: number;  
  label: string;  
  done: boolean;  
}
```

#### Logique Component

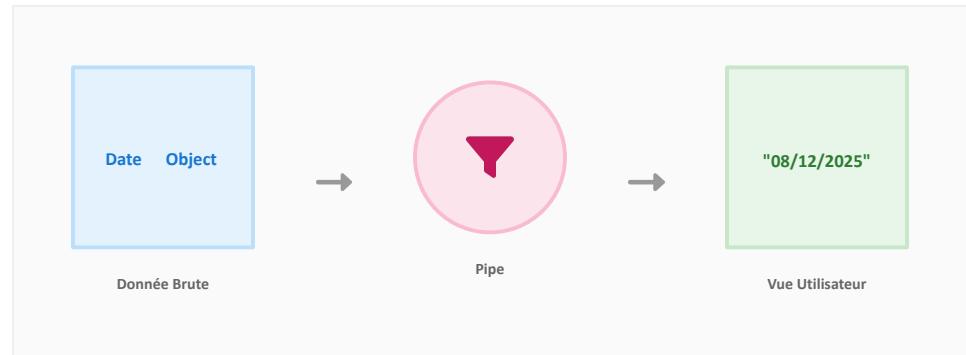
```
todos: Todo[] = [];  
  
addTodo(label: string){  
  const newTodo = {  
    id: Date.now(),  
    label,  
    done: false  
  };  
  this.todos.push(newTodo);  
}  
  
deleteTodo(todo: Todo){  
  this.todos = this.todos.filter(todo => todo.id !== todo.id);  
}
```

# Les Pipes

## Transformation d'Affichage

Les pipes sont des fonctions simples utilisées dans les templates pour transformer une valeur brute en une valeur affichable.

- ⌚ Modifie uniquement l'affichage
- 💾 Ne modifie pas la donnée source
- >\_ Inspiré des pipes Unix



```
 {{ valeur|nomDuPipe }}
```

# Pipes Intégrés

Angular fournit une collection de pipes prêts à l'emploi pour les transformations de données courantes.

## DatePipe

Formate une date

```
{{ today | date }}
```



Jun 15, 2023

## UpperCase

Transforme en majuscules

```
{{ 'hello' | uppercase }}
```



HELLO

## CurrencyPipe

Formate un montant

```
{{ 123.45 | currency }}
```



\$123.45

## JsonPipe

Affiche l'objet (Débug)

```
{{ user | json }}
```



```
{ "id": 1, "name": "Bob" }
```

# Pipes Paramétrés

Les pipes peuvent accepter un ou plusieurs **paramètres** pour affiner leur comportement et personnaliser la sortie.

valeur | pipeName :**param1** :**param2**

- Séparateur : deux-points :
- L'ordre des paramètres est strict
- Accepte des valeurs statiques ou dynamiques

## Format de Date

```
{{ today | date :'dd/MM/yyyy' }}
```

→ 25/12/2023

## Devise (Code & Symbole)

```
{{ price | currency :'EUR':'symbol' }}
```

→ €42.50

## Chaîne de caractères (Slice)

```
{{ 'Angular' | slice :0:3 }}
```

→ Ang

# Safe Navigation Operator

Gérer les données asynchrones sans crash

## ⚠ Le Problème

Si user est null (ex: chargement API), l'accès direct provoque une erreur fatale.

```
<p>{{ user.address.city }}</p>
```

✖ `TypeError: Cannot read property 'address' of undefined`

## 🛡 La Solution

L'opérateur `?.` vérifie si la valeur est définie avant d'accéder à la propriété suivante.

```
<p>{{ user?.address?.city }}</p>
```

✓ **Résultat :**  
Rien ne s'affiche (null), mais l'application ne plante pas.

L'opérateur `?.` (aussi appelé "Elvis Operator" ou "Optional Chaining") court-circuite l'évaluation si la valeur de gauche est `null` ou `undefined`.

# Configuration des Locales

## Internationalisation (i18n)

Par défaut, Angular utilise la locale américaine (en-US). Pour formater correctement les dates et nombres en français, il faut configurer l'application.

### Impact sur l'affichage

EN-US	12/31/2023
FR-FR	31/12/2023
EN-US	1,234.56
FR-FR	1 234,56

### app.module.ts

```
// 1. Importer les données de locale
import { localeFr } from '@angular/common/locales/fr';
import { registerLocaleData } from '@angular/common';
import { LOCALE_ID } from '@angular/core';

// 2. Enregistrer la locale
registerLocaleData(localeFr, 'fr');

@NgModule({
  providers: [
    // 3. Définir la locale par défaut
    { provide: LOCALE_ID, useValue: 'fr' }
  ]
})
export class AppModule {}
```

# Créer un Pipe Personnalisé

```
import { Pipe , PipeTransform } from '@angular/core' ;  
  
@Pipe ({  
name : 'expo'  
})  
export class ExponentialPipe implements PipeTransform {  
  
transform (value : number , exponent : number = 1): number {  
return Math .pow (value , exponent );  
}  
}
```

1

2

3

## Le Décorateur @Pipe

Définit les métadonnées. La propriété name est le nom que vous utiliserez dans le HTML (ex: {{ val | expo }}).

## L'Interface PipeTransform

Oblige la classe à implémenter la méthode transform. C'est le contrat standard d'Angular.

## La Méthode transform

Reçoit la valeur d'entrée et les arguments optionnels. Elle doit retourner la valeur transformée.

# Pipes Pures vs Impures



## Pures

Déclencheur

### Changement de Référence

- ✓ Comportement par défaut.
- ✓ Exécution rapide et optimisée.

- ℹ Ignore les mutations internes d'objets ou tableaux (ex:

array.push ).



## Impures

Déclencheur

### Chaque Cycle de Détection

- ⚠ S'exécute à chaque événement (clic, frappe, timer...).
- ✓ Détecte les mutations internes.
- ❗ Peut impacter les performances.

```
@Pipe({  
  name: 'myFilter',  
  pure: false  
})
```

# Atelier 6 : Pipe Personnalisé

## Objectifs

### 1 Générer le Pipe

Utilisez Angular CLI pour créer un pipe nommé PhoneFormat.

Terminal

```
ng generate pipe pipes/phone-format
```

### 2 Implémenter la Logique

Dans transform(), formatez une chaîne de 10 chiffres en format lisible (ex: 06 12 34 56 78).

phone-format.pipe.ts

```
transform(value: string): string {  
  if (!value) return '';  
  // Exemple simple : regex pour grouper par 2  
  return value.replace(/(\d{2})(?=\d)/g, '$1 ');  
}
```

### 3 Utiliser dans la Vue

Appliquez le pipe sur un numéro de téléphone brut dans votre template HTML.

app.component.html

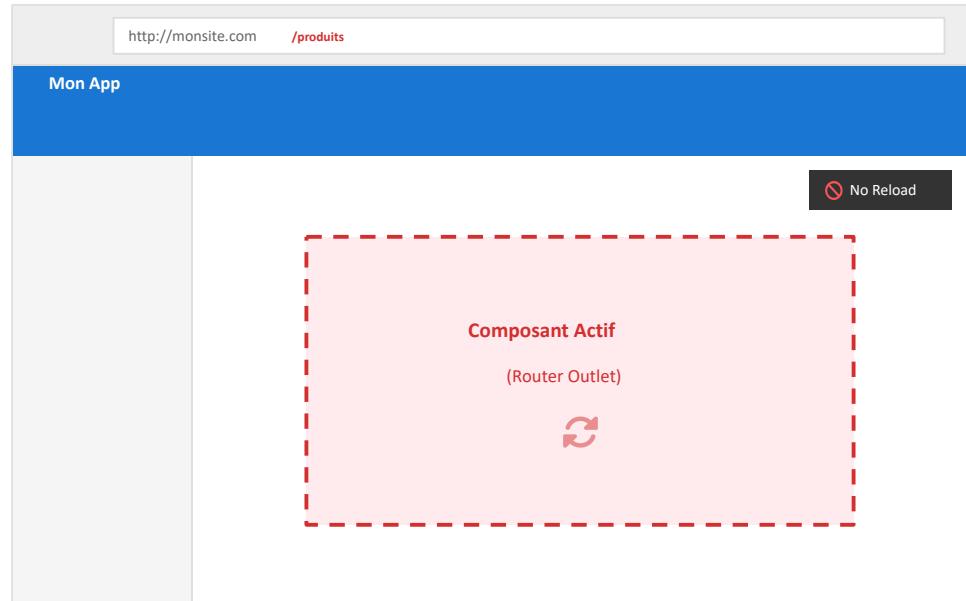
```
<p>Tél : {{ '0612345678' | phoneFormat }}</p>
```

# Le Routage (Routing)

## Single Page Application (SPA)

Une application Angular ne charge qu'une seule page HTML (index.html). La navigation est simulée par le remplacement dynamique des composants.

- ⚡ Pas de rechargement de page
- ➡ Navigation fluide et instantanée
- 🔗 L'URL change pour refléter l'état



Le "Shell" (Header/Sidebar) reste fixe.  
Seul le contenu central change.

# Configuration des Routes

## Tableau de Routes

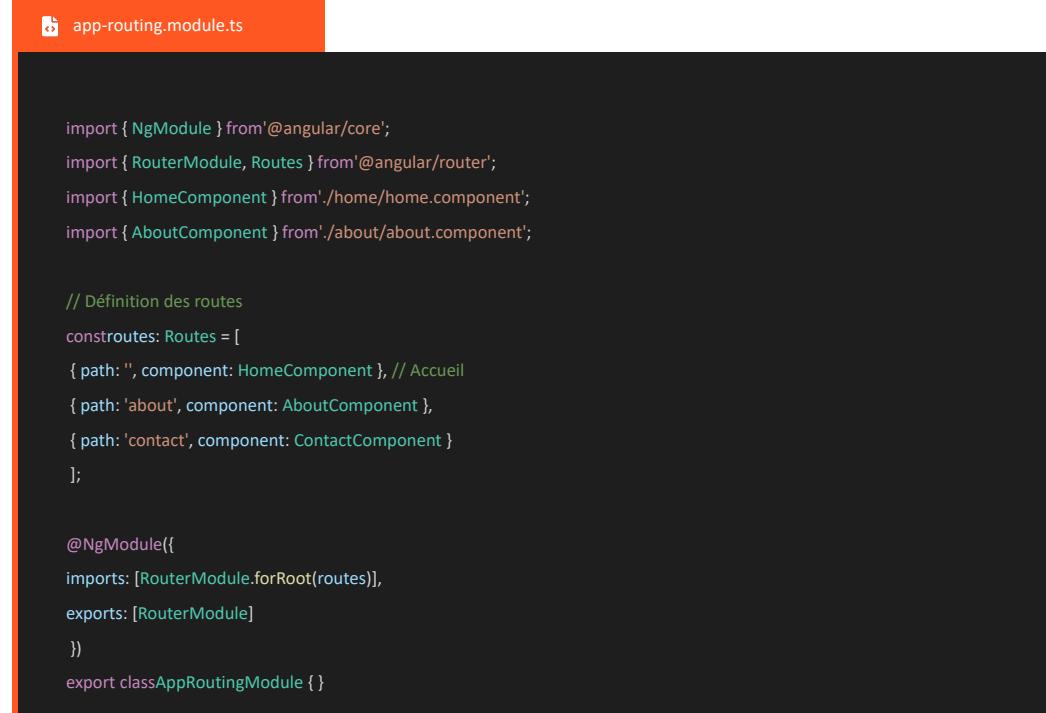
On définit un tableau d'objets de type `Routes`. Chaque objet associe un `path` (URL) à un `component`.

## Initialisation

La méthode `RouterModule.forRoot()` configure le service de routage à la racine de l'application.

## Ordre Important

Le routeur parcourt le tableau du haut vers le bas. La première correspondance gagne ("First Match Wins").



The screenshot shows a code editor window with a dark theme. The title bar says "app-routing.module.ts". The code inside the file is:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { AboutComponent } from './about/about.component';

// Définition des routes
const routes: Routes = [
  { path: '', component: HomeComponent }, // Accueil
  { path: 'about', component: AboutComponent },
  { path: 'contact', component: ContactComponent }
];

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

# Router Outlet

## Le Point d'Insertion

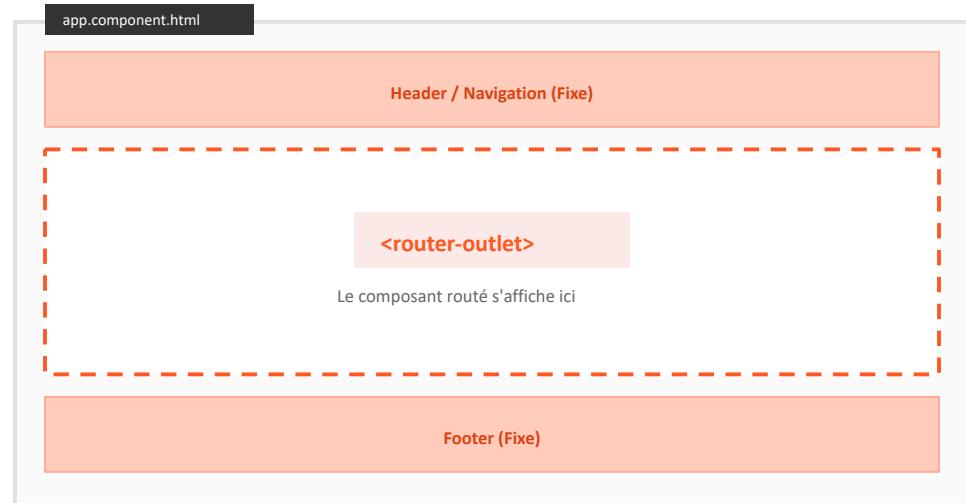
La directive <router-outlet> agit comme un marqueur dynamique dans votre template. C'est ici que le routeur va charger le composant correspondant à l'URL active.

## Content Dynamique

Le composant change, mais le reste de la page (header, menu) reste fixe.

## </> Directive Angular

Fait partie du module RouterModule .



```
<!-- Structure typique d'une SPA -->
<app-header></app-header>
```

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

```
<app-footer></app-footer>
```

# La Directive routerLink

## Navigation Interne

Pour naviguer entre les vues sans recharger la page, on remplace l'attribut standard href par la directive Angular routerLink.



### Attention :

L'utilisation de href provoque un rechargeement complet de l'application (perte de l'état, lenteur).



### HREF

Rechargement complet



### routerLink

Navigation SPA fluide

#### Lien Statique (Chaîne)

```
<a routerLink      ="/contact"    >  
Contactez-nous  
</a>
```

#### Lien Dynamique (Tableau)

```
<a [routerLink]   = "[ '/user', userId, 'details' ]"      >  
Profil Utilisateur  
</a>
```

# Paramètres de Route

## Définition (Route)

1

```
{ path : 'produit/:id', component : ProductComponent }
```

Utilisez le deux-points : pour déclarer un segment dynamique.



## Navigation (URL)

2

```
http://monsite.com/produit/42
```

L'utilisateur navigue vers une URL contenant la valeur.



## Récupération (Component)

3

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

Injectez ActivatedRoute pour lire le paramètre.

# Query Strings

http://monsite.com/produits

?page=2&sort=price

Paramètres Optionnels

## Navigation (Envoi)

Utilisez la directive queryParams pour ajouter des paramètres à l'URL.

```
<a routerLink="/produits"
[queryParams]="{ page: 2, sort: 'price' }">
    Page Suivante
</a>
```

 Idéal pour : Pagination, Filtres, Recherche.

## Récupération (Lecture)

Utilisez ActivatedRoute pour observer les changements.

```
constructor(privateRoute: ActivatedRoute) {}

ngOnInit() {
    this.route.queryParams.subscribe(params => {
        this.page = params['page'];
        this.sort = params['sort'];
    });
}
```

# Observer les Changements

## ⚠ Le Piège du Snapshot

Angular réutilise l'instance du composant si seule la partie paramètre de l'URL change (ex: de /user/1 à /user/2).

Dans ce cas, ngOnInit n'est pas réexécuté, et le snapshot ne se met pas à jour.

Réutilisation du Composant



## La Solution : Souscription (Observable)

```
ngOnInit () {  
  // Écouter le flux de changements  
  
  this .route .paramMap .subscribe (params => {  
  
    // Exécuté à chaque changement d'URL  
  
    this .userId = params .get ('id');  
    this .loadUserData (this .userId );  
  
  });  
}
```

i Note : Le Router gère automatiquement la désinscription de cet Observable spécifique lors de la destruction du composant.

# Routes Imbriquées

## Hiérarchie de Vues

Les routes peuvent avoir des **routes enfants**. Cela permet de créer des interfaces complexes où un composant parent affiche différents sous-composants.



### Important :

Le composant parent (ex: ProductComponent) doit impérativement contenir sa propre balise `<router-outlet>` pour afficher les enfants.

```
{ path: 'product/:id',
  component: ProductComponent,
  children: [
    { path: 'overview', component: OverviewComponent },
    { path: 'specs', component: SpecsComponent }
  ]
}
```

Structure Visuelle

ProductComponent (Parent)

Titre du Produit, Onglets...

<router-outlet> (Enfant)

# Redirections & Wildcard



## Route par Défaut

Redirige automatiquement l'utilisateur lorsqu'il arrive sur la racine du site (/).

**Important :** Toujours utiliser pathMatch: 'full' pour éviter les boucles infinies.

```
{ path: '',
  redirectTo: '/home',
  pathMatch: 'full' }
```



## Route Wildcard (404)

Intercepte toutes les URL non reconnues.

**Règle d'Or :** Doit être déclarée en dernier dans le tableau des routes (First Match Wins).

```
// À la fin du tableau routes[]
{ path: '**',
  component: PageNotFoundComponent }
```

# Navigation Programmée

## 1. Injection du Service

```
import { Router } from '@angular/router';  
  
constructor (private router : Router) {}
```

## 2. Utilisation dans une Méthode

```
onSave () {  
  // Logique métier (ex: sauvegarde)  
  this .dataService .save ().subscribe (() => {  
  
  // Redirection après succès  
  this .router .navigate(['/dashboard']);  
  
});  
}
```

### Action Utilisateur

Clic sur un bouton, soumission de formulaire...



### Logique TypeScript

Validation, appels API, calculs...



### Navigation

Changement de route via le Router



# Les Guards (Gardiens)

## Sécuriser la Navigation

Les Guards sont des interfaces qui permettent d'autoriser ou d'empêcher l'accès à une route en fonction de certaines conditions logiques.

-  Lock Authentification (Login requis)
-  Key Autorisation (Rôle Admin)
-  Save Modifications non sauvegardées



# Implémentation : CanActivate

```
@Injectable({ providedIn: 'root' })  
export class AuthGuard implements CanActivate {  
  
  constructor(  
    private auth: AuthService,  
    private router: Router  
  ) {}  
  
  canActivate(): boolean {  
    if (this.auth.isLoggedIn()) {  
      return true;  
    }  
  
    // Non connecté : Redirection  
    this.router.navigate(['/login']);  
    return false;  
  }  
}
```

- 1** **Injection des Services**  
On injecte AuthService pour vérifier l'état et Router pour rediriger si besoin.
  
  - 2** **Vérification**  
La méthode canActivate est appelée automatiquement par le routeur avant la navigation.
-   
**TRUE**  
Accès Autorisé

  
**FALSE**  
Accès Refusé + Redirection

# Appliquer un Guard

## Configuration de la Route

Pour activer la protection, ajoutez la propriété canActivate à la définition de la route dans votre fichier de routage.

### Format Tableau

path: 'login', component: LoginComponent,

### Validation Totale

pour laquelle il faut retourner true

```
const routes: Routes = [
  { path: 'login', component: LoginComponent },

  {
    path: 'dashboard',
    component: DashboardComponent,
    canActivate: [AuthGuard]
  }
];
```

 Route Protégée

# Types de Guards



## CanActivate

Vérifie si l'utilisateur peut accéder à une route spécifique.

Avant l'entrée



## CanActivateChild

Vérifie l'accès pour toutes les routes enfants d'une route parente.

Avant l'entrée (Enfants)



## CanDeactivate

Vérifie si l'utilisateur peut quitter la route actuelle (ex: formulaire non sauvegardé).

Avant la sortie



## CanMatch

Décide si une route peut être chargée/utilisée. Remplace souvent CanLoad.

Avant le chargement



## Resolve

Récupère des données obligatoires avant d'activer la route. Assure que le composant a ses données prêtes.

Pendant la navigation

# Lazy Loading

## Chargement à la Demande

Le Lazy Loading permet de ne charger le code d'un module que lorsque l'utilisateur navigue vers la route associée.

-  Démarrage plus rapide
-  Bundle initial plus léger
-  Économie de bande passante

```
// app-routing.module.ts
{
  path: 'admin',
  loadChildren: () => import('./admin/admin.module')
  .then(m => m.AdminModule)
}
```

Architecture des Fichiers (Bundles)



Les "Chunks" sont téléchargés séparément via le réseau.

# routerLinkActive

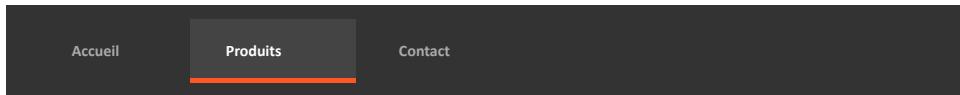
## Style de Navigation

Cette directive ajoute automatiquement une classe CSS à l'élément HTML lorsque la route associée est active.

### ! Option Exacte

Pour la route racine (/), utilisez {exact: true}. Sinon, elle sera considérée comme active pour **toutes** les pages (car toutes commencent par /).

Résultat Visuel :



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

<a routerLink="/products" routerLinkActive="active-link">
    Produits
</a>
</nav>
```

# Fragments d'URL

## Configuration Requise



Pour activer le défilement automatique vers les ancrées (comme en HTML standard), vous devez configurer le routeur.

```
RouterModule.forRoot(routes, { anchorScrolling: 'enabled' })
```

## </> Dans le Template

```
<!-- Lien vers l'ancre -->
<a routerLink="/page" fragment="contact">
  Aller au Contact
</a>

<!-- Cible -->
<div id="contact">
  Formulaire...
</div>
```

## 🔗 Via le Code (TypeScript)

```
scrollToSection() {
  this.router.navigate(['/page'], {
    fragment: 'contact'
  });

  // URL générée :
  // /page#contact
}
```

# Navigation Relative

## Contexte Courant

Permet de naviguer par rapport à la route actuellement active, plutôt que depuis la racine de l'application.

### Analogie Système de Fichiers

```
/products
  ├── /list
  └── /details (Vous êtes ici)
      └── /edit

      ..../ = Remonter vers /products
      ./edit = Aller vers /products/edit
```

### 1. Injection Nécessaire

```
constructor ( router : Router , route : ActivatedRoute ) {}
```

### 2. Navigation Relative

```
goBack () {
  // Remonter d'un niveau
  this .router .navigate (['..'], {
    relativeTo : this .route
  });
}
```

# Atelier 7 : Mise en place du Routage



## Objectif

Créer une application de navigation simple avec 3 pages distinctes et un menu de navigation.

- ✓ Page Accueil
- ✓ Page À Propos
- ✓ Page Contact

1

### Générer les Composants

```
ng g c home  
ng g c about  
ng g c contact
```

2

### Configurer les Routes (app-routing.module.ts)

```
const routes = [  
  { path: "", component: HomeComponent },  
  { path: 'about', component: AboutComponent },  
  { path: 'contact', component: ContactComponent }  
];
```

3

### Mettre à jour le Template (app.component.html)

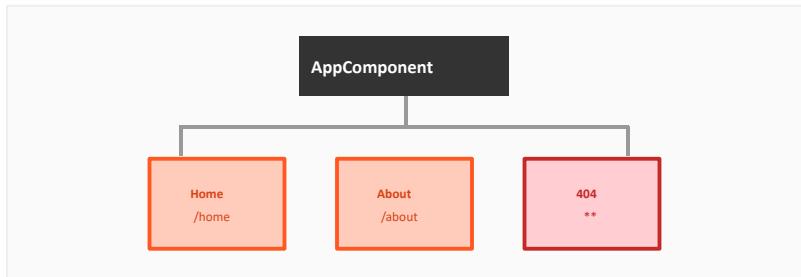
```
<nav>  
  <a routerLink="/">Accueil</a>  
  <a routerLink="/about">À Propos</a>  
</nav>  
<router-outlet></router-outlet>
```

# Atelier 8 : Application Multi-Vues

## Objectifs

- 1 Créer 4 composants : **Home**, **About**, **Contact**, **NotFound**.
- 2 Configurer les routes dans `app-routing.module.ts`.
- 3 Ajouter une redirection de " " vers '/home'.
- 4 Gérer les erreurs 404 avec une route Wildcard \*\*.
- 5 Créer une barre de navigation avec `routerLink` et `routerLinkActive`.

## Structure du Site



### Rappel :

```
<nav>
<a routerLink="/home">Accueil</a>
</nav>
<router-outlet></router-outlet>
```

# Gestion des Formulaires

Les formulaires sont essentiels pour interagir avec l'utilisateur : connexion, création de profil, saisie de données. Angular propose deux approches distinctes pour gérer la saisie, la validation et l'état des champs.

vs



## Template Driven

Approche basée sur le template HTML. Simple à mettre en place, idéale pour les formulaires simples. Utilise massivement [(ngModel)].

FormsModule



## Reactive Forms

Approche basée sur le modèle (TypeScript). Plus robuste, scalable et testable. Gestion explicite des flux de données via FormControl.

ReactiveFormsModule

# Template Driven Forms

## Approche "Template First"

La logique du formulaire est principalement définie dans le template HTML. C'est l'approche la plus simple pour démarrer.

- ⌚ Binding Bidirectionnel `([(ngModel)])`
- ⌚ Peu de code TypeScript
- ✓ Idéal pour formulaires simples

Prérequis (AppModule)

imports: [ `FormsModule` ]

```
<form#f="ngForm"(ngSubmit)="onSubmit(f)">  
  
  <inputtype="text"  
    name="username"  
    [(ngModel)]="user.name"  
    required>  
  
  <button>Envoyer</button>  
  </form>
```

 L'attribut `name` est **obligatoire** pour que `ngModel` fonctionne dans un formulaire.

# Reactive Forms

## Prérequis (AppModule)

```
imports: [ ReactiveFormsModule ]
```

## TypeScript (Component)

```
export class LoginComponent {
  email = new FormControl("");
  password = new FormControl("");
}
```

## HTML (Template)

```
<input type="email" [formControl] = "email" >
<input type="password" [formControl] = "password" >
```

## Approche "Code First"

Le modèle du formulaire est défini explicitement dans la classe du composant. Le template HTML ne fait que se lier à ce modèle existant.

 Flux de données synchrone

 Facilement testable (Unit Tests)

 Idéal pour formulaires complexes

 Supporte les Observables

# Comparaison des Approches

## Template Driven

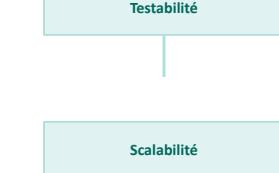
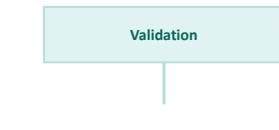
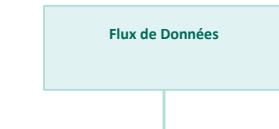
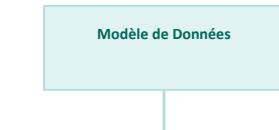
Non structuré (Mutable)

Asynchrone (UI vers Modèle)

Directives HTML

Difficile (Nécessite TestBed)

Faible (Formulaires simples)



## Reactive Forms

Structuré & Immutable

Synchrone (Contrôle total)

Fonctions / Validateurs

Facile (Pas de DOM requis)

Élevée (Formulaires complexes)

# Validateurs Standards



## required

Le champ doit contenir une valeur non vide.



## min/max length

Longueur minimale ou maximale de la chaîne de caractères.



## min/max

Valeur numérique minimale ou maximale autorisée.



## pattern

Doit correspondre à une expression régulière (Regex).



## email

Vérifie si la valeur respecte le format d'une adresse email valide.

### </> Template Driven (Attributs HTML)

```
<input type="text"  
      name="email"  
      [(ngModel)]="user.email"  
      required  
      email>
```

### ⚙️ Reactive Forms (Classe Validators)

```
this.emailCtrl = new FormControl('', [  
  Validators.required,  
  Validators.email,  
  Validators.minLength(5)  
]);
```

# États du Formulaire



## Visite (Focus/Blur)

Indique si l'utilisateur est entré puis sorti du champ.

touched

untouched



## Modification

Indique si la valeur a été modifiée par l'utilisateur.

dirty

pristine



## Validité

Indique si la valeur respecte toutes les règles de validation.

valid

invalid

## Pattern UX Recommandé

N'affichez les erreurs que si le champ est invalide ET que l'utilisateur a interagi avec (touched ou dirty).

```
<div*ngIf="name.invalid && (name.dirty || name.touched)">
```

```
<div*ngIf="name.errors?.['required']">
```

Ce champ est requis.

```
</div>
```

```
</div>
```

# Afficher les Erreurs

## Le Bon Moment

Il ne faut pas afficher les erreurs dès le chargement de la page. Attendez que l'utilisateur ait interagi avec le champ.

Email

✖ L'email est invalide.

## ⚠ La Condition Idéale

### INVALID && (DIRTY || TOUCHED)

"Le champ est invalide ET (l'utilisateur a tapé quelque chose OU a quitté le champ)"

```
<input name="email" ngModel required email  
#emailRef = "ngModel" >  
  
<div *ngIf="emailRef.invalid && (emailRef.dirty || emailRef.touched)"  
class="error-msg" >  
  
<div *ngIf="emailRef.errors?.['required']" >  
L'email est requis.  
</div >  
  
<div *ngIf="emailRef.errors?.['email']" >  
Format invalide.  
</div >
```

# Soumettre le Formulaire

## Template HTML

```
<form(ngSubmit)="onSubmit(f)"#f="ngForm">  
  
  <!-- Champs du formulaire -->  
  
  <button type="submit"  
    [disabled]="f.invalid">  
    Envoyer  
  </button>  
  
</form>
```

## Component TypeScript

```
onSubmit(form: NgForm) {  
  
  if (form.invalid) {  
    return; // Sécurité supplémentaire  
  }  
  
  console.log('Données :', form.value);  
  // Appel au service API ici...  
}
```



Bonne Pratique : Utilisez toujours l'événement (ngSubmit) sur la balise <form> plutôt que (click) sur le bouton. Cela permet de gérer la soumission via la touche "Entrée" du clavier

# Validateur Personnalisé

## Définition de la Fonction

```
export function forbiddenNameValidator (control : AbstractControl): ValidationErrors | null {  
  
  const forbidden = '/admin/i'.test(control.value);  
  
  return forbidden  
    ? { forbiddenName : { value : control.value } }  
    : null;  
}
```

## Utilisation dans le Composant

```
this.form = new FormGroup ({  
  username : new FormControl ('', [  
    Validators.required,  
    forbiddenNameValidator // Pas de parenthèses !  
  ])  
});
```



## Signature Simple

Un validateur est une simple fonction qui prend un AbstractControl en entrée.



## Valeur de Retour

La fonction doit retourner un objet si l'erreur existe, ou null si tout est valide.

Valide :	null
Invalide :	{ key: value }



## Intégration

Ajoutez simplement la référence de la fonction dans le tableau des validateurs du FormControl.

# Validateurs Asynchrones



## Saisie Utilisateur

L'utilisateur tape son email.



## État PENDING

Le formulaire attend la réponse.



## Appel API

Vérification en base de données.



## Résultat

Retourne null ou { error: true }.



Les validateurs asynchrones sont passés en 3ème argument du FormControl.

```
functionuniqueEmailValidator(api: ApiService): AsyncValidatorFn {
  return (control: AbstractControl) => {
    returnapi.checkEmail(control.value).pipe(
      map(exists => exists ? { emailTaken: true } : null),
      catchError(() => of(null))
    );
  };
}

// Utilisation
this.email = newFormControl("", [Validators.required], // Sync
[uniqueEmailValidator(this.api)] // Async
);
```

# FormArray : Champs Dynamiques

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

Une alternative au FormGroup pour gérer une collection de contrôles dont le nombre n'est pas connu à l'avance.

## Cas d'usage fréquents :

 Liste de tags ou mots-clés

 Numéros de téléphone multiples

 Lignes d'un panier d'achat

```
TypeScript (Initialisation & Ajout)  
  
this .form = new FormGroup ({  
hobbies : new FormArray ([])  
});  
  
get hobbies () {  
return this .form .get ('hobbies' ) as FormArray ;  
}  
  
addHobby () {  
this .hobbies .push (new FormControl (''));  
}
```

```
HTML (Itération)  
  
< div formArrayName = "hobbies" >  
< div *ngFor = "let hobby of hobbies.controls; index as i" >  
< input [formControlName] = "i" >  
< button (click) = "removeHobby(i)" >X</ button >  
</ div >  
</ div >
```

# FormGroup Imbriqué

Formulaire Principal (userForm)

Groupe: adresse

TypeScript

```
this .userForm = new FormGroup {{  
    nom : new FormControl (""),  
    email : new FormControl (""),  
    adresse : new FormGroup {{  
        rue : new FormControl (""),  
        ville : new FormControl (""),  
        codePostal : new FormControl ("")  
    }}  
}};
```

HTML

```
<form [formGroup] ="userForm" >  
<input formControlName ="nom" >  
  
<div formGroupName="adresse" >  
<h3 >Adresse</ h3 >  
<input formControlName ="rue" >  
<input formControlName ="ville" >  
</div >
```

# Réinitialiser le Formulaire

## Nettoyage Complet

La méthode `reset()` ne se contente pas d'effacer les valeurs.

Elle restaure également l'état "vierge" du formulaire.

### Impact sur l'État

Value	"texte"	→	null
Dirty	true	→	false
Touched	true	→	false
Valid	...	→	recalculé

### Réinitialisation Simple (Tout à null)

```
// Reactive Forms  
this.myForm.reset();  
  
// Template Driven (via ViewChild ou référence)  
form.resetForm();
```

### Réinitialiser avec Valeurs par Défaut

```
this.myForm.reset({  
  firstName: 'John',  
  lastName: 'Doe',  
  email: ''  
});  
// Le formulaire est maintenant "Pristine" avec ces valeurs.
```

# Contrôler l'État



**enable()**Active le champ (éditable).



**disable()**Désactive le champ (lecture seule).



## Attention au Piège !

Un champ désactivé est exclu de l'objet form.value. Il disparaît complètement du résultat JSON.

Solution : Utilisez `form.getRawValue()` pour tout récupérer.

TypeScript (Basculer l'état)

```
toggleEditMode () {
  if( this .emailCtrl .disabled  ){
    this .emailCtrl .enable  ();
  } else {
    this .emailCtrl .disable  ();
  }
}
```

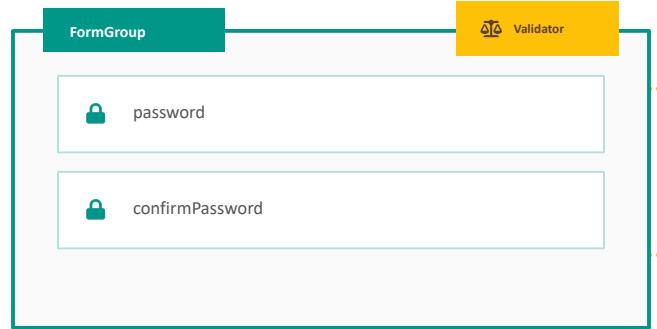
Récupérer les données

```
// Si 'email' est désactivé :

console .log (this .form .value );
// -> { nom:'Dupont' } (email manquant)

console .log (this .form .getRawValue ());
// -> { nom:'Dupont', email:'test@test.com' }
```

# Validation Croisée



## Concept Clé :

Le validateur doit être attaché au *parent* (le Groupe) pour avoir accès aux deux enfants simultanément.

### 1. La Fonction de Validation

```
const matchValidator : ValidatorFn = ( g: AbstractControl ) => {
  const pass = g.get('password')?.value;
  const confirm = g.get('confirmPassword')?.value;

  return pass === confirm ? null : { mismatch : true };
};
```

### 2. Configuration du FormGroup

```
this .form = new FormGroup ({{
  password : new FormControl (''),
  confirmPassword : new FormControl ('')
}, { validators: matchValidator }});
```

### 3. Affichage HTML

```
<div *ngIf ="form.errors?['mismatch']" >
  Les mots de passe ne correspondent pas.
</div >
```

# Accessibilité (a11y)



## Labels Explicites

Associez toujours un `<label>` à son champ via l'attribut `for` et l'`id`.



## Lier les Erreurs

Utilisez `aria-describedby` pour lier le message d'erreur au champ de saisie.



## Indiquer l'État

Utilisez `aria-invalid="true"` pour signaler aux lecteurs d'écran qu'un champ est en erreur.



## Annonces Dynamiques

Utilisez `aria-live="assertive"` pour que les erreurs soient lues dès leur apparition.



Angular permet de lier dynamiquement les attributs ARIA.

```
<label for="emailId">Email</label>

<input type="email"
       id="emailId"
       [FormControl]="emailCtrl"
       [attr.aria-invalid]="emailCtrl.invalid"
       aria-describedby="emailError">

<div id="emailError" role="alert">
  <span *ngIf="emailCtrl.invalid && emailCtrl.touched">
    L'email est requis.
  </span>
</div>
```

# Patterns Courants



## Sous-Formulaires

Découper un formulaire géant en plusieurs composants réutilisables (ex: `AdresseComponent`, `IdentiteComponent`).

```
viewProviders: [[ provide: ControlContainer, ... ]]
```



## Formulaires Dynamiques

Générer le formulaire (HTML et Contrôles) entièrement à partir d'une configuration JSON reçue du serveur.

```
Metadata -> FormGroup + *ngFor
```



## Custom Controls (CVA)

Créer vos propres composants UI (`StarRating`, `DatePicker`) qui s'intègrent nativement avec `ngModel` et `formControl`.

```
implements ControlValueAccessor
```

Interface `ControlValueAccessor` (Le Saint Graal des composants de formulaire)

```
interface ControlValueAccessor {  
  writeValue(obj: any): void; // Modèle -> Vue  
  registerOnChange(fn: any): void; // Vue -> Modèle  
  registerOnTouched(fn: any): void; // Gestion du blur  
}
```

# Formulaire Multi-Étapes



## Stratégie "Single Parent"

Créez un seul FormGroup parent qui contient tout. Cela permet de conserver l'état global même si l'utilisateur navigue entre les étapes.

## Validation Progressive

Vérifiez la validité du sous-groupe actuel (ex: form.get('step1').valid) avant d'autoriser le passage à l'étape suivante.

Précédent

Suivant

### TypeScript Structure

```
this.mainForm = new FormGroup ({  
  account : new FormGroup ({ ... }), // Étape 1  
  address : new FormGroup ({ ... }), // Étape 2  
  payment : new FormGroup ({ ... }) // Étape 3  
});  
  
currentStep = 1;
```

### HTML View Switching

```
<form [formGroup] = "mainForm" >  
  
<div *ngIf = "currentStep === 1" formGroupName = "account" >  
  <!-- Champs Compte -->
```

# Erreurs Serveur



Envoi du Formulaire



Erreur 422 (Unprocessable)

{ email: "Déjà pris" }



Affichage dans l'UI

## Stratégie :

Le serveur est la source de vérité. Si la validation client passe mais que le serveur refuse, il faut réinjecter l'erreur dans le formulaire.

## Gestion de l'erreur API

```
onSubmit () {
  this.authService.register(this.form.value).subscribe ({
    next: () => this.router.navigate(['/home']),
    error: (err) => {
      // Si l'email est déjà pris
      if (err.status === 422 && err.error.email) {
        this.emailCtrl.setErrors({
          serverError: err.error.email
        });
      }
    }
  });
}
```

## Affichage HTML

```
<div *ngIf="emailCtrl.errors['serverError']" >
  {{emailCtrl.errors['serverError']}}
</div>
```

# Atelier 9 : Formulaire de Contact

## Objectif

Créer un formulaire de contact simple utilisant l'approche **Template Driven**.

## Cahier des Charges :

- Champ Nom** : Requis.
- Champ Email** : Requis et format email valide.
- Champ Message** : Requis, min 10 caractères.
- Bouton** : Désactivé si le formulaire est invalide.
- Soumission** : Afficher l'objet complet dans la console.

Maquette

Nom \*

Email \*

Message (min 10) \*

ENVOYER

## Indices

N'oubliez pas d'importer `FormsModule` .  
Utilisez `[(ngModel)]` pour le binding.  
Ajoutez `name="..."` à chaque input.  
Utilisez `#f="ngForm"` pour accéder à l'état global.

# Atelier 10 : Formulaire de Contact

## Objectifs

Créez un composant ContactComponent utilisant les **Reactive Forms**.

- ✓ **Champs** : Nom, Email, Sujet, Message.
- ✓ **Validation** : Tous requis. Email valide. Message > 10 caractères.
- ✓ **UX** : Afficher les erreurs uniquement si le champ est "touched".
- ✓ **Submit** : Bouton désactivé si invalide. Log en console au clic.

## Bonus Challenge ⭐

Ajoutez une checkbox "S'abonner à la newsletter". Si cochée, un champ "Fréquence" (Hebdo/Mensuel) apparaît et devient requis.

Résultat Attendu...

Email \*

⚠ Email invalide

Sujet \*

Message \*  
  
0/10

ENVOYER

# Les Services

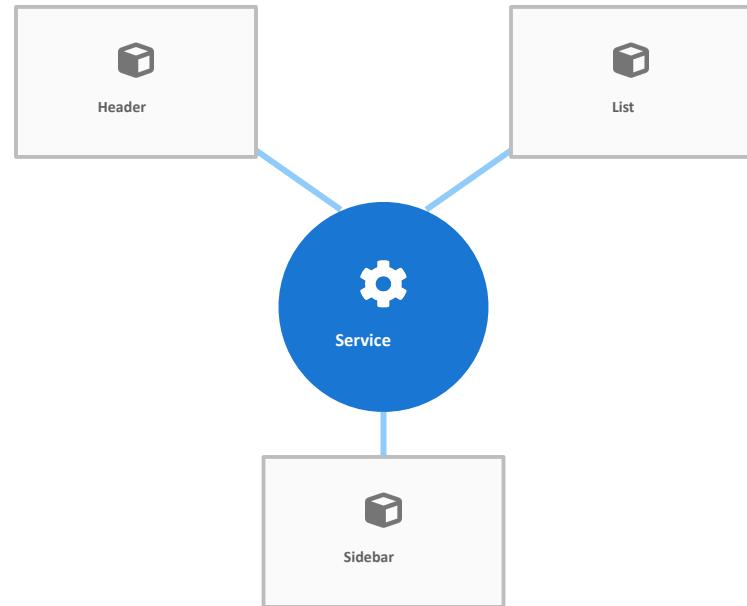
## Définition

Une classe TypeScript dont le but est d'organiser et de partager du code métier, des données ou des fonctions utilitaires.

 Logique Métier (Business Logic)

 Accès aux Données (API Calls)

 Partage d'État entre Composants



# Créer un Service

```
→ng generate service services/user
```

```
CREATE src/app/services/user.service.spec.ts (360 bytes)
```

```
CREATE src/app/services/user.service.ts (135 bytes)
```

 [user.service.ts \(Le code\)](#)

 [user.service.spec.ts \(Les tests\)](#)

user.service.ts

```
import { Injectable } from '@angular/core' ;  
  
@Injectable({  
  providedIn: 'root'  
})  
export class UserService {  
  
  constructor() {}  
  
  getUsers() {  
    return [ 'Alice' , 'Bob' ];  
  }  
}
```

## `@Injectable({ providedIn: 'root' })`

Ce décorateur est **obligatoire**. La propriété `providedIn: 'root'` indique à Angular de créer une instance unique (Singleton) disponible dans toute l'application.

# Injecter un Service



## Interdit

Ne faites jamais new DataService() vous-même. Cela briserait le système d'injection et le partage d'état.



## Constructor Injection

Déclarez simplement le service comme argument du constructeur. Angular détecte le type et fournit l'instance.

*Le mot-clé private crée automatiquement une propriété de classe accessible via this.*

user.component.ts

```
import { Component, OnInit } from '@angular/core';
import { DataService } from './data.service';

@Component({ ... })
export class UserComponent implements OnInit {

  // Injection via le constructeur
  constructor (private dataService : DataService) {}

  ngOnInit () {
    // Utilisation immédiate
    const users = this.dataService.getUsers();
    console.log(users);
  }
}
```

# Injection de Dépendances (DI)

## Le Concept

Un **Design Pattern** où les objets reçoivent leurs dépendances d'une source externe plutôt que de les créer eux-mêmes.

*"Ne m'appelez pas, je vous appellerai."*

Le Principe Hollywood (Inversion de Contrôle)

- ✓ Permet de changer facilement d'implémentation.
- ✓ Facilite énormément les tests unitaires (Mocking).

✗ Sans DI (Couplage Fort)

```
classCar {  
    engine;  
    constructor() {  
        // La classe crée sa propre dépendance  
        this.engine = newEngine(); // ⚡ Rigide  
    }  
}
```

✓ Avec DI (Couplage Faible)

```
classCar {  
    engine;  
    // La classe demande une dépendance  
    constructor(engine: Engine) {  
        this.engine = engine; // ✅ Flexible  
    }  
}
```

# Pourquoi la DI ?



## Couplage Faible

Les composants ne dépendent pas d'une implémentation spécifique (ex: new Service()), mais d'une **abstraction** ou d'un jeton. Cela rend le code plus flexible et modulaire.



## Testabilité

C'est l'avantage majeur. Lors des tests unitaires, on peut facilement injecter des **Mocks** ou des **Spies** à la place des vrais services (ex: éviter les appels HTTP réels).



## Réutilisabilité

Les services sont conçus pour être **agnostiques** de la vue. Un même service 'AuthService' peut être utilisé par le Login, le Header, et les Guards sans duplication de code.



## Maintenabilité

La configuration des dépendances est centralisée (dans les providers). Changer une implémentation pour toute l'application se fait en **une seule ligne** de code.

# Enregistrer des Providers



## Niveau Root (Global)

Le service est un **Singleton** partagé par toute l'application. C'est la méthode recommandée par défaut.

Recommandé



## Niveau Composant

Une **nouvelle instance** du service est créée pour chaque instance du composant. Le service est détruit avec le composant.

Isolation

### Syntaxe Moderne (Tree-shakable)

```
@Injectable ({  
  providedIn : 'root'  
})  
export class UserService { ... }
```

### Syntaxe Composant (Scoped)

```
@Component ({  
  selector : 'app-user-list' ,  
  templateUrl : './user-list.component.html' ,  
  providers : [ UserService ] // Instance locale !  
})  
export class UserListComponent { ... }
```

# Le Pattern Singleton

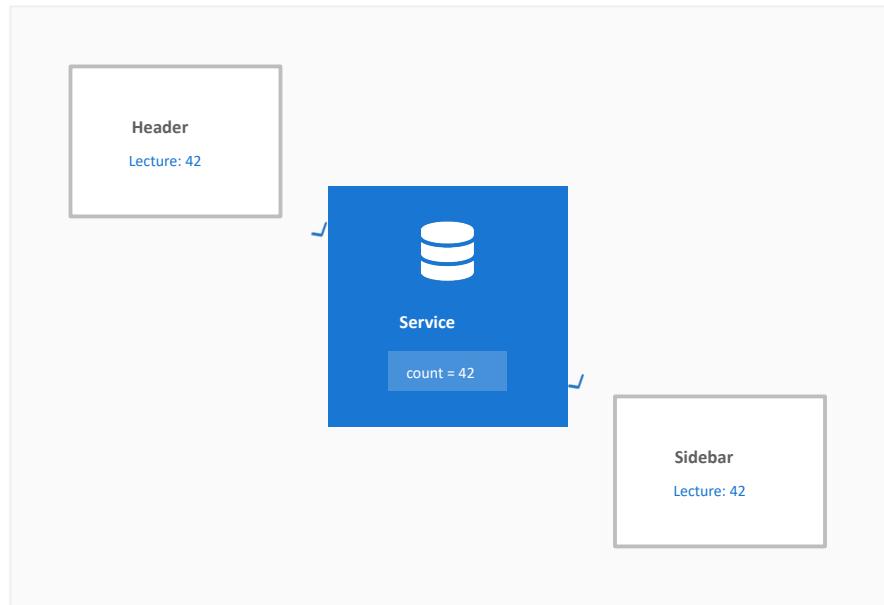
## Instance Unique

Par défaut (avec providedIn: 'root'), Angular crée une **seule et unique instance** de votre service pour toute l'application.

## Source de Vérité

Cela transforme le service en un lieu idéal pour stocker et partager l'état de l'application.

-  Profil Utilisateur connecté
-  Contenu du Panier
-  Préférences de Langue



# Services avec État



## Composants (Éphémères)

Détruits à chaque changement de route. Les données locales sont perdues.



## Services (Persistants)

Restent en mémoire tant que l'application tourne. Idéal pour stocker des données.

### Cas d'Usage Typiques :

Panier d'achat

Profil Utilisateur

Thème (Dark/Light)

Notifications

cart.service.ts

```
export class CartService {  
  
    // État privé (Encapsulation)  
    private items : Product[] = [];  
  
    // Méthode pour modifier l'état  
    addToCart (product : Product) {  
        this.items.push(product);  
    }  
  
    // Méthode pour lire l'état  
    getItems () {  
        return this.items;  
    }  
  
    clearCart () {  
        this.items = [];  
    }  
}
```

# Services & Observables (RxJS)

## Le Pattern "Store"

Pour gérer un état partagé (ex: panier, utilisateur connecté), on utilise un `BehaviorSubject` encapsulé dans le service.



### Privé : `BehaviorSubject`

La source de vérité. On peut écrire dedans (`next()`).



### Public : `Observable`

Le flux exposé. Les composants s'y abonnent (`subscribe()`).

cart.service.ts

```
export class CartService {  
  
    // 1. Source privée (Initialisée à 0)  
    private _count = new BehaviorSubject <number>(0);  
  
    // 2. Flux public (Lecture seule)  
    public count$ = this._count.asObservable();  
  
    // 3. Méthode pour modifier l'état  
    addToCart () {  
        const current = this._count.value;  
        this._count.next(current + 1);  
    }  
}
```

header.component.ts

```
ngOnInit () {  
    this.cartService.count$.subscribe (n => {  
        console.log(`Cart count: ${n}`);  
    });  
}
```

# Injection Avancée : InjectionToken

## ! Le Problème

Les **Interfaces** TypeScript disparaissent à la compilation. On ne peut pas les utiliser comme "clé" pour l'injection. De plus, injecter une simple string est ambigu.

## 🔑 La Solution

**InjectionToken** crée un jeton unique (une clé) qui permet d'injecter des valeurs primitives, des objets de configuration ou des interfaces.

### 1 Créer le Token

```
export const API_URL = new InjectionToken<string>('api.url');
```

### 2 Fournir la Valeur

```
providers: [
  { provide: API_URL, useValue: 'https://api.monsite.com' }
]
```

### 3 Injecter (avec Décorateur)

```
constructor(@Inject(API_URL) private url: string) {
  console.log(this.url);
}
```

# Factory Providers



Ingrédients (deps)



Factory Function



Service Instance

## Cas d'Usage

Utilisez `useFactory` lorsque la création du service dépend d'une condition dynamique (ex: est-ce que l'utilisateur est admin ?) ou d'une configuration chargée au runtime.

### 1. La Fonction Factory

```
const loggerFactory = ( user : UserService ) => {
  return user.isAdmin ? new AdminLogger() : new SimpleLogger();
};
```

### 2. Le Provider

```
@NgModule({
  providers: [
    {
      provide: LoggerService,
      useFactory: loggerFactory,
      deps: [UserService]
    }
  ]
})
```

# Value Providers

## Injection de Valeurs

Parfois, vous n'avez pas besoin d'instancier une classe. Vous voulez simplement injecter un objet de configuration, une chaîne de caractères ou une constante.

`useValue` injecte la valeur telle quelle, sans aucune transformation.



Configuration Globale (AppConfig)



URL de l'API (Base URL)



Feature Flags (Toggle)

### 1. L'Objet de Configuration

```
const MY_CONFIG = {  
  apiUrl : 'https://api.example.com' ,  
  theme : 'dark' ,  
  retryCount : 3  
};
```

### 2. Le Provider (useValue)

```
providers : [  
  {  
    provide : APP_CONFIG_TOKEN ,  
    useValue: MY_CONFIG  
  }  
]
```

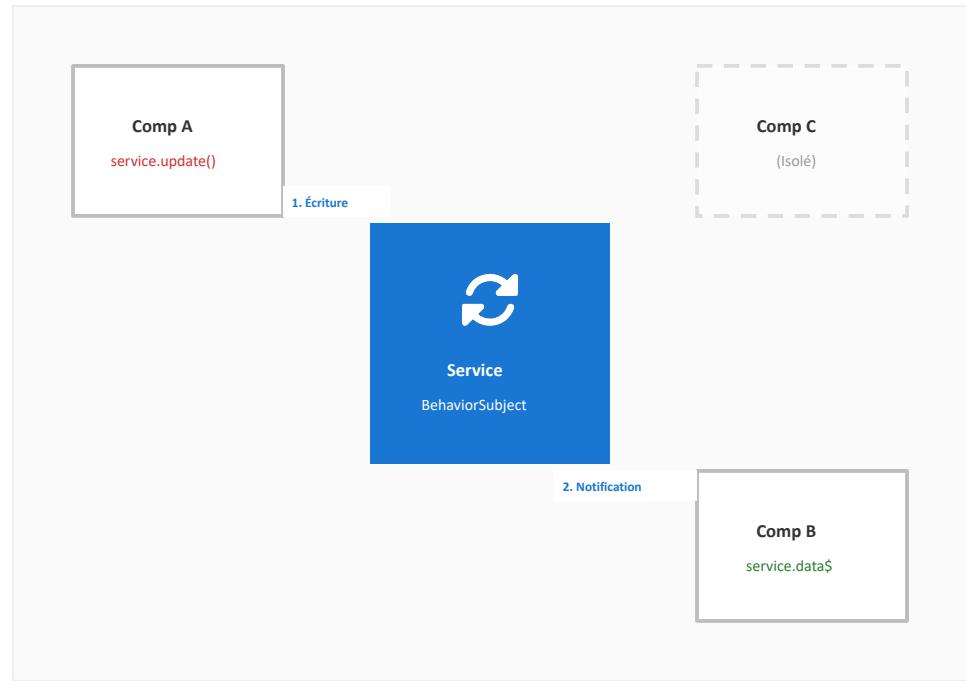
# Partage d'État

## Le Problème

Passer des données via `@Input()` à travers 5 niveaux de composants ("Prop Drilling") rend le code rigide et difficile à maintenir.

## La Solution

Le Service agit comme un **Bus de Données**. N'importe quel composant peut s'y connecter directement pour lire ou écrire l'état, sans passer par ses parents.



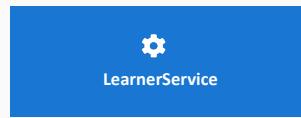
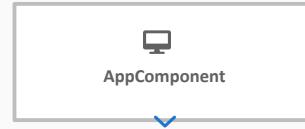
# Atelier 11 : Votre Premier Service

## Objectifs

Externaliser la gestion des données dans un service dédié.

- ✓ **Création :** Générer LearnerService.
- ✓ **Données :** Un tableau de strings ['Alice', 'Bob'].
- ✓ **Méthodes :** Implémenter getAll() et add(name: string).
- ✓ **Utilisation :** Injecter dans AppComponent et afficher la liste.

```
→ ng generate service services/learner
```



*Le composant "consomme" le service*

## Aide-mémoire

1. constructor(private learnerService: LearnerService) {}
2. ngOnInit() { this.list = this.learnerService.getAll(); }
3. N'oubliez pas d'importer le service !

# Atelier 12 : Architecture & Refactoring

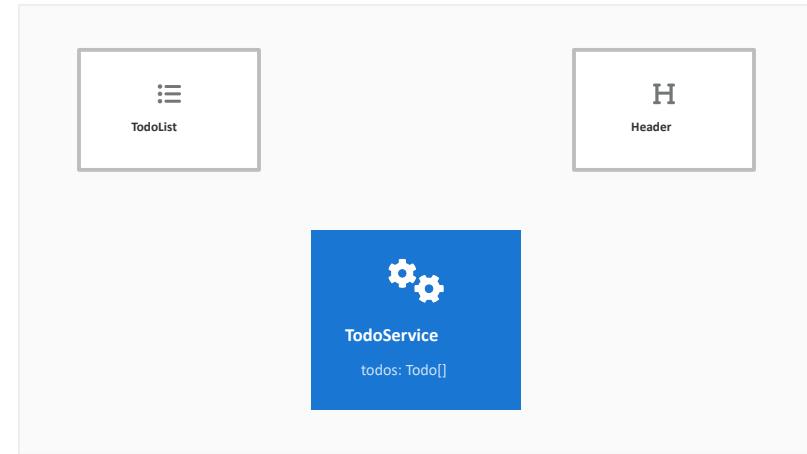
## Objectifs

Migrer la logique de la TodoList vers une architecture orientée Services.

- 1 Générer TodoService.
- 2 Déplacer le tableau todos et les méthodes (add, delete) dans le service.
- 3 Injecter le service dans TodoListComponent.
- 4 Remplacer la manipulation directe par des appels au service.

### Bonus : État Partagé 🌟

Injectez aussi le service dans HeaderComponent pour afficher le nombre de tâches en cours (ex: "3 tâches restantes").



```
todo.service.ts (Squelette)

export class TodoService {
  private todos : Todo[] = [];

  getTodos() { return this.todos; }
  addTodo(t: Todo) { this.todos.push(t); }
}
```

# Le Client HTTP

Angular fournit un module complet, HttpClient, pour communiquer avec des serveurs distants via le protocole HTTP.



Basé sur les Observables (RxJS)



Réponses et Erreurs Typées



Intercepteurs de Requêtes



Facile à Tester (Mocking)

Fonctionnalité	Fetch API	HttpClient
Paradigme	Promise	Observable
Annulation	Complexe (AbortController)	Simple (unsubscribe)
Progression	Non supporté	Supporté (Upload)
JSON Parsing	Manuel (.json())	Automatique

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

# Configuration HTTP

## Activation Globale

Avant de pouvoir injecter HttpClient dans vos services, vous devez configurer le provider au niveau racine de l'application.

Recommandé (Standalone)

Utilisez la fonction provideHttpClient() dans la configuration de l'application.

Legacy (Modules)

Importez HttpClientModule dans votre AppModule.

app.config.ts

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

export const appConfig: ApplicationConfig = {
  providers: [
    provideRouter(routes),
    provideHttpClient() // <-- Ici
  ]
};
```

MODERNE

app.module.ts

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

@NgModule({
  imports: [
    BrowserModule,
    HttpClientModule
  ]
})
```

LEGACY

# Requête GET

## Récupérer des Données

La méthode `get()` effectue une requête HTTP GET standard. Elle attend une réponse JSON par défaut et tente de la parser automatiquement.

## Typage Fort (Generics)

Utilisez les génériques `<Type>` pour indiquer à TypeScript la forme des données attendues. Cela active l'autocomplétion et la vérification d'erreurs.



### Attention : Observable "Froid"

La requête n'est envoyée que lorsque vous appelez `.subscribe()`. Si personne n'écoute, rien ne se passe !

user.service.ts

```
getUsers (): Observable<User[]> {
  // Typage du retour de l'API
  return this.http.get<User[]>('https://api.com/users');
}
```

user-list.component.ts

```
ngOnInit () {
  this.userService.getUsers().subscribe(users => {
    // 'users' est typé comme User[]
    this.userList = users;
  });
}
```

# Requête POST

## Envoyer des Données

La méthode POST est utilisée pour créer une nouvelle ressource sur le serveur. Elle nécessite un **corps (body)** contenant les données à envoyer.

### Signature

```
post<T>(url, body, options?)
```

ⓘ Le body est automatiquement sérialisé en JSON.

### todo.service.ts

```
createTodo (todo : Todo ): Observable <Todo > {
  return this .http .post <Todo >(
    'https://api.example.com/todos' ,
    todo // Le payload
  );
}
```

### add-todo.component.ts

```
save () {
  const newTodo = { title : 'Apprendre Angular' };
  this .todoService .createTodo (newTodo ).subscribe (res => {
    console .log ('Créé avec succès ! ID:' , res .id );
  });
}
```

# Requête PUT

## Mise à Jour Complète

La méthode **PUT** est utilisée pour remplacer intégralement une ressource existante par une nouvelle version.

Contrairement à POST, PUT est **idempotent** : envoyer la même requête plusieurs fois aura toujours le même effet (la ressource aura l'état final spécifié).

PUT /api/users/ :id

user.service.ts

```
updateUser (id : number , user : User ): Observable <User> {  
  const url = `https://api.example.com/users/${id}`;  
  
  // 2ème argument : le corps (body) de la requête  
  return this .http .put <User>(url , user );  
}
```

user-edit.component.ts

```
saveChanges () {  
  this .userService .updateUser (1, this .form .value )  
  .subscribe (updatedUser => {  
    console .log ('Mise à jour réussie !' , updatedUser );  
  });  
}
```

# Requête DELETE

## Suppression

La méthode `delete()` permet de supprimer une ressource sur le serveur. Elle prend généralement l'identifiant unique de la ressource dans l'URL.

### Attention à l'UI

Une fois la suppression confirmée par le serveur, n'oubliez pas de mettre à jour votre interface (ex: retirer l'élément du tableau local) pour refléter le changement.

`data.service.ts`

```
deleteUser (id: number ): Observable <void> {
  return this.http.delete <void>(`https://api.com/users/${id}`);
}
```

`user-list.component.ts`

```
onDelete (id: number ) {
  this.dataService.deleteUser (id).subscribe (() => {
    // Mise à jour de l'interface locale
    this.users = this.users.filter (u=> u.id !== id);
  });
}
```

# Gestion des Erreurs

## Ne jamais ignorer l'échec

Les requêtes HTTP peuvent échouer pour de nombreuses raisons. Angular encapsule ces échecs dans un objet `HttpErrorResponse`.



### Erreur Client / Réseau

Pas d'internet, DNS introuvable, erreur JS.



### Erreur Serveur (Backend)

Status 404 (Not Found), 500 (Internal Error), 403 (Forbidden).

```
user.component.ts

this .http .get ('/api/users'    ).subscribe  (({
next : ( data  ) => {
console  .log ('Succès :' , data  );
},
error : (err : HttpErrorResponse      ) => {
if( err .status  ===  404 ){
console  .error ('Utilisateur introuvable'      );
} else {
console  .error ('Erreur :' , err .message   );
}
}
}));
```

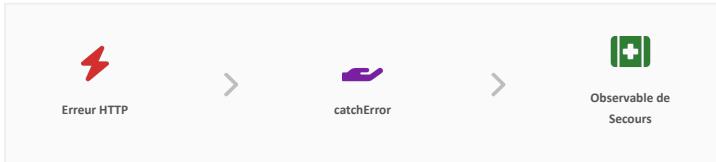
# L'Opérateur catchError

## Le Filet de Sécurité

L'opérateur catchError intercepte une erreur dans le flux Observable et vous permet de décider de la suite :

**Retourner**: une valeur par défaut (ex: liste vide).

**Retrapper**: l'erreur (ou une nouvelle erreur) vers le composant.



### Stratégie 1 : Valeur par défaut

```
searchUsers  (term : string ){  
  return  this .http .get (`/api/users?q=${term}`      ).pipe (  
    catchError(err => of([]))           // Retourne un tableau vide  
  );  
}
```

### Stratégie 2 : Relancer l'erreur

```
getData  (){  
  return  this .http .get (`/api/data`  ).pipe (  
    catchError  (err => {  
      console .error ('Log serveur:', err );  
      return  throwError  (()=> new  Error ('Oups !' ));  
    })  
  );  
}
```

# Les Intercepteurs

## Middleware HTTP

Un intercepteur est une fonction qui s'exécute pour chaque requête HTTP sortante et chaque réponse entrante. C'est le point idéal pour la logique transversale.



Token Auth (Bearer)

Logging

Gestion d'Erreurs

Spinner Loading

auth.interceptor.ts

FONCTIONNEL

```
export const authInterceptor : HttpInterceptorFn = ( req , next )=>{  
  const token = localStorage.getItem ('token');  
  
  // Les requêtes sont immuables, on doit cloner !  
  const clonedReq = req.clone ({  
    setHeaders : {  
      Authorization : `Bearer ${token}`  
    }  
  });  
  
  return next (clonedReq);  
};
```

app.config.ts

```
providers : [  
  provideHttpClient (  
    withInterceptors ([authInterceptor])  
  )
```

# Gestion du Loading



## Astuce RxJS

Utilisez l'opérateur finalize() pour garantir que le chargement s'arrête, même si la requête échoue (crash).

### component.ts

```
loadData () {
  this.isLoading = true;

  this.api.getData()
    .pipe(
      finalize(() => this.isLoading = false)
    )
    .subscribe (data => this.data = data);
}
```

### template.html

```
<div *ngIf="isLoading; else content" >
<app-spinner></app-spinner>
</div>

<ng-template #content >
<ul> ...Liste des données... </ul>
</ng-template>
```

# Typage des Données

## Le Contrat

Ne laissez pas vos réponses API en any. Définissez une **Interface** qui décrit la structure exacte des données attendues.

- ✓ Autocomplétion dans l'IDE
- ✓ Détection d'erreurs à la compilation
- ✓ Code auto-documenté
- ✓ Refactoring sécurisé

### 1. Définir l'Interface

```
export interface Product {  
  id : number ;  
  name : string ;  
  price : number ;  
  inStock : boolean ;  
}
```



### 2. Utiliser le Générique

```
getProducts (): Observable <Product[]> {  
  return this.http.get<Product[]>('/api/products');  
}
```

// Dans le composant :

```
this.service.getProducts().subscribe(products => {  
  console.log(products[0].name); // OK !  
  console.log(products[0].titre); // Erreur TypeScript !
```

# Async / Await

## ⌚ Style Linéaire

Bien qu'Angular soit construit autour de RxJS, vous pouvez préférer la syntaxe async/await pour des opérations simples (une seule requête).

Pour cela, convertissez l'Observable en Promise.

```
lastValueFrom(obs$)
```

### ⚠ Limitation

Ne fonctionne que pour les Observables qui se "terminent" (comme HTTP). Ne pas utiliser pour des flux continus (WebSocket, Clics).

```
getData () {
  this.http.get('/api/items').subscribe ({
    next : (data) => console.log(data),
    error : (err) => console.error(err)
  });
}
```

Standard (RxJS)



```
async getData () {
  try {
    // Conversion Observable -> Promise
    const data = await lastValueFrom(this.http.get('/api/items'));
    console.log(data);
  } catch (err) {
    console.error('Erreur catchée:', err);
  }
}
```

Moderne (Async/Await)

# Les Observables

## Le Flux de Données

Un Observable est un objet qui émet des valeurs dans le temps. C'est le concept central de la programmation réactive (RxJS).



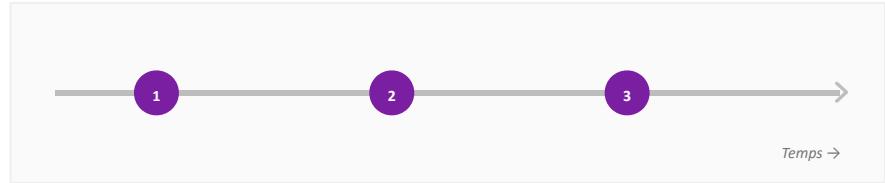
**Asynchrone** : Gère les événements futurs (clics, HTTP, timers).



**Multiple** : Peut émettre 0, 1 ou plusieurs valeurs (contrairement à une Promise).



**Paresseux (Lazy)** : Ne démarre que si on s'y abonne (subscribe).



### Création d'un Observable

```
const stream$ = new Observable (observer => {
  observer .next (1);
  observer .next (2);
  observer .next (3);
  observer .complete ();
});

// Utilisation
stream$ .subscribe (val => console .log (val ));
```

# La Souscription

## L'Activateur

Un Observable est "froid" (lazy). Il ne fait rien tant que personne ne l'écoute. La méthode subscribe() déclenche l'exécution.

### next (Valeur)

Appelé à chaque nouvelle donnée émise (0 à N fois).

### error (Erreur)

Appelé si une erreur survient. Arrête le flux.

### complete (Fin)

Appelé quand il n'y a plus de données. Arrête le flux.

## Syntaxe Moderne (Objet Observer)

```
const subscription = myObservable$.subscribe ({
  next : (value) => {
    console .log ('Reçu : ' , value );
  },
  error : (err) => {
    console .error ('Problème : ' , err );
  },
  complete : () => {
    console .log ('Terminé !' );
  }
});
```

# Unsubscribe & Nettoyage

## ☢️ Fuites de Mémoire

Si vous souscrivez (subscribe) manuellement sans jamais vous désabonner, l'Observable continue d'émettre même après la destruction du composant.

Cela crée des "fuites de mémoire" et des comportements inattendus (ex: code exécuté en double, erreurs console).



### ★ Meilleure Pratique : Async Pipe

```
<div *ngFor = "let item of items$ | async">  
{{ item.name }}  
</div>  
// Angular gère l'unsubscribe automatiquement !
```

HTML

### </> Moderne : takeUntilDestroyed

```
constructor() {  
this.data$.pipe(takeUntilDestroyed()).subscribe();  
}
```

TS (v16+)

### 🔧 Legacy : ngOnDestroy

```
ngOnDestroy() {  
this.subscription.unsubscribe();  
}
```

# Opérateurs RxJS

## Transformation de Flux

Les opérateurs sont des fonctions pures qui prennent un Observable en entrée et retournent un nouvel Observable transformé. Ils s'utilisent dans la méthode .pipe().



**filter** Ne laisse passer que certaines valeurs.



**map** Transforme chaque valeur émise.



**tap** Effet de bord (log) sans modification.

### Exemple Complet

```
import { map, filter, tap } from 'rxjs/operators' ;  
  
this .http .get <User []>('api/users' ).pipe (  
// 1. Log pour debug  
tap (users => console .log ('Reçu:' , users )),  
  
// 2. Transformer : Garder seulement les noms  
map (users => users .map (u=> u.name )),  
  
// 3. Filtrer : Garder les noms commençant par 'A'  
map (names => names .filter (n=> n.startsWith ('A')))  
.subscribe (result => {  
console .log ('Résultat final:' , result  );  
});
```

# Atelier 13 : Requête GET

## Objectif

Récupérer une liste d'utilisateurs depuis une API publique et l'afficher dans votre application Angular.

API Endpoint (JSONPlaceholder)

GET <https://jsonplaceholder.typicode.com/users>

// Retourne un tableau d'objets User

1

### Créer le Service

Générez un service UserService avec la CLI.

```
ng g s services/user
```

2

### Implémenter la Méthode

Injectez HttpClient et créez une méthode getUsers() qui retourne l'Observable.

3

### Souscrire dans le Composant

Dans AppComponent,appelez le service et stockez le résultat.

4

### Afficher la Liste

Utilisez \*ngFor (et idéalement le pipe async) pour afficher les noms des utilisateurs.

# Atelier 14 : Client API REST

## ☰ Objectif

Créer un service pour récupérer une liste d'utilisateurs depuis une API publique et l'afficher dans un composant.

<https://jsonplaceholder.typicode.com/users>

1 Générer UserService

2 Définir l'interface User (id, name, email)

3 Implémenter getUsers() avec HttpClient

4 Afficher la liste avec \*ngFor et async

### Structure Attendue (Interface)

```
export interface User {  
  id : number ;  
  name : string ;  
  email : string ;  
  website : string ;  
}
```

### Structure Attendue (Service)

```
@Injectable ({ providedIn : 'root' })  
export class UserService {  
  private apiUrl = 'https://jsonplaceholder...' ;  
  
  constructor (private http : HttpClient) {}  
  
  getUsers (): Observable <User []> {  
    // TODO: Retourner le GET typé  
  }  
}
```

# Déploiement : Introduction

## De Localhost au Monde

Le développement se fait avec `ng serve` (rapide, en mémoire), mais la production nécessite des fichiers statiques optimisés.

### DEV (`ng serve`)

Compilation JIT

Fichiers lourds

Source Maps

### PROD (`ng build`)

Compilation AOT

Minifiés & Tree-shakés

Optimisé



### 1. Build (Construction)

Transformation du TypeScript/HTML en JavaScript pur. Compilation Ahead-of-Time (AOT).



### 2. Optimisation

Minification, Uglification, Tree Shaking (suppression du code mort).



### 3. Hébergement

Dépôt des fichiers statiques (`index.html`, `.js`, `.css`) sur un serveur web (Apache, Nginx, AWS S3).

# Compiler pour Production

```
user@project:~$ng build

Building for production...
✓ Browser application bundle generation complete.
✓ Copying assets complete.
✓ Index html generation complete.
```



## Minification

Le code est compressé, les espaces supprimés et les variables renommées pour réduire la taille.



## Tree Shaking

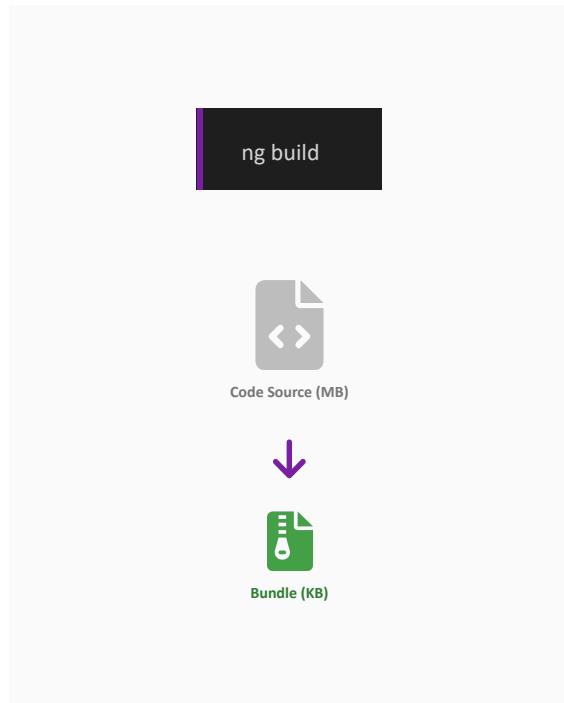
Le code mort (non utilisé) est éliminé automatiquement du bundle final.



## Cache Busting

Les fichiers générés ont un hash unique (ex: main.a1b2.js) pour forcer la mise à jour du cache navigateur.

# Optimisations Automatiques



## Minification

Suppression de tous les caractères inutiles : espaces, sauts de ligne, commentaires. Le code devient illisible pour l'humain mais très compact.



## Uglification

Renommage des variables et fonctions avec des noms courts (ex: myLongVariableName devient a). Réduit drastiquement la taille.



## Tree Shaking

"Secouer l'arbre" pour faire tomber les feuilles mortes. Le compilateur détecte et supprime le code importé mais jamais utilisé.



## AOT Compilation

Ahead-of-Time. Les templates HTML et CSS sont compilés en JavaScript pur pendant le build, supprimant le besoin d'embarquer le compilateur Angular.

# AOT vs JIT

## JIT

Just-in-Time Compilation



- 🌐 Compilation dans le **navigateur**
- 🎒 Bundle **plus lourd** (inclus le compilateur)
- ⌚ Démarrage **plus lent**
- 💻 Usage : Développement (historique)

## AOT

Ahead-of-Time Compilation



- 💻 Compilation lors du **build**
- 🎒 Bundle **léger** (compilateur exclu)
- ⚡ Rendu **instantané**
- 🛠️ Détecte les erreurs de template **avant** l'exécution



Depuis Angular 9, AOT est activé par défaut, même en mode développement (`ng serve`), pour garantir la sécurité du typage dans les templates.

# Déployer sur Serveur

Une application Angular compilée n'est qu'un ensemble de fichiers statiques (HTML, JS, CSS). Elle peut être hébergée presque partout.



## Hébergeurs Statiques

La solution moderne. Connecté à Git, déploiement automatique à chaque push, HTTPS et CDN inclus par défaut. Idéal pour les SPA.

Exemples

Vercel    Netlify    GitHub Pages  
          Firebase



## Serveurs Web

La solution classique d'entreprise. Vous gérez la configuration du serveur. Nécessite de configurer la réécriture d'URL (Rewrite Rules).

Exemples

Apache HTTPD    Nginx    IIS



## Cloud & Conteneurs

Pour les architectures complexes ou micro-services.  
Hébergement sur stockage objet ou via conteneurisation.

Exemples

AWS S3 + CloudFront    Docker  
                          Kubernetes

# Configuration Apache

## ⚠ Le Problème 404

Si un utilisateur rafraîchit la page /users/123, Apache cherche ce dossier sur le disque. Il ne le trouve pas et renvoie une erreur 404.

## ✓ La Solution

Rediriger toutes les requêtes inconnues vers index.html pour laisser le routeur Angular gérer l'URL.



Requis : mod\_rewrite

.htaccess

`RewriteEngine On`

*# Si le fichier ou dossier existe physiquement, on le sert*

`RewriteCond %{DOCUMENT_ROOT}%{REQUEST_URI} -f [OR]`

`RewriteCond %{DOCUMENT_ROOT}%{REQUEST_URI} -d`

`RewriteRule ^ - [L]`

*# Sinon, on redirige tout vers index.html*

`RewriteRule ^ /index.html[L]`

Placez ce fichier dans le dossier src/ et ajoutez-le aux assets dans angular.json pour

# Configuration Nginx

## ⚠️ Le Problème "404"

Si un utilisateur rafraîchit la page /users/12, Nginx va chercher ce fichier physique. Il n'existe pas !

## ✓ La Solution SPA

On doit dire au serveur : "Si tu ne trouves pas le fichier, renvoie toujours index.html". Angular gérera ensuite la route.

/etc/nginx/sites-available/default

```
server {  
    listen 80;  
    server_name mon-app.com;  
    root /var/www/mon-projet/dist/browser;  
    index index.html;  
  
    # La règle magique pour les SPA  
    location / {  
        try_files $uri $uri/ /index.html;  
    }  
  
    # Cache pour les assets statiques (optionnel)  
    location ~* \.(js|css|png|jpg|jpeg|gif|ico)$ {  
        expires 1y;  
        add_header Cache-Control "public, no-transform";  
    }  
}
```

# Variables d'Environnement

## Configuration

Ne hardcodez jamais vos URLs d'API. Angular utilise le dossier `src/environments/` pour gérer les différences entre le développement et la production.

```
environment.ts
```

```
export const environment = {  
  production: false,  
  apiUrl: 'http://localhost:3000'  
};
```

```
environment.prod.ts
```

```
export const environment = {  
  production: true,  
  apiUrl: 'https://api.myapp.com'  
};
```

### Utilisation dans un Service

```
import { environment } from './environments/environment';  
  
@Injectable(...)  
export class UserService {  
  // Angular utilisera automatiquement le bon fichier  
  private url = environment.apiUrl + '/users';  
  
  getUsers() {  
    return this.http.get(this.url);  
  }  
}
```

Code Source

```
import { environment }
```



Build (ng build)

Remplacement de fichier



Bundle Final

Contient la config Prod

# Atelier 15 : Déploiement

## Objectif

Générer le bundle de production optimisé et simuler un déploiement sur un serveur statique local pour vérifier que tout fonctionne hors du mode dev.

## Outil Requis

Nous utiliserons http-server (via npx) pour servir les fichiers statiques comme le ferait Apache ou Nginx.

1

### Compiler le Projet

```
ng build --configuration production  
# Crée le dossier /dist/mon-projet avec les fichiers optimisés
```

2

### Lancer le Serveur de Test

```
npx http-server ./dist/mon-projet  
# Démarré un serveur web léger sur le dossier de build
```

3

### Vérifier

```
Ouvrir http://127.0.0.1:8080  
# Vérifiez la console (F12) : aucune erreur ne doit apparaître !
```

# Bonnes Pratiques



## Mode Strict

Activez toujours strict: true dans tsconfig.json. Cela élimine les erreurs de type null ou undefined avant même l'exécution.

```
"compilerOptions": {  
  "strict": true,  
  "noImplicitAny": true  
}
```



## OnPush Strategy

Utilisez ChangeDetectionStrategy.OnPush pour les composants de présentation. Angular ne vérifiera la vue que si les inputs changent.

```
@Component({  
  changeDetection: ChangeDetectionStrategy.OnPush  
})
```



## Async Pipe

Évitez les subscribe() manuels dans les composants. Préférez le pipe async dans le template pour gérer automatiquement la souscription.

```
<!-- Pas de fuite de mémoire -->  
<div *ngFor="let user of users$ | async">
```

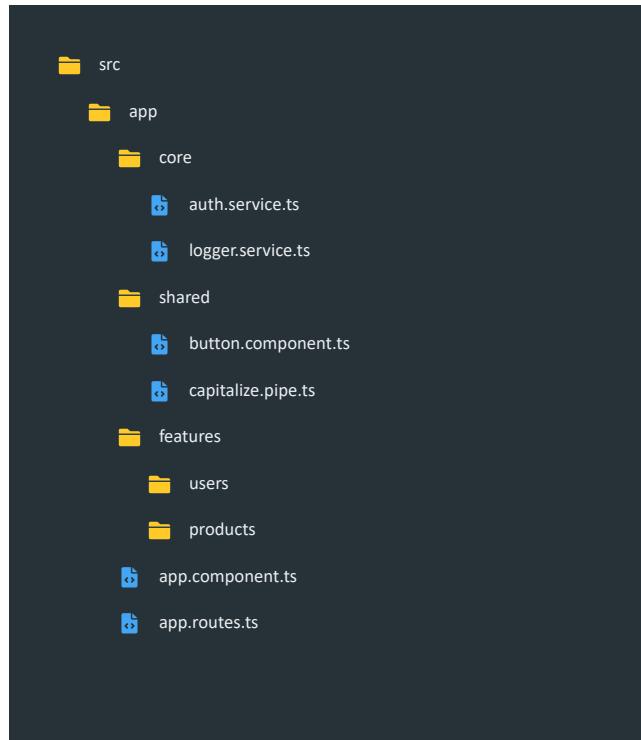


## Smart vs Dumb

Séparez les composants "Intelligents" (qui appellent les services) des composants "Bêtes" (qui ne font qu'afficher des @Input).

```
UserPage (Smart) -> UserList (Dumb)  
// UserList ne connaît pas le Service !
```

# Structure du Projet



## Core

Singletons

Contient les services globaux (Auth, Logger), les Guards, les Intercepteurs et les modèles globaux. Chargé une seule fois dans AppModule (ou root).

## Shared

Réutilisable

Contient les composants UI "dumb" (boutons, cartes), les Pipes et les Directives utilisés partout. Ne doit PAS avoir de dépendances vers Core.

## Features

Métier

Modules fonctionnels (Utilisateurs, Admin, Dashboard). Contient les pages, le routing spécifique et la logique métier. Souvent chargés en Lazy Loading.



Principe LIFT : Locate, Identify, Flat, Try-Dry

# Conventions de Nommage

## Le Pattern Universel

nom-fonctionnalite.type.ts

Type	Nom de Fichier	Nom de Classe
Component	hero-list.component.ts	HeroListComponent
Service	user-profile.service.ts	UserProfileService
Pipe	initials.pipe.ts	InitialsPipe
Directive	highlight.directive.ts	HighlightDirective
Interface	hero.model.ts	Hero

```
src/app/heroes/
  hero-list.component.ts
  hero-list.component.html
  hero-list.component.css
  hero-list.component.spec.ts
```

✗ HeroList.ts / heroList.html

✓ hero-list.component.ts

# Composants Réutilisables

## Smart Component (Container)

Gère la logique, les services, et l'état.



## Dumb Component (UI)

Reçoit des données (@Input) et émet des événements (@Output).

### Principe Clé

Un composant réutilisable ne doit jamais injecter de services pour récupérer ses propres données. Il doit être "pur" et dépendre uniquement de ses Inputs.

#### user-card.component.ts (Dumb)

```
@Component ({ ... })
export class UserCardComponent {
  // Entrée : Données pures
  @Input () user !: User;

  // Sortie : Événement pur
  @Output () delete = new EventEmitter <number>();

  onDelete () {
    this.delete.emit (this.user.id);
  }
}
```

#### Utilisation (Parent)

```
<app-user-card
  [user] = "currentUser"
  (delete) = "removeUser($event)" >
</app-user-card>
```

# Services Réutilisables

## Pattern Generic

Ne réécrivez pas les méthodes CRUD (Create, Read, Update, Delete) pour chaque entité. Utilisez les Génériques TypeScript pour créer un service de base.

ResourceService<T>



UserService  
(extends Resource)



ProductService  
(extends Resource)

resource.service.ts

```
export class ResourceService<T> {
  constructor (
    protected http : HttpClient ,
    protected endpoint : string
  ) {}

  list (): Observable<T[]> {
    return this .http .get <T>(`${apiUrl}/${this.endpoint}`);
  }

  create (item : T): Observable<T> {
    return this .http .post <T>(`${apiUrl}/${this.endpoint}` , item );
  }
}
```

user.service.ts

```
@Injectable (...)

export class UserService extends ResourceService<User> {
```

# Lazy Loading

Sans Lazy Loading (Eager)



Avec Lazy Loading



-60%

Temps de chargement initial

```
app.routes.ts

const routes : Routes = [
{
  path : 'admin',
  // Import dynamique (Promise)
  loadChildren : () => import ('./admin/admin.module')
  .then (m => m.AdminModule)
}
];
```



Démarrage ultra-rapide de l'application



Économie de bande passante (mobile)



Meilleure modularité du code

# Tree Shaking

## Le Concept

Un processus d'optimisation qui "secoue" votre projet pour faire tomber le code mort (non utilisé). Seul le code réellement importé est inclus dans le bundle final.



Activé par défaut en Production

### Import Global

Lourd (Tout le module)

```
// Importe TOUTE la librairie lodash
import * as _ from 'lodash';

_.map([1, 2], n => n * 2);
```



### Import Nommé

Léger (Juste la fonction)

```
// Importe UNIQUEMENT la fonction map
import { map } from 'lodash-es';

map([1, 2], n => n * 2);
```

# Performance



## ChangeDetection.OnPush

Désactive la détection de changement automatique. Angular ne met à jour la vue que si les références des @Input changent.



## trackBy Function

Indispensable pour \*ngFor. Permet à Angular d'identifier les éléments uniques et d'éviter de redessiner toute la liste lors d'une mise à jour.



## Pipes Pur

Évitez les appels de fonction dans le template {{ heavyCalc() }}. Utilisez des Pipes qui mettent en cache le résultat.

### Optimized Component

```
@Component ({{
  selector : 'app-user-list',
  template : `
    <div *ngFor= "let user of users;      trackBy: trackById"      ">
      {{ user.name |      formatName      }}
    </div>
  `,
  changeDetection : ChangeDetectionStrategy .OnPush
})
export class UserListComponent {
  @Input () users : User [] = [];

  trackById (index : number , user : User ) : number {
    return user .id ;
  }
}
```

# Sécurité



## Protection XSS

Angular traite toutes les valeurs comme non fiables par défaut. Lors de l'interpolation {{ value }} ou du binding de propriété, Angular **assainit** (sanitize) le contenu pour retirer les scripts malveillants.

```
<div [innerHTML]="html">  
// Les balises <script> sont retirées automatiquement
```



## Zone de Danger

Parfois, vous devez forcer l'affichage de contenu HTML (ex: CMS). L'utilisation de DomSanitizer désactive la protection d'Angular. À utiliser avec extrême précaution !

```
this.sanitizer.  
bypassSecurityTrustHtml(val)
```



## Checklist

- ✓ Éviter l'accès direct au DOM
- ✓ Utiliser CSP (Content Security Policy)
- ✓ Mettre à jour Angular régulièrement
- ✓ Auditer les dépendances npm



Règle d'or : Ne faites jamais confiance aux données provenant de l'utilisateur ou d'une API externe sans validation.

# Les Tests Automatisés

## E2E

End-to-End  
(Cypress, Playwright)

## Intégration

Interaction Composants  
( TestBed )

## Unitaires

Fonctions isolées, Services  
( Jasmine, Jest )



## Filet de Sécurité

Les tests capturent les régressions instantanément. Si vous cassez une fonctionnalité existante en modifiant du code, le test échoue.



## Refactoring Serein

Vous pouvez améliorer la structure de votre code sans peur, car les tests garantissent que le comportement reste le même.



Angular est "Opinionated" : Tout projet généré via la CLI inclut déjà un framework de test configuré et prêt à l'emploi.

# Jasmine Framework

## BDD Framework

Jasmine est un framework de test orienté comportement (Behavior Driven Development). Il fournit la syntaxe pour écrire les tests de manière lisible, proche de l'anglais.

- Pas de dépendances externes
- Assertions incluses (`expect`)
- Spies & Mocks intégrés

```
describe('CalculatorService', () => {  
  let service: CalculatorService;  
  
  beforeEach(() => {  
    service = new CalculatorService();  
  });  
  
  // Le test unitaire  
  it('should add two numbers', () => {  
    const result = service.add(2, 3);  
    expect(result).toBe(5);  
  });  
  
  it('should subtract numbers', () => {  
    expect(service.sub(5, 2)).toEqual(3);  
  });  
});
```

Suite

Setup

Spec (Test)

Assertion

# Karma



## Le Test Runner

Karma n'est pas un framework de test, c'est un exécuteur. Il lance un serveur web, ouvre de vrais navigateurs, injecte votre code et rapporte les résultats.



Connected

Chrome / Firefox / Edge

karma.conf.js

```
module.exports = function(config) {
  config.set({
    frameworks: ['jasmine', '@angular-devkit/build-angular'],
    browsers: ['Chrome'],
    reporters: ['progress', 'kjhtml'],
    autoWatch: true,
    restartOnFileChange: true
  });
};
```



Mode Watch : Relance les tests à chaque sauvegarde



CI/CD Ready : Supporte ChromeHeadless

# Erreurs Courantes

## ExpressionChanged...

L'erreur classique ! Vous modifiez une donnée *après* qu'Angular l'ait vérifiée (souvent dans ngAfterViewInit).

```
ngAfterViewInit
() {
  this
  .
  loading
  =
  false
;
}
```

```
setTimeout
() => {
  this
```

Solution

## Fuite de Mémoire

Oublier de se désabonner d'un Observable infini (comme interval ou router.events).

```
obs$ 
.
subscribe
(...)

// Reste actif même après destruction !
```

Risque

Solution

## Fonctions dans le Template

Appeler une fonction dans {{ }} force Angular à la réexécuter à chaque cycle de détection.

```
{

  calculateTotal
() {}

}

// Exécuté 100x par seconde
```

Lent

```
{

  total
  |
  currency
}

// Utilisez un Pipe pur !
```

Solution

# Debugging

## </> Dans le Code

Utilisez le mot-clé debugger pour forcer l'arrêt de l'exécution dans le navigateur et inspecter les variables locales.

```
updateUser() {  
  const data = ...;  
  // Arrêt ici !  
  debugger;  
  this.save(data);  
}
```

💡 Les Source Maps (.map) permettent de voir le code TypeScript original dans Chrome, pas le JS compilé.



## Angular DevTools

L'extension officielle pour Chrome/Firefox. Indispensable pour comprendre la structure de l'application.

- Visualiser l'arbre des composants
- Inspecter les @Input / @Output
- Modifier l'état en direct
- Profiler les performances (cycles de détection)



## Dans le Template

Le pipe json est votre meilleur ami pour visualiser rapidement le contenu d'un objet ou d'un Observable.

```
<pre>  
{{ user$ | async | json }}  
</pre>
```

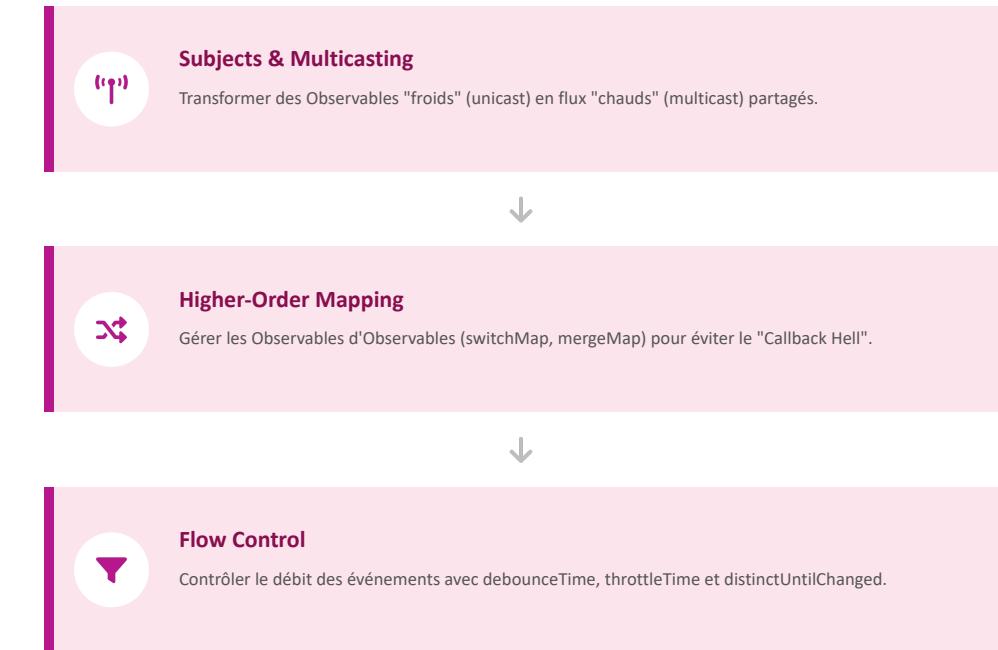
Utilisez la balise <pre> pour conserver le formatage et l'indentation du JSON.

# RxJS Avancé

Au-delà des simples requêtes HTTP, RxJS est un moteur puissant pour gérer l'état, le temps et les événements complexes.



## Reactive Extensions



# Subjects

## 💡 Multicasting

Contrairement à un Observable classique qui est **Unicast** (chaque abonné a sa propre exécution), un Subject est **Multicast**.

C'est comme une émission de radio en direct : tout le monde reçoit le même signal au même moment.



A

B

C

Un Subject est spécial car il est à la fois **Observable** (on peut s'y abonner) et **Observer** (on peut y pousser des valeurs).

### Exemple d'utilisation

```
const news$ = new Subject<string>();

// Abonné 1
news$.subscribe(v=> console.log('Abonné A:', v));

// Abonné 2
news$.subscribe(v=> console.log('Abonné B:', v));

// Émission manuelle (Multicast)
news$.next('Breaking News!');

// Sortie :
// Abonné A: Breaking News!
// Abonné B: Breaking News!
```

# BehaviorSubject

## La Mémoire

Contrairement au Subject classique, le **BehaviorSubject** retient la dernière valeur émise. Tout nouvel abonné reçoit immédiatement cette valeur, même s'il arrive "en retard".



*"C'est comme un tableau d'affichage. Le message reste écrit dessus. Si vous entrez dans la pièce, vous voyez le message actuel tout de suite."*

```
// 1. Valeur initiale OBLIGATOIRE
const user$ = new BehaviorSubject('Guest');

// 2. Émission d'une nouvelle valeur
user$.next('Alice');

// 3. Abonnement TARDIF
user$.subscribe(val => console.log(val));
// -> Affiche immédiatement "Alice" !

// 4. Accès synchrone (Spécifique au BehaviorSubject)
console.log(user$.value); // "Alice"
```



# ReplaySubject

## ⌚ Le "Magnétophone"

Contrairement au BehaviorSubject qui ne retient que la *dernière* valeur, le ReplaySubject peut mémoriser les *N dernières valeurs* et les renvoyer aux nouveaux abonnés.



### Exemple de Buffer

```
// On garde en mémoire les 2 dernières valeurs
const subject = new ReplaySubject(2);

subject .next ('Un' );
subject .next ('Deux' );
subject .next ('Trois' );

// Abonnement tardif
subject .subscribe (val => console .log (val ));

// Output immédiat :
// "Deux"
// "Trois"
```

### 💡 Cas d'usage

Idéal pour afficher un historique récent (ex: les 5 derniers messages d'un chat) ou pour mettre en cache plusieurs résultats d'API.

# Opérateurs de Mapping

Appelés "Higher Order Mapping Operators", ils gèrent la stratégie de souscription lorsque des Observables sont imbriqués (ex: un clic déclenche une requête HTTP).

## switchMap

Le Zapeur

Annule l'Observable précédent dès qu'une nouvelle valeur arrive. Idéal pour les recherches (typeahead).



## concatMap

La File

Attend que le précédent soit terminé avant de lancer le suivant. Ordre garanti. Idéal pour des opérations séquentielles (ex: updates).



## mergeMap

Le Parallèle

Souscrit à tout en parallèle. Ne s'annule jamais. Idéal pour des opérations indépendantes (ex: suppressions multiples).



## exhaustMap

Le Bloqueur

Ignore les nouvelles valeurs tant que l'Observable en cours n'est pas terminé. Idéal pour empêcher le spam de clics (ex: bouton login).



# switchMap

## L'Annulateur

switchMap est l'opérateur le plus utilisé pour les effets de bord (API). À chaque nouvelle émission source, il **annule** (unsubscribe) l'Observable intérieur précédent s'il n'est pas terminé.



## Cas d'usage : Autocomplétion

Évite les "Race Conditions" où une ancienne réponse arrive après la nouvelle et écrase le bon résultat.

```
search.component.ts
```

```
this .searchControl .valueChanges .pipe (
```

```
// Attendre que l'utilisateur arrête de taper
```

```
debounceTime (300),
```

```
// Annuler la requête précédente si nouvelle frappe
```

```
switchMap (term => this .api .search (term ))
```

```
).subscribe (results => {
```

```
this .items = results ;
```

# mergeMap

## Le Parallélisme

mergeMap souscrit à chaque nouvel Observable interne **sans annuler les précédents**. Tout s'exécute en parallèle.

Source

1

2

Résultat (Parallèle)

1a

2a

1b

2b

Exemple : Suppressions Multiples

```
const idsToDelete = [ 1, 2, 3];

from (idsToDelete).pipe (
  // Lance les 3 requêtes en même temps
  mergeMap (id => this.http.delete (`/api/items/${id}`))
).subscribe (
  res => console.log ('Un item supprimé')
);
```



### Attention à la charge

Si vous émettez 1000 valeurs, mergeMap lancera 1000 requêtes simultanées ! Utilisez le paramètre concurrent pour limiter (ex: mergeMap(fn, 5)).

# concatMap

## ↓ 1 L'Ordre Respecté

concatMap met les requêtes en file d'attente. Il attend impérativement que l'Observable précédent soit *terminé* (complete) avant de souscrire au suivant.



Exécution séquentielle : 1, puis 2, puis 3.

### Exemple : Sauvegarde en série

```
// On veut sauvegarder une liste d'items un par un
from ([1, 2, 3]).pipe (
  // concatMap garantit l'ordre 1 -> 2 -> 3
  concatMap (id => this .http .post ('/save' , id ))
).subscribe (res => {
  console .log ('Sauvegardé !');
});
```

- ✓ Opérations CRUD dépendantes (Créer puis Modifier)
- ✓ Envoi de messages dans l'ordre exact
- ✓ Animations séquentielles

# debounceTime

## ⌚ Le Temporisateur

Cet opérateur ignore les nouvelles valeurs tant qu'une période de silence n'est pas respectée. Il n'émet que la dernière valeur après que le flux se soit calmé.



### Recherche Optimisée

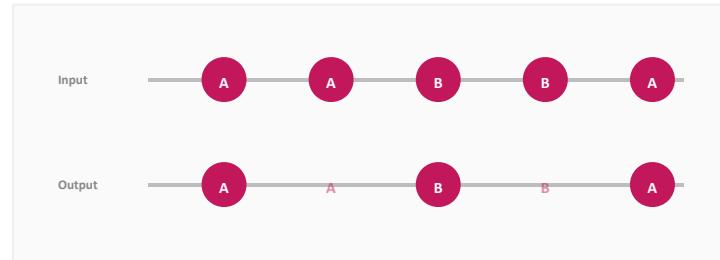
```
this .searchControl .valueChanges .pipe (  
    // Attend 300ms de pause  
    debounceTime (300 ),  
  
    // Évite de rechercher 2x la même chose  
    distinctUntilChanged (),  
  
    // Annule la requête précédente si nouvelle  
    switchMap (term => this .api .search (term ))  
).subscribe (results => ...);
```



# distinctUntilChanged

## Anti-Doublons

Cet opérateur compare la valeur actuelle avec la précédente. Si elles sont identiques, la nouvelle valeur est ignorée.



### Exemple Basique

```
import { of } from 'rxjs';
import { distinctUntilChanged } from 'rxjs/operators';

of(1, 1, 2, 2, 3, 1).pipe (
  distinctUntilChanged()
).subscribe (val => console .log (val ));

// Output: 1, 2, 3, 1
```

### ⚠️ Attention aux Objets !

Par défaut, il utilise l'égalité stricte (==). Deux objets avec le même contenu sont des références différentes !

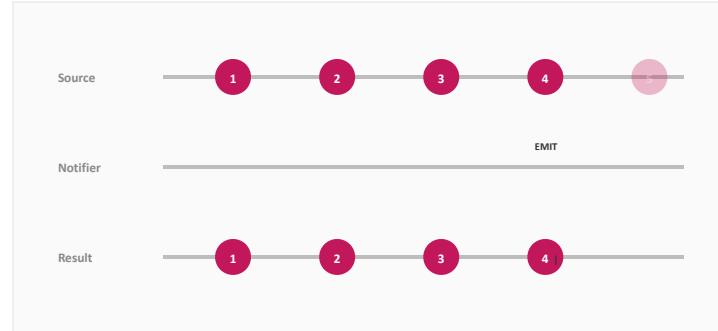
Pour comparer des objets, passez une fonction de comparaison :  
distinctUntilChanged((prev, curr) => prev.id === curr.id)

# takeUntil



## Le "Kill Switch"

Cet opérateur écoute un **deuxième Observable** (le notificateur). Dès que ce notificateur émet une valeur, takeUntil complète immédiatement l'Observable source et se désabonne.



### Le Pattern "ngOnDestroy"

```
export class MyComponent implements OnDestroy {  
  private destroy$ = new Subject<void>();  
  
  ngOnInit() {  
    interval(1000).pipe(  
      // Arrête tout quand destroy$ émet  
      takeUntil(this.destroy$)  
    ).subscribe(...);  
  }  
  
  ngOnDestroy() {  
    // Déclenche la fin de tous les abonnements  
    this.destroy$.next();  
    this.destroy$.complete();  
  }  
}
```

C'est la méthode recommandée pour éviter les fuites de mémoire dans les composants Angular. Plus propre que de stocker manuellement chaque Subscription.



# Directives Personnalisées



## Étendre le HTML

Les directives permettent d'attacher un comportement personnalisé à des éléments du DOM existants. C'est le moyen idéal pour encapsuler de la logique UI réutilisable.

- ✓ Manipuler le style
- ✓ Écouter les événements
- ✓ Modifier la structure (DOM)

```
@Directive({
  selector: '[appHighlight]',
  standalone: true
})
export class HighlightDirective {
  constructor(privateel: ElementRef) {
    this.el.nativeElement.style.backgroundColor = 'yellow';
  }
}
```

Décorateur

Sélecteur d'attribution

Logique

Utilisation dans le template

```
<p appHighlight>Texte surligné !</p>
```

# Directive d'Attribut

## Le Décorateur

Une directive d'attribut modifie l'apparence ou le comportement d'un élément DOM existant. Elle ne l'ajoute pas et ne le supprime pas.



highlight.directive.ts

```
@Directive  {{
  selector : '[appHighlight]'    // Sélecteur CSS d'attribut
})
export class  HighlightDirective  {
  constructor  (private  el : ElementRef ) {
    // Accès direct au DOM natif
    this .el .nativeElement .style .backgroundColor = 'yellow' ;
  }
}
```

app.component.html

```
<p> Texte normal </p>

<!-- Utilisation comme un attribut HTML standard -->
<p  appHighlight  >Texte surligné par Angular ! </p>
```

# Directive Structurelle



## Manipuler le DOM

Une directive structurelle modifie la structure du DOM en ajoutant ou supprimant des éléments. Elle est toujours précédée d'un astérisque \*.

```
<div * appDelay="1000">...</div>
```

Syntaxe Sucré



```
<ng-template [appDelay]="1000">  
<div>...</div>  
</ng-template>
```

Réalité Angular

L'astérisque transforme l'élément en ng-template.

### TemplateRef

C'est le contenu ("le tampon"). Ce que nous voulons afficher.

### ViewContainerRef

C'est le conteneur ("la feuille"). Là où nous allons insérer le contenu.

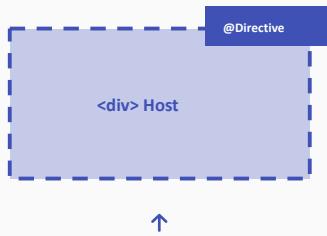
delay.directive.ts

```
@Directive({ selector : '[appDelay]' })  
export class DelayDirective {  
  constructor(  
    private templateRef : TemplateRef<any>,  
    private viewContainer : ViewContainerRef  
  ) {}  
  
  @Input() set appDelay(time : number) {  
    setTimeout(() => {  
      // Insère le template dans le conteneur  
      this.viewContainer.createEmbeddedView(this.templateRef);  
    }, time);  
  }  
}
```

# @HostListener

## L'Écouteur

Ce décorateur permet à une directive de s'abonner aux événements DOM (click, hover, keydown) de l'élément sur lequel elle est placée (l'hôte).



La directive capture l'événement "mouseenter" déclenché par la souris sur le div.

highlight.directive.ts

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

@Directive({ selector: '[appHighlight]' })
export class HighlightDirective {

    // Écoute l'entrée de la souris
    @HostListener('mouseenter') onMouseEnter() {
        this.highlight('yellow');
    }

    // Écoute la sortie
    @HostListener('mouseleave') onMouseLeave() {
        this.highlight(null);
    }

    // Avec arguments
    @HostListener('click', ['$event']) onClick(e) {
        console.log('Click coordinates:', e.clientX);
    }
}
```

# @HostBinding

## Le Lieur

Ce décorateur permet de lier une propriété de la classe de la directive à une propriété de l'élément hôte (style, classe, attribut).

Variable TS  
this.color



DOM Style  
background

*Angular synchronise automatiquement la valeur. Si la variable change, le style change.*

highlight.directive.ts

```
export class HighlightDirective {  
  
    // Lie la propriété style.backgroundColor de l'hôte  
    // à la variable 'backgroundColor' ci-dessous  
    @HostBinding ('style.backgroundColor')  
    backgroundColor : string ;  
  
    @HostListener ('mouseenter') onMouseEnter () {  
        // Modifie simplement la variable !  
        this .backgroundColor = 'yellow' ;  
    }  
  
    @HostListener ('mouseleave') onMouseLeave () {  
        this .backgroundColor = null ;  
    }  
}
```



Plus propre que `ElementRef.nativeElement`



Fonctionne pour les classes : `@HostBinding('class.active')`

# Atelier 16 : Directive Personnalisée

## Objectif

Créer une directive [appRainbow] qui change la couleur du texte et de la bordure aléatoirement à chaque frappe clavier.

Hello World!

(La couleur change quand on tape)

1

### Générer la directive

Utilisez la CLI pour créer la structure de base.

```
ng generate directive rainbow
```

2

### Préparer les Bindings

Utilisez @HostBinding pour lier la couleur et la bordure de l'élément hôte à des propriétés de votre directive.

3

### Écouter le Clavier

Utilisez @HostListener('keydown') pour détecter quand l'utilisateur tape.

4

### Logique Couleur

Dans le handler, générez une couleur hexadécimale aléatoire et assignez-la à vos variables liées.

```
constcolor = '#' + Math.floor(Math.random()*16777215).toString(16);
```

# Ressources



## Documentation

### [angular.dev](#)

Nouveau

La nouvelle maison d'Angular. Tutoriels interactifs, playground et documentation moderne.

### [angular.io](#)

L'ancienne documentation officielle, toujours une référence pour les versions antérieures à la v17.

### [Angular Blog](#)

[blog.angular.io](http://blog.angular.io) - Pour suivre les annonces de sorties et les évolutions futures.



## Communauté

### [Stack Overflow](#)

Tag [angular]. La plus grande base de connaissances pour résoudre vos bugs.

### [Discord Angular](#)

Chat en direct avec d'autres développeurs et parfois l'équipe Core.

### [Reddit r/Angular](#)

Discussions, partage de projets et veille technologique.



## Écosystème

### [RxJS.dev](#)

Documentation officielle de la librairie réactive. Indispensable.

### [Angular Material](#)

[material.angular.io](http://material.angular.io) - Composants UI officiels respectant le Material Design.

### [NgRx](#)

[ngrx.io](http://ngrx.io) - Gestion d'état réactive (Redux pattern) pour les applications complexes.

# Documentation Officielle



## Angular.dev

<https://angular.dev>

La nouvelle documentation officielle. Plus moderne, interactive et complète que l'ancien angular.io.



### Guides & Concepts

Pour comprendre la philosophie (Components, DI, Signals).



### API Reference

La liste exhaustive de toutes les classes, directives et pipes.



### CLI Reference

Toutes les commandes (ng generate, ng build, options).



### Playground

Testez du code Angular directement dans votre navigateur.

# La Communauté

Angular possède l'une des communautés les plus vastes et actives. Ne restez pas bloqué seul, rejoignez les discussions !



## Discord Officiel

Le lieu idéal pour le chat en direct. Des milliers de développeurs connectés pour aider sur des problèmes spécifiques ou discuter des nouveautés.



## Stack Overflow

La base de connaissance ultime. Utilisez le tag [angular]. La plupart de vos erreurs ont déjà été résolues ici.



## Reddit

Rejoignez r/angular pour des discussions plus générales, des articles de blog, des débats sur l'architecture et des news.



## Twitter / X

Suivez le compte officiel @angular et les nombreux GDEs pour être au courant des RFCs et des sorties de versions.



## Événements Majeurs

### ng-conf

La conférence officielle (Salt Lake City)

### AngularConnect

La plus grande conf en Europe

### Meetups Locaux

Cherchez "Angular [Votre Ville]" sur Meetup.com



Suivez les Google Developer Experts (GDE) pour du contenu de qualité.

# Projets Pratiques

La meilleure façon d'apprendre est de construire. Voici trois idées de projets pour consolider vos acquis, du niveau intermédiaire à expert.



Intermédiaire

## Dashboard Admin

Créez une interface d'administration avec une sidebar, plusieurs pages de statistiques et des tableaux de données.

### Concepts Clés

- Routing & Lazy Loading
- Guards (Auth)
- Directives Structurelles



Avancé

## E-commerce

Une boutique en ligne avec catalogue, panier d'achat persistant et processus de commande multi-étapes.

### Concepts Clés

- Services & DI
- RxJS (Gestion du panier)
- Reactive Forms (Checkout)



Expert

## Chat App

Une application de messagerie instantanée avec liste de contacts, historique et notifications en temps réel.

### Concepts Clés

- RxJS Avancé (WebSockets)
- Optimisation (ChangeDetection)
- Pipes Impurs / Async

# Prochaines Étapes



## Modern Angular

### ✓ Signals

Le nouveau standard de réactivité. Plus performant et plus simple que RxJS pour l'état synchronisé.

### ✓ Standalone APIs

Construire des applications sans NgModules. Plus léger, plus facile à apprendre.



## Architecture

### ✓ State Management

Maîtriser NgRx ou Akita pour gérer des états complexes partagés globalement.

### ✓ Design Patterns

Container/Presenter, Facade Pattern, et Clean Architecture.



## Enterprise Scale

### ✓ Nx Monorepos

Gérer plusieurs applications et librairies dans un seul dépôt avec des outils puissants.

### ✓ SSR & Hydration

Optimiser le SEO et le temps de chargement initial avec le rendu côté serveur.



# Félicitations !

Vous avez terminé la formation  
Angular pour Débutants



Codez



Testez



Déployez



Merci de votre attention !