# Injecter un service angular

## 1. Objectifs

- Recevoir un service avec la fonction `inject()` ou par le constructeur, et passer de l'un à l'autre
- Injecter un service dans un composant, une directive, un pipe, un autre service ou une fonction
- Utiliser les services fournis par Angular : `ElementRef`, `Title`, `DOCUMENT`, `LOCALE_ID`, `DestroyRef`
- Comprendre le **contexte d'injection** et éviter l'erreur NG0203
- Écrire ses propres **fonctions d'injection** (`injectXxx()`)
- Contrôler la recherche avec les options `optional`, `self` et `skipSelf`, et choisir entre `providers` et `viewProviders`
- Remplacer un service par un faux dans un test

## 1. Présentation

- Le chapitre précédent a montré comment **écrire** un service. Celui-ci s'intéresse à l'autre côté : comment le **recevoir**. Injecter un service, c'est demander à Angular l'instance dont on a besoin au lieu de la créer soi-même.
- En pratique, presque tout passe par une seule fonction, `inject()`. Mais il faut savoir **où** on peut l'appeler, **à quel moment**, et comment dire à Angular **où chercher**.

- Pour suivre ce chapitre, continuez dans le projet `premiers-pas-angular` et créez un dossier `src/app/injecter/`. Tous les exemples ont été testés avec Angular 22.2 (composants `OnPush` par défaut, sans Zone.js).

## 1. Deux façons de recevoir un service

- Un composant peut recevoir un service de deux façons : avec la fonction `inject()`, dans l'initialisation d'un champ, ou comme **paramètre du constructeur**. Le résultat est le même : la même instance du service.

```
import { Component, Service, inject, signal } from '@angular/core';

@Service()
export class Journal {
```

apcpedagogie.com

```
  private readonly _lignes = signal<string[]>([]);
  readonly lignes = this._lignes.asReadonly();

  ecrire(texte: string): void {
    this._lignes.update((l) => [...l, texte]);
  }
}

// 1. La fonction inject(), dans l'initialisation d'un champ (écriture recommandée)
@Component({
  selector: 'app-avec-inject',
  template: `<button (click)="journal.ecrire('clic avec inject()')">inject()</button>`,
})
export class AvecInject {
  protected readonly journal = inject(Journal);
}

// 2. Un paramètre du constructeur (écriture classique, encore très répandue)
@Component({
  selector: 'app-avec-constructeur',
  template: `<button (click)="journal.ecrire('clic avec le
constructeur')">constructeur</button>`,
})
export class AvecConstructeur {
  constructor(protected readonly journal: Journal) {}
}

@Component({
  selector: 'app-deux-ecritures',
  imports: [AvecInject, AvecConstructeur],
  template: `
    <app-avec-inject /> <app-avec-constructeur />
    <ol>
      @for (l of journal.lignes(); track $index) { <li>{{ l }}</li> }
    </ol>
  `,
})
export class DeuxEcritures {
  protected readonly journal = inject(Journal); // la même instance pour les trois composants
}
```

```
[inject()] [constructeur]
(clic sur les deux boutons)
```

apcpedagogie.com

```
1. clic avec inject()
2. clic avec le constructeur      ← les trois composants partagent le même Journal
```

## Deux façons de recevoir un service

La fonction inject() (recommandée) ou un paramètre du constructeur

**Avec inject()**

```
export class Fiche {
  private readonly journal = inject(Journal);
  private readonly title = inject(Title);
}
```

**Avec le constructeur**

```
export class Fiche {
  constructor(
    private readonly journal: Journal,
    private readonly title: Title,
  ) {}  }
```

| | inject() | constructeur |
|---|---|---|
| Où ? | champ, constructeur, fonction | paramètres du constructeur |
| Type requis | une classe ou un InjectionToken | une classe (sinon NG2003) |
| Options | `inject(X, { optional: true })` | `décorateurs @Optional()…` |
| Dans un @Service | oui | `non : erreur NG2028` |
| Hors d'une classe | oui (fonctions d'injection) | non |

- `inject()` est l'écriture recommandée par la documentation actuelle : le type est déduit, les options s'écrivent simplement, et elle fonctionne aussi hors d'une classe.
- L'injection par le constructeur reste valable dans un composant, une directive, un pipe ou un `@Injectable`, et on la rencontre dans beaucoup de projets. Elle est en revanche **refusée** dans un `@Service` (erreur NG2028, vue au chapitre précédent).
- Le constructeur a besoin d'un type qui existe à l'exécution : une classe. Avec une interface, la compilation échoue : `NG2003: No suitable injection token for parameter 'config' of class 'Fiche'`. On utilise alors un `InjectionToken` et `inject()`.

- Pour convertir du code existant, Angular fournit une migration automatique :

```
$ ng generate @angular/core:inject --path src/app/annonces
UPDATE src/app/annonces/tableau-annonces.ts (1154 bytes)
```

apcpedagogie.com

1. ## Où peut-on injecter ?

    - `inject()` fonctionne partout où **Angular crée l'objet**: un composant, mais aussi une directive, un pipe ou un autre service. Dans l'exemple suivant, le service `Langue` est injecté à plusieurs endroits :
        - dans le service `Traductions`, qui en dépend ;
        - dans la directive `appLangue`, qui injecte aussi `ElementRef` pour atteindre son élément hôte ;
        - dans le composant, qui change la langue ;
        - et le pipe `traduire` injecte `Traductions`.

```
import { Component, Directive, ElementRef, Pipe, PipeTransform, Service, effect, inject, signal
} from '@angular/core';

type Code = 'fr' | 'en';

// Un service…
@Service()
export class Langue {
  private readonly _code = signal<Code>('fr');
  readonly code = this._code.asReadonly();
  choisir(code: Code): void {
    this._code.set(code);
  }
}

// … injecté dans un autre service
@Service()
export class Traductions {
  private readonly langue = inject(Langue);
  private readonly mots: Record<string, Record<Code, string>> = {
    bonjour: { fr: 'Bonjour', en: 'Hello' },
    panier: { fr: 'Votre panier', en: 'Your cart' },
  };
  t(cle: string): string {
    return this.mots[cle]?.[this.langue.code()] ?? cle;
  }
}

// … dans un pipe (pure: false, car le résultat dépend de la langue, pas seulement de la clé)
@Pipe({ name: 'traduire', pure: false })
export class TraduirePipe implements PipeTransform {
```

apcpedagogie.com

```
  private readonly traductions = inject(Traductions);
  transform(cle: string): string {
    return this.traductions.t(cle);
  }
}

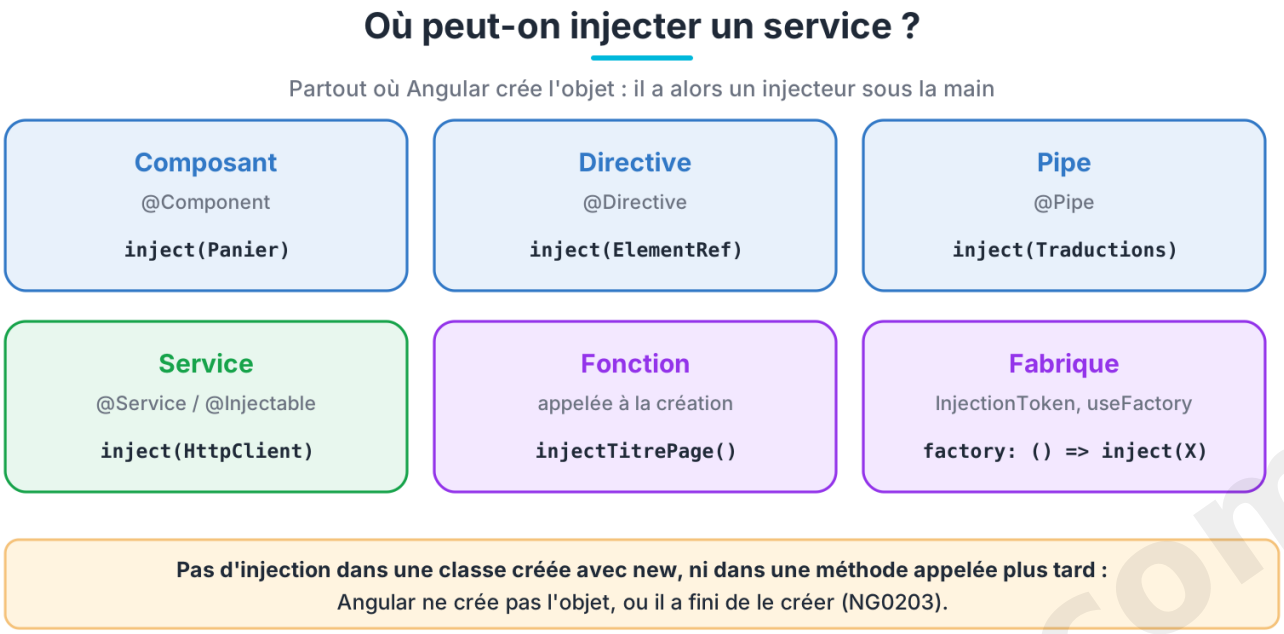
// … et dans une directive, avec l'élément hôte (ElementRef)
@Directive({ selector: '[appLangue]' })
export class LangueHote {
  private readonly hote = inject<ElementRef<HTMLElement>>(ElementRef);
  private readonly langue = inject(Langue);
  constructor() {
    effect(() => this.hote.nativeElement.setAttribute('lang', this.langue.code()));
  }
}

@Component({
  selector: 'app-ou-injecter',
  imports: [TraduirePipe, LangueHote],
  template: `
    <div appLangue>
      <p>{{ 'bonjour' | traduire }} ! {{ 'panier' | traduire }}</p>
    </div>
    <button (click)="langue.choisir('fr')">FR</button>
    <button (click)="langue.choisir('en')">EN</button>
  `,
})
export class OuInjecter {
  protected readonly langue = inject(Langue);
}
```

```
Bonjour ! Votre panier            <div lang="fr">
(clic sur EN)
Hello ! Your cart                 <div lang="en">
```

- Le pipe est déclaré `pure: false`: un pipe pur ne recalcule que si sa valeur d'entrée change, et ici la clé (`'bonjour'`) reste la même quand la langue change.
- `inject<ElementRef<HTMLElement>>(ElementRef)` précise le type de l'élément : `nativeElement` est alors un `HTMLElement`.

apcpedagogie.com




## 1. Injecter les services d'Angular

- Angular fournit lui-même de nombreux services, que l'on injecte exactement comme les siens :
  - `Title` (de `@angular/platform-browser`) lit et change le titre de l'onglet ;
  - `DOCUMENT` donne le document de la page, et `document.defaultView` la fenêtre ;
  - `LOCALE_ID` est un **jeton** (`InjectionToken`) : il ne fournit pas un service mais une valeur, la langue configurée ;
  - `DestroyRef` prévient de la destruction du composant ; `ElementRef` donne l'élément hôte.

```
import { Component, DOCUMENT, DestroyRef, LOCALE_ID, inject, signal } from '@angular/core';
import { Title } from '@angular/platform-browser';

@Component({
  selector: 'app-services-angular',
  template: `
    <p>Langue de l'application : {{ locale }}</p>
    <p>Largeur de la page : {{ largeur() }} px</p>
    <button (click)="renommer()">Changer le titre de l'onglet</button>
    <p>Titre actuel : {{ titre() }}</p>
  `,
})
export class ServicesAngular {
```

```
  private readonly title = inject(Title);                 // le titre de l'onglet
  private readonly document = inject(DOCUMENT);           // le document de la page
  protected readonly locale = inject(LOCALE_ID);          // un jeton : la langue configurée
  private readonly destroyRef = inject(DestroyRef);       // la destruction du composant

  protected readonly titre = signal(this.title.getTitle());
  protected readonly largeur = signal(this.document.documentElement.clientWidth);

  constructor() {
    const fenetre = this.document.defaultView!;
    const mesurer = () => this.largeur.set(this.document.documentElement.clientWidth);
    fenetre.addEventListener('resize', mesurer);
    this.destroyRef.onDestroy(() => fenetre.removeEventListener('resize', mesurer));
  }

  protected renommer(): void {
    this.title.setTitle('Injecter un service');
    this.titre.set(this.title.getTitle());
  }
}
```

```
Langue de l'application : fr
Largeur de la page : 900 px
(clic sur le bouton)              Titre actuel : Injecter un service   (et l'onglet change)
(fenêtre réduite à 700 px)        Largeur de la page : 700 px
```

- Pourquoi `inject(DOCUMENT)` plutôt que la variable globale `document`? Parce qu'il se remplace dans un test et fonctionne aussi quand l'application est rendue sur le serveur.
- L'écouteur `resize` est retiré à la destruction du composant : sans cela, il resterait actif.

## 1. Le contexte d'injection

- `inject()` a besoin de savoir **qui** demande, pour chercher dans le bon injecteur. Angular ne le sait que pendant la **création** de l'objet : c'est le **contexte d'injection**.
- On peut donc appeler `inject()` dans l'initialisation d'un champ, dans le constructeur, ou dans une fonction appelée depuis l'un d'eux. Pas dans une méthode appelée plus tard, au clic par exemple.
- Quand on a besoin d'injecter plus tard, on garde l'`Injector` dans un champ, puis on le passe à `runInInjectionContext()`, ou à `effect()` avec l'option `injector`:

apcpedagogie.com

```
import { Component, Injector, effect, inject, runInInjectionContext, signal } from
'@angular/core';
import { Journal } from './deux-ecritures';

@Component({
  selector: 'app-contexte',
  template: `
    <button (click)="compteur.update((n) => n + 1)">+1 ({{ compteur() }})</button>
    <button (click)="suivre()">Suivre le compteur</button>
    <button (click)="mauvais()">inject() dans une méthode</button>
    <ul>
      @for (l of journal.lignes(); track $index) { <li>{{ l }}</li> }
    </ul>
  `,
})
export class Contexte {
  // Contexte d'injection : l'initialisation des champs…
  protected readonly journal = inject(Journal);
  private readonly injector = inject(Injector);
  protected readonly compteur = signal(0);

  constructor() {
    // … et le constructeur
    this.journal.ecrire('Composant créé');
  }

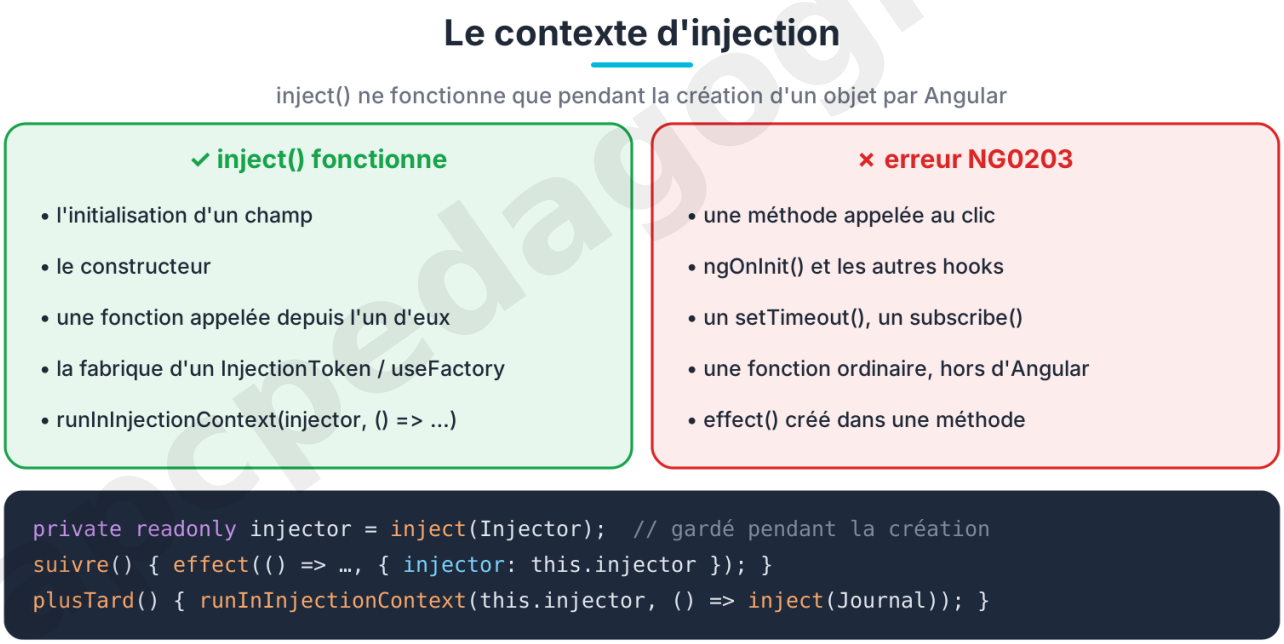
  // Une méthode n'est PAS un contexte d'injection : on fournit l'injecteur gardé dans un champ
  protected suivre(): void {
    effect(() => this.journal.ecrire(`Compteur : ${this.compteur()}`), { injector: this.injector
});
  }

  protected mauvais(): void {
    try {
      inject(Journal);
    } catch (e) {
      this.journal.ecrire((e as Error).message.split('.')[0]);
    }
    // runInInjectionContext crée un contexte à partir d'un injecteur
    runInInjectionContext(this.injector, () => {
      inject(Journal).ecrire('inject() dans runInInjectionContext : OK');
    });
  }
}
```

apcpedagogie.com

```
Composant créé
(+1, puis « Suivre le compteur », puis +1 deux fois)
Compteur : 1
Compteur : 2
Compteur : 3
(« inject() dans une méthode »)
NG0203: The `Journal` token injection failed
inject() dans runInInjectionContext : OK
```

- Le message complet : `NG0203: The `Journal` token injection failed. `inject()` function must be called from an injection context such as a constructor, a factory function, a field initializer, or a function used with `runInInjectionContext`.`
- `effect()` utilise lui aussi le contexte d'injection : créé dans une méthode sans l'option `injector`, il provoque la même erreur NG0203.
- `ngOnInit()` n'est **pas** un contexte d'injection : on injecte dans les champs, et on utilise ces champs dans `ngOnInit()`.




## 1. Écrire ses fonctions d'injection

- Puisque `inject()` marche dans une fonction appelée pendant la création, on peut regrouper un comportement réutilisable dans une **fonction d'injection**. Par convention, son nom commence par `inject`.

- Ici, `injectTitrePage()` change le titre de l'onglet et le remet en partant, et `injectDefilement()` renvoie la position du défilement sous forme de signal :

```
import { Component, DOCUMENT, DestroyRef, Signal, computed, inject, signal } from
'@angular/core';
import { Title } from '@angular/platform-browser';

// Une fonction d'injection : elle appelle inject(), donc elle s'utilise dans un contexte
d'injection
export function injectTitrePage(titre: string): void {
  const title = inject(Title);
  const ancien = title.getTitle();
  title.setTitle(titre);
  inject(DestroyRef).onDestroy(() => title.setTitle(ancien)); // remis en partant
}

// Une autre : la position du défilement, sous forme de signal, nettoyée automatiquement
export function injectDefilement(): Signal<number> {
  const fenetre = inject(DOCUMENT).defaultView!;
  const position = signal(Math.round(fenetre.scrollY));
  const mesurer = () => position.set(Math.round(fenetre.scrollY));
  fenetre.addEventListener('scroll', mesurer, { passive: true });
  inject(DestroyRef).onDestroy(() => fenetre.removeEventListener('scroll', mesurer));
  return position.asReadonly();
}

@Component({
  selector: 'app-page-aide',
  template: `
    <p>Défilement : {{ defilement() }} px</p>
    @if (enBas()) { <p>Vous avez parcouru plus de 300 px.</p> }
  `,
})
export class PageAide {
  constructor() {
    injectTitrePage('Aide · Mon application');
  }
  protected readonly defilement = injectDefilement();
  protected readonly enBas = computed(() => this.defilement() > 300);
}

@Component({
  selector: 'app-fonctions-inject',
  imports: [PageAide],
```

apcpedagogie.com

```
  template: `
    <button (click)="visible.update((v) => !v)">{{ visible() ? 'Fermer' : 'Ouvrir' }}
l'aide</button>
    @if (visible()) { <app-page-aide /> }
  `,
})
export class FonctionsInject {
  protected readonly visible = signal(false);
}
```

```
(Ouvrir l'aide)         onglet : « Aide · Mon application »     Défilement : 0 px
(défilement de 450 px)  Défilement : 450 px
                        Vous avez parcouru plus de 300 px.
(Fermer l'aide)         l'ancien titre revient, l'écouteur est retiré
```

- Le composant tient en trois lignes : toute la mécanique (écouteur, nettoyage, titre) est dans des fonctions réutilisables dans n'importe quel composant.
- Une fonction d'injection s'appelle dans un contexte d'injection : un champ ou le constructeur, jamais une méthode.
- Elle peut renvoyer une valeur (ici un signal en lecture seule) ou ne rien renvoyer et simplement agir.

1. **Les options de inject()**
    - Quand on écrit `inject(X)`, Angular cherche un fournisseur de `X` en partant de l'élément qui demande, puis remonte de parent en parent jusqu'à l'injecteur racine. S'il ne trouve rien : erreur NG0201. Un second argument modifie cette recherche :
        - `{ optional: true }`: renvoyer `null` au lieu de l'erreur ;
        - `{ self: true }`: ne chercher que dans l'injecteur de l'élément lui-même ;
        - `{ skipSelf: true }`: commencer au parent, en ignorant son propre fournisseur ;
        - `{ host: true }`: arrêter la recherche au composant hôte.
    - Dans l'exemple, chaque bloc fournit son propre `Niveau`, qui demande celui du bloc parent avec `skipSelf` et `optional` pour calculer sa profondeur :

```
import { Component, Injectable, inject, input } from '@angular/core';

// Chaque bloc fournit son propre Niveau, qui cherche celui du bloc parent
@Injectable()
export class Niveau {
```

apcpedagogie.com

```
  // skipSelf : ignorer l'injecteur qui me fournit et chercher au-dessus
  // optional : renvoyer null s'il n'y a rien (le bloc le plus haut)
  private readonly parent = inject(Niveau, { skipSelf: true, optional: true });
  readonly profondeur: number = this.parent ? this.parent.profondeur + 1 : 1;
}

@Component({
  selector: 'app-bloc',
  providers: [Niveau],
  template: `
    <div class="bloc" [style.margin-left.px]="(niveau.profondeur - 1) * 20">
      <strong>{{ titre() }}</strong> : niveau {{ niveau.profondeur }}
    </div>
    <ng-content />
  `,
})
export class Bloc {
  readonly titre = input.required<string>();
  protected readonly niveau = inject(Niveau, { self: true }); // self : uniquement mon propre
injecteur
}

@Component({
  selector: 'app-options',
  imports: [Bloc],
  template: `
    <app-bloc titre="Cours">
      <app-bloc titre="Chapitre">
        <app-bloc titre="Section" />
      </app-bloc>
      <app-bloc titre="Annexe" />
    </app-bloc>
  `,
})
export class Options {}
```

```
Cours : niveau 1
    Chapitre : niveau 2
        Section : niveau 3
    Annexe : niveau 2
```

- Sans `skipSelf`, `Niveau` se demanderait lui-même : `NG0200: Circular dependency detected for `Niveau`. Path: Bloc -> Niveau -> Niveau.`

apcpedagogie.com

- Sans `optional`, le bloc « Cours », qui n'a pas de parent, provoquerait `NG0201: No provider found for \`Niveau\`.`
- `self: true` dans `Bloc` garantit que le bloc utilise son propre `Niveau`. Si l'on retire ses `providers`, l'erreur est immédiate (`NG0201: No provider for Niveau found in NodeInjector`) au lieu d'emprunter en silence celui du parent.
- Les blocs imbriqués par `<ng-content>` ont pour parent le bloc qui les entoure dans le template : la recherche suit cette imbrication.




```
// le Niveau du bloc parent, ou null pour le bloc le plus haut
private readonly parent = inject(Niveau, { skipSelf: true, optional: true });
readonly profondeur = this.parent ? this.parent.profondeur + 1 : 1;
```

## 1. providers ou viewProviders ?

- Un composant peut fournir un service de deux façons. `providers` le rend visible par sa vue **et** par le contenu qu'on lui projette avec `<ng-content>`. `viewProviders` le réserve à sa **vue**:

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

@Injectable()
export class Couleur {
  nom = 'par défaut';
}

@Component({
```

```
  selector: 'app-affiche-couleur',
  template: `<span>{{ couleur?.nom ?? 'aucune' }}</span>`,
})
export class AfficheCouleur {
  protected readonly couleur = inject(Couleur, { optional: true });
}

function couleur(nom: string): Couleur {
  const c = new Couleur();
  c.nom = nom;
  return c;
}

@Component({
  selector: 'app-cadre-providers',
  imports: [AfficheCouleur],
  providers: [{ provide: Couleur, useValue: couleur('bleu') }],
  template: `<p>providers · vue : <app-affiche-couleur /> · contenu projeté : <ng-content
/></p>`,
})
export class CadreProviders {}

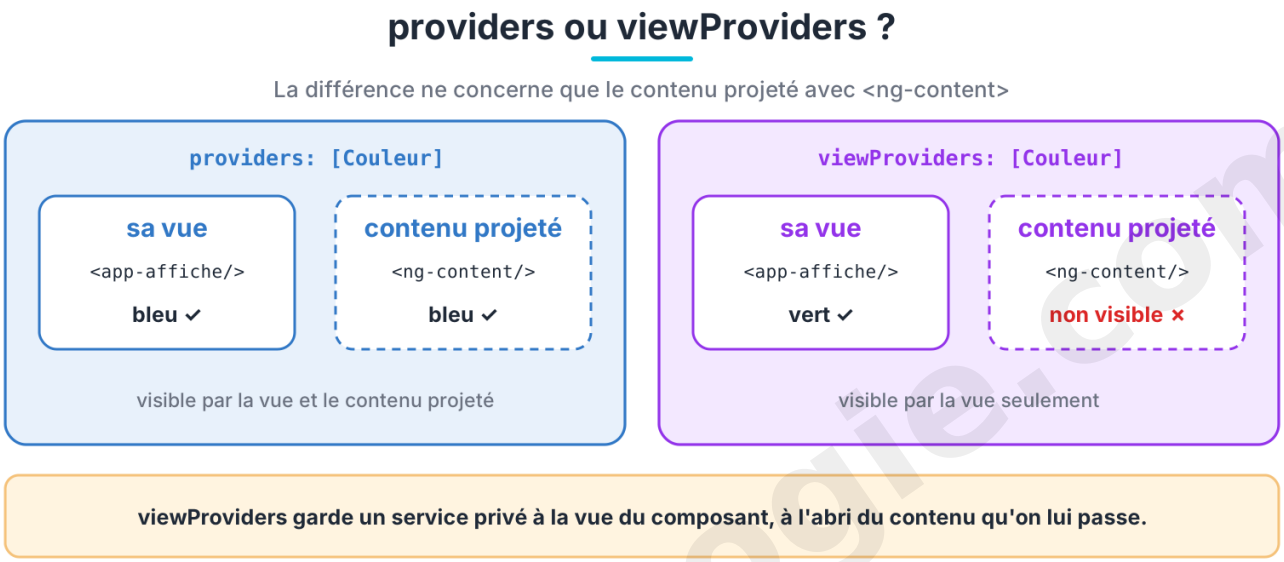
@Component({
  selector: 'app-cadre-view-providers',
  imports: [AfficheCouleur],
  viewProviders: [{ provide: Couleur, useValue: couleur('vert') }],
  template: `<p>viewProviders · vue : <app-affiche-couleur /> · contenu projeté : <ng-content
/></p>`,
})
export class CadreViewProviders {}

@Component({
  selector: 'app-view-providers',
  imports: [CadreProviders, CadreViewProviders, AfficheCouleur],
  template: `
    <app-cadre-providers><app-affiche-couleur /></app-cadre-providers>
    <app-cadre-view-providers><app-affiche-couleur /></app-cadre-view-providers>
  `,
})
export class ViewProviders {}
```

```
providers · vue : bleu · contenu projeté : bleu
viewProviders · vue : vert · contenu projeté : aucune
```

apcpedagogie.com

- Le contenu projeté (le `<app-affiche-couleur />` placé entre les balises du cadre) ne voit pas le service des `viewProviders`: il cherche au-dessus du cadre, ne trouve rien, et `optional` renvoie `null`.
- `viewProviders` sert surtout aux composants de bibliothèque, pour garder un service interne à l'abri du contenu que l'utilisateur leur passe. Dans une application, `providers` suffit presque toujours.




1. **Remplacer un service dans un test**
    - Un composant qui injecte un service dépend de lui. Pour le tester sans serveur, on **remplace** le service par un faux, dans les `providers` du module de test. Le composant ne voit pas la différence : il demande `Meteo` et reçoit l'objet fourni.

```
import { Component, Service, inject, signal } from '@angular/core';
import { HttpClient } from '@angular/common/http';

export interface Bulletin {
  ville: string;
  temperature: number;
  ciel: string;
}

@Service()
export class Meteo {
  private readonly http = inject(HttpClient);
  private readonly _bulletin = signal<Bulletin | null>(null);
```

apcpedagogie.com

```
  readonly bulletin = this._bulletin.asReadonly();

  actualiser(): void {
    this.http.get<Bulletin>('meteo.json').subscribe((b) => this._bulletin.set(b));
  }
}

@Component({
  selector: 'app-carte-meteo',
  template: `
    @if (meteo.bulletin(); as b) {
      <p class="bulletin">{{ b.ville }} : {{ b.temperature }} °C, {{ b.ciel }}</p>
    } @else {
      <p class="bulletin">Aucun bulletin.</p>
    }
    <button (click)="meteo.actualiser()">Actualiser</button>
  `,
})
export class CarteMeteo {
  protected readonly meteo = inject(Meteo);
}
```

```
import { TestBed } from '@angular/core/testing';
import { signal } from '@angular/core';
import { CarteMeteo, Meteo, Bulletin } from './meteo';

// Un faux service : mêmes membres publics, aucune requête HTTP
class FausseMeteo {
  readonly bulletin = signal<Bulletin | null>(null);
  appels = 0;
  actualiser(): void {
    this.appels++;
    this.bulletin.set({ ville: 'Tunis', temperature: 18, ciel: 'nuageux' });
  }
}

describe('CarteMeteo', () => {
  let faux: FausseMeteo;

  beforeEach(() => {
    faux = new FausseMeteo();
    TestBed.configureTestingModule({
      providers: [{ provide: Meteo, useValue: faux }], // remplace le vrai service
    });
```

apcpedagogie.com

```
  });

  it('affiche « Aucun bulletin. » au départ', async () => {
    const fixture = TestBed.createComponent(CarteMeteo);
    await fixture.whenStable();
    expect(fixture.nativeElement.textContent).toContain('Aucun bulletin.');
  });

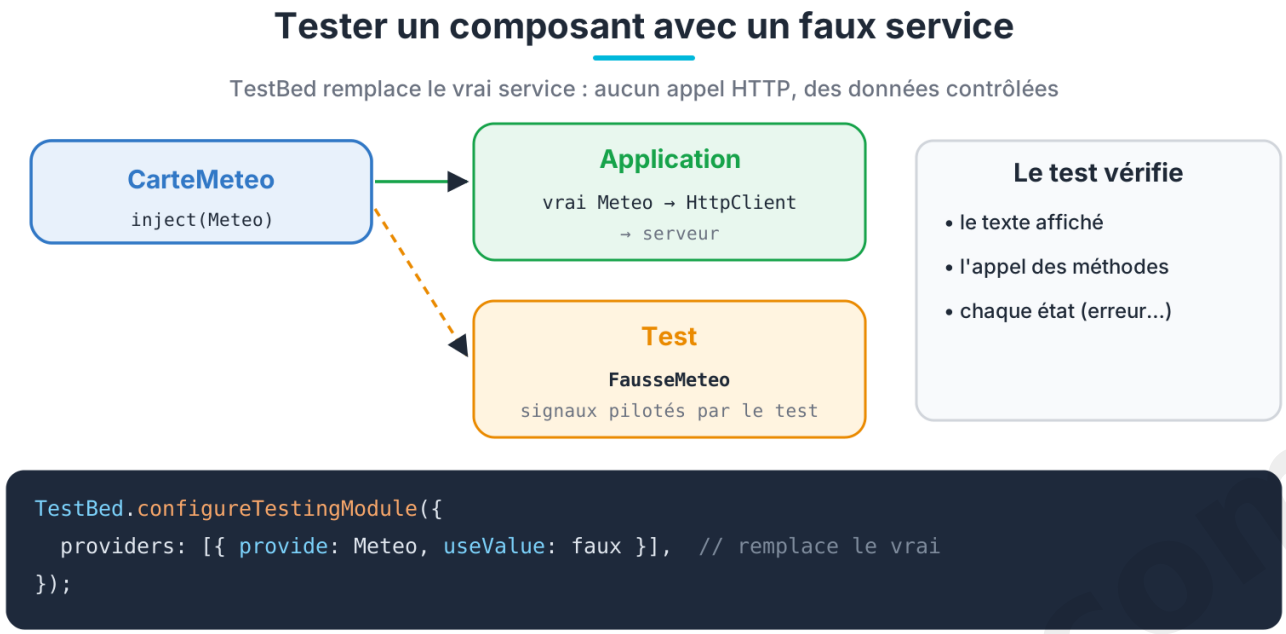
  it('demande un bulletin au clic sur Actualiser', async () => {
    const fixture = TestBed.createComponent(CarteMeteo);
    await fixture.whenStable();
    fixture.nativeElement.querySelector('button').click();
    await fixture.whenStable();
    expect(faux.appels).toBe(1);
    expect(fixture.nativeElement.textContent).toContain('Tunis : 18 °C, nuageux');
  });
});
```

```
$ ng test --include src/app/injecter/meteo.spec.ts
 ✓ CarteMeteo > affiche « Aucun bulletin. » au départ
 ✓ CarteMeteo > demande un bulletin au clic sur Actualiser

 Test Files  1 passed (1)
      Tests  2 passed (2)
```

- Le faux service a les **mêmes membres publics** que le vrai (`bulletin`, `actualiser()`) ; il compte en plus les appels pour que le test puisse les vérifier.
- `useValue` fournit un objet déjà créé ; `useClass: FausseMeteo` laisserait Angular le créer. C'est le mécanisme vu dans les principes de l'injection de dépendances.
- `await fixture.whenStable()` attend que la vue soit à jour, sans Zone.js.

apcpedagogie.com




```
TestBed.configureTestingModule({
  providers: [{ provide: Meteo, useValue: faux }],  // remplace le vrai
});
```

## 1. Aide-mémoire

apcpedagogie.com

## Aide-mémoire : injecter un service

| | |
|---|---|
| Injecter (recommandé) | `private readonly carnet = inject(Carnet);` |
| Écriture classique | `constructor(private readonly carnet: Carnet) {}` |
| Migrer vers inject() | `ng generate @angular/core:inject` |
| Élément hôte (directive) | `inject<ElementRef<HTMLElement>>(ElementRef)` |
| Services d'Angular | `inject(Title) · inject(DOCUMENT) · inject(LOCALE_ID)` |
| Dépendance facultative | `inject(X, { optional: true }) // ou null` |
| Chercher chez le parent | `inject(X, { skipSelf: true })` |
| Seulement chez soi | `inject(X, { self: true })` |
| Plus tard, hors création | `inject(Injector) + runInInjectionContext()` |
| effect() dans une méthode | `effect(fn, { injector: this.injector })` |
| Fonction d'injection | `function injectXxx() { const a = inject(A); … }` |
| Service privé à la vue | `viewProviders: [X]` |
| Faux service en test | `providers: [{ provide: X, useValue: faux }]` |
| Erreurs | `NG0201 introuvable · NG0203 hors contexte · NG0200 cycle` |

1. **Activité 1 : surligner un glossaire**
   - Créer le service `Recherche`: un signal privé `terme` exposé en lecture seule et une méthode `chercher(texte)` qui le met en minuscules, sans espaces autour.
   - Créer la directive `[appSurligner]`: elle injecte `ElementRef` et `Recherche`, expose un `computed` public `trouve` (le texte de l'élément contient le terme, sans tenir compte des majuscules) et ajoute la classe `trouve` à son hôte (fond jaune).
   - Le composant `Glossaire` affiche un champ (`ChampRecherche`, qui injecte lui aussi `Recherche`), six définitions marquées `appSurligner`, et le nombre de définitions trouvées, calculé avec `viewChildren(Surligner)`.

apcpedagogie.com

classe

4 définition(s) contiennent « classe »

**Composant** : une classe avec un template, qui affiche une partie de l'écran.

**Service** : une classe sans vue, qui garde les données et la logique.

**Directive** : une classe qui ajoute un comportement à un élément.

**Pipe** : une classe qui transforme une valeur dans le template.

**Injecteur** : l'objet qui crée et fournit les services.

**Signal** : une valeur qui prévient Angular quand elle change.

---

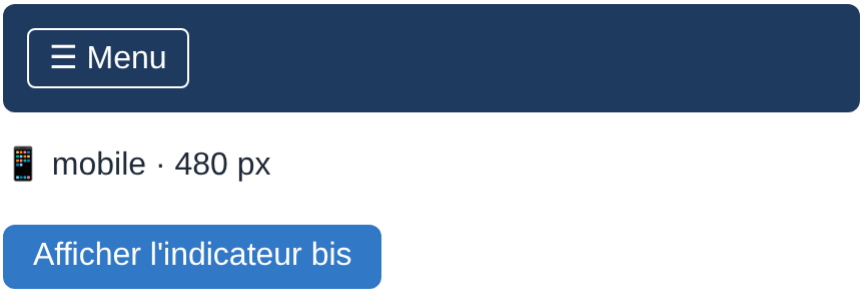
- Vérifier : « classe » → 4 définitions ; « SERVICES » → 1 ; champ vide → « Tapez un mot pour le surligner. ».
- Piège : lire `textContent` dans l'initialisation d'un champ de la directive donne un texte vide. Pourquoi ? Où faut-il le lire ?

1. ## Activité 2 : un écran adaptatif

   - Écrire la fonction d'injection `injectLargeurFenetre()`: elle renvoie un signal en lecture seule de `innerWidth`, mis à jour à chaque `resize`, et retire son écouteur à la destruction (« Écouteur resize retiré » dans la console).
   - Écrire `injectAppareil(largeur = injectLargeurFenetre())`: un signal `'mobile'` (moins de 600 px), `'tablette'` (moins de 1024 px) ou `'ordinateur'`.
   - `IndicateurEcran` affiche « ▭ tablette · 800 px » ; `MenuAdaptatif` affiche un bouton « ☰ Menu » sur mobile et quatre liens sinon. Un bouton affiche ou masque un second indicateur.




---




- Vérifier : sous 600 px, le menu devient « ☰ Menu » ; masquer le second indicateur écrit une seule fois « Écouteur resize retiré ».
- Pourquoi `IndicateurEcran` passe-t-il sa propre largeur à `injectAppareil()`? Combien d'écouteurs y aurait-il sans cela ?

1. **Activité 3 : moderniser un composant**
    - Voici un composant écrit avec l'injection par le constructeur. Le copier dans `src/app/annonces/tableau-annonces.ts` et l'afficher :

```
import { Component, DestroyRef, Service, signal } from '@angular/core';
import { Title } from '@angular/platform-browser';

@Service()
export class Annonces {
  private readonly _liste = signal(['Réunion des formateurs lundi', 'Examen de fin de module le
15']);
  readonly liste = this._liste.asReadonly();
  retirer(i: number): void {
    this._liste.update((l) => l.filter((_, j) => j !== i));
  }
}

// Ancienne écriture : les dépendances arrivent par le constructeur
@Component({
  selector: 'app-tableau-annonces',
  template: `
    <h4>Annonces ({{ annonces.liste().length }})</h4>
    <ul>
      @for (a of annonces.liste(); track a; let i = $index) {
        <li>{{ a }} <button (click)="annonces.retirer(i)">x</button></li>
      } @empty {
```

```
        <li>Aucune annonce.</li>
      }
    </ul>
  `,
})
export class TableauAnnonces {
  constructor(
    protected readonly annonces: Annonces,
    private readonly title: Title,
    destroyRef: DestroyRef,
  ) {
    const ancien = this.title.getTitle();
    this.title.setTitle('Annonces');
    destroyRef.onDestroy(() => this.title.setTitle(ancien));
  }
}
```

- Le convertir à la main en `inject()` : `annonces` et `title` deviennent des champs, et le constructeur n'a plus de paramètre.
- Annuler la modification, lancer la migration automatique `ng generate @angular/core:inject --path src/app/annonces`, puis comparer avec votre version.

**Annonces (2)**

Réunion des formateurs lundi ✕

Examen de fin de module le 15 ✕

---

- Vérifier : l'application fonctionne comme avant ; l'onglet s'appelle « Annonces » ; ✕ retire une annonce.
- Où la migration a-t-elle placé `DestroyRef`, qui n'était pas un champ ? Pourquoi ?

1. ## Activité 4 : un plan numéroté automatiquement

    - Créer le service `Numero` (`@Injectable()`, sans `providedIn`) : il injecte le `Numero` parent avec `{ skipSelf: true, optional: true }` et calcule sa `profondeur` (0 sans parent) et son `chemin` (« 1 », « 1.2 », « 1.2.1 »…), en demandant à son parent le rang suivant parmi ses enfants.

apcpedagogie.com

- Le composant `Plan` fournit un `Numero` racine et projette son contenu. Le composant `Partie` (entrée `titre`) fournit son propre `Numero`, l'injecte avec `{ self: true }`, affiche le numéro puis le titre, et projette ses sous-parties.
- Une partie placée hors d'un `Plan` s'affiche sans numéro.

---

1 Les services
1.1 Générer un service
1.2 Un service d'état
1.2.1 Le signal privé
1.2.2 Les méthodes
2 Injecter un service
2.1 La fonction inject()
2.2 Les options de inject()
3 Le routage

---

Une partie hors du plan

---

- Vérifier : 1, 1.1, 1.2, 1.2.1, 1.2.2, 2, 2.1, 2.2, 3, et « Une partie hors du plan » sans numéro.
- Que se passe-t-il si l'on retire `skipSelf`? Et si l'on retire `optional`?

## 1. Activité 5 : un annuaire testé sans serveur

- Placer `formateurs.json` dans `public/`. Créer le service `Annuaire` (`HttpClient`, signaux privés `formateurs` et `etat` exposés en lecture seule, méthode `charger()`) et le composant `ListeFormateurs`, qui appelle `charger()` à sa création et affiche « Chargement… », la liste, ou « Annuaire indisponible. ».
- Écrire `annuaire.spec.ts` avec une classe `FauxAnnuaire` fournie par `{ provide: Annuaire, useValue: faux }`, et trois tests : `charger()` est appelé une fois ; la liste affiche deux formateurs ; l'état `'erreur'` affiche le message.

---

Mme Gharbi · Angular

M. Jaziri · TypeScript

Mme Hamdi · HTML et CSS

M. Ferchichi · Bases de données

---

apcpedagogie.com

- Vérifier : `ng test` affiche 3 tests réussis, sans qu'aucune requête soit envoyée.
- Pourquoi le faux service peut-il avoir des signaux publics modifiables, alors que le vrai les protège ?

- Les corrigés des cinq activités, testés avec Angular 22, sont fournis dans l'archive de la leçon (dossier `corriges/`).

## 1. Erreurs fréquentes

| Message ou symptôme | Cause | Solution |
|---|---|---|
| `NG0203: The `Journal` token injection failed. `inject()` function must be called from an injection context …` | `inject()` dans une méthode, `ngOnInit()` ou un `setTimeout()` | Injecter dans un champ, ou utiliser `runInInjectionContext()` |
| NG0203 sur `effect()` | `effect()` créé dans une méthode | `effect(fn, { injector: this.injector })` |
| `NG0201: No provider found for `Niveau`.` | Service sans `providedIn` et aucun `providers` au-dessus | Le fournir, ou `inject(X, { optional: true })` |
| `NG0201: No provider for Niveau found in NodeInjector` | `{ self: true }` alors que l'élément ne fournit pas le service | Ajouter `providers: [Niveau]` au composant |
| `NG0200: Circular dependency detected for `Aa`. … Path: Aa -> Bb -> Aa.` | Deux services qui s'injectent l'un l'autre, ou un service qui se demande lui-même | Revoir le découpage ; `skipSelf` pour viser le parent |
| `NG2003: No suitable injection token for parameter 'config' …` | Paramètre du constructeur typé par une interface | Un `InjectionToken` et `inject(TOKEN)` |
| NG2028 dans un `@Service` | Injection par le constructeur | `inject()` dans un champ |
| Le contenu projeté ne trouve pas le service | Service fourni dans `viewProviders` | Utiliser `providers` |
| Le pipe ne se met pas à jour | Pipe pur qui dépend d'un service | `pure: false`, ou passer la valeur en argument |
| Le texte lu par une directive est vide | `textContent` lu à la création de la directive | Le lire plus tard, par exemple dans un `computed` |

apcpedagogie.com

## 1. À retenir

- `inject(X)` dans un champ est l'écriture recommandée ; l'injection par le constructeur reste valable hors des `@Service`, et `ng generate @angular/core:inject` fait la migration.
- On injecte dans un composant, une directive, un pipe, un service, une fabrique ou une fonction appelée à la création.
- `inject()` ne fonctionne que dans un contexte d'injection : champ, constructeur, fabrique, `runInInjectionContext()`. Sinon : NG0203.
- Une fonction `injectXxx()` regroupe un comportement réutilisable (écouteur, nettoyage, signal).
- `optional` renvoie `null`, `self` limite la recherche à soi, `skipSelf` la commence au parent.
- `viewProviders` réserve un service à la vue du composant, sans le contenu projeté.
- En test, `{ provide: X, useValue: faux }` remplace un service sans toucher au composant.

> *Un composant ne fabrique pas ses outils : il les demande au bon moment, au bon endroit. Bien maîtrisée, l'injection rend le code court, réutilisable et facile à tester.*

---

**Sommaire du cours Angular**

Source : https://apcpedagogie.com/injecter-un-service-angular · téléchargé le 07/11/2026 · © APC Pédagogie · apcpedagogie.com

apcpedagogie.com