esprit
Se former autrement
HONORIS UNITED UNIVERSITIES

Les services et les observables dans Angular

United Nations Educational, Scientific and Cultural Organization
UNESCO Chair "Project-based learning" ESPRIT School of engineering, Tunisia
EUR-ACE® Délivrée par la Commission des Titres d'Ingénieur
CONFERENCE DES GRANDES ÉCOLES
cdio CONCEIVE DESIGN IMPLEMENT OPERATE

- Les services
- Injection de dépendances
- Exemple d’application
- Les observables
- RxJS
- Exercices sur les observables et les opérateurs RxJS

Les services

Service - Définition

- Un service est une ressource réutilisable et partagée entre plusieurs éléments.
- Il s’agit généralement d’une classe ayant un objectif étroit et bien défini.
- Il offre des propriétés et des fonctionnalités dont l’application a besoin

Service - Rôles

- Centraliser la logique métier: Les services permettent de regrouper la logique métier de l'application dans des classes dédiées, plutôt que de la disperser dans les composants.
- Partager des données: Les services peuvent être injectés dans plusieurs composants, permettant ainsi de partager des données et des fonctionnalités entre eux.
- Abstraction des dépendances: Les services permettent d'abstraire les dépendances (comme les appels HTTP) et de les rendre accessibles aux composants qui en ont besoin.
- Testabilité: Les services étant découplés des composants, ils sont plus faciles à tester de manière unitaire.

Service - Création

- Nous pouvons écrire manuellement la classe du service tout en respectant la charte associée.
- Angular CLI permet de générer un service à l'aide de la commande

ng generate service nomservice

- Si un service est à créer dans un répertoire bien déterminé, il faut alors préciser le nom du répertoire au niveau la commande

ng generate service rep\nomservice

Service - déclaration

- Pour déclarer qu'une classe est un service dans Angular, il suffit de créer une classe TypeScript et de la décorer avec le décorateur @Injectable().

```
@Injectable()
export class UsersService { ... }
```

- Le décorateur @Injectable indique qu'il s'agit d'un service qui peut être injecté dans d'autres parties de l'application.

Services - Squelette

- **@Injectable()**: décorateur qui permet de fournir les métadonnées permettant à Angular d'injecter le service dans un composant en tant que dépendance.
- **providedIn: 'root'** : fournir le service au niveau racine. Ce provider est généré par défaut en créant un service avec l'outil CLI. Angular crée une instance unique et partagée de ce service et l'injecte dans toute classe qui le demande

```
import { Injectable } from '@angular/cor
@Injectable({
  providedIn: 'root'
})
export class UserService {

  constructor() {}

}
```

Service – Injection dans un composant ou service

Un service est injecté au niveau de la classe qu'elle va l'utiliser que ce soit un service ou un composant à travers le constructor.

```
constructor(private ds:DataService) { }
```

« ds » représente le nom de l'instance du service

« DataService » représente le nom du service souhaité.

Injection de dépendances

Injection des dépendances - Présentation

- C’est un design pattern qui consiste à séparer l’instanciation d’une dépendance et son utilisation.
- L’injector du système d’injection de dépendance permet de créer ou réutiliser une dépendance.
- Il appelle uniquement les dépendances dont l’application aura besoin lors de l’exécution..
- Les services dans la plus part du temps sont utilisées comme des singletons çad un service est créé seulement une fois pour le cycle de vie de l’application.

Injection des dépendances – Avantages

L'injection des dépendances assure :

Le découplage: Le composant n'a pas besoin de connaître l'implémentation du service, il peut simplement l'utiliser.

La testabilité: Le service peut être facilement remplacé par un mock lors des tests unitaires du composant.

La réutilisabilité: Le service peut être injecté dans d'autres composants ou services de l'application.

Injection des dépendances - Principe

Lorsque Angular découvre qu'un composant dépend d'un service:

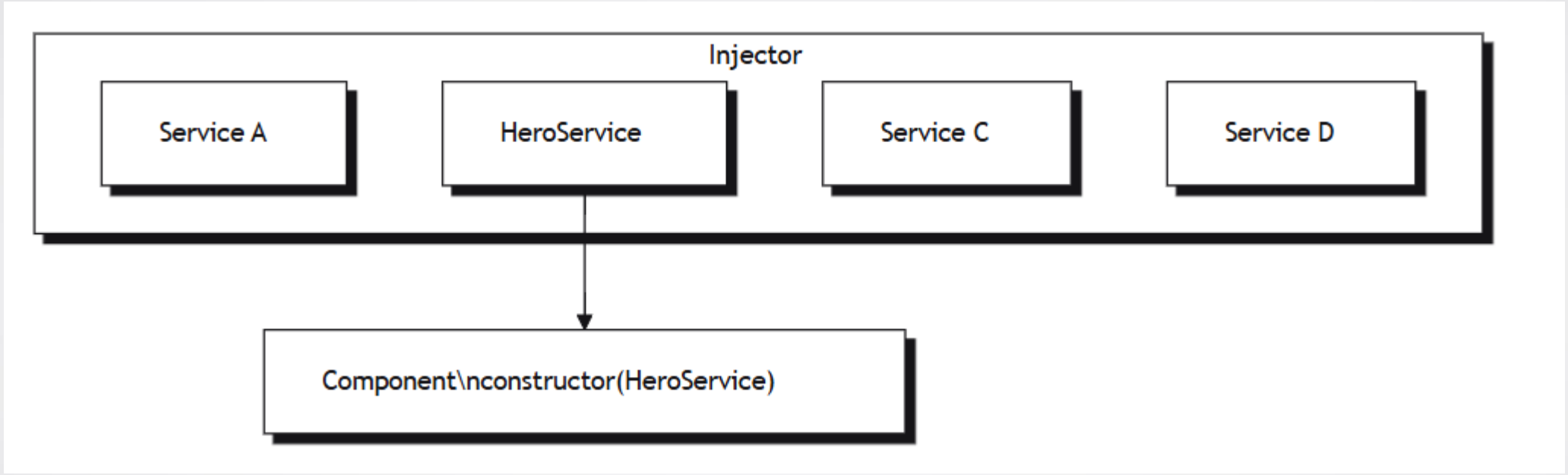
1- il vérifie d'abord si l'injecteur « injector » possède des instances existantes de ce service.

2- Si aucune instance du service demandé n'existe encore, l'injecteur « injector » en crée une en utilisant le fournisseur enregistré « provider », puis l'ajoute à l'injecteur « injector » avant de retourner le service à Angular.

3- L'injecteur « injector » fournit l'instance du service au composant

RQ: Même principe si un service dépend d'un service

Injection des dépendances - Principe



Injection de dépendance : ModuleInjector

- Angular ne possède pas un seul injecteur, mais une arborescence d'injecteurs

```
Root Injector
    ├── ModuleInjector (Module lazy-loaded)
    ├── ModuleInjector (Autre module lazy-loaded)
    └── ComponentInjectors...
```

- Le Root Injector est créé au démarrage.
- Chaque module lazy-loaded a son propre ModuleInjector.
- Chaque composant peut avoir un ComponentInjector

Injection de dépendance : ModuleInjector

Le ModuleInjector est un injecteur attaché à un NgModule

Il contient :

- les services déclarés dans providers: [] du module,
- les services référencés par providedIn: <ce module>,
- les services provenant d'un module importé (qui fournit des services).

Angular l'utilise pour instancier les services quand un composant en a besoin.

À quoi sert un ModuleInjector ?

1- Isoler des instances:

Chaque module lazy-loaded aura **sa propre instance** d'un service.

2- Partager des services:

Si deux modules partagent le même ModuleInjector, ils partagent les **mêmes instances** des services (singleton).

3- Contrôler le scope, par exemple:

- providedIn: 'root' → root injector
- providedIn: 'any' → chaque ModuleInjector
- providedIn: AdminModule → si AdminModule est lazy-loaded → instance isolée

Provider – Définition du provider d’un service

Afin de pouvoir instancier un service, Angular a besoin d'un "provider" lui indiquant **comment produire l'instance de ce service**. Cela se fait généralement via la propriété

- **providedIn** de **@Injectable()**
- **providers** de **@Component()**
- **providers** de **@NgModule()**
- **providers de AppConfig (*pour les projets standalones*)**

Vous devez alors enregistrer au moins un fournisseur « provider » du service à injecter.

Provider - Erreur

- Au cas où aucun provider n’est enregistré pour un service XService vous obtenez l’erreur suivante:

```
StaticInjectorError(xModule)[xxxxxxx -> xxxxxxxx]:
StaticInjectorError(Platform: core)[xxxxxx -> xxxxxxx]:
NullInjectorError: No provider for XService!
```

- Il faut alors définir le provider de XService

Définir un provider dans providers []

provider	description
@NgModule({ ….. providers: [SomeService] })	• L'enregistrement du service dans un @NgModule le rendra disponible **pour toutes les déclarations du module et pour tout autre module qui partage le même ModuleInjector.** • Une instance du service par module • **Déclarer un service de cette manière fait de sorte que ce service est toujours inclus dans votre application, même si le service n'est pas utilisé.**
@Component({ ….. providers: [SomeService] })	• L'enregistrement du service dans un @Component le rendra disponible **pour tout instance de ce composant ainsi que pour les autres composants et directives utilisés dans son template.** • Avec chaque nouvelle instance du composant, une nouvelle instance du service est créée. • **Déclarer un service de cette manière fait en sorte que ce service soit toujours inclus dans votre application, même si le service n'est pas utilisé.**
app.config.ts export const appConfig: ApplicationConfig = {providers: [{ provide: SomeService },]}; main.ts: bootstrapApplication(AppComponent, appConfig)	• L'enregistrement du service à travers ApplicationConfig le rendra disponible dans toute l'application. • La même insatnce sera partagée pour tous. • **Déclarer un service de cette manière fait de sorte que ce service est toujours inclus dans votre application, même si le service n'est pas utilisé.**

ProvidedIn - Valeurs

provider	description
@Injectable({ providedIn: 'root' })	Une seule instance du service disponible dans toute l'application (y compris les modules lazy loaded)
@Injectable({ providedIn: 'any' })	Une instance du service est créée **pour chaque ModuleInjector** qui l'utilise donc **pour chaque module lazy loaded** Le reste de l'application y compris les modules qui ne sont pas lazy loaded partage l'instance disponible au niveau du module racine.
@Injectable({ providedIn: 'plateform' })	Ceci est utile si vous avez plusieurs applications angular en cours d'exécution sur une seule page. Il s'agit d'une option utile si vous utilisez Angular Elements, où ils peuvent partager une seule instance de service entre eux
@Injectable({ providedIn: NomModule })	En fournissant le nom d'un module spécifique (par exemple 'my-module'), le service est enregistré uniquement dans ce module et ses sous-modules. Cela signifie que chaque instance du module aura sa propre instance du service.

ProvidedIn - Avantages

Avantages de providedIn:

- Simple et rapide à implémenter
- Ne charger le code des services qu'à la première injection
- Si le service n'est jamais utilisé, le code du service sera entièrement retiré du build final de l'application.
- *L'utilisation de providedIn permet aux optimisateurs de code Angular et JavaScript de supprimer efficacement les services non utilisés (appelé tree-shaking).*

Exemple d’application

Exemple d’application

- Soit un service appelé LogService

```
@Injectable({
providedIn : 'root'
})
export class LogService {
log(msg: any) { console.log(msg); }
error(msg: any) { console.error(msg); }
warn(msg: any) { console.warn(msg); }
}
```

- Ce service offre trois méthodes : log(msg), error(msg) et warn(msg). Ces méthodes sont souvent utilisées par diverses classes.

Injecter un service

- Injecter un service au niveau du (composant ou service ou classe) qui a besoin

```
constructor(private lservice:LogService){}
```

- Utiliser l'une ou les méthodes de ce service en fonction du besoin.

```
log(){
this.lservice.log("finish");
}
```

Injection dans le constructor() ou inject()

Situation	Solution classique	Solution moderne avec inject()
Dans un composant	constructor(private s: Service)	Pas utile
Dans un service	constructor(private s: Service)	private s = inject(Service)
Dans une factory provider	❌ impossible	✅ inject(Service)
Dans un guard, resolver	constructor(private s: Service)	✅ const s = inject(Service)
Dans une fonction de composition	❌ impossible	✅ inject(Service)

Les observables

Les observables - Définition

- Les observables représentent un nouveau standard pour la gestion des données asynchrones.
- Un **Observable** représente un *flux de données asynchrones* c'est-à-dire l'observable peut envoyer plusieurs valeurs au fil des temps d'une façon asynchrone.

- Les observables sont annulables (unsubscribe())

Les observables - Définition

- Ils ne sont pas une caractéristique spécifique à Angular.
- Angular utilise largement les observables dans le système d'événements et le service HTTP.
- Angular entrain d'utiliser les observables de la bibliothèque RxJS (bibliothèque Reactive X pour le langage JavaScript)
- Un **Observable est "froid",** il ne fait rien tant que personne ne s'y abonne.

Les observables dans Angular

Angular utilise les observables comme interface pour gérer diverses opérations asynchrones courantes. Tels que:

- Les classes EventEmitter extends observable.
- Les modules Router et Forms utilisent des observables pour écouter et répondre aux événements entrés par l'utilisateur.
- Le module HttpClientModule utilise des observables pour gérer les requêtes et les réponses HTTP.

```
getPizzas(): Observable <Pizza[]> {
 return this.http.get<Pizza[]>(urlAPI) ;
}
```

Les observables - inscription

- L’inscription à un observable peut se faire avec:

  Les pipes asynchrones « asyncPipe »

  La fonction subscribe()

- Une fois l’inscription à un observable est établie, ce dernier émet le flux des données.
- Un traitement doit être fait une fois le flux des données est émis.

La méthode subscribe()

- Un **Observable** ne fait *rien* tant que tu n'appelles pas subscribe().
- La fonction subscribe() permet de s'inscrire à un observable pour récupérer les données.
- Elle peut recevoir **jusqu'à trois callbacks** :

```
.subscribe({
  next: (data) => console.log('Données reçues', data),
  error: (err) => console.error('Erreur', err),
  complete: () => console.log('Requête terminée')
});
```

Callback	Rôle
next()	Appelé quand des **données** arrivent
error()	Appelé si une **erreur** survient
complete()	Appelé quand le flux est **terminé**

Les observables - async pipe

- Les **pipes asynchrones** (le **async pipe**) décrivent un mécanisme Angular qui permet d'**afficher automatiquement la valeur d'un Observable ou d'une Promise dans le template**, *sans avoir besoin de subscribe dans le TypeScript*.
- Le **async pipe** :
  - s'abonne **automatiquement** à un Observable/Promise
  - affiche la dernière valeur émise
  - se désabonne **automatiquement** quand le composant est détruit
  - met à jour la vue automatiquement grâce au *change detection*

  => **C'est un subscribe automatique dans le template.**

Async pipe – Exemple avec of()

- of() crée un Observable qui **émet immédiatement** la valeur puis **se complète**.

Component.ts

```
// Observable simple qui émet une seule valeur
  message$ = of('Bonjour depuis Observable !');

  // Observable de tableau
  users$ = of([
    { id: 1, name: 'Ameni' },
    { id: 2, name: 'Nour' },
    { id: 3, name: 'Sami' }
  ]);
```

Component.html

```
<h2>Message :</h2>
<p>{{ message$ | async }}</p>

<h2>Liste d'utilisateurs :</h2>
<ul>
  <li *ngFor="let user of users$ | async">
    {{ user.name }}
  </li>
</ul>
```

Async pipe – Exemple avec HttpClient

Service.ts

```
constructor(private http: HttpClient) {}

  getUsers(): Observable<any[]> {
    return this.http.get<any[]>('https://jsonplaceholder.typicode.com/users');
  }
```

Component.ts

```
  users$ = this.userService.getUsers();  // Observable HttpClient

  constructor(private userService: UserService) {}
}
```

Component.html

```
<h2>Liste des utilisateurs</h2>

<ul>
  <li *ngFor="let user of users$ | async">
    {{ user.name }} ({{ user.email }})
  </li>
</ul>
```

Async Pipe VS subscribe()

Le async pipe est recommandé lorsque :

- Tu veux afficher des données dans le template
- Tu n'as pas besoin d'effectuer une action impérative dans le TypeScript:

  Exemple : tu ne veux pas faire de calcul, appel serveur en chaîne, logique métier dans subscribe, etc.
- Tu veux éviter les fuites mémoire

  async pipe *unsubscribe automatiquement*, contrairement à subscribe().
- Tu veux un code propre et déclaratif

subscribe() est nécessaire quand:

- Tu dois **manipuler la valeur dans le TS** avant l'affichage
- Tu veux **lancer une action dans subscribe (ex : navigation, toast)**

La méthode pipe()

- pipe() est **une méthode des Observables** qui sert à **enchaîner des opérateurs RxJS** pour transformer les données, gérer les erreurs, filtrer…
- un "pipeline" où tu fais passer les données et appliques des transformations **dans l'ordre**.

```
this.http.get('/api/users')
  .pipe(
    map(users => users.filter(u => u.active)),   // transformation
    tap(() => console.log('Request done')),      // effet secondaire
  )
  .subscribe(result => console.log(result));
```

HttpClient → pipe(map → filter → tap) → subscribe()

RxJS

RxJS – C’est quoi ?

- RxJS est une **bibliothèque JavaScript** qui permet de travailler avec **Reactive X (programmation réactive) en JavaScript**
- C’est l’outil que Angular utilise pour gérer :
  - les **Observables**
  - les **Streams** (flux de données)
  - les **opérateurs** (map, filter, tap, switchMap…)
  - les **événements asynchrones** (HTTP, formulaires, WebSockets, timers…)
- RxJS permet de manipuler des données comme un flux (stream), sur lequel on peut écouter, transformer et réagir.

Les opérateurs RxJS

Il existe **2 grands types d'opérateurs RxJS** :

- **Pipeable Operators :** Ceux que tu mets dans pipe()
  Ex : map, filter, tap, switchMap, catchError, etc.
- ***Creation Operators :*** Ce sont les opérateurs qui **créent un Observable**. Ils **ne vont pas dans pipe()**.

RxJS - Pipeable Operators

Élément	Rôle
Observable	Flux de données (ex: réponse HTTP)
pipe()	Permet d’appliquer des opérateurs au flux
map()	transforme les données
filter()	garde/ignore certaines valeurs
tap()	exécuter une action sans modifier
catchError()	gérer les erreurs

RxJS - Creation operator

Creation operator	Rôle
of()	Crée un observable à partir de valeurs
from()	Crée un observable à partir d'un tableau / Promise
interval()	Observable qui émet un nombre toutes les x ms
timer()	Observable après un délai
throwError()	Observable qui émet une erreur
forkJoin()	Combine plusieurs observables (une fois tous terminés)
combineLatest()	Combine plusieurs observables en continu
EMPTY	Observable vide
NEVER	Observable qui n'émet jamais

Exercices sur les observables et les opérateurs RxJS

Exercice 1 - Créer un Observable simple avec of() et l’afficher

Objectifs :

- Créer un Observable
- S’abonner avec un async pipe
- Manipuler une donnée simple

Consignes :

- Dans un composant, crée un Observable nommé message$ avec of("Bienvenue dans les Observables !")
- Affiche sa valeur dans le template avec async.

Résultat attendu :

- Le texte “Bienvenue dans les Observables !” s'affiche grâce au async pipe.

Exrcice 2 - Transformer un Observable avec map()

Objectif :

- Introduire pipe()
- Utiliser un opérateur de transformation

Consignes :

- Crée un Observable numbers$ contenant [1, 2, 3] avec of()
- Utilise map() pour transformer le tableau en [2, 4, 6]
- Affiche la version transformée avec async

Résultat attendu :

- Le template affiche : [2,4,6]

Exercice 3 — Utiliser interval() pour faire un compteur

Objectif :

- Découvrir un Observable infini
- Comprendre que async pipe gère l'abonnement automatiquement

Consignes :

- Crée un Observable counter$ = interval(1000)
- Affiche la valeur dans le template avec async

Résultat attendu :

- Un compteur s'incrémente toutes les secondes.

RQ: Ajoute .pipe(take(5)) pour arrêter après 5 valeurs.

Exercice 4 — Observer les changements d’un formulaire

Objectif :

- Utiliser un observable réel d’Angular (valueChanges)
- Observer un flux d’événements

Consignes:

- Crée un FormControl : myControl = new FormControl('');
- Abonne-le au template : <input [formControl]="myControl">
- Utilise valueChanges pour afficher ce que tape l’utilisateur :

TS : value$ = this.myControl.valueChanges;

HTML : {{ value$ | async }}

Résultat attendu :

- Chaque caractère saisi s’affiche en temps réel.

Exrcice 5 – s’inscrire à un observable avec subscribe()

Dans un composant Angular :

1. **Crée un Observable :** of("foulen")
2. **Dans le subscribe(), applique un traitement :**
   - Convertir le texte en MAJUSCULE
   - Ajouter un préfixe "Bonjour "
     Exemple de résultat final : **"BONJOUR FOULEN"**

   !!!Tu dois faire ça **à l’intérieur du subscribe**, pas dans le template.
3. **Stocke la valeur transformée dans la propriété result.**
4. **Affiche result dans le template :**

Exercice 1 - Correction

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

@Component({
  selector: 'app-example1',
  templateUrl: './example1.component.html'
})
export class Example1Component {
  message$ = of("Bienvenue dans les Observables !");
}
```

```
<p>{{ message$ | async }}</p>
```

Exercice 2 - Correction

```
import { Component } from '@angular/core';
import { of, map } from 'rxjs';

@Component({
  selector: 'app-example2',
  templateUrl: './example2.component.html'
})
export class Example2Component {

  numbers$ = of([1, 2, 3]).pipe(
    map(numbers => numbers.map(n => n * 2))  // => [2, 4, 6]
  );
}
```

```
<p>{{ numbers$ | async | json }}</p>
```

Exercice 3 - Correction

```
import { Component } from '@angular/core';
import { interval, take } from 'rxjs';

@Component({
  selector: 'app-example3',
  templateUrl: './example3.component.html'
})
export class Example3Component {

  counter$ = interval(1000).pipe(
    take(5)  // facultatif, mais évite un flux infini
  );
}
```

```
<h2>Compteur :</h2>
<p>{{ counter$ | async }}</p>
```

Exercice 4 - Correction

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

@Component({
  selector: 'app-example4',
  templateUrl: './example4.component.html'
})
export class Example4Component {

  myControl = new FormControl('');
  value$ = this.myControl.valueChanges;
}
```

```
<input [formControl]="myControl" placeholder="Tape quelque chose">
<p>Valeur : {{ value$ | async }}</p>
```

Exercice 5 - Correction

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

@Component({
  selector: 'app-example',
  templateUrl: './example.component.html'
})
export class ExampleComponent implements OnInit {
  result = '';
  ngOnInit(): void {
    const name$ = of("foulen");

    name$.subscribe(value => {
      const upper = value.toUpperCase();
      this.result = "BONJOUR " + upper;
    });
  }
}
```

```
<p>{{ result }}</p>
```

Néthographie

- https://angular.io/guide/
- https://guide-angular.wishtack.io/angular
- https://developer.mozilla.org/fr/docs/Web/HTTP/CORS
- https://betterprogramming.pub/setup-a-proxy-for-api-calls-for-your-angular-cli-app-6566c02a8c4d

Merci de votre attention