

Formation Angular



Angular

- Framework basé sur Typescript, CSS et HTML, développé par Google
- Permet la création d'applications web monopage (SPA)
- Successeur de AngularJS, en 2016

Versions

- Angular 14 (02/06/2022)
 - Composants standalone
 - Liaison sur les attributs protected
 - inject()
- Angular 15 (16/11/2022)
 - host directive
 - Intercepteur fonctionnel
- Angular 16 (03/05/2023)
 - Signals
 - @Input requis
 - takeUntilDestroyed()
- Angular 17 (08/11/2023)
 - Blocs @if @for et @switch directement dans le template

Etude de l'application de démonstration d'Angular

- Installation d'angular

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

- Crédit du projet

```
ng new
```

- Le premier exemple est basé sur le projet qui était initialisé avec la version 16 d'Angular

Lancement de l'application

```
ng serve
```

- ng serve permet de tester l'application sur votre machine
- Le lancement de l'application se découpe en deux parties :
 - Le compilateur transforme votre projet Angular en fichiers HTML/CSS et javascript
 - Un serveur local est créé en sur la machine, qui héberge ces fichiers (avec Vite depuis la version 17, Webpack avant)

main.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { InterpolationComponent } from './01-interpolation/interpolation.component';
import { AppModule } from './app/app.module';

// Module
platformBrowserDynamic().bootstrapModule(AppModule).catch(err => console.error(err));

/* ou */

// Composant standalone
bootstrapApplication(InterpolationComponent).catch(err => console.error(err));
```

- Point d'entrée de l'application
- Permet d'initialiser l'application avec un module (en haut) ou un composant standalone (en bas)
- Dans une application normale, ce fichier est très rarement modifié

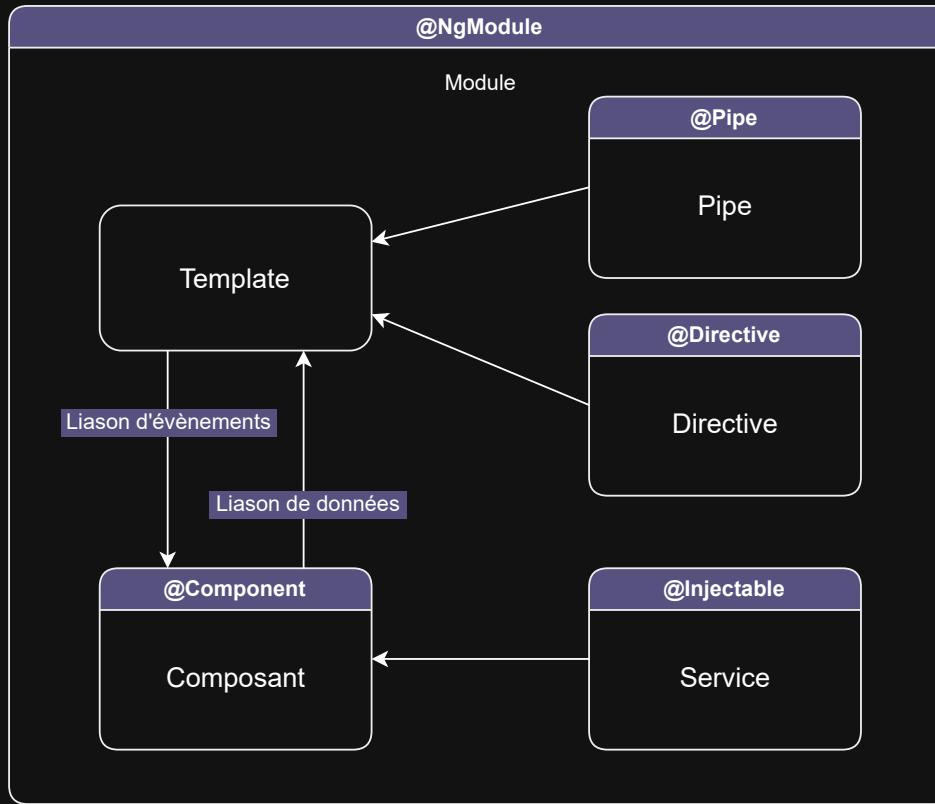
index.html

Comparaison du index.html du projet Angular et du code source du site web généré

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>FormationApp</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

```
<html lang="en">
  <head>
    <script type="module" src="/@vite/client"></script>
    <meta charset="utf-8">
    <title>FormationApp</title>
    <base href="/">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="icon" type="image/x-icon" href="favicon.ico" style="<...>">
    <link rel="stylesheet" href="styles.css" style="<...>">
  </head>
  <body>
    <app-root _ngcontent-ng-c1593039801 ng-version="17.3.5">
      <div _ngcontent-ng-c1593039801 class="content" flex="<...>">
        <h1 _ngcontent-ng-c1593039801>Torticols</h1>
        <h3 _ngcontent-ng-c1593039801>Liste des colis</h3>
        <input _ngcontent-ng-c1593039801 type="hidden" style="<...>">
        <div _ngcontent-ng-c1593039801 class="card-container" flex="<...>">
          <h3 _ngcontent-ng-c1593039801>Details</h3>
          <div _ngcontent-ng-c1593039801 class="box" ng-reflect-ng-switch="<...>">
            </div>
        </div>
      </app-root>
      <script src="polyfills.js" type="module"></script>
      <script src="main.js" type="module"></script>
    </body>
  </html>
```

Classes Angular



Modules

- Un module est un regroupement des autres classes Angular
 - Les composants, directives et pipes sont déclarés dans des modules
 - Les modules peuvent exporter une partie de leurs déclarations pour les autres modules
 - Chaque application Angular possède au moins un module, le module racine, utilisé par Angular pour lancer l'application
-
- Depuis Angular 14 et les composants standalones, il est possible de faire une application entièrement sans modules
 - A partir de la version 17, les composants sont standalone par défaut

app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';

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

app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';

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

Décorateur @NgModule

- Indique à Angular que la classe AppModule est un module
- Utilisation de métadonnées pour paramétrier le module

app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';

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

Déclarations

- Déclaration de tous les composants, directives et pipes du module
- Les éléments déclarés sont ceux qui appartiennent à ce module, et non ceux exportés

app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';

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

Modules importés

- Le module BrowserModule contient les services essentiels pour que l'application fonctionne sur navigateur
- Réexporte le module CommonModule, qui contient les directives et pipes basiques de Angular
- Il est également possible d'importer des composants standalone

app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';

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

Composant racine de l'application

- Au démarrage de l'application, Angular recherche dans le DOM, le premier élément correspondant au sélecteur du composant racine et l'injecte à cet endroit

Composants

- Brique de base d'une application Angular
- A chaque composant est assigné un template

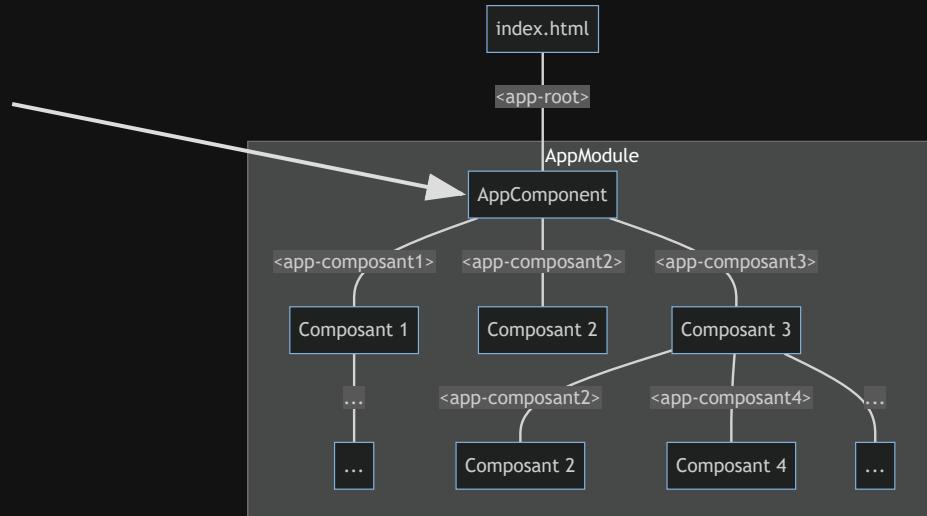
Composant

- Fichier Typescript
- Détermine le comportement du template

Template

- Fichier HTML avec des additions propres à Angular
- Peut avoir son propre fichier de style
- Informe le composant des actions utilisateur

Découpage en composants



- Une application Angular se présente sous la forme d'une arborescence de composants, et contient au moins un composant, le composant racine
- Chaque composant est assigné à un selecteur
- A chaque fois que le selecteur apparaît dans un template, Angular crée une nouvelle instance du composant et l'injecte à cet endroit

Exemple

YouTube

Rechercher

Accueil

Tous Jeux vidéo Squeezie Musique En direct Comédie à sketches Actualités Débats Chill-out Histoire Musiques d'ambiance Émissions de cuisine Travaux d'intérieur Destinations touristiques Jeux d'action et d'aventure Comédies Arts visuels Voitures Vidéos >

Se connecter

Accueil

Shorts

Abonnements

Bibliothèque

Historique

Connectez-vous à YouTube pour cliquer sur 'J'aime', ajouter un commentaire et vous abonner.

Lamborghini vs. Déchiqueteuse

71 M de vues • il y a 3 jours

Les amours de vacances - La chronique de Douilly

France Inter • 122 k vues • il y a 1 jour

Le premier jour - Palmashow

Palmashow • 2 M de vues • il y a 3 mois

ACTU

Un taureau sur le siège passager

Le Parisien • 0:54

États-Unis : arrêté au volant de sa voiture avec un taureau d'une tonne su...

Le Parisien • 293 k vues • il y a 4 jours

JE GRIND LA RANKED ! (Fortnite)

Gotagatv • 18 k vues • il y a 4 heures

TIKTOK MA FAIT ACHETER ÇA ! #4 (et c'est dingue !!)

Atelier de Roxane • 351 k vues • il y a 4 jours

J'AI ACHETÉ UN VRAI BUNKER (on survit dedans pendant 24h)

Amikem • 3.9 M de vues • il y a 1 mois

Musique Relaxante pour Dormir, Méditer et Détresser • "Flying" par Peder B....

Soothing Relaxation • 403 M de vues • il y a 7 ans

RÉSUMÉ : La Lazio et Mattéo Guendouzi font chuter Naples au Maradona !

belin SPORTS France • 493 k vues • il y a 3 jours

on piège des abonnés avec nos sosies

SQUEZZIE • 8,8 M de vues • il y a 1 an

Discours de Florian Philippot au Congrès d'Arras !

FLORIAN PHILIPPOT • 71 k vues • Diffusé il y a 3 jours

J'ai Brisé 100 MYTHES Interdits de Minecraft JAVA vs BEDROCK !!

LINED • 63 k vues • il y a 17 heures

AliExpress

FastGoodCuisine • 26.49

Le Secret du Roi Salomon | Film Complet en Français | Fantastique...

BoxOffice | SCIENCE-FICTION | Films Co... • 296 k vues • il y a 9 jours

MYTHO PAS #4 : LA POLICE S'INVITE, ON TESTE LEURS MAX à...

Bodytime • 119 k vues • il y a 5 jours

Si Vous Construisez Une Maison, Je Paierai !

MrBeast Gaming • 140 M de vues • il y a 2 ans

誰よりも働くスーパーお母さん！！神様に選ばれた綺麗な人が営む東京町中...

うどんそん 関東 Udonsoba • 61 k vues • il y a 2 jours

Gaspard Proust face à Manuel Bompard : "Ici, c'est la tour du Mordor pour lui"

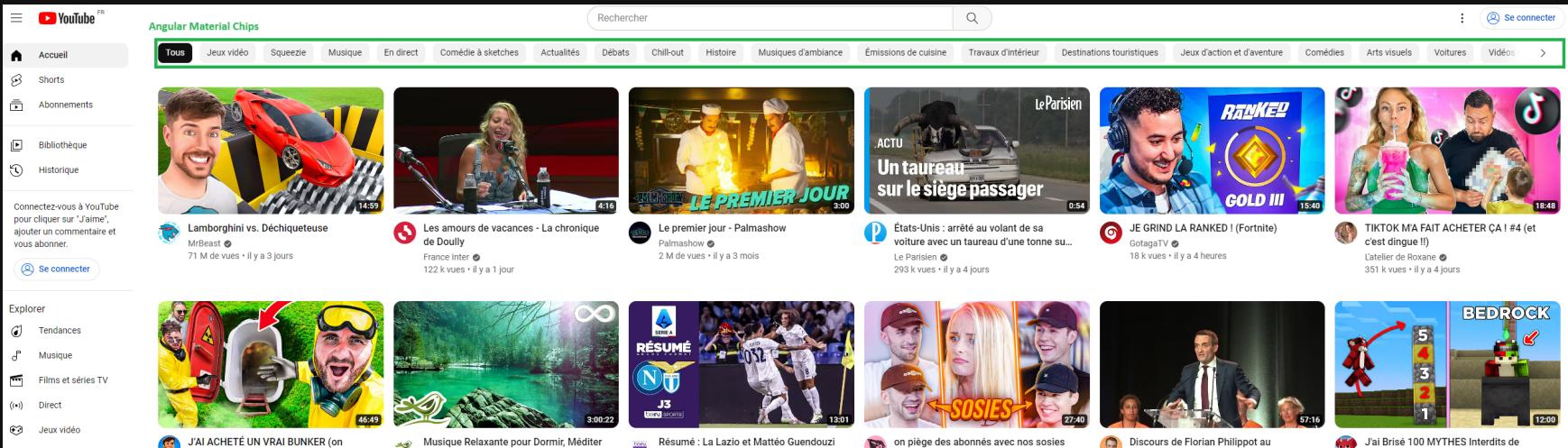
Europe 1 • 7,5 k vues • il y a 2 heures

Rechercher

Paramètres

Désactiver les cookies

Utilisation de bibliothèques



- La bibliothèque Angular Materials fournit des composants réutilisables
- La SNCF possède sa propre bibliothèque de composants : WCS

app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Torticolis';
}
```

app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Torticolis';
}
```

Décorateur @Component

- Indique à Angular que la classe AppComponent est un composant
- Utilisation de métadonnées pour paramétriser le composant

app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Torticolis';
}
```

Sélecteur CSS du composant

Exemple dans index.html

```
...
<body>
  <app-root></app-root>
</body>
...
```

app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Torticolis';
}
```

Template HTML associé au composant :

- Sous la forme d'un fichier (comme dans cet exemple)
- Sous forme de texte, dans ce cas utiliser l'attribut template à la place

app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Torticolis';
}
```

Fichier de style utilisé :

- Comme pour le template, soit texte soit fichier
- style et styleUrls sont des tableaux, donc plusieurs entrées possibles
- Par défaut le style est propre au composant
- Un fichier styles.css est présent à la racine pour les styles globaux au projet

app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Torticolis';
}
```

Code du composant

app.component.html

```
<div class="content">

<h1>{{ title }}</h1>

<!-- Next Steps -->
<h3>Liste des colis</h3>

<input type="hidden" #selection>

<div class="card-container">
  <button class="card" (click)="selection.value = 'D1AB7E3C-5D23-D646-4375-9B7872B52289'" tabindex="0">
    Lampe
  </button>

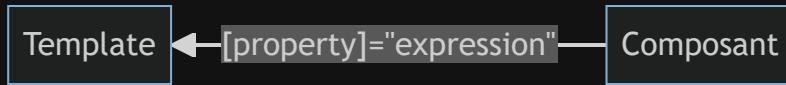
  <button class="card" (click)="selection.value = 'C9CE1525-59A7-577D-EE9E-B29065C0EB63'" tabindex="0">
    Chaise
  </button>
  ...
</div>
```

Interpolation



- Dans l'exemple précédent, `{{ title }}` est remplacé par Torticolis
- Il y a l'intérieur de l'interpolation une expression de template qui est évaluée et convertie en string
- Il est possible d'accéder aux propriétés et méthodes du composant dans l'expression
- **⚠ Les méthodes dans les expressions doivent être simples, prévisibles et sans effet de bord**
- **⚠ Aux interpolations avec les tableaux et objets**

Property binding



- La syntaxe [] indique à Angular de lier une expression avec une propriété d'un élément du template
- Le lien se fait uniquement du composant vers le template
- Si la propriété du composant change, le template sera modifié en conséquence
- ⚠ Mais pas l'inverse

Template

```
...  
<img [src]="image" />  
...
```



Composant

```
...  
image = 'https://t.ly/B0f_Z'  
...
```

Attribute binding

- **⚠️** Les attributs HTML sont différents des propriétés du DOM
- Il est possible de lier à un attribut, en préfixant avec "attr"
- Exemple :

```
<input [disabled]="condition">  
<input [attr.disabled]="condition ? 'disabled' : null">
```

- Les deux input ont le même comportement (disabled si condition est vraie)
- **⚠️** La propriété disabled attend un booléen, alors que l'attribut disabled attend un string
- **⚠️** Tous les attributs n'ont pas de propriété correspondante, et inversement
- **⚠️** Il peut y avoir des différences de comportement entre attribut et propriété (value d'un input)
- Privilégiez l'utilisation des propriétés

Class / Style binding

Classe

```
<div [class.class1]="boolean"> ... </div>
```

```
<div [class]="classExpression"> ... </div>
```

```
classExpression = 'class1 class2'
```

```
classExpression = { class1: true, class2: false }
classExpression = ['class1', 'class2']
```

- Ici le div aura la classe "class1" si l'attribut "boolean" du composant est truthy
- classExpression peut être sous forme de string, de tableau ou d'objet

Style

```
<div [style.fontSize]="fontSize"> ... </div>
<div [style.fontSize.px]="size"> ... </div>
```

```
<div [style]="styleExpression"> ... </div>
```

```
styleExpression = 'color: blue; font-size: 50px'
styleExpression = { color: 'blue', 'font-size': '50px' }
```

- Il est possible de lier un style qui a une unité avec un nombre
- styleExpression peut être sous forme de string ou d'objet
-  Même problème pour les tableaux et objets que l'interpolation

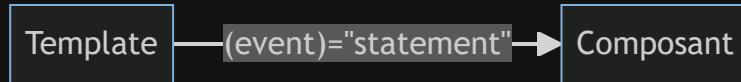
Template expression

- Une expression de template est très similaire à du JavaScript, avec quelques exceptions
 - Il n'est pas possible de faire des assignements (`=`, `+=`, `-=`, ...) ni d'incrément/décrément (`++/- -`)
 - Les opérateurs binaires (`|`, `&`) sont interdits
 - Il n'est pas possible de chainer les expressions avec ; or ,
 - Pas de `new`, `typeof` ou `instanceof`
 - L'opérateur binaire `|` (OR) de JavaScript est remplacé par le pipe
 - La fonction `$any()` pour cast une expression en any
-  Privilégier des expressions simples, avec la logique dans le composant

Exemples

```
 {{ 120 + 10 }} 
```

Event binding



- Lie un évènement avec un template statement

```
<button (click)="i=i+1">Incrémenter</button>
```

- Il est possible d'utiliser les méthodes du composant

```
<button (click)="reset()">Réinitialiser</button>
```

- Il est possible de récupérer des informations sur l'évènement grâce à \$event

```
<input (input)="logInput($event)" />
```

```
logInput(event: Event){  
  console.log(event);  
}
```

- Le type de l'objet \$event dépend du type d'évènement, qui sont basés sur l'interface Event
- <https://developer.mozilla.org/fr/docs/Web/API/Event>

Template statement

- Comme les expressions, les templates statements sont proches du JavaScript, avec les différences suivantes :
 - L'assignement basique (=) est permis
 - Le chainage des expressions avec ; ou , est permis
 - La fonction \$any() est autorisée
 - new, typeof et instanceof sont interdits
 - Les opérateurs (+=, -=, ++ et --) sont interdits
 - Les opérateurs binaires sont interdits
 - L'opérateur pipe est interdit

Exemples

```
<button (click)="120 + 10">Test</button>
```

Directives

- Les directives sont des classes Angular, avec le décorateur @Directive
- Elles sont rattachées à des éléments HTML par un sélecteur, et modifient le comportement de l'élément
- Les “Structural Directive” ajoutent ou suppriment des éléments du DOM

```
<ng-container *ngIf="monTableau.length else empty">
```

- Les “Attribute directive” modifient l'apparence ou le comportement d'un élément du DOM

```
<div [ngStyle]="styleExpression"> ... </div>
```

- Les composants sont également des directives

```
export declare interface Component extends Directive {
```

- Des directives sont fournies par Angular, mais il est possible de créer ses propres directives

Structural Directives (*ngIf, *ngFor, *ngSwitch)

- Les directives structurelles ajoutent ou suppriment des éléments du DOM
- Les balises *ngIf, *ngFor, *ngSwitch sont très similaires aux structures de contrôle (if, for et switch) que l'on peut utiliser dans d'autres langages
- Le * indique une syntaxe raccourcie, par exemple :

```
<div *ngIf="condition">Texte affiché si la condition est vraie</div>
```

- Est en réalité un raccourci pour :

```
<ng-template [ngIf]="condition">
  <div>Texte affiché si la condition est vraie</div>
</ng-template>
```

- La balise <ng-template> est une balise utilisée par Angular pour porter des directives structurelles, sans ajouter d'élément au DOM
- <ng-template> n'est pas affichée par défaut

*ngIf

- Affiche l'élément uniquement si la condition est vraie

```
<div *ngIf="condition">Texte affiché si la condition est vraie</div>
```

- La balise <ng-container> permet de contenir un *ngIf sans créer un élément

```
<ng-container *ngIf="condition">Texte affiché si la condition est vraie<ng-container>
```

- Il est possible de faire un "else" en utilisant un <ng-template>

```
<ng-container *ngIf="loaded; else loading">
  ...
</ng-container>
<ng-template #loading>
  L'application est en cours de chargement
</ng-template>
```

*ngFor

- L'élément à l'intérieur du *ngFor va être dupliqué pour chaque item du tableau monTableau

```
<div *ngFor="let item of monTableau">{{item.name}}</div>
```

- Il est possible d'utiliser les propriétés suivantes à l'intérieur d'un *ngFor
 - index (number) : index de l'objet en cours
 - count (number) : taille totale de l'iterable
 - first (boolean) : vrai si l'objet est le premier de l'iterable
 - last (boolean) : vrai si l'objet est le dernier de l'iterable
 - even (boolean) : vrai si l'index de l'objet est pair
 - odd (boolean) : vrai si l'index de l'objet est impair

```
<div [class.red]="odd" *ngFor="let item of monTableau; index as i; let tailleTotale=count; odd as odd">  
    {{item}} ({{ i }}) / {{ tailleTotale }}  
</div>
```

Built-in control flow

- Depuis la version 17, il est possible de créer des blocs structurels directement dans le template
- Par exemple, avec le @if :

```
<ng-container *ngIf="loaded; else loading">
  ...
</ng-container>
<ng-template #loading>
  L'application est en cours de chargement
</ng-template>
```

```
@if(loaded) {
  ...
} else {
  L'application est en cours de chargement
}
```

- Il est possible de faire un else facilement sans passer par un ng-template
- Il est également possible de faire un else if

- Avec un @for :

```
<div *ngFor="let item of monTableau">{{item.name}}</div>
```

```
@for(item of monTableau; track item.id) {  
  <div>{{item.name}}</div>  
} @empty {  
  <div>Le tableau est vide</div>  
}
```

- Le bloc @for à des meilleures performances que le *ngFor
- Le track, qui remplace le trackBy, est obligatoire

Attributes Directives (ngModel, ngStyle, ngClass)

- Les directives d'attributs modifient l'aspect et le comportement des éléments du DOM

```
<div maDirective
```

- [ngStyle] et [ngClass] ont le même comportement que [style] et [class], mais détectent le changement du contenu d'un objet ou tableau

```
<div [ngClass]="classExpression"[ngStyle]="styleExpression"
```

- [(ngModel)] lie une propriété du composant à un input et le modifie lorsque la valeur de l'input change

```
<input [(ngModel)]="data" />
```

- ⚠ Ne pas oublier l'import de FormsModule pour utiliser ngModel
- Notez l'écriture de [(ngModel)], () à l'intérieur de [], aussi connu sous le nom de 'banana-in-a-box syntax'

Exercice

- Le but de l'exercice est de créer un petit lecteur vidéo :
 - Un clic sur la vidéo lance ou pause la vidéo
 - On peut faire défiler la vidéo avec la molette
 - On peut déplacer la vidéo sur la page
- https://developer.mozilla.org/en-US/docs/Web/API/HTML_DOM_API
- <https://developer.mozilla.org/fr/docs/Web/API/Event>
- <https://developer.mozilla.org/en-US/docs/Web/CSS/transform>

Template variables

- Dans le template, il est possible d'utiliser le symbole # pour déclarer une variable de template

```
<input #classe [value]="classeString" />
<input #style [value]="styleString" />
```

- La variable référence l'élément, et peut être utilisée de partout dans le template

```
<div>La valeur de mon input est {{ classe.value }}</div>
<button (click)="maj(classe.value, style.value)">Mise à jour</button>
```

- Le type de cette variable dépend d'où elle est déclarée
 - Pour un élément HTML standard, l'interface correspondante (HTMLInputElement pour un input)
 - Pour un composant, le type du composant
- https://developer.mozilla.org/en-US/docs/Web/API/HTML_DOM_API

Priorité et portée des variables

- Les variables de templates sont globales dans le template du composant, sauf celles à l'intérieur d'une directive structurelle
- Une variable de template peut être utilisée sur plusieurs éléments, dans ce cas la première sera prioritaire
- Les variables de template peuvent avoir le même nom que des attributs du composant
- Si plusieurs variables ont le même nom dans le template, la priorité suivante est appliquée
 - Variables définies dans une directive structurelle
 - Variable de template
 - Attribut du composant
-  Utiliser des noms différents pour les variables

Standalone components

```
@Component({
  standalone: true,
  imports: [CommonModule, OtherStandaloneComponent],
  selector: 'standalone-component',
  template: ' ... ',
  styles: [ ... ]
})
export class StandaloneComponent {
```

- Depuis Angular 14, il est possible de créer des composants standalone (ainsi que des directives et pipes), sans passer par des modules
- Dans le décorateur, il suffit d'ajouter le flag `standalone` à vrai
- Il faut également spécifier les dépendances directement dans le composant
- Si l'on souhaite réutiliser le composant standalone, on peut l'importer directement (dans un module ou directement dans un autre composant standalone)

Standalone components

```
bootstrapApplication(StandaloneComponent).catch(err => console.error(err));
```

- On peut bootstrap directement sur un composant standalone

Pipes

Pipes



- Un pipe est une classe Angular, qui possède une méthode de transformation
- Un pipe est utilisée dans le template pour transformer une expression en une autre
- Les pipes sont utilisés avec l'opérateur |

```
 {{ entrée | pipe }}
```

Pipes



- Un pipe est une classe Angular, qui possède une méthode de transformation
- Un pipe est utilisée dans le template pour transformer une expression en une autre
- Les pipes sont utilisés avec l'opérateur |

```
 {{ entrée | pipe | secondPipe }}
```

- Il est possible de faire des enchainements de pipes, où la sortie d'un pipe devient l'entrée de la suivante

Créer un pipe

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'exponential',
  standalone: true
})
export class ExponentialPipe implements PipeTransform {

  transform(value: number, exponent = 1): number {
    return Math.pow(value, exponent);
  }
}
```

Créer un pipe

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'exponential',
  standalone: true
})
export class ExponentialPipe implements PipeTransform {

  transform(value: number, exponent = 1): number {
    return Math.pow(value, exponent);
  }
}
```

Décorateur @Pipe

- Indique à Angular que la classe est un pipe
- Utilisation de métadonnées pour paramétriser le pipe
- Un pipe peut être standalone

Créer un pipe

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'exponential',
  standalone: true
})
export class ExponentialPipe implements PipeTransform {

  transform(value: number, exponent = 1): number {
    return Math.pow(value, exponent);
  }
}
```

Sélecteur CSS du pipe

```
...
{{ 12 | exponential:2 | exponential | exponential:3 }}
...
```

Créer un pipe

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'exponential',
  standalone: true
})
export class ExponentialPipe implements PipeTransform {

  transform(value: number, exponent = 1): number {
    return Math.pow(value, exponent);
  }
}
```

Implémentation :

- Un pipe implémente l'interface PipeTransform
- La méthode transform est une fonction variadique, et a au moins un paramètre, l'expression à transformer
- **⚠️** Attention au type d'entrée de sortie des pipes

Exercice

- Créer un pipe qui tronque une chaîne de caractères trop longue, et ajoute "..." à la fin
- La taille maximale de la chaîne en entrée est paramétrable, avec une valeur par défaut de 10

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'tronque',
  standalone: true
})
export class TronquePipe implements PipeTransform {

  ...
}
```

Pipes Angular

- Un certain nombre de pipes sont disponibles dans le package CommonModule :
- UpperCasePipe (uppercase) et LowerCasePipe (lowercase) : Convertit la valeur respectivement en majuscules ou minuscules
- DatePipe (date) : Formatte une date
- DecimalPipe (number), PercentPipe (percent), CurrencyPipe (currency) : Formatte un nombre
- JsonPipe (json) : Convertit un objet sous forme JSON (utile pour débug)
- AsyncPipe (async) : Souscrit à un Observable ou une Promise et retourne la dernière valeur émise
- <https://angular.io/api?type=pipe>

Pourquoi les pipes ?

Performance

- Angular execute le pipe uniquement lorsque l'expression d'entrée change

```
 {{ toUpperCase(texte) }}
```

```
 {{ texte | toUpperCase }}
```

Lisibilité

- En particulier lors de chaînage de pipes, l'expression finale est bien plus lisible

```
 {{ toUpperCase(lowercase(format(transformation(texte)))) }}
```

```
 {{ texte | transformation | format | lowercase | toUpperCase }}
```

Pipe pur et impur

- La fonction de transformation du pipe doit être pure
 - La valeur de sortie doit dépendre uniquement des valeurs d'entrée
 - La fonction ne modifie pas de valeurs en dehors de ses variables locales
- **⚠ Utilisation du pipe avec les tableaux et les objets**
- Un pipe est pur par défaut, mais il est possible de le rendre 'impur'

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

- **⚠ Angular ne 'devine' pas si le pipe est pure ou impur, c'est à vous de lui indiquer selon l'usage**
- Un pipe impur n'a plus l'optimisation du pipe pur, et sera executé aussi souvent qu'une méthode

Pipes et objets

The screenshot shows a component with a dark background and a light gray header bar. The header bar contains the text "Ajouter" and "Supprimer". Below the header is a list of five items, each consisting of a number and a button labeled "Incrément index". The numbers are 0, 1, 2, 3, and 4. At the bottom of the list, there is a text element "Somme des éléments : 10".

0	Incrément index 0
1	Incrément index 1
2	Incrément index 2
3	Incrément index 3
4	Incrément index 4

Ajouter Supprimer Somme des éléments : 10

- Ce composant affiche les éléments d'un tableau
- Il est possible d'incrémenter les éléments, et d'ajouter ou supprimer un élément
- La somme totale des éléments est affichée, en utilisant un pipe
- Problème : La somme n'est pas mise à jour si l'utilisateur modifie les éléments

Solution 1

- Utiliser un pipe impure

```
@Pipe({  
  standalone: true,  
  name: 'sumPipe',  
  pure: false  
})
```

- ✓ Solution simple
 - ✗ Peut impacter les performances si l'opération est complexe (tri, ...)
-
- Exercice : trouver une solution sans passer par un pipe impure

Solution 2

- Forcer le changement de référence manuellement (le tableau devient immuable)

```
increment(index: number): void {
  this.tableau = [...this.tableau]
  this.tableau[index] += 1
}

addElement(): void {
  this.tableau = [...this.tableau]
  this.tableau.push(this.tableau.length)
}

removeElement(): void {
  this.tableau = [...this.tableau]
  this.tableau.pop()
}
```

- ✓ Meilleures performances : le pipe est appelé uniquement lorsque la valeur change
- ✗ Il faut faire la modification si l'on ajoute une opération, ou si on réutilise le pipe ailleurs
- ✗ Le tableau est recréé à chaque fois, et il est reparcouru entièrement pour calculer la somme

Solution 3

- Déléguer la logique au composant, le pipe n'est plus utilisé

```
Somme des éléments : {{ somme }}
```

```
somme: number = this.tableau.reduce((a, b) => a + b, 0)

...

increment(index: number): void {
  this.tableau[index] += 1
  this.somme += 1
}

addElement(): void {
  this.somme += this.tableau.length
  this.tableau.push(this.tableau.length)
}

removeElement(): void {
  let n = this.tableau.pop()
  if(typeof n === 'number') {
    this.somme -= n
  }
}
```

✓ Meilleures performances

✗ Il faut implémenter la logique de la somme dans le composant

- Il n'y a pas de méthode meilleure dans tous les cas de figure
- C'est à vous de vous adapter selon l'usage

Découpage en composants

@Input

```
@Input({  
  alias: 'alias',  
  required: true,  
  transform: (input: number) => input * 10  
})  
attribut: number = 0
```

- Le décorateur @Input lie l'attribut d'un composant à une propriété de son élément

```
alias?: string
```

- Par défaut, la propriété liée à le même nom que l'attribut. L'alias permet de définir une propriété différente (ici alias)

```
<app-fils [alias]="value" > </app-fils>
```

@Input

```
@Input({  
  alias: 'alias',  
  required: true,  
  transform: (input: number) => input * 10  
})  
attribut: number = 0
```

- Le décorateur @Input lie l'attribut d'un composant à une propriété de son élément

```
required?: boolean
```

- Depuis Angular 16, il est possible d'obliger le binding de la propriété
- Si required est à true, une erreur sera renvoyée à la compilation si l'on essaye de créer le composant sans lier la propriété

```
<app-fils></app-fils> ←!— ici erreur de compilation —→
```

@Input

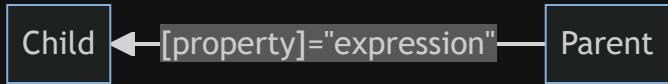
```
@Input({  
  alias: 'alias',  
  required: true,  
  transform: (input: number) => input * 10  
})  
attribut: number = 0
```

- Le décorateur @Input lie l'attribut d'un composant à une propriété de son élément

```
transform?: (value: any) => any
```

- Appelle un callback pour transformer la propriété en entrée avant de l'assigner à l'attribut
- Ici la fonction multiplie simplement l'entrée par 10, mais il est possible de changer le type d'entrée
- **⚠** Comme souvent avec les décorateurs, le compilateur ne vérifie pas que le type de retour de vote callback corresponde à celui de l'attribut à lier

Input



```
<app-fils [alias] = "value" > </app-fils>
```

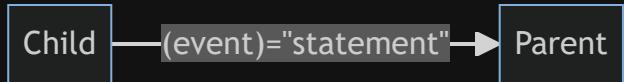
- La syntaxe dans le template du père est la même que pour se lier à une propriété
- **⚠** Comme pour le property binding, un changement de la propriété chez le fils n'est pas répercuté chez le père

@Output

```
@Output({  
  alias: 'alias',  
})  
event: EventEmitter<number> = new EventEmitter()  
...  
...  
event.emit(value)  
...
```

- Le décorateur @Output permet au composant fils d'émettre des évènements
- Il est possible de définir un alias comme pour l'input
- La variable doit être du type EventEmitter<T>, avec T le type de valeurs à émettre
- Chaque appel à emit() dans le composant fils envoie un évènement dans le composant père

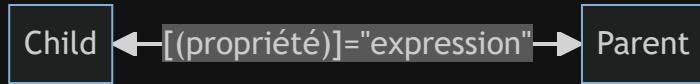
Output



```
<app-fils (event)="value=$event" > </app-fils>
```

- La syntaxe dans le template du père est la même que pour se lier à un évènement d'un élément HTML
- La variable \$event permet de récupérer directement la valeur renvoyée par le composant fils

Two-way binding



- La donnée peut être modifiée par le père et le fils
- Combinaison d'une property et d'un event binding, avec une syntaxe raccourcie

```
<app-fils [(value)]="value"> ... </app-fils>
<app-fils [value]="value" (valueChange)="value=$event"> ... </app-fils>
```

- C'est ce qui est fait lors de l'utilisation de la directive ngModel

```
<input [(ngModel)]="data" />
<input [ngModel]="data" (ngModelChange)="data=$event" />
```

@Input et @Output

- Dans le composant fils, une variable @Input x doit avoir comme @Output xChange

```
...
@Input()
value: number = 0

@Output()
valueChange: EventEmitter<number> = new EventEmitter()
...
```

- ⚠ Il est nécessaire d'appeler manuellement la méthode emit() dans le composant fils

```
...
addValue() {
  this.value++
  this.valueChange.emit(this.value)
}
...
```

Exercices (1/2)

- Refactoriser l'application de démo, en créant un composant pour la liste des topics, et un composant pour la description

Exercices (2/2)

- Partant de la classe suivante :

```
class Tree {  
    constructor(public value: string, public children: Tree[] = []) {}  
}
```

- Créer un composant permettant d'afficher une arborescence :
- On peut afficher ou cacher les fils d'un élément de l'arbre en cliquant dessus

Recursive Component

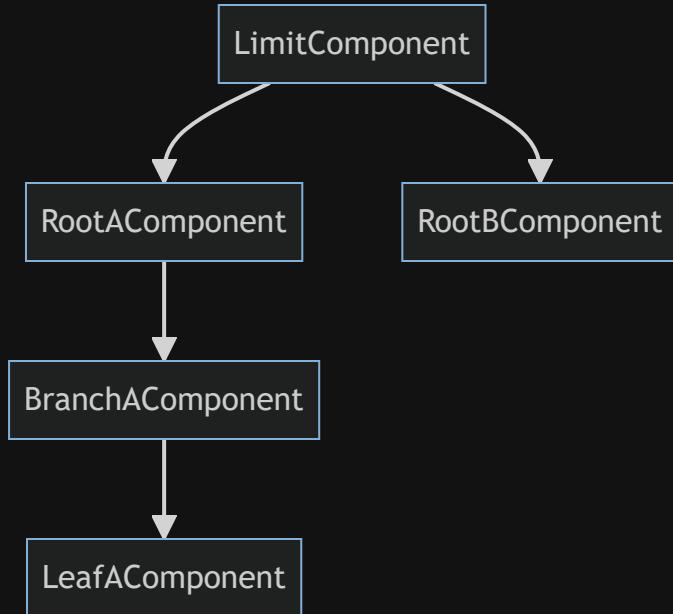
root

 folder

 readme

Limites

- Considérons l'arborescence suivante :



Services et injection de dépendances

Injection de dépendances

- L'injection de dépendances est un des principes fondamentaux d'Angular, qui met en relation des consumers (qui ont besoin d'une ressource) et des providers (qui fournissent une ressource)
- Les providers sont des services
- Les consumers peuvent être des composants, pipes, directives ou d'autres services
- L'injecteur gère une liste de providers, sous forme de singletons, et les fournit aux consumers qui en ont besoin

Services

- Un service est simplement une classe, annotée avec le décorateur `@Injectable()`

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

Utilisaton

- Dans le constructeur

```
constructor(public myService: MyService) { ... }
```

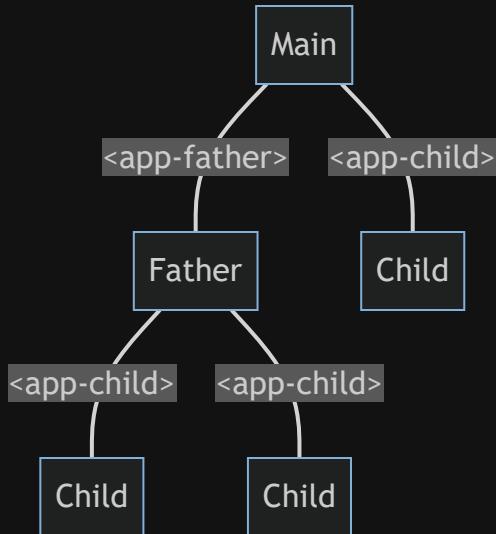
- Depuis Angular 14, avec la fonction inject()

```
myService = inject(MyService)
```

- Notre composant peut maintenant utiliser le service
- Pas besoin d'instancier myService, c'est l'injecteur qui s'en occupe pour nous

Définition des providers

- Pour la suite, considérons l'arborescence de composants suivante :



- Un service CounterService permet de stocker une valeur numérique, et de l'incrémenter
- Le composant father à une dépendances vers ce service, et le composant child deux

Définition des providers

- Une application Angular comporte plusieurs injecteurs, avec chacun une liste de providers
 - root : unique pour l'application*
 - ElementInjector : spécifique pour un composant (ou une directive)
- *En réalité, l'injecteur root est un EnvironmentInjector (anciennement ModuleInjector), et il y a plusieurs EnvironmentInjector dans une application Angular

root

- On peut déclarer un service dans l'injecteur root dans les métadonnées de l'annotation @Injectable

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

- On peut également les définir dans les providers des modules

```
@NgModule({  
  ...  
  providers: [CounterService],  
})  
export class AppModule { }
```

- Pour une application standalone, on peut le définir dans bootstrapApplication()

```
bootstrapApplication(RootComponent, {  
  providers: [ CounterService ]  
}).catch(err => console.error(err));
```

Commentaires

- Pour les services que l'on crée, et que l'on souhaite inclure dans l'injecteur root, il est préférable de le déclarer dans le décorateur (avec providedIn: 'root').
- Tous les providers de vos modules importés se retrouvent dans l'injecteur root (sauf dans le cas du lazy-loading), les providers d'un module ne sont donc pas limités aux composants de votre module

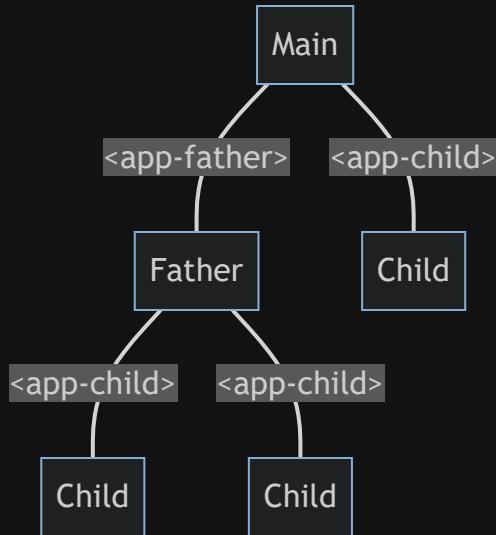
Element Injector

- Les composants et les directives ont également leur propre injecteur

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

- Dans ce cas, l'instance de CounterService est différente pour chaque composant
- L'injecteur est également accessible par les éléments fils
- Angular met à disposition des providers par défaut dans l'Element Injector, par exemple l'élément du DOM du composant, ou l'instance du composant lui-même

Priorité de résolution



- Lorsque un élément doit résoudre une dépendance, il cherche d'abord dans son propre injecteur
- Puis il remonte les injecteurs parents un à un jusqu'à tomber sur l'élément racine
- A la fin, il demande à l'injecteur root
- Si la dépendance n'est toujours pas résolue, une exception est levée

Token d'injection

```
providers: [  
  CounterService  
]
```

```
providers: [  
  { provide: CounterService, useClass: CounterService }  
]
```

- Ici, les deux écritures sont équivalentes
- L'injecteur d'Angular assigne un token (une clé) à chaque provider pour l'identifier
- Pour une classe, par défaut le token est la classe elle-même

Token d'injection

- Dans mon exemple, pour utiliser deux instances différentes du CounterService pour mon composant fils, il faut un moyen de distinguer les deux
- Il est possible (mais déconseillé) d'utiliser un string comme token
- Angular fournit une classe dédiée : `InjectionToken<T>`

```
const FIRST_TOKEN = new InjectionToken<CounterService>('first_token')

{ provide: FIRST_TOKEN, useClass: CounterService},
```

- Lorsque l'on instancie un token, il y a un paramètre obligatoire, une description.
- Ce paramètre ne sert que en débug (c'est cette description qui est affichée par exemple lors d'une dépendance manquante)

Token d'injection

- Pour utiliser le token :

```
@Inject(FIRST_TOKEN) public firstCounterService: CounterService,  
  
firstCounterService = inject(FIRST_TOKEN)
```

- `InjectionToken` est une générique, avec `T` le type du service auquel il est associé
- La fonction `inject()` est capable d'inférer le type de retour lors de l'utilisation d'un `InjectionToken`
- **⚠ Par contre avec le décorateur `@Inject`, c'est à vous de vous assurer du bon type**

useClass

```
providers: [
  { provide: CounterService, useClass: BetterCounterService }
]
```

- En modifiant useClass, il est possible de spécifier un autre service pour le même token
- Le token ne change pas, par contre une instance de BetterCounterService sera utilisée à la place de CounterService pour satisfaire la dépendance
-  C'est à vous de vous assurer que la classe BetterCounterService est compatible avec la classe originale

useExisting

```
export abstract class MinimalCounterService {  
    abstract value: number;  
}
```

```
{ provide: MinimalCounterService, useExisting: CounterService },
```

```
@Inject(MinimalCounterService) public firstCounterService: MinimalCounterService,
```

- useExisting permet de mapper un token vers un autre service qui existe déjà
- Ici mon service MinimalCounterService est une version réduite de CounterService, qui permet d'afficher la valeur mais pas de la modifier
- En utilisant useExisting, la recherche de dépendance va être déportée sur le token que l'on souhaite utiliser à la place : il faut s'assurer que ce nouveau token à un provider quelque part
- Utiliser useClass à la place de useExisting créerait une nouvelle instance de CounterService

useValue

```
const URL = new InjectionToken<string>('url')

{ provide: URL, useValue: 'http://localhost:4200/' },

@Inject(URL) public url: string,
```

- useValue permet d'associer une valeur fixe à un token
- useValue peut être utilisée pour associer un objet à un token, mais c'est à nous de l'instancier
- Peut être utilisée avec un InjectionToken pour définir des interfaces ou des types primitifs comme dépendances
- Peut également être utilisé lors des tests unitaires pour faire des mocks de services

useFactory

```
export const ENV = new InjectionToken<string>('Environment')
...
export function dataServiceFactory(environment: string): DataService {
  ...
}

{
  provide: DataService,
  useFactory: dataServiceFactory,
  deps: [ENV]
},
```

- useFactory permet de définir un callback pour instancier notre dépendance
- En utilisant deps, on peut spécifier des paramètres pour notre callback. deps est constitué de tokens qui sont résolus et passés dans l'ordre au callback
- On peut utiliser une factory pour fournir des implémentations différentes pour un même service, selon un paramètre (penser au pattern factory)

- Il est possible de définir un provider au niveau de l'environnement injector lors de la déclaration d'un token

```
export const HELLO_TOKEN = new InjectionToken<string>('bonjour', {  
  providedIn: 'root',  
  factory: () => 'Bonjour'  
})
```

- factory est un callback, qui est appelée pour instancier la dépendance
- Il n'est pas possible de spécifier des paramètres à factory

Modification de la recherche

- Du coté du consumer, il est possible de modifier la recherche de provider
- Cette modification se représente par des décorateurs (avec le constructeur) ou des options (avec inject)
- `@Optional()` : Retourne null plutot qu'une exception si la dépendance n'est pas résolue

```
@Optional() public firstCounterService: CounterService,  
  
firstCounterService = inject(CounterService, {optional: true} )
```

- `inject()` renvoie un union type `CounterService | null`

Modification de la recherche

- Du côté du consumer, il est possible de modifier la recherche de provider
- Cette modification se représente par des décorateurs (avec le constructeur) ou des options (avec inject)
- `@SkipSelf` : Saute l'élément courant

```
@SkipSelf() public firstCounterService: CounterService,  
firstCounterService = inject(CounterService, {skipSelf: true} )
```

Modification de la recherche

- Du côté du consumer, il est possible de modifier la recherche de provider
- Cette modification se représente par des décorateurs (avec le constructeur) ou des options (avec inject)
- `@Self` : Arrête la recherche à l'élément courant

```
@Self() public firstCounterService: CounterService,  
  
firstCounterService = inject(CounterService, {self: true} )
```

Modification de la recherche

- Du côté du consumer, il est possible de modifier la recherche de provider
- Cette modification se représente par des décorateurs (avec le constructeur) ou des options (avec inject)
- @Host : Arrête la recherche à l'élément hôte (pour les directives et contenu projeté)

```
@Host() public firstCounterService: CounterService,  
firstCounterService = inject(CounterService, {host: true} )
```

Exercices

- Sur le composant fils, faire en sorte que le deuxième compteur soit indépendant du premier
- Sur le composant fils, faire en sorte que le deuxième compteur incrémentente le compteur deux par deux
- Sur le composant père, utiliser le service minimal pour laisser uniquement la possibilité au compteur père de consulter le compteur
- Inverser la résolution de dépendances pour premier compteur du composant fils :
 - Un provider est cherché dans la hierarchie des composants en ignorant le provider du composant fils
 - Si aucun provider n'est trouvé utiliser un provider par défaut dans le composant fils
- Pour tester l'inversion, vous pouvez fournir uniquement un counterService dans le composant père, les deux fils du composant père vont partager le compteur, et le composant fils seul va avoir son propre compteur

Exercices

- Dans l'application de démo, créer un service ColisService qui mets à disposition la liste des colis

Style

Encapsulation de style

- Chaque composant à son propre style associé
- Il existe 3 stratégies d'encapsulation de style :
 - ViewEncapsulation.Emulated (par défaut)
 - ViewEncapsulation.ShadowDom
 - ViewEncapsulation.None
- La préconisation Angular est de considérer les styles comme des éléments privés du composant
- Il est déconseillé de combiner les différentes stratégies dans un même projet

ViewEncapsulation.Emulated

- A l'époque des premières version d'Angular, le ShadowDom n'était pas pris en charge par tous les navigateurs
- Angular a crée sa propre version simulée du ShadowDom
- Angular crée un attribut unique pour chaque composant
- Angular ajoute cet attribut dans chaque élément dans son template

```
<div _ngcontent-ng-c3975064098 class="toolbar">  
  ...  
</div>
```

- Un selecteur d'attribut est ajouté dans le fichier de style final

```
.toolbar[_ngcontent-ng-c3975064098] {  
  ...  
}
```

Pseudo-class selecteurs

- Les selecteurs :host et :host-context normalement utilisables avec un shadow DOM sont également utilisables avec l'encapsulation émulée d'Angular

:host

- Permet d'ajouter des styles au sélecteur du composant lui même

:host-context

- Donne la possibilité de définir des styles depuis le composant parent

::ng-deep

- Le sélecteur ::ng-deep permet d'appliquer un sélecteur pour toute l'application
- Combinée avec :host, le selecteur est appliquée uniquement pour le composant et ses fils
- Peut être utilisée pour modifier le style d'un composant externe
- Dépréciée depuis longtemps, mais sans alternative pour le moment

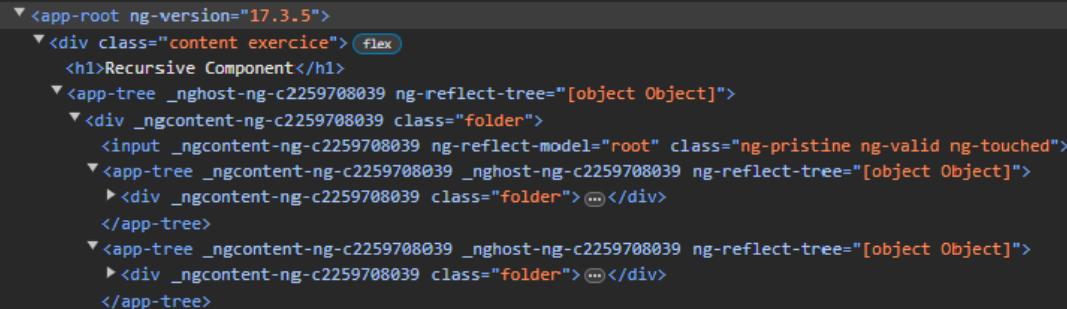
Host binding et Host listener

Host listener

- Retour sur le composant folder :

```
<div (click)="onClick($event)" class="folder">
  <input [(ngModel)]="tree.value" />
  @if(expanded) {
    @for (child of tree.children; track $index) {
      <app-tree [tree]="child"/>
    }
  }
</div>
```

- Angular génère un élément dans le HTML pour chaque composant, et ici un div qui encapsule tout le composant



```
▼ <app-root ng-version="17.3.5">
  ▼ <div class="content exercice" flex
    <h1>Recursive Component</h1>
    ▼ <app-tree _ngcontent-ng-c2259708039 ng-reflect-tree="[object Object]">
      ▼ <div _ngcontent-ng-c2259708039 class="folder">
        <input _ngcontent-ng-c2259708039 ng-reflect-model="root" class="ng-pristine ng-valid ng-touched">
      ▼ <app-tree _ngcontent-ng-c2259708039 _ngcontent-ng-c2259708039 ng-reflect-tree="[object Object]">
        ▶ <div _ngcontent-ng-c2259708039 class="folder">...</div>
        </app-tree>
      ▼ <app-tree _ngcontent-ng-c2259708039 _ngcontent-ng-c2259708039 ng-reflect-tree="[object Object]">
        ▶ <div _ngcontent-ng-c2259708039 class="folder">...</div>
        </app-tree>
```

- On a déjà vu qu'il est possible de spécifier du style directement sur les composants avec le sélecteur :host
- De manière similaire, on peut écouter des évènements directement sur l'élément du composant
- Pour cela on utilise la propriété host dans le décorateur @Component

```
@Component({  
  ...  
  host: {  
    '(click)': 'onClick($event)'  
  }  
})
```

- Il est aussi possible d'utiliser le décorateur @HostListener directement sur la méthode dans le composant

```
@HostListener('click', ['$event'])  
onClick(event: Event) {
```

- La préconisation d'Angular est d'utiliser host plutôt que le décorateur @HostListener

Host binding

- Il est également possible de faire du style/class binding avec l'élément hôte du composant
- Pour cela on utilise également la propriété host

```
@Component({  
  ...  
  host: {  
    'class': 'box',  
    '[style.fontSize.px]': 'fontSize',  
  }  
})
```

- Ou le décorateur @HostBinding

```
@HostBinding('style.fontSize.px')  
fontSize = 60  
  
@HostBinding('style.color')  
color = 'red'
```

Exercice

- Refaire le composant tree-composant en enlevant le div global

Organisation d'un projet Angular

Smart vs dumb components

- On distingue deux types de composants :
 - Des composants de présentation, ou “dumb components”
 - Des composants conteneurs, ou “smart components”
- Les composants de présentation s’occupent de l’affichage des données, et n’ont que pas connaissance du reste de l’application (ils communiquent avec des @Input ou @Output)
- Les composants de présentation peuvent être réutilisés facilement
- Les composants conteneurs ont connaissance de l’état de l’application, et donnent aux composants de présentation uniquement les données dont ils ont besoin

Bonnes pratiques SNCF

- <https://dev.sncf/docs/frontend/angular/>
- Les bonnes pratiques présentées sont celles de la SNCF, pas des vérités absolues !

Création d'un projet

```
ng new --skip-git --inline-style --inline-template --directory ./ --routing --skip-tests --style scss {nom-du-projet}-cl
```

inline-style et inline-template

- Lors de la création d'un nouveau composant avec ng generate , le style et le template sont dans le même fichier
- La préconisation de la SNCF est de faire des single file component

skip-tests

- ng generate ne génère pas les fichiers de tests

ng generate

- <https://angular.io/cli/generate>

- Permet de générer et modifier automatiquement les fichiers du projet Angular pour ajouter des nouveaux éléments (Module, Composant, ...)
- Nomme automatiquement les classes et les fichiers en suivant les bonnes pratiques (<https://angular.dev/style-guide>)

- Le comportement de ng generate dépend du fichier angular.json à la racine du projet

Organisation d'une application

- src/
 - app/ --Composants et services nécessaires au fonctionnement de l'application
 - assets/ --Contient les images, fonts, ressources de l'application
 - config/ --Fichier(s) de configuration qui seront utilisés par la shared lib de configuration
 - environments/
 - features/ --Contient les features modules
 - generated/ --Contient tous les fichiers générés par des outils de génération de code
 - shared/ --Contient tout ce qui est partagé à travers l'application
 - styles/

app/

 services/

 components/

 pipes/

 .../

 app.module.ts

 app-routing.module.ts

- Point d'entrée de l'application
- Contient les éléments nécessaires au démarrage de l'application

App module

- Importe d'autres features modules
- Importe le shared module ou des éléments standalone si besoin
- N'exporte rien

features/

```
feature/
  services/
  components/
  ...
  feature.module.ts
  feature.routing.module.ts
other-feature/
  services/
  components/
  ...
  other-feature.module.ts
  other-feature.routing.module.ts
```

- Chaque feature représente une partie de l'application

Feature module

- Le feature module est composé de tous les éléments spécifiques à une feature
- Le feature module est importé une seule fois, par le AppModule ou un autre feature module
- Importe le shared module ou des éléments standalone si besoin
- Importe d'autres features modules

shared/

```
components/  
enums/  
types/  
services/  
utils/  
...  
(shared.module.ts)
```

- Contient les éléments réutilisables de l'application
- Les composants réutilisés sont de bon candidats pour être standalone, ce qui évite de faire un shared.module

Shared module

- Exporte tous les éléments réutilisables de l'application
-  Ne pas fournir de providers dans le shared module

environments/

environment.ts

environment.development.ts

- Contient les fichiers spécifiques à un environnement (généralement des constantes)
- Il est possible de générer les fichiers et la configuration associée avec la commande

```
ng generate environments
```

- Exemple de fichier environment.ts

```
export const environment = {  
  production: true,  
};
```

- Et le fichier environment.development.ts associé

```
export const environment = {  
  production: false,  
};
```

environments/

- Pour l'utiliser, on importe le fichier original dans le reste du programme

```
import { environment } from './environments/environment';  
  
console.log(environment.production);
```

- Le fichier sera remplacé selon l'environnement

View queries

@ViewChild

- Retour sur les variables de template :

```
<audio #audio src=".. /assets/sample.mp3" ></audio>
<button (click)="audio.play()">Play</button>
<button (click)="audio.pause()">Pause</button>
```

- Dans certains cas, on veut pouvoir manipuler les éléments du template dans le composant
- Pour cela, Angular met à disposition les décorateurs d'attributs @ViewChild et @ViewChildren

```
@ViewChild('audio')
audioRef: ElementRef<HTMLAudioElement>
```

- ViewChild fait une requête sur le template, et assigne le premier résultat correspondant à l'attribut du template

- Il est possible de requérir :
- Un élément du dom avec une variable de template associée

```
@ViewChild('audio')  
audioRef: ElementRef<HTMLAudioElement>
```

- Un composant ou une directive

```
@ViewChild(ChildComponent)  
componentRef: ChildComponent
```

```
@ViewChild(HighlightDirective)  
directiveRef: HighlightDirective
```

- Un provider dans les composants ou directives fils

```
@ViewChild(CounterService)  
childServiceRef: CounterService
```

Quelques précisions sur ViewChild :

- ViewChild renvoie uniquement le premier résultat correspondant à la requête
- Sur le type du retour :
 - **⚠️** Comme souvent avec les décorateurs, pas de type-checking
 - Pour un composant ou une directive, le type de retour est la classe elle-même
 - Pour un élément HTML, le type de retour est l'interface correspondant encapsulée dans un `ElementRef<T>`
 - Si la requête n'a pas de résultat, le retour est `undefined`
- La requête n'est pas récursive : elle est limitée au template de composant
- Si au cours de la vie de votre composant le résultat de la requête change, la valeur de l'attribut change également

@ViewChildren

- Le fonctionnement de @ViewChildren est très similaire à @ViewChild, sauf qu'il renvoie tous les résultats de la requête dans une QueryList
- Il est possible de fournir plusieurs sélecteurs, séparés par une virgule
- Comme pour ViewChild, le contenu de la QueryList change avec le composant
- Il est possible de récupérer un observable sur une QueryList qui notifie les subscribers des changements

read

- La propriété `read` permet de changer le type de retour de la requête,
- Exemple pour avoir l'élément du DOM correspondant à un composant fils, plutôt que le composant lui-même

```
@ViewChild(ChildComponent, {read: ElementRef})
childRef!: ElementRef<HTMLElement>
```

- Ou une directive sur un composant en particulier

```
@ViewChild(ChildComponent, {read: HighlightDirective})
childDirectiveRef?: HighlightDirective
```

- **⚠️** Manipuler directement le DOM peut rendre votre application vulnérable

Exercice

- Coder un composant qui permet de choisir une piste audio parmi une liste
- Lorsque l'on change de piste, la lecture actuelle est remise à zero

static (ViewChild seulement)

- Le paramètre static dans le décorateur permet de changer le comportement de la view query
- La requête n'est effectuée qu'une seule fois, après l'initialisation du composant

Lifecycle Hooks

Cycle de vie des composants

- Considérons le composant suivant, qui prends une taille en @Input, et affiche un tableau de cette taille rempli de nombre aléatoires

```
export class TableauComponent {  
  
  @Input({required: true})  
  size!: number  
  
  numberArray: number[] = [ ...Array(this.size)].map(() => Math.round(Math.random()*10))  
  
}
```

- Un seul élément est crée dans le tableau, pourquoi ?

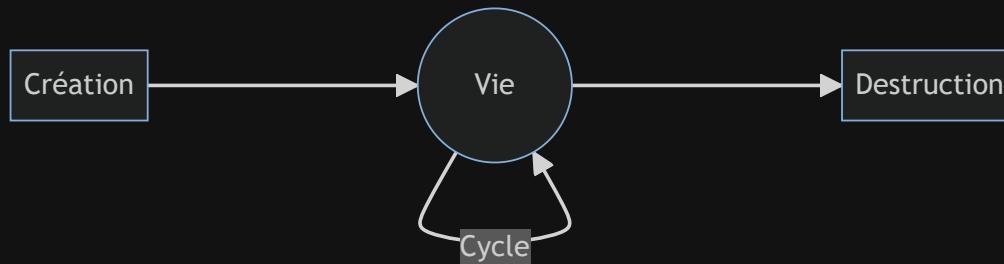
- Lors de l'appel au constructeur du composant, les bindings ne sont pas encore définis, this.size est donc undefined
- Autre exemple : avec une View Query

```
@ViewChild('audio')  
audioRef!: ElementRef<HTMLAudioElement>
```

- Dans cet exemple, audioRef est encore undefined dans le constructeur

```
constructor() {  
  this.audioRef.nativeElement.play()  
}
```

Cycle de vie des composants



- Le cycle de vie d'un composant Angular comporte trois parties :
 - Création du composant
 - Vie du composant (detection cycle)
 - Destruction du composant
- Pour chaque composant, il est possible de définir des hooks, des méthodes qui seront appelées automatiquement à des moments précis du cycle de vie
- Il est également possible de définir des hooks pour des directives, pipes et services

Liste des hooks

Création

- constructor()
- ngOnInit()
- ngAfterContentInit()
- ngAfterViewInit()

Vie

- ngOnChanges()
- ngDoCheck()
- ngAfterContentChecked()
- ngAfterViewChecked()

Destruction

- ngOnDestroy()

- Les hooks de vie du composant sont également appelés lors de la création
- A chaque hook est associé une interface

Hooks de création

ngOnInit()

- Appelé après que les bindings du composants aient été initialisés (et après le premier ngOnChanges)

ngAfterViewInit()

- Appelé une fois après que le template ait été initialisé

Detection de changement

- La vie d'une application Angular est rythmée par les cycles de détection de changement
- A chaque cycle de détection de changement, Angular met à jour l'affichage de l'application
- Un cycle de détection de changements se déclenche lors des évènements suivants :
 - Evènements du DOM surveillé par Angular
 - setTimeout() and setInterval()
 - Requêtes HTTP

ngOnChanges(changes: SimpleChanges)

- Appelé une première fois après que les attributs @Input soient initialisés
- Puis à chaque fois qu'un attribut @Input est modifié
- Un paramètre SimpleChanges permet de récupérer les changements

```
export declare interface SimpleChanges {
  [propName: string]: SimpleChange;
}

export declare class SimpleChange {
  constructor(previousValue: any, currentValue: any, firstChange: boolean);

  previousValue: any;
  currentValue: any;
  firstChange: boolean;
  isFirstChange(): boolean;
}
```

ngDoCheck()

- Appelé à chaque detection cycle
- Peut être utilisé pour détecter des changements que ngOnChanges ne prend pas en compte, comme des mutations d'objets
- Appelé avant qu'Angular ne mette à jour le template
- **⚠️** Eviter d'appeler des méthodes coûteuses

ngAfterViewChecked()

- Appelé à chaque detection cycle, une fois que le template ait été mis à jour
- **⚠️** Ne pas modifier la valeur des attributs interpolés dans les hooks ngAfterViewInit() et ngAfterViewChecked(), au risque d'avoir un état incohérent entre la valeur des attributs et ce qui est affiché à l'écran

Cycle de détection de changement

- Un évènement déclenche le cycle de détection de changement
- Parent ngOnChanges()
- Parent ngDoCheck()
- Appel des méthodes du template parent
- Parent view update
- Child ngOnChanges()
- Child ngDoCheck()
- Appel des méthodes du template child
- Child view update
- Child afterViewChecked()
- Parent afterViewChecked()

Hooks de destruction

ngOnDestroy()

- Appelé une fois lors de la destruction du composant
- Peut être utilisé pour unsubscribe à des Observables

ExpressionChangedAfterItHasBeenCheckedError

- Les méthodes dans le template sont appelées deux fois en développement
- Angular effectue un contrôle pour s'assurer que rien n'ait changé après la mise à jour de l'affichage
- L'erreur NG0100: ExpressionChangedAfterItHasBeenCheckedError, est lancée par Angular en développement uniquement si une valeur change après que la détection ait eu lieu

ChangeDetectionStrategy

- Lors d'un cycle de détection de changement, Angular ne sait pas quels composants doivent être actualisés
- Angular à choisi par défaut de faire une vérification sur tous les composants à chaque cycle
- Cette méthode à l'avantage d'être simple au développeur à implémenter
- Par contre elle peut conduire à des problèmes de performances, surtout pour des grosses applications

ChangeDetectionStrategy.OnPush

- La stratégie OnPush permet de réduire le nombre de vérification à faire lors de la détection de changement
- L'option se configure au niveau des métadonnées du décorateur @Component

```
changeDetection: ChangeDetectionStrategy.OnPush
```

- Le composant possède un booléen *dirty*, qui symbolise la nécessité d'actualiser le composant au prochain cycle de détection de changement
- Les hooks ngDoCheck() et ngAfterViewChecked() d'un composant sont appelés, même si celui-ci n'est pas dirty

ChangeDetectionStrategy.OnPush

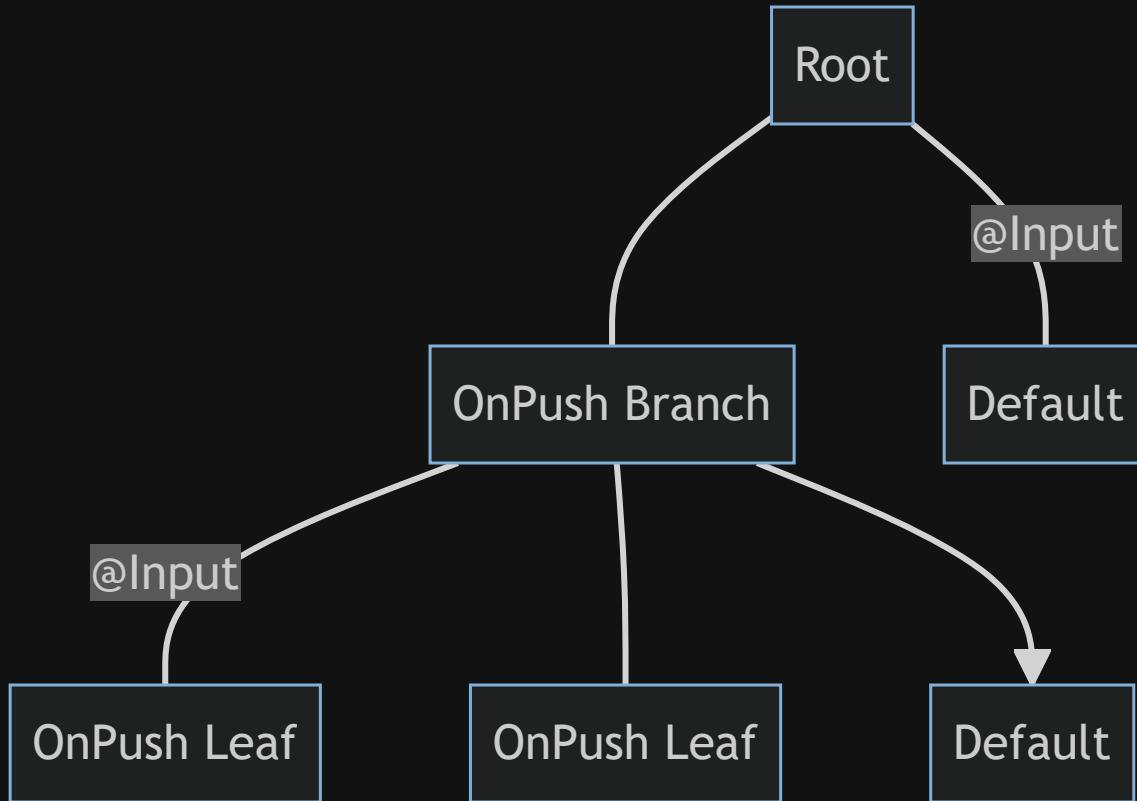
- Un composant est marqué automatiquement à dirty avec les actions suivantes
 - Un évènement du DOM lié à ce composant à lieu
 - Une variable @Input ou @Output de ce composant change
- Lorsque un composant est marqué dirty, tous ses parents sont également marqués
- Il est également possible de marquer un composant dirty manuellement en injectant le service ChangeDetectorRef

```
cdr = inject(ChangeDetectorRef)
```

- Et en utilisant la méthode markForCheck()

```
this.cdr.markForCheck()
```

Exemple



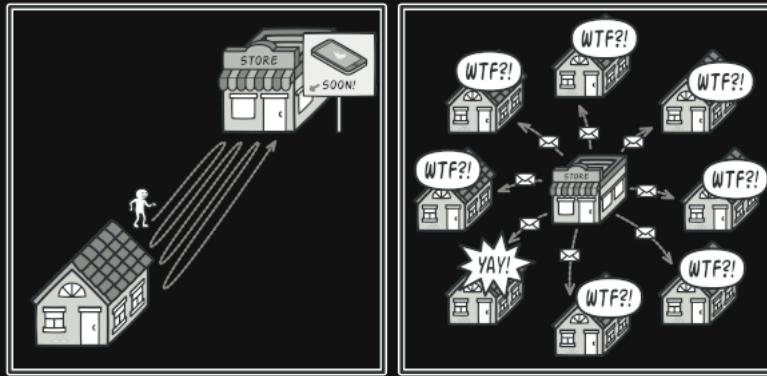
Limites

- Certaines actions courantes dans une application Angular ne mettent pas automatiquement le composant à dirty
 - Mutations d'objets dans un @Input
 - setTimer et setInterval
 - Services

RxJS et Observables

Design pattern Observer

- Mise en situation : deux objets, un Customer et un Store
- Le Customer est intéressé par un objet, qui devrait être disponible *bientôt*



- Le Customer peut rendre visite régulièrement au Store pour savoir si le produit est disponible
 - Beaucoup de temps perdu
- Le Store peut envoyer à tous les Customer une notification lorsque un produit est disponible
 - Beaucoup de messages inutiles

Design pattern Observer

- Le design pattern Observer propose une solution à ce problème :
 - Notre Customer s'abonne au Store
 - Le Store maintient une liste des Customer abonnés
 - Lorsque le Store a un nouveau produit, il notifie les Customer abonnés
- <https://refactoring.guru/design-patterns/observer>

RxJS et Observable

- En se basant sur ce concept, la librairie RxJS permet de gérer les séquences d'événements asynchrones
- L'élément principal est l'Observable qui peut :
 - Émettre de 0 à une infinité de valeurs
 - Peut être en erreur
 - Peut être complété
- Un observable en état d'erreur ne va plus jamais émettre de valeur, ni passer en état de complétion.
- Un observable qui a complété ne va plus jamais émettre de valeur, ni passer en état d'erreur.

Créer un Observable

- La classe (générique) RxJS Observable représente un observable
- Il est possible de créer un observable en utilisant le constructeur

```
const observable = new Observable((subscriber) => {
  subscriber.next(1);
  subscriber.next(2);
  subscriber.next(3);
  setTimeout(() => {
    subscriber.next(4);
    subscriber.complete();
  }, 1000);
});
```

- next() pour envoyer une valeur (synchrone ou asynchrone)
- error() pour envoyer une erreur
- complete() pour signaler que l'observable est terminé

Créer un Observer

- L'interface (également générique) RxJS Observer représente un observateur qui va s'abonner à un observable et consommer ses valeurs

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

- Les callbacks next, error et complete sont appelés lors des émissions respectives de l'observable
- Il est possible de définir un observateur partiel

Souscrire à un observable

- On peut s'abonner à un observable en utilisant la méthode subscribe() de celui-ci

```
subscription: Subscription = this.observable$.subscribe(this.observer)
```

- La classe RxJS Subscription représente un abonnement, retour de la méthode subscribe()
- Un observable est dit *lazy*, c'est à dire qu'il n'est exécuté que lors de l'abonnement d'un observateur
- L'observable s'exécute pour chaque abonnement
- Un observable peut émettre potentiellement à l'infini, on peut utiliser l'abonnement pour se désabonner

```
this.subscription.unsubscribe()
```

- Dans mon exemple précédent, que ce passe t'il si j'ajoute un console.log() dans le setInterval ?

```
setInterval(() => {
  console.log("tick")
  subscriber.next(4);
}, 1000);
```

- Malgré que l'abonnement se soit arrété, le log continue de s'afficher
- Compléter l'observable ne résout pas le problème non plus

TeardownLogic

- Il est nécessaire de libérer manuellement certaines ressources utilisées par les observables
- Un exemple : setInterval()
- Pour cela, on peut définir une TeardownLogic qui libère les ressources à la fin de l'observable

```
const observable = new Observable((subscriber) => {
  subscriber.next(1);
  subscriber.next(2);
  subscriber.next(3);
  let intervalId = setTimeout(() => {
    subscriber.next(4);
    subscriber.complete();
  }, 1000);
  return () => clearInterval(intervalID)
});
```

Async pipe

- Angular met à disposition le pipe async pour gérer facilement les observables
- Le pipe async s'abonne à un Observable (ou une Promise) et renvoie la dernière valeur émise
- A chaque utilisation du pipe, un nouvel abonnement est fait à l'observable
- Composant :

```
obs$ = interval(10)
```

- Template :

```
{{ obs$ | async }}
```

- Le pipe gère automatiquement le désabonnement à l'observable
- **⚠** à l'utilisation du pipe async avec le *ngIf

Exercice

- Nous allons tenter de recréer une version simplifiée du pipe async (uniquement avec des Observables)
- Qu'est ce que fait le pipe async ?
 - On passe un observable au pipe dans le template (pipeTransform)
 - Le pipe async gère l'abonnement et le désabonnement de l'observable
 - Le pipe async retourne la dernière valeur émise par l'observable
- Comment gérer le désabonnement ?
- Est ce que le pipe async est pur ou impur ?
- Plus avancé : que ce passe t'il lorsque je change l'observable dans le template ?

Observable et stratégie OnPush

- Au sein d'un composant avec la stratégie OnPush, que ce passe t'il lorsque l'un de ses observable émet une nouvelle valeur ?
- Rappel des conditions pour mettre un composant dirty :
 - Un évènement du DOM lié à ce composant à lieu
 - Une variable @Input ou @Output de ce composant change
- Aucune des deux conditions n'est remplie pour mettre le composant dirty, l'affichage n'est donc pas mis à jour
- ⚠ Bien que le composant ne se mette pas à jour, l'observable émet ses valeurs, et les cycles de détection de changement ont lieu
- Et avec le pipe async ?
 - Le pipe async met automatiquement le composant dirty

EMPTY, NEVER

```
const EMPTY: Observable<never>;
```

- EMPTY complète immédiatement sans émettre de valeur ni d'erreur

```
const NEVER: Observable<never>;
```

- NEVER n'émet rien, ni ne complète

Opérateurs de création

- En pratique, on crée rarement un observable nous même
- En plus de EMPTY et NEVER, RxJS fournit des opérateurs pour créer des observables
- Les opérateurs de création sont des **fonctions**, qui retournent des observables
- Angular fournit également des observables (HttpClient, ReactiveFormsModule, ...)

of(), from()

```
of<T>(... args: T[]): Observable<T>
```

- of() crée un observable à partir d'une liste de valeurs

```
from<T>(input: T): Observable<T>
```

- from() crée an observable à partir d'un tableau, une promesse, un objet itérable ou "observable-like"
- Quelle est la différence entre les deux appels :

```
of([1,2,3])  
from([1,2,3])
```

timer(), interval()

```
timer(due: number | Date): Observable<0>
```

- Crée un observable qui attend un temps donné (en ms), ou une date, et émet 0

```
interval(period: number = 0): Observable<number>
```

- Crée un observable qui émet une séquence incrémentale à une certaine période

```
timer(startDue: number | Date, intervalDuration: number): Observable<number>
```

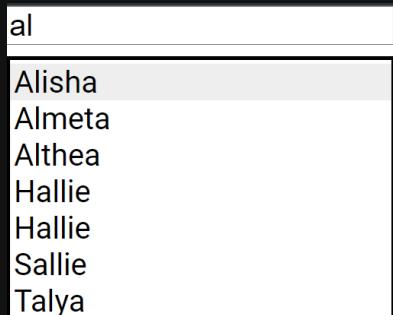
- Crée un observable qui attend un temps donné (en ms), ou une date, puis émet une séquence incrémentale à une certaine période

Exercice

- Dans la suite du cours, nous allons essayer d'implémenter le composant suivant :
- Considérons un serveur qui expose une liste de clients à l'url : <http://localhost:3000/clients>

```
export type Client = {
  id: string,
  firstname: string,
  lastname: string,
  age: number
}
```

- Créer un composant “searchbar avec autocomplétion” (uniquement sur le firstname)



fromEvent()

```
fromEvent<T>(target: any, eventName: string): Observable<T>;
```

- Fonction fournie par RxJS
- Crée un observable à partir d'un évènement (eventName) d'un élément (target) du DOM
- L'observable généré émet une valeur à chaque fois que l'évènement est déclenché

fromEvent()

- Dans une application Angular, on peut utiliser une ViewQuery pour récupérer les éléments du DOM
- Template :

```
<input #input />
```

- Composant :

```
@ViewChild('input')
input!: ElementRef<HTMLInputElement>

...

ngAfterViewInit() {
  fromEvent(this.input.nativeElement, 'input').subscribe( ... )
}
```

HttpClient

- Le service HttpClient d'Angular fournit des méthodes pour faire des requêtes HTTP
- Un seul provider pour HttpClient dans l'application. On peut définir un provider sur le AppModule, ou dans la méthode bootstrapApplication pour un composant standalone

```
bootstrapApplication(HttpClientComponent, {  
  providers: [  
    provideHttpClient()  
  ]  
}).catch(err => console.error(err));
```

- Puis on injecte le service HttpClient

```
private _http = inject(HttpClient)
```

- En général le HttpClient n'est pas injecté directement dans les composants, on passe par un service intermédiaire

Remarques sur les méthodes de HttpClient

- Le service HTTP client fournit des méthodes, permettant de faire des requêtes HTTP
- Les méthodes renvoient des observables
- La requête HTTP n'est émise que lors de l'abonnement à l'observable (lazy)
- Une requête est envoyée pour chaque abonnement à l'observable
- L'observable émet la réponse une fois retournée par le serveur
- Le désabonnement de ces observables est géré par Angular

GET / get()

- La requête GET permet de récupérer des données
- La méthode get() de HttpClient prend deux paramètres d'entrée, l'url à requêter et des paramètres optionnels

```
get<T>(url: string, options): Observable<T>
```

- Par défaut, les données de retour sont supposées au format JSON, et converties en Object
- Il est possible de spécifier un type de retour de l'observable de la méthode get()

```
posts$ = this.http.get<Articles[]>("http://localhost:3000/articles")
```

- **⚠** Il n'y a aucune garantie que le type des données renvoyées par le serveur corresponde au type spécifié

json-server

- Package node pour simuler un service REST facilement
- Installation :

```
npm install -g json-server
```

- Utilisation :

```
json-server --watch {file.json}
```

- Le serveur est prêt à être utilisé

json-server

- Quelques options qui vont nous servir pour notre exercice :
- Add _like to filter (RegExp supported)

```
GET /articles?title_like=server
```

- Add _sort and _order (ascending order by default)

```
GET /articles?_sort=views&_order=asc
```

```
GET /articles/1/comments?_sort=votes&_order=asc
```

- For multiple fields, use the following format:

```
GET /articles?_sort=user,views&_order=desc,asc
```

HttpParams

- Pour certaines requêtes, il est possible de passer des paramètres dans l'url

```
return this._http.get<Articles[]>(`${this.baseUrl}?author_like=${filter}&_sort=title&public=true`)
```

- Plutôt que de tout mettre directement dans l'url lors de l'appel à get(), il est possible d'utiliser un objet HttpParams dans les options
- La classe HttpParam permet de stocker les paramètres sous forme d'une liste clé/valeur
- L'objet HttpParam est immuable, toutes les opérations renvoient un nouvel objet

```
let options = {
  params: new HttpParams()
    .append('author_like', filter)
    .append('_sort', 'title')
    .append('public', true)
}

return this._http.get<Articles[]>(`${this.baseUrl}`, options)
```

Exercice

- Dans le composant AutocompleteComponent :
 - Avec l'opérateur fromEvent(), créer un observable qui émet sur les évènement de type input de l'élément input du template du composant
 - Faire un observer qui s'abonne à cet observable, et qui à partir de l'évènement émit, affiche un console.log de la valeur dans l'élément input
- Dans le service ClientService :
 - Compléter la méthode getFilteredSortedClients() : notre méthode génère un observable en utilisant la méthode get() du client http
 - Cet observable doit nous renvoyer la liste des clients filtrés avec le filtre passé en paramètre, et triés par ordre alphabétique
 - Pour le filtre et le tri, ne considérer que le firstname du client

Operateurs pipeables

- Essayons maintenant de combiner nos deux observables :

```
clients?: Client[]

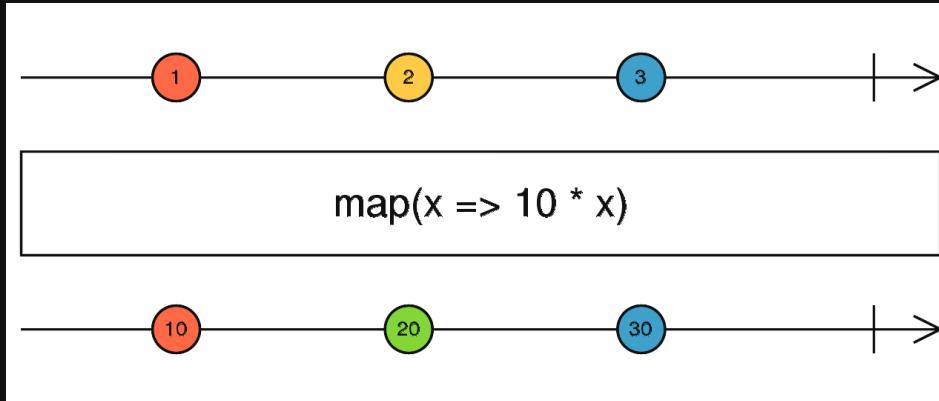
this.subscription = fromEvent(this.input.nativeElement, 'input').subscribe(
  ev => {
    let input = (ev.target as HTMLInputElement).value
    this.clientService.getFilteredSortedClients(input).subscribe(
      c => this.clients = c
    )
  }
)
```

- Notre exemple pourtant simple n'est pas si facile à écrire
- Le code se complexifie vite, avec des subscribe dans des subscribe, et des abonnements intermédiaires à gérer
- Ce phénomène est connu sous le nom de *Subscription Hell*

Operateurs pipeables

- L'une des forces de RxJS est de pouvoir faire des opérations sur les observables, c'est à dire transformer un opérateur en un autre
- Un opérateur pipeables sont des fonctions, qui prennent des observables en entrée, et renvoient des observables en sortie
- Ces opérateurs ne modifient pas les observables d'entrée, mais ils créent de nouveaux observables
- L'opérateur gère l'abonnement et le désabonnement de tous les observables intermédiaires.
- Pour créer des opérateur pipeables, on va en général utiliser des factories, fournies par RxJS
 - Une factory est une fonction, qui renvoie un opérateur
 - Un opérateur est une fonction, qui renvoie un observable

Exemple et Marble Diagram



- Un observable en entrée : il génère les nombres 1, 2, 3, puis complète
- La factory `map(x => 10 * x)` génère un opérateur
- Cet opérateur transforme mon observable d'entrée en un nouvel observable qui génère 10, 20, 30, puis complète
- Vous pouvez utiliser les Marble Diagram pour vous aider à visualiser les opérateurs

map()

```
map<T, R>(project: (value: T, index: number) => R): OperatorFunction<T, R>
```

- La fonction map() est une factory, qui renvoie des OperatorFunction<T, R>
- Un OperatorFunction<T, R> est une fonction qui prend en entrée un Observable<T> et renvoie un Observable<R>
- Le callback project, paramètre de la fonction transforme les valeurs de type T émises par l'observable d'entrée en valeurs de type R émises par l'observable de sortie
- Le type T peut être le même que le type R
- L'utilisation est similaire à celle de map() avec des tableaux

Pipes (RxJS)

- `map()` renvoie une fonction
- L'écriture suivante est correcte

```
obs$ = map((i: number) => 10 * i)(this.interval$)
```

- En pratique, cette écriture est peu lisible, et encore plus si on veut enchaîner plusieurs opérateurs
- Pour résoudre le problème, la classe Observable à une méthode `pipe()`
- **⚠ Pipe RxJS ≠ Pipe Angular**

Pipes (RxJS)

- La méthode pipe() de la classe Observable prends un OperatorFunction<T, R>, et renvoie un nouvel observable
- L'exemple précédent peut être écrit de la manière suivante

```
obsPipe$ = this.interval$.pipe(  
  map((i: number) => 10 * i)  
)
```

- Et pour les types de l'OperatorFunction ?
 - T doit être du même type que l'observable qui appelle pipe()
 - R est le type de retour de l'observable

Pipes (RxJS)

- On peut passer plusieurs OperatorFunction à la suite comme paramètres de la méthode pipe, ainsi :

```
obsPipe$ = this.interval$.pipe(  
  map((i: number) => 10 * i)  
).pipe(  
  map((i: number) => 'To String : ' + i)  
)
```

- Peut être écrit :

```
obsPipe$ = this.interval$.pipe(  
  map((i: number) => 10 * i),  
  map((i: number) => 'To String : ' + i)  
)
```



- Il faut s'assurer que le type de chaque opérateur de la chaîne est compatible avec le type suivant
- L'ordre des opérateurs dans la chaine est important

Operateurs RxJS

- Nous allons voir dans la suite quelques exemple d'opérateurs mis à disposition par RxJS
- <https://rxjs.dev/api/operators>
- <https://rxjs.dev/operator-decision-tree>
- Le but n'est pas de connaître par cœur tous les observables !

filter()

```
filter<T>(predicate: (value: T, index: number) => boolean): MonoTypeOperatorFunction<T>
```

- MonoTypeOperatorFunction<T> représente un opérateur qui revoie un observable de même type que son entrée (OperatorFunction<T, T>)
- A chaque fois que l'observable source émet, la valeur émise est évaluée par le callback predicate, si il retourne true, l'observable renvoyé émet
- Que fait l'observable suivant ?

```
obs$ = interval(1000).pipe(  
  filter( i => i % 2 === 0 )  
)
```

take()

```
take<T>(count: number): MonoTypeOperatorFunction<T>
```

- Crée un observable qui émet uniquement les n premières valeurs de l'observable source, puis complète
- Si l'observable source complète avant d'avoir émis n valeurs, l'observable généré complète également

first()

```
first<T, D>(  
    predicate?: (value: T, index: number, source: Observable<T>) => boolean,  
    defaultValue?: D  
) : OperatorFunction<T, T | D>
```

- Crée un observable qui émet uniquement la première valeur de l'observable source, puis complète
- Cependant quelques différences avec take(1) :
 - Si l'observable source complète avant d'avoir émis une valeur, first émet une erreur, sauf si une valeur par défaut est définie
 - Il est possible de définir un prédictat, dans ce cas uniquement la première valeur qui satisfait ce prédictat sera renvoyée

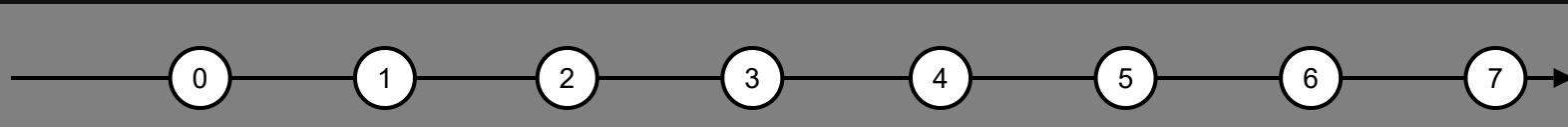
last()

```
last<T, D>(  
    predicate?: (value: T, index: number, source: Observable<T>) => boolean,  
    defaultValue?: D  
) : OperatorFunction<T, T | D>
```

- Similaire à first, mais renvoie la dernière valeur émise par l'observable source
- Pour s'assurer de la dernière valeur, l'observable généré par last émet uniquement lorsque l'observable source complète

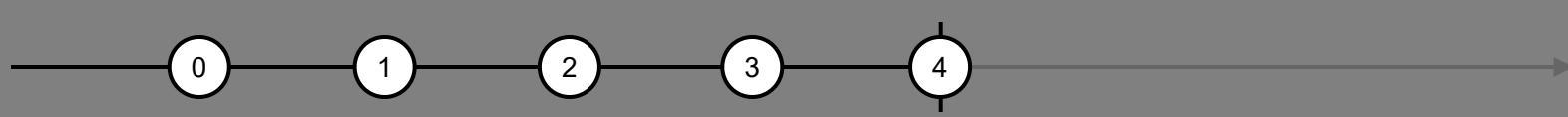
Exemple

- `interval(1000)`



- Toutes les secondes, une valeur est émise, l'observable ne complète jamais

- `take(5)`



- Les 5 premières valeurs sont émises, puis l'observable complète

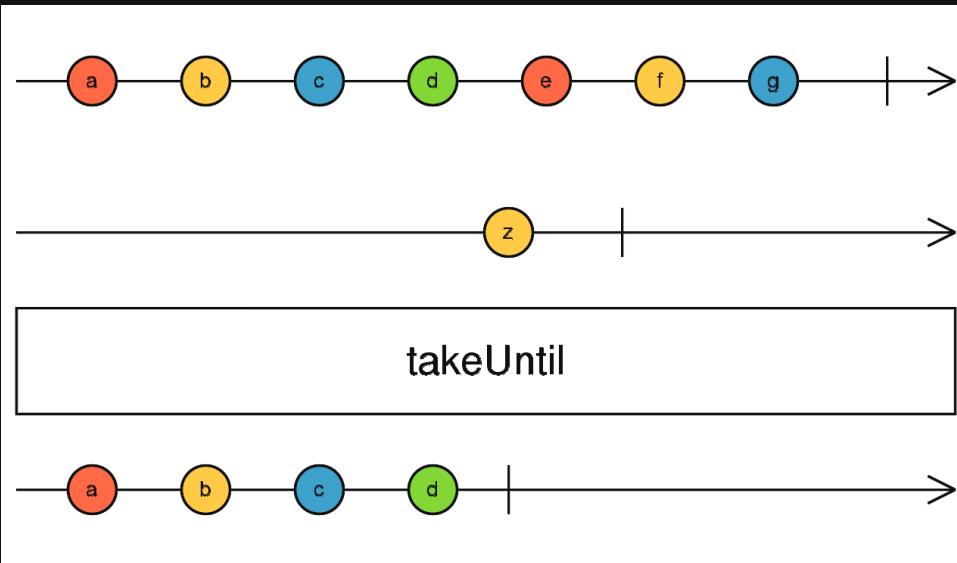
- `last(x => x < 2)`



- Lorsque l'observable, complète, la première valeur qui respecte la condition est émise

takeUntil()

```
takeUntil<T>(notifier: ObservableInput<any>): MonoTypeOperatorFunction<T>
```



- Émet les valeurs de l'observable source jusqu'à ce qu'une valeur de l'observable notifier soit émise, puis complète

reduce(), scan()

```
reduce<V, A>(  
    accumulator: (acc: V | A, value: V, index: number) => A,  
    seed?: any  
) : OperatorFunction<V, V | A>
```

- Applique une fonction d'accumulation sur les valeurs émises par la source
 - La valeur renvoyée par la fonction d'accumulation est assignée au paramètre acc de l'appel suivant
 - Il est possible de définir une seed, valeur initiale de l'accumulateur
- reduce() émet une seule fois, lorsque l'observable source complète, alors que scan() émet la valeur accumulée à chaque émission de la source

tap()

```
tap(observerOrNext?: (value: T) => void) ): MonoTypeOperatorFunction<T>
```

- L'opérateur tap() permet de créer un effet de bord sur un opérateur
- Les valeurs émises par l'observable créé sont les mêmes que l'observable source
- tap() est très utile pour débugger vos observables
- **⚠️** Si votre opérateur source émet des objets, ils peuvent être mutés dans le tap()

Higher-order Observables

- Retour sur notre exemple d'autocomplétion
- Que ce passe t'il si j'essaie de combiner nos deux observables avec un map ?

```
...
clients$?: Observable<Client[]>

ngAfterViewInit() {
  this.clients$ = fromEvent(this.input.nativeElement, 'input').pipe(
    map(ev => this.clientService.getFilteredSortedClients((ev.target as HTMLInputElement).value))
  )
}
...
```

- Il y a une erreur de compilation, c'est normal, car notre map ne renvoie pas un Observable<Client[]>, mais un Observable<Observable<Client[]>>

Higher-order Observables

- Un observable qui envoie des observables est appelé un *higher-order observable*
- Dans la plupart des cas, un tel observable n'est pas utilisable directement, on veut convertir cet observable en un observable émettant des valeurs
- Cette opération (appelée *flatten*) est possible via des opérateurs

mergeAll(), concatAll(), switchAll(), exhaustAll()

- Ces 4 opérateurs s'abonnent à tous les observables générés par l'higher-order observable, et émettent les valeurs émises par les observables intermédiaires
- Il est tellement courant d'utiliser ces opérateurs après un map(), que des opérateurs spécifiques existent, ainsi :

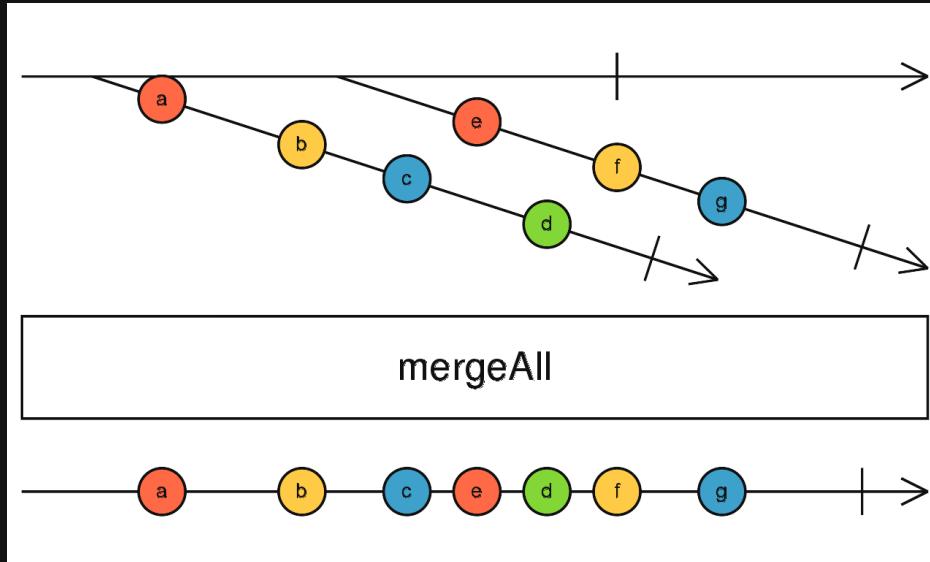
```
obs$.pipe(  
  map(callback),  
  mergeAll()  
)
```

- Peut être simplifié en :

```
obs$.pipe(  
  mergeMap(callback),  
)
```

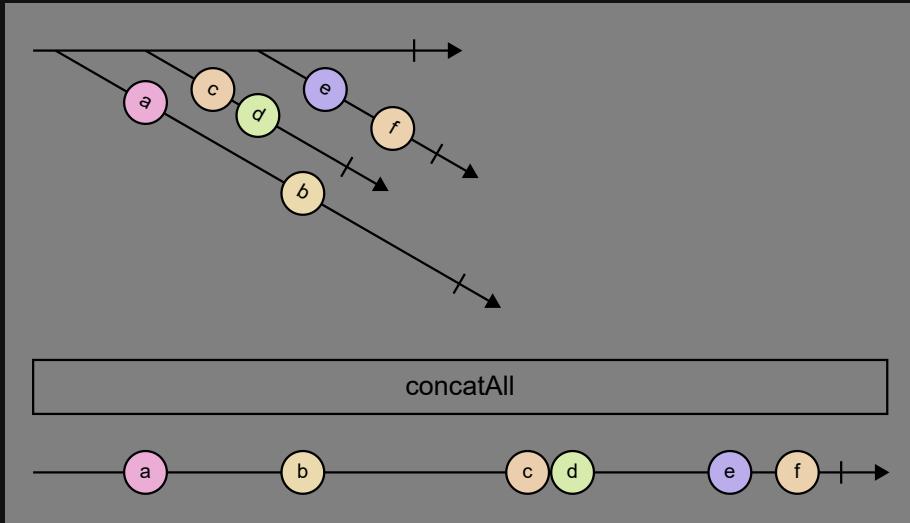
- Il est également possible d'utiliser concatMap(), switchMap(), exhaustMap() de la même manière
- Ces 4 opérateurs sont très similaires, mais ont quelques différences, que nous allons voir maintenant

mergeAll()



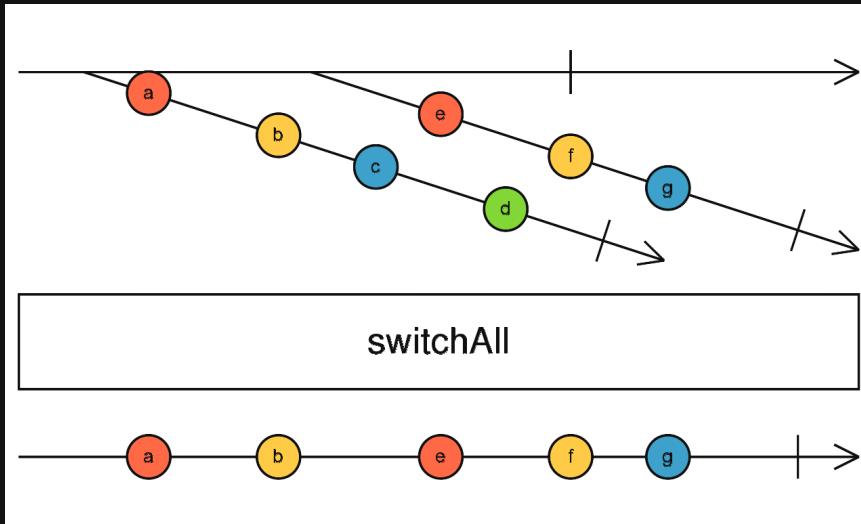
- `mergeAll()` s'abonne à tous les observables et émet toutes les valeurs émises au fur et à mesure

concatAll()



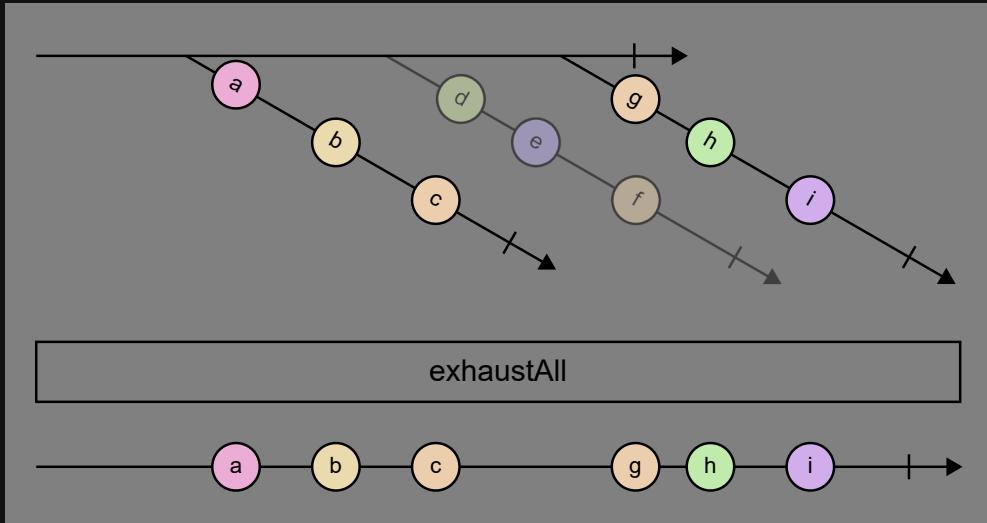
- N'émet les valeurs que d'un observable à la fois, concatAll() attend que le premier ait fini d'émettre avant de passer au deuxième, puis au troisième, ...
- Si le premier observable ne complete jamais, les suivants ne seront jamais pris en compte
- Garde en mémoire une file de tous les observables qu'il a à traiter

switchAll()



- N'émet les valeurs que d'un observable à la fois
- À chaque fois qu'un nouvel observable est émis, `switchAll()` se désabonne à l'observable en cours et passe au nouvel observable
- Une partie des valeurs émises par les observables intermédiaires sont perdues

exhaustAll()



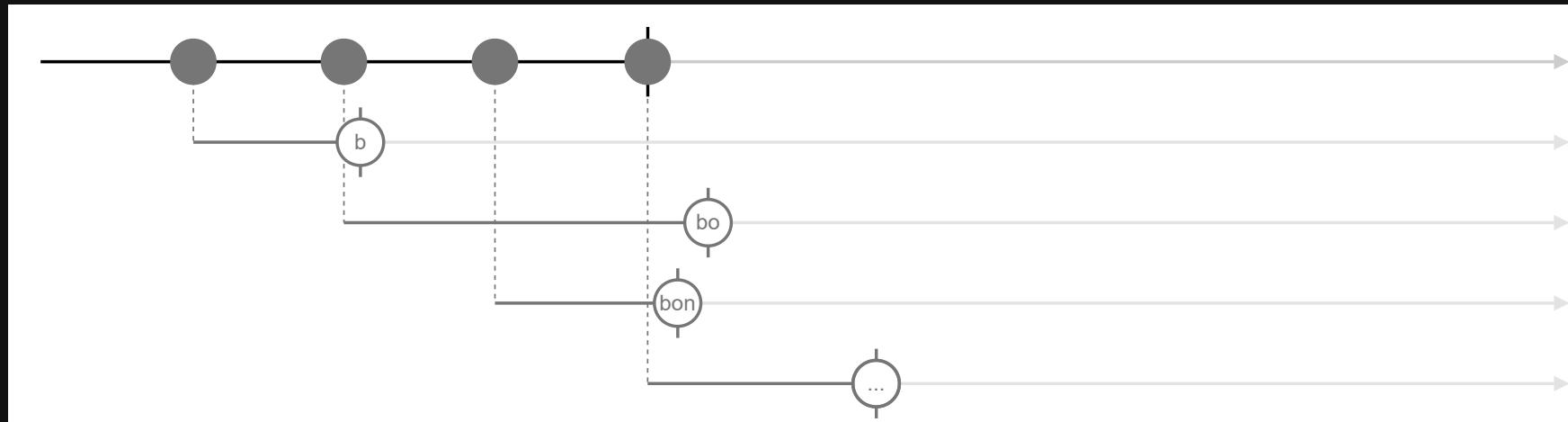
- N'émet les valeurs que d'un observable à la fois
- `exhaustAll()` va ignorer tous les observables émis tant que l'observable en cours n'a pas complété
- Une partie des observables intermédiaires sont perdues

Exemple

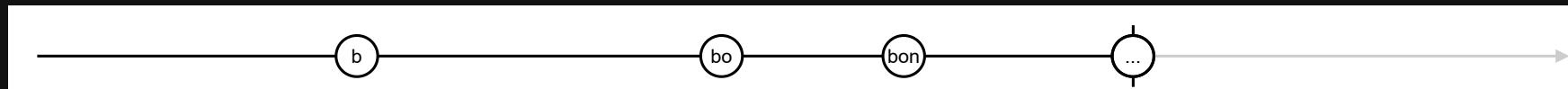
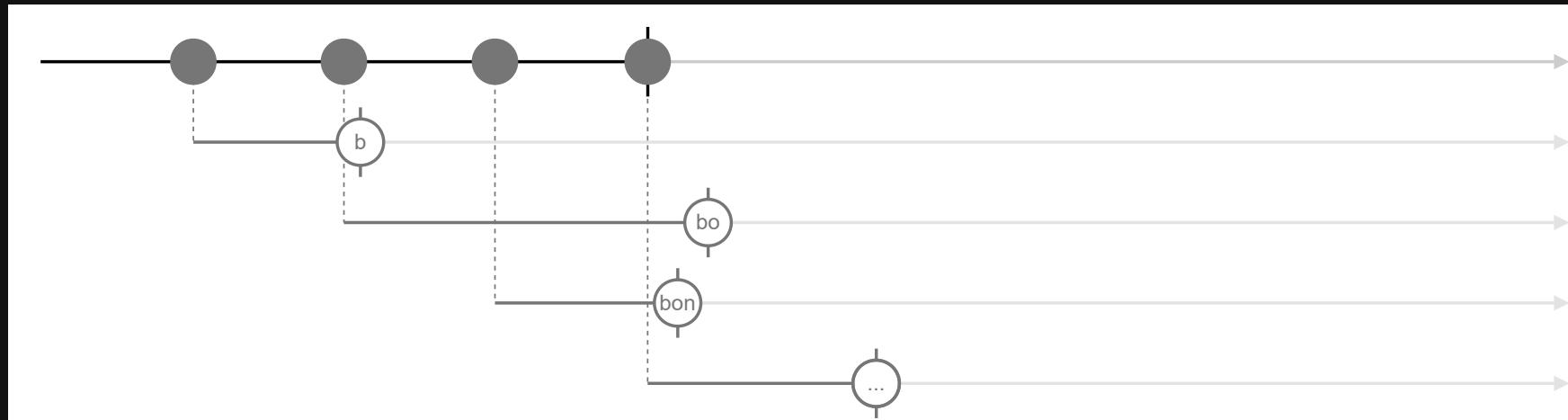
- De retour sur notre exemple d'autocomplétion
- Nous allons devoir utiliser un de ces opérateurs pour lier nos deux observables ensemble, mais lequel ?
- Avec notre serveur en local, la réponse sera à chaque fois très rapide. Ce qui n'est pas toujours le cas en pratique
- La différence de comportement entre les opérateurs va devenir cruciale si le serveur a un délai de réponse, que l'on va pouvoir simuler avec json-server

```
--delay, -d          Add delay to responses (ms)
```

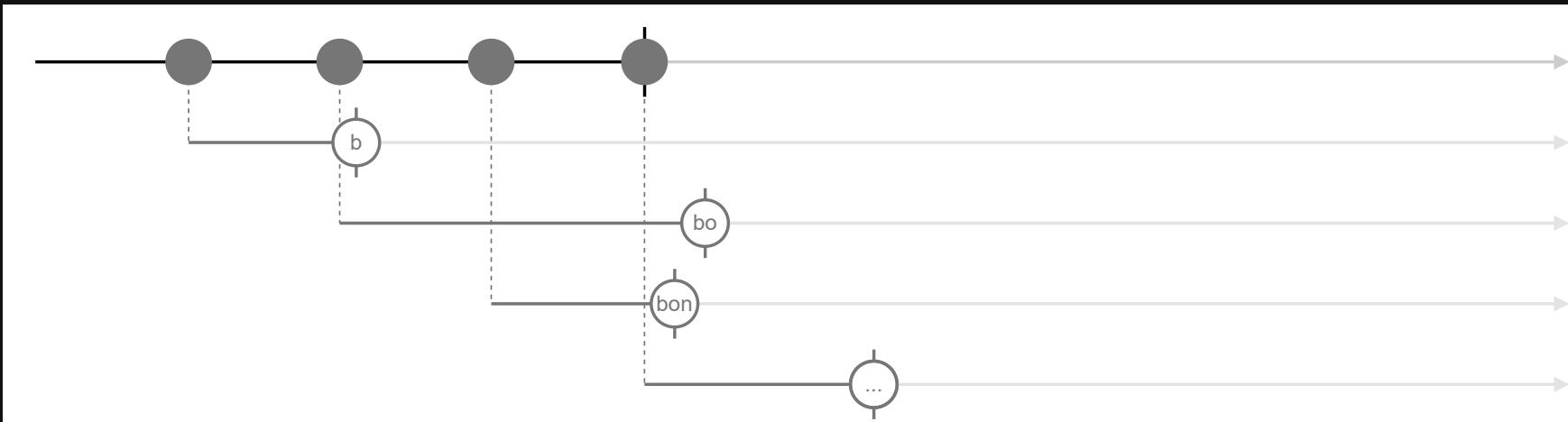
Avec mergeMap()



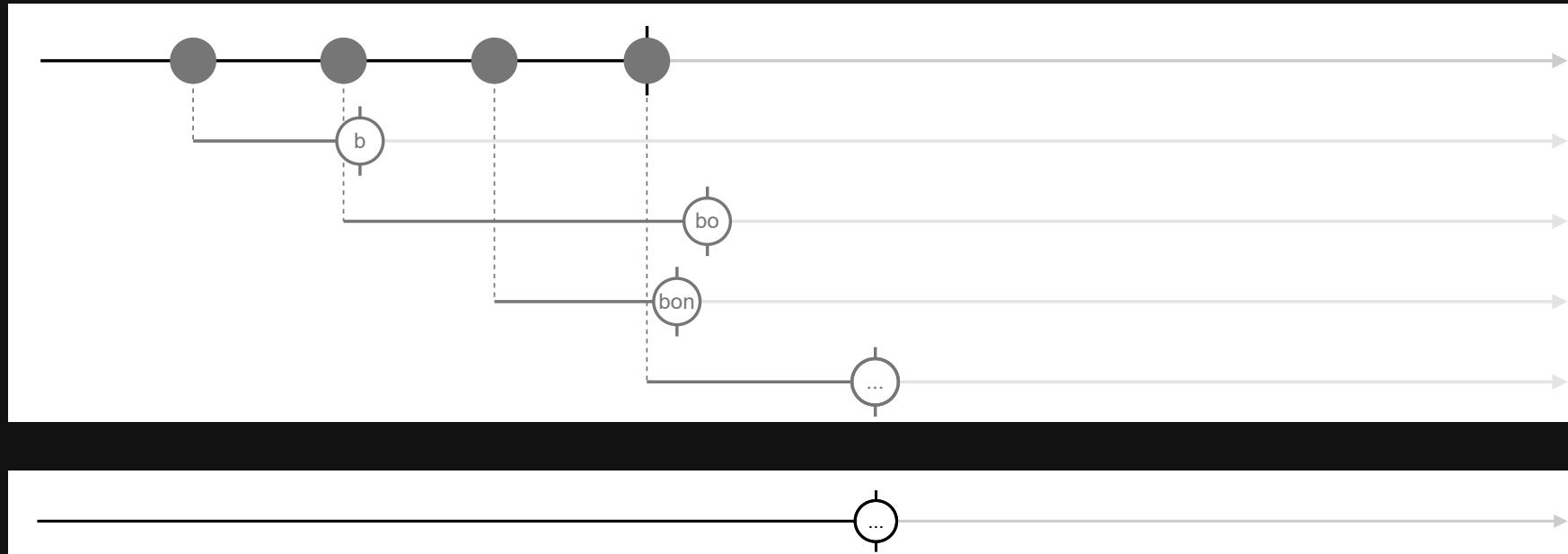
Avec concatMap()



Avec exhaustMap()



Avec switchMap()



Exercice :

- Implémenter l'observable game\$:
 - Lorsque l'on s'abonne à game\$, un timer de 5 secondes est déclenché
 - Pendant ce timer, game\$ compte tous les clicks utilisateur faits sur le document
 - A la fin du timer, game\$ envoie le nombre de clicks, et complète

Implémentation finale

- Pour ce cas, l'opérateur à utiliser est le switchMap
- Quelques idées pour améliorer le composant d'autocomplétion :
 - Attendre que l'utilisateur arrête de taper pour lancer la requête
 - Ne pas faire deux fois la même requête de suite
 - Attendre 3 caractères pour lancer la requête

Gestion d'erreur avec les opérateurs

- retry(n) va tenter de s'abonner jusqu'à n fois à un observable si celui ci émet une erreur

```
retry<T>(configOrCount: number | RetryConfig = Infinity): MonoTypeOperatorFunction<T>
```

- catchError() remplace l'observable source par un autre en cas d'erreur
- catchError() prend un callback en paramètre
- Ce callback prend deux paramètres, l'erreur à "catch", et optionnellement l'observable source, et renvoie un nouvel observable

```
catchError<T, O extends ObservableInput<any>>(selector): OperatorFunction<T, T | ObservedValueOf<O>>
selector: (err: any, caught: Observable<T>) => O
```

(penser à catch() en Java par exemple)

- Exemple, je considère un observable `obs$` qui émet 1, 2 puis une erreur



- L'opérateur `retry(2)` va se réabonner jusqu'à deux fois lors de l'émission de l'erreur

```
obs$.pipe(retry(2))
```



- A partir du même observable que la slide précédente



- Avec l'opérateur catchError(), je substitue un autre observable lors d'une erreur

```
obs$.pipe(  
  catchError(() => interval(1000))  
)
```



- Dans cet exemple, lors de l'erreur de l'observable initial, je le remplace par interval(1000)

State managment

State management

- Retour sur le service CounterService
- Notre service gère un état (la valeur du compteur) qui peut être partagé par plusieurs consumers
- Pour le moment, cette gestion n'est pas faite de manière réactive
- Que se passerait-il si on veut utiliser notre service avec des composants OnPush ?
 - Les composants ne sont pas marqués dirty
- Nous allons voir comment utiliser RxJS pour gérer les états de notre application de manière réactive

"Hot" et "Cold" observable

- On distingue le producteur (celui qui produit les données) de l'observable (responsable de la distribution des données)
- Un observable est dit "cold" lorsque un producteur est créé par l'observable pour chaque abonnement
- Un observable est dit "hot" lorsque le producteur est créé en dehors de l'abonnement
-  Un observable même "hot" n'est pas multicast

Subject

- Un Subject est un type particulier d'Observable

```
export class Subject<T> extends Observable<T> ...
```

- Comme un observable, on peut définir des observateurs qui s'abonnent au subject

```
subject = new Subject()  
  
subject.subscribe(observer1)  
subject.subscribe(observer2)
```

- A la différence des Observables, les valeurs émises par le Subject sont multicast
- Dans l'exemple précédent, cela implique que l'on aura une seule execution du subject pour les deux observateurs (à la différence d'un observable classique, où chaque observateur aurait eu sa propre exécution)

Subject

- La classe Subject implémente les méthodes next(v), error(e) et complete()

```
subject.next(v)
subject.error(e)
subject.complete()
```

- Lorsque on appelle next(v), le subject émet une valeur à ses observateurs
- Lorsque on appelle complete(), le subject complète
- Lorsque on appelle error(v), le subject envoie une erreur

Subject

- Le nom des méthodes next(), error() et complete() n'a pas été choisi par hasard
- Comme il implémente ces méthodes, notre subject est également un observateur
- Comme un observateur, notre subject peut s'abonner à un observable :

```
observable$ = new Observable( ... )  
  
observable$.subscribe(subject)
```

- Dans ce cas, le subject est utilisé comme intermédiaire pour émettre les valeurs d'un observable en multicast à plusieurs observateurs

BehaviorSubject

- Le BehaviorSubject est un type particulier de Subject

```
class BehaviorSubject<T> extends Subject<T> {
```

- Le BehaviorSubject à une notion de valeur courante (la dernière valeur émise), qu'il est possible de récupérer

```
behaviorSubject.getValue()
```

- Lorsque un observateur s'abonne au BehaviorSubject, il reçoit immédiatement la valeur courante
- Un BehaviorSubject doit toujours avoir une valeur courante, pour cela il est nécessaire d'en donner une par défaut lorsque on instancie un BehaviorSubject

```
behaviorSubject = new BehaviorSubject(0)
```

State management avec RxJS

- Nous allons voir une manière d'implémenter le state management avec RxJS, en refactorisant notre CounterComposant de manière réactive :
- La base de notre state management va être un BehaviorSubject

```
private _state = new BehaviorSubject(0)
```

- On ne veut en général pas exposer directement le BehaviorSubject dans notre service, pour limiter ce que peuvent faire les utilisateurs du service, pour cela on peut convertir un Subject en Observable

```
observable$ = this._state.asObservable()
```

State management avec RxJS

- Les consumers peuvent s'abonner à l'observable pour récupérer l'état du compteur

```
reactiveCounterService = inject(ReactiveCounterService)  
  
{{ reactiveCounterService.counter$ | async }}
```

- Et l'incrémenter avec une méthode du service

```
incrementValue() {  
  this._counter.next(this._counter.getValue() + 1)  
}
```

State management et objets

- Très souvent, on va vouloir représenter notre état sous la forme d'un objet
- Comme d'habitude, il y a une différence de comportement entre un objet (référence) et un type primitif (valeur)
- Il est très important que les abonnés à l'observable ne modifient pas directement l'état, car l'état serait modifié pour les autres abonnés sans qu'il y ait de notification

Exercice - Application magasin

- Implémenter un service "panier" avec RxJS
 - L'état du panier est stocké dans un BehaviorSubject
 - L'utilisateur peut ajouter ou enlever un objet dans le panier, et vider le panier
- Implémenter le service "produits"
 - Une méthode permet de récupérer la liste des produits disponibles depuis un serveur distant

State management

- Il existe des librairies dédiées au state management pour Angular (NgRx, NGXS), mais elles ne sont pas obligatoires
- Dans le futur, state management avec des Signals ?

Retour sur HttpClient

Méthodes HTTP

- Les méthodes pour les requêtes HTTP courantes sont également implémentées dans le service `HttpClient` :
 - `post()` / POST
 - `put()` / PUT
 - `delete()` / DELETE
 - `patch()` / PATCH
 - `head()` / HEAD
 - `options()` / OPTIONS

POST / post()

- La requête post permet d'envoyer des données au serveur

```
_httpClient.post(url, body, options): Observable<Object>
```

- Comme pour la requête get(), le type de retour de la méthode est un observable, qui renvoie le contenu du body de la réponse du serveur, supposée au format JSON et convertie en objet

Typage

- Par défaut le client http de Angular retourne uniquement le body de la réponse de serveur, au format JSON convertie en object
- On peut utiliser les options dans les méthodes du client pour changer cela :

```
responseType?: 'json' (default) | 'arraybuffer' | 'blob' | 'text'
```

```
observe?: 'body' (default) | 'events' | 'response'
```

- Le type de retour de l'observable change avec les paramètres

HttpInterceptor

HttpInterceptor

- Dans une application Angular, il y a des traitements que l'on veut appliquer sur toutes les requêtes que l'on envoie, ou sur toutes les réponses que l'on reçoit
- Pour cela, on va définir des HttpInterceptor



- Les intercepteurs se composent sous forme de chaîne
- Le dernier intercepteur de la chaîne est **HttpBackend**, qui s'occupe de l'envoi de la requête

HttpInterceptor

- La manière historique de faire un intercepteur est un service injecté :

```
@Injectable()  
export class NoopInterceptor implements HttpInterceptor {
```

- Angular nous fournit l'interface HttpInterceptor comme modèle pour créer un intercepteur

```
interface HttpInterceptor {  
  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>>  
}
```

- Cette interface ne contient qu'une méthode, intercept, que nous allons étudier

- Nous allons d'abord voir un intercepteur qui ne fait rien :

```
intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
  return next.handle(req);
}
```

- La méthode intercept prend 2 paramètres :
 - req : la requête http envoyée par intercepteur précédent
 - next : le traitement de l'intercepteur suivant dans la liste
- HttpHandler est une classe abstraite qui représente le traitement d'un intercepteur

```
abstract class HttpHandler {
  abstract handle(req: HttpRequest<any>): Observable<HttpEvent<any>>
}
```

- Le retour de la méthode est l'observable avec la réponse
- L'observable est renvoyé à l'intercepteur précédent

Utilisation (avec DI)

- Nous avons maintenant défini un intercepteur, mais comment l'utiliser ?
- On a vu que notre provider est un service injectable, on va donc utiliser l'injection de dépendances
 - Pour cela, Angular nous met à disposition un token d'injection : HTTP_INTERCEPTORS

```
{ provide: HTTP_INTERCEPTORS, useClass: NoopInterceptor, multi: true }
```

- Quelques remarques :
 - Le paramètre "multi: true" indique qu'il est possible de spécifier plusieurs providers pour une dépendance (ici un provider par intercepteur)
 - **⚠️** L'ordre des providers à une importance
 - Les intercepteurs sont une dépendance optionnelle de HttpClient, il faut donc les fournir à linstanciation du HttpClient, dans le même provider ou un parent

Intercepteur fonctionnel

- Depuis la version 14 d'Angular, il est possible de déclarer les intercepteurs sous forme de fonction plutôt que sous forme de classe
- L'écriture est très similaire au service, sauf qu'on utilise directement des fonctions
- Dans les versions antérieures d'Angular, les intercepteurs étaient sous forme de service car c'était la seule manière de pouvoir injecter d'autres services
- La fonction inject() a permis de pouvoir créer des intercepteurs fonctionnels

Intercepteur fonctionnel

- Exemple de NoopInterceptor fonctionnel :

```
export const noopIntercept: HttpInterceptorFn = (
  req: HttpRequest<unknown>, next: HttpHandlerFn
): Observable<HttpEvent<unknown>> => {
  return next(req)
}
```

- Le type HttpInterceptorFn ressemble à la méthode intercept de l'interface HttpInterceptor

```
export declare type HttpInterceptorFn =
  (req: HttpRequest<unknown>, next: HttpHandlerFn) => Observable<HttpEvent<unknown>>;
```

- Le type HttpHandlerFn ressemble à la méthode handle de l'interface HttpHandler

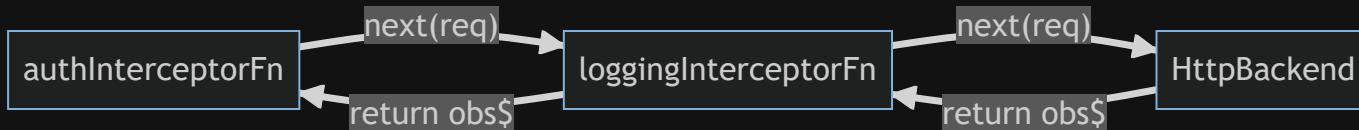
```
export declare type HttpHandlerFn = (req: HttpRequest<unknown>) => Observable<HttpEvent<unknown>>;
```

Intercepteur fonctionnel

- Utilisation de l'intercepteur :

```
provideHttpClient(withInterceptors(  
    [noopIntercept]  
)  
)
```

Chaine d'intercepteurs



- Dans la chaîne, il y a un empilement des appels des intercepteurs
 - authInterceptorFn appelle loggingInterceptor qui appelle le HttpBackend
 - HttpBackend retourne un observable à LoggingInterceptor qui retourne un observable à HttpBackend
- Pendant la chaîne, les différents intercepteurs peuvent modifier la requête, mais aussi la réponse
- Cet empilement signifie que LoggingInterceptor va voir la requête après qu'elle soit passée dans tous les intercepteurs de la chaîne, mais la réponse en premier

HttpRequest

- HttpRequest est une classe représentant une requête Http. Une instance est créée par HttpClient et passe dans les différents intercepteurs
- L'objet HttpRequest de Angular est immutable, il est nécessaire de faire une copie à chaque intercepteur, pour cela, on utilise la méthode clone()

```
const clone = req.clone({  
  headers: req.headers.set('Authorization', authToken)  
});
```

- Les headers sont aussi un objet immutable, la requête clonée à aussi un clone des headers

HttpContext

- Les intercepteurs sont appelés pour chaque requête Http
- Parfois, on veut pouvoir avoir un traitement particulier pour certaines requêtes
- Pour cela, peut définir un HttpContext dans une requête, qui est une map de HttpContextToken, auquel on peut définir une valeur particulier dans chaque requête
- HttpContextToken<T> est une classe générique

HttpContext

- Définition du token :

```
export const IS_LOGGING_ENABLED = new HttpContextToken<boolean>(() => false)
```

- Lorsque l'on définit un token, on doit définir une valeur par défaut dans le constructeur

- Utilisation dans l'intercepteur :

```
if (req.context.get(IS_LOGGING_ENABLED) === true) {  
  return ...;  
}  
...
```

- Utilisation dans le HttpClient :

```
return this._http.get(this.baseUrl, {  
  context: new HttpContext().set(IS_LOGGING_ENABLED, true)  
})
```

TrackBy

TrackBy

- Tous les éléments à l'intérieur de la balise portant la directive *ngFor (ou @for) sont dupliqués pour chaque membre de la collection
- Lorsque un membre de la collection change, Angular détruit tous les éléments correspondants, et recrée des nouveaux éléments avec le nouveau membre
- Le fait de détruire et de recréer des éléments du DOM est une opération couteuse, qui peut ralentir les performances de l'application si elle est faite trop souvent
- Pour savoir si un membre de la collection par défaut, Angular compare les références des objets

- On peut définir une autre fonction pour comparer les objets

```
<div *ngFor="let item of collection; trackBy: trackByFunction">  
  ...
```

```
...  
trackByFunction(index: number, item: any): any {  
  return item  
}  
...
```

- index est la position de l'objet dans la liste, item est une référence à l'objet
- Le retour de la fonction trackBy est utiliser pour déterminer si l'objet a changé
- Il est très important que la fonction trackBy renvoie un résultat unique pour chaque élément de la collection
- Avec @for, définir cette fonction est obligatoire

Routage

Single Page Application

- Historiquement, les sites webs étaient constitués d'une multitude de pages, avec une architecture qui ressemble à celle d'un répertoire de fichiers
- Le problème de cette approche est que une nouvelle page entière doit être chargée dès que l'utilisateur change de page, ce qui peut être lent
- Pour résoudre ce problème, les SPA (Single Page Application) ont vu le jour
- Le but d'une SPA est d'avoir une page unique, qui est modifiée dynamiquement par le navigateur, sans avoir à charger une nouvelle page, avec éventuellement des données récupérées depuis le serveur
- Le principe du routage dans une SPA permet d'associer des URL à des états de l'application. Il va permettre à l'utilisateur de naviguer sur l'application grâce à l'URL, comme il le ferait sur un site classique

Routage

- Dans une application Angular, un service gère le routage. Comme le HttpClient, il n'est pas activé par défaut
- En général, on crée un module à part :

```
const routes: Routes = [
  { path: 'colis', component: ColisComponent },
  { path: 'livraisons', component: LivraisonsComponent }
]

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

- Le RouterModule associe des composants à des URLs

Routage

- Il faut indiquer à Angular où insérer le composant, pour cela, on utilise une balise spécifique dans le template :

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

- RouterOutlet est une directive, qui va être remplacée par le module de routage d'Angular par le composant correspondant à la route

Routes

- Le type Routes est un tableau de Route

```
const routes: Routes = [
  { path: 'colis', title: 'Colis', component: ColisComponent },
  { path: 'livraisons', title: 'Livraisons', component: LivraisonsComponent },
  { path: '', redirectTo: '/colis', pathMatch: 'full' },
  { path: '**', component: PageNotFoundComponent },
]
```

- Le router va essayer de matcher l'URL avec ses paths, dans l'ordre du tableau
- title permet de changer le titre de la page pour une route spécifique
- pathMatch: 'full' indique que l'URL complète doit matcher avec le path de la route, l'autre valeur (par défaut est 'prefix')
- ** est une **wildcard** qui va matcher avec toutes les URLs

Navigation

- Il est possible de faire des liens classiques dans l'application pour naviguer, mais dans ce cas, la page est rechargée, ce qui annule l'intérêt du routage
- Angular met à disposition la directive routerLink pour naviguer dans l'application
- Une autre directive, routerLinkActive permet d'associer une classe à un élément si le lien pointe sur la route courante

```
<button routerLink="/colis" routerLinkActive="selected-item">Gestion des colis</button>
```

- Dans un composant, on peut injecter le service Router et utiliser la méthode navigate() pour rediriger l'utilisateur

```
router = inject(Router)  
...  
this.router.navigate(['/colis']);
```

- Un / dans l'URL de navigation indique un chemin absolu

Découpage modulaire et lazy-loading

```
app/
  feature/
    feature.module.ts
    feature.routing.module.ts
  other-feature/
    ...
    other-feature.module.ts
    other-feature.routing.module.ts
  app.module.ts
  app.routing.module.ts
```

- Dans une application modulaire, le module principal gère le routing jusqu'à chaque feature et chaque module feature gère son propre sous-routing
- En général les features modules sont lazy-loadés
- **⚠** La configuration des sous-routes est différente si le module est lazy-loadés

FeatureRoutingModule (pas de lazyload)

```
const routes: Routes = [
  {
    path: 'colis',
    title: 'Colis',
    component: ColisComponent,
  }
]

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

- Chaque feature module à son propre sous module de routage
- La seule différence avec le routing principal est l'utilisation de `forChild()` au lieu de `forRoot()` dans les imports
- Dans ce cas, les routes sont au même niveau que celles du routeur principal

Child routes

- L'objet Route possède un paramètre children qui est un tableau de Route, il est donc possible de définir une arborescence de routes :

```
const routes: Routes = [
{
  path: 'colis',
  title: 'Colis',
  component: ColisComponent,
  children: [
    { path: 'details', component: ColisDetailsComponent },
  ]
},
...
]
```

- Pour que les routes filles soient utilisées, le composant ColisComponent doit avoir lui aussi un appel à la directive RouterOutlet dans son template

Lazy-loading

- Lorsque l'application devient plus complexe, charger toute l'application en une seule fois peut devenir long
- Il devient nécessaire de découper l'application en plusieurs sous-parties, qui seront téléchargées uniquement lorsque elles sont nécessaires
- Ce principe s'appelle le lazy-loading
- Le lazy-loading à l'avantage de réduire le chargement initial de l'application, mais oblige le client à faire des appels réguliers vers le serveur pour récupérer les parties manquantes

AppRoutingModule

```
const routes: Routes = [
{
  path: 'colis',
  title: 'Colis',
  loadChildren: () => import('../features/colis/colis.module').then(m => m.ColisModule)
},
{
  path: 'livraisons',
  title: 'Livraisons',
  loadChildren: () => import('../features/livraisons/livraisons.module').then(m => m.LivraisonsModule)
}
]
```

- Dans le AppRoutingModule, on ne charge plus des composants, mais des modules
- **⚠️** Attention à ne pas importer en plus les features modules dans le AppModule

FeatureRoutingModule (lazyload)

```
const routes: Routes = [
  {
    path: '',
    title: 'Colis',
    component: ColisComponent,
    children: [
      { path: 'details', component: ColisDetailsComponent },
    ]
  }
]

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

- Le path utilisé pour chargé le module est réutilisé en préfixe

Paramètres

- Il est possible d'avoir des routes avec des paramètres dans l'URL
- Par exemple pour consulter un colis en particulier

```
{ path: ':colisId', component: ColisDetailComponent },
```

- Le composant peut récupérer les paramètres d'url avec le service ActivatedRoute :

```
router = inject(ActivatedRoute)  
...  
id = route.snapshot.paramMap.get('colisId');
```

- Snapshot représente un instantané de l'état de la route

Paramètres

- Lorsque l'on navigue avec le router, Angular ne recrée pas un composant si les paramètres changent
- Dans ce cas, le snapshot n'est pas adapté car il ne reflètera pas le changement de paramètre
- Mais il est possible de récupérer les paramètres sous forme d'observable :

```
route = inject(ActivatedRoute)

constructor() {
  route.paramMap.subscribe((params: ParamMap) => {
    const id = params.get('colisId')!;
  });
}
```

Guards

- Les guards permettent de restreindre l'accès à certaines parties de l'application
- Il existe plusieurs types de guards, nous allons voir le plus simple canActivate, qui permet de restreindre l'accès à une route
-  Les Guards ne dispensent pas de faire un contrôle d'accès au niveau du serveur backend

CanActivate (déprécié)

- Les guards sont définis sous la forme de service
- Angular fournit une interface CanActivate, qui définit ce que notre service soit implémenter pour pouvoir être utilisé comme guard canActivate
- Il s'agit d'une méthode : canActivate()

```
@Injectable()
class CanActivateLivraisons implements CanActivate {
  ...
  canActivate(
    route: ActivatedRouteSnapshot, state: RouterStateSnapshot
  ): MaybeAsync<GuardResult> {
  ...
}
```

Guards fonctionnels

- L'implémentation des Guards sous forme de service est dépréciée depuis Angular 16
- De la même manière que les intercepteurs, il faut maintenant définir les guards de manière fonctionnelle :

```
export const isAuthenticatedGuard: CanActivateFn = (route, state) => {
  const authenticationService = inject(AuthenticationService)
  const router = inject(Router);

  return authenticationService.getStatus()
    ? true
    : router.parseUrl("/connexion")
}
```

CanActivateFn

```
type CanActivateFn = (route: ActivatedRouteSnapshot, state: RouterStateSnapshot) => MaybeAsync<GuardResult>
```

- La méthode canActivate() renvoie un objet de type MaybeAsync<GuardResult>

```
type MaybeAsync<T> = T | Observable<T> | Promise<T>
```

- Le retour d'un guard peut être synchrone ou asynchrone, dans le cas d'un retour asynchrone, le routeur attend une émission avant de continuer

```
type GuardResult = boolean | UrlTree
```

- Le retour d'un guard peut être soit un boolean, qui représente si l'utilisateur peut accéder à la route ou non. Au lieu de renvoyer false, le guard peut renvoyer une url pour rediriger l'utilisateur

Fin