Configurer l'injecteur Angular

1. Objectifs

- Comprendre ce qu'est un **provider**: un jeton et une recette
- Savoir **où** configurer l'injecteur : `app.config.ts`, un composant, une route, un injecteur d'environnement
- Utiliser les quatre recettes : `useClass`, `useValue`, `useFactory` et `useExisting`
- Créer des `InjectionToken`, avec ou sans valeur par défaut, et utiliser les options de `@Service()`
- Fournir plusieurs valeurs pour un même jeton avec `multi: true`
- Écrire sa propre fonction `provideXxx()` avec `makeEnvironmentProviders()`
- Exécuter du code au démarrage avec `provideAppInitializer()` et `provideEnvironmentInitializer()`
- Créer des injecteurs d'environnement enfants et configurer l'injecteur dans les tests

1. Présentation

- Dans les chapitres précédents, on a écrit des services (`@Service()`) puis on les a reçus avec `inject()`. Entre les deux, il y a l'**injecteur**: c'est lui qui sait quel objet donner pour chaque demande.
- **Configurer l'injecteur**, c'est lui donner des **providers**: des instructions du type « quand on te demande `Stockage`, crée un `StockageMemoire` ». On peut ainsi changer d'implémentation, fournir une configuration, choisir selon le contexte ou préparer l'application avant son affichage, **sans modifier les composants**.

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

1. Un provider : un jeton et une recette

- Un provider est un objet qui répond à deux questions :
  - `provide`: **quel jeton** est demandé ? C'est ce qu'on passe à `inject()`: une classe ou un `InjectionToken`;
  - `useClass`, `useValue`, `useFactory` ou `useExisting`: **quelle recette** utiliser pour obtenir la valeur.

apcpedagogie.com

- Quand on écrit `providers: [Panier]`, c'est une abréviation de `providers: [{ provide: Panier, useClass: Panier }]`: le jeton et la classe à créer sont les mêmes.



1. Où configurer l'injecteur ?

- Angular n'a pas un seul injecteur, mais un **arbre** d'injecteurs. Chaque niveau peut recevoir des providers :
  - **la racine**: le tableau `providers` de `app.config.ts`, les services `@Service()`, les jetons avec `factory`;
  - **les injecteurs d'environnement enfants**: les `providers` d'une route, ou `createEnvironmentInjector()`;
  - **les injecteurs d'élément**: les `providers` (et `viewProviders`) d'un composant ou d'une directive.
- Quand un composant appelle `inject(X)`, la recherche part de son propre élément et **remonte**: le provider le plus proche gagne. Ici, le jeton `SOURCE` a une valeur par défaut, que le composant `Zone` remplace pour son contenu. `NOM_APPLICATION` est fourni dans `app.config.ts`:

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

```
// Un jeton avec une valeur par défaut : utilisable même si personne ne le fournit
export const SOURCE = new InjectionToken<string>('SOURCE', {
  factory: () => 'valeur par défaut du jeton',
});

// Fourni dans app.config.ts : { provide: NOM_APPLICATION, useValue: 'Premiers pas Angular' }
export const NOM_APPLICATION = new InjectionToken<string>('NOM_APPLICATION');

@Component({
  selector: 'app-lire-source',
  template: `<p>{{ etiquette }} → <strong>{{ source }}</strong></p>`,
})
export class LireSource {
  protected readonly source = inject(SOURCE);
  protected readonly etiquette = inject(NOM_APPLICATION);
}

@Component({
  selector: 'app-zone',
  providers: [{ provide: SOURCE, useValue: 'providers du composant Zone' }],
  template: `<div class="zone"><ng-content /></div>`,
  styles: `.zone { border: 2px dashed #3178c6; padding: 4px 12px; border-radius: 8px; }`,
})
export class Zone {}

@Component({
  selector: 'app-niveaux',
  imports: [LireSource, Zone],
  template: `
    <app-lire-source />
    <app-zone>
      <app-lire-source />
    </app-zone>
  `,
})
export class Niveaux {}
```

```
// app.config.ts (extrait)
export const appConfig: ApplicationConfig = {
  providers: [
    provideBrowserGlobalErrorListeners(),
    provideRouter(routes),
    provideHttpClient(),
```

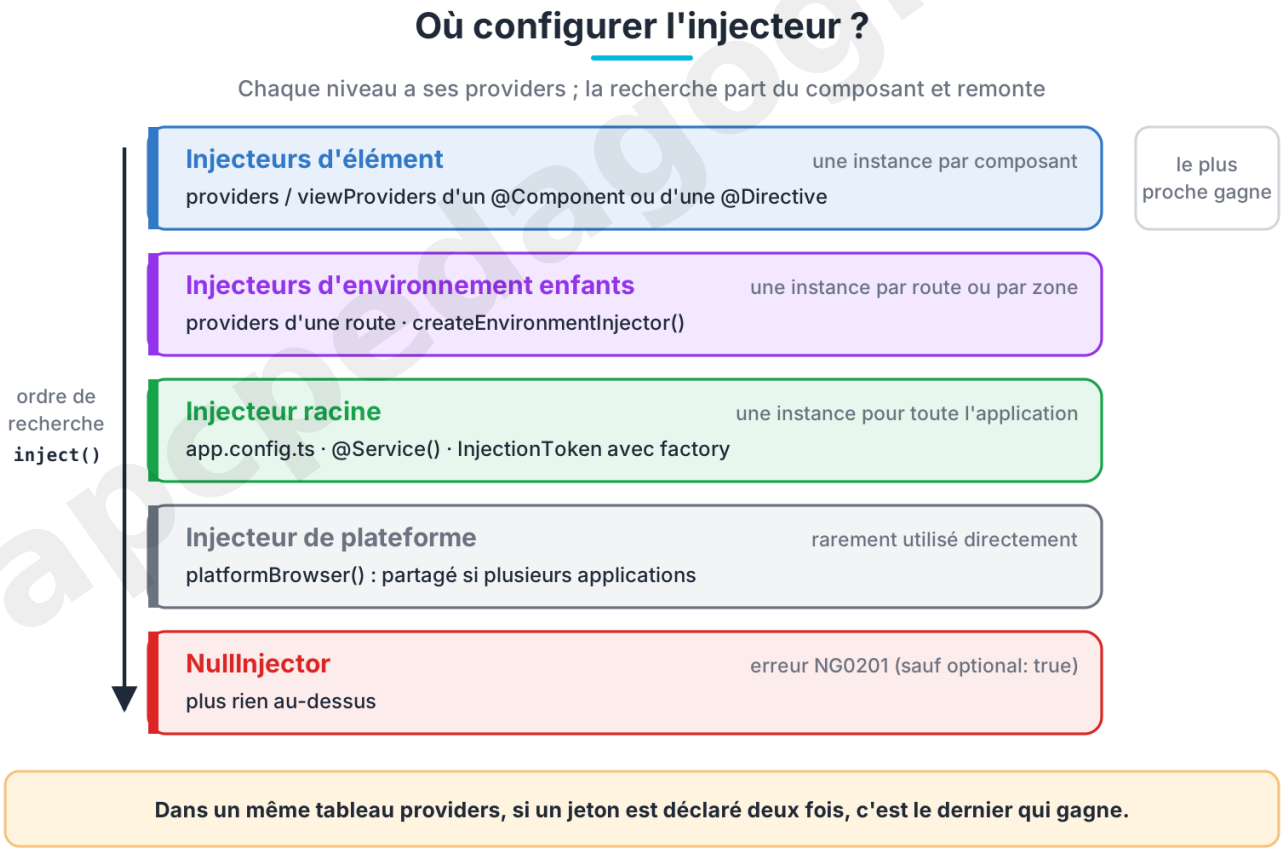
apcpedagogie.com

```
    { provide: NOM_APPLICATION, useValue: 'Premiers pas Angular' },
    provideSuivi({ prefixe: 'APP' }),   // fonction écrite plus loin
    provideParametres(),                // fonction écrite plus loin
  ],
};
```

Premiers pas Angular → **valeur par défaut du jeton**

Premiers pas Angular → **providers du composant Zone**

- Le premier `<app-lire-source>` ne trouve aucun provider de `SOURCE`: la **fabrique** du jeton est utilisée.
- Le second est dans `<app-zone>`: il trouve le provider du composant avant d'atteindre la racine.
- Dans un même tableau `providers`, si un jeton est déclaré deux fois, c'est le **dernier** qui gagne.



1. ## useClass : choisir l'implémentation

   - `useClass` demande à Angular de **créer une instance** d'une classe. Le jeton est souvent une **classe abstraite** qui décrit le contrat : les composants dépendent du contrat, la configuration choisit l'implémentation.
   - Ici, le même composant `Brouillon` est utilisé deux fois, avec deux stockages différents :

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

// Le contrat : ce que les composants demandent
export abstract class Stockage {
  abstract readonly nom: string;
  abstract lire(cle: string): string;
  abstract ecrire(cle: string, valeur: string): void;
}

// Première recette : en mémoire (perdu au rechargement)
@Injectable()
export class StockageMemoire extends Stockage {
  readonly nom = 'mémoire';
  private readonly valeurs = new Map<string, string>();
  lire(cle: string) { return this.valeurs.get(cle) ?? ''; }
  ecrire(cle: string, valeur: string) { this.valeurs.set(cle, valeur); }
}

// Deuxième recette : dans le navigateur (conservé au rechargement)
@Injectable()
export class StockageNavigateur extends Stockage {
  readonly nom = 'localStorage';
  lire(cle: string) { return localStorage.getItem(cle) ?? ''; }
  ecrire(cle: string, valeur: string) { localStorage.setItem(cle, valeur); }
}

@Component({
  selector: 'app-brouillon',
  template: `
    <label>Brouillon ({{ stockage.nom }})
      <input [value]="texte()" (input)="modifier($any($event.target).value)" />
    </label>
  `,
})
export class Brouillon {
```

apcpedagogie.com

```
  readonly cle = input.required<string>();
  protected readonly stockage = inject(Stockage);
  protected readonly texte = signal('');

  ngOnInit() {
    this.texte.set(this.stockage.lire(this.cle()));
  }

  protected modifier(valeur: string) {
    this.texte.set(valeur);
    this.stockage.ecrire(this.cle(), valeur);
  }
}

// Le même composant, deux configurations différentes
@Component({
  selector: 'app-brouillon-memoire',
  imports: [Brouillon],
  providers: [{ provide: Stockage, useClass: StockageMemoire }],
  template: `<app-brouillon cle="brouillon-a" />`,
})
export class BrouillonMemoire {}

@Component({
  selector: 'app-brouillon-navigateur',
  imports: [Brouillon],
  providers: [{ provide: Stockage, useClass: StockageNavigateur }],
  template: `<app-brouillon cle="brouillon-b" />`,
})
export class BrouillonNavigateur {}

@Component({
  selector: 'app-use-class',
  imports: [BrouillonMemoire, BrouillonNavigateur],
  template: `
    <app-brouillon-memoire />
    <app-brouillon-navigateur />
  `,
})
export class UseClass {}
```

apcpedagogie.com

Brouillon (mémoire) Relire le chapitre Brouillon (localStorage) Préparer le quiz

```
(saisie dans les deux champs, puis rechargement de la page)
Brouillon (mémoire)       : vide          ← perdu
Brouillon (localStorage)  : Préparer le quiz   ← conservé
```

- `Brouillon` ne connaît ni `StockageMemoire` ni `StockageNavigateur`: il demande `inject(Stockage)`.
- Une classe abstraite peut servir de jeton, car elle existe à l'exécution (contrairement à une interface). Sans provider, l'erreur est : `NG0201: No provider found for `Stockage`.`
- Les deux implémentations portent `@Injectable()` sans `providedIn`: elles ne sont pas fournies automatiquement, seulement là où on les configure.

1. useValue et les jetons d'injection

- `useValue` donne une valeur **déjà prête**: un objet de configuration, une chaîne, un nombre, un faux service.
- Pour fournir une valeur qui n'est pas une classe (une interface, une chaîne…), on crée un `InjectionToken`. Son paramètre de type donne le type de `inject()`. Avec l'option `factory`, le jeton a une **valeur par défaut** à la racine, qu'un provider peut remplacer localement :

```
import { Component, InjectionToken, inject } from '@angular/core';
import { CurrencyPipe } from '@angular/common';

// Une interface n'existe pas à l'exécution : on crée un jeton pour la représenter
export interface ConfigEcole {
  nom: string;
  ville: string;
  annee: string;
}
export const CONFIG_ECOLE = new InjectionToken<ConfigEcole>('CONFIG_ECOLE');

// Un jeton avec fabrique : fourni automatiquement à la racine, remplaçable
export const DEVISE = new InjectionToken<string>('DEVISE', { factory: () => 'TND' });

@Component({
```

apcpedagogie.com

```
  selector: 'app-entete-ecole',
  imports: [CurrencyPipe],
  template: `
    <h4>{{ config.nom }} · {{ config.ville }}</h4>
    <p>Année {{ config.annee }} · Inscription : {{ 350 | currency: devise : 'symbol' : '1.0-0'
}}</p>
  `,
})
export class EnteteEcole {
  protected readonly config = inject(CONFIG_ECOLE);
  protected readonly devise = inject(DEVISE);
}

// Ce composant remplace seulement la devise pour ce qu'il contient
@Component({
  selector: 'app-zone-euro',
  providers: [{ provide: DEVISE, useValue: 'EUR' }],
  template: `<ng-content />`,
})
export class ZoneEuro {}

@Component({
  selector: 'app-jetons',
  imports: [EnteteEcole, ZoneEuro],
  providers: [
    {
      provide: CONFIG_ECOLE,
      useValue: { nom: 'Centre de formation APC', ville: 'Sousse', annee: '2026-2027' },
    },
  ],
  template: `
    <app-entete-ecole />
    <app-zone-euro><app-entete-ecole /></app-zone-euro>
  `,
})
export class Jetons {}
```

apcpedagogie.com

Centre de formation APC · Sousse

Année 2026-2027 · Inscription : 350 TND

Centre de formation APC · Sousse

Année 2026-2027 · Inscription : 350 €

- `CONFIG_ECOLE` n'a pas de fabrique : sans provider, `NG0201: No provider found for `InjectionToken CONFIG_ECOLE``.
- `DEVISE` a une fabrique : sans rien configurer, il vaut `'TND'`; `ZoneEuro` le remplace par `'EUR'` pour ce qu'il contient.
- La chaîne passée au constructeur (`'CONFIG_ECOLE'`) ne sert qu'aux messages d'erreur : deux jetons créés avec le même nom restent deux jetons différents.



1. useFactory : fabriquer selon le contexte

- `useFactory` confie la création à une **fonction**. Elle s'exécute dans un contexte d'injection : elle peut donc appeler `inject()` pour lire d'autres jetons et choisir ce qu'elle renvoie.
- Ici, le journal est détaillé en mode développement et discret en mode production :

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

export type Mode = 'developpement' | 'production';
export const MODE = new InjectionToken<Mode>('MODE', { factory: () => 'developpement' });

export abstract class Journal {
  abstract readonly lignes: () => readonly string[];
  abstract ecrire(message: string): void;
}

class JournalDetaille extends Journal {
  private readonly _lignes = signal<string[]>([]);
  readonly lignes = this._lignes.asReadonly();
  ecrire(message: string) {
    const heure = new Date().toLocaleTimeString('fr-FR');
    this._lignes.update((l) => [...l, `[${heure}] ${message}`]);
  }
}

class JournalDiscret extends Journal {
  private readonly nombre = signal(0);
  readonly lignes = () => [`${this.nombre()} message(s) reçu(s), non affichés`];
  ecrire() {
    this.nombre.update((n) => n + 1);
  }
}

// La fabrique choisit la recette au moment de la création
export const JOURNAL_PROVIDER: Provider = {
  provide: Journal,
  useFactory: () => (inject(MODE) === 'production' ? new JournalDiscret() : new
JournalDetaille()),
};

@Component({
  selector: 'app-console',
  providers: [JOURNAL_PROVIDER],
  template: `
    <button (click)="journal.ecrire('Bouton cliqué')">Écrire</button>
    @for (l of journal.lignes(); track $index) { <div><code>{{ l }}</code></div> }
  `,
})
export class Console {
  protected readonly journal = inject(Journal);
```

apcpedagogie.com

```
}

@Component({
  selector: 'app-console-production',
  imports: [Console],
  providers: [{ provide: MODE, useValue: 'production' }],
  template: `<app-console />`,
})
export class ConsoleProduction {}

@Component({
  selector: 'app-fabrique',
  imports: [Console, ConsoleProduction],
  template: `
    <h4>Mode développement</h4><app-console />
    <h4>Mode production</h4><app-console-production />
  `,
})
export class Fabrique {}
```

Mode développement

Écrire

```
[16:37:45] Bouton cliqué
[16:37:45] Bouton cliqué
```

Mode production

Écrire

```
2 message(s) reçu(s), non affichés
```

- La fabrique est appelée **une seule fois par injecteur**: chaque `<app-console>` fournit `JOURNAL_PROVIDER` et a donc son propre journal.
- `inject(MODE)` dans la fabrique cherche à partir du composant : la console placée dans `ConsoleProduction` trouve `'production'`.
- Les classes `JournalDetaille` et `JournalDiscret` ne sont pas exportées : seule la fabrique les connaît.
- Dans du code ancien, on rencontre `useFactory: (mode: Mode) => …, deps: [MODE]`. `deps` liste les jetons passés en paramètres ; avec `inject()`, il n'est plus nécessaire.

apcpedagogie.com

1. **useExisting : un alias**
   - `useExisting` dit : « pour ce jeton, renvoie l'objet déjà fourni pour cet **autre** jeton ». C'est un **alias**. À ne pas confondre avec `useClass`, qui crée une nouvelle instance.
   - Cas typique : un service a été renommé, mais d'anciens composants utilisent encore l'ancien nom :

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

let compteur = 0;

@Injectable()
export class Notifications {
  readonly numero = ++compteur; // numéro de l'instance
  private readonly _nombre = signal(0);
  readonly nombre = this._nombre.asReadonly();
  envoyer() { this._nombre.update((n) => n + 1); }
}

// L'ancien nom, encore utilisé par d'anciens composants
export abstract class AncienNotifService {
  abstract readonly numero: number;
  abstract nombre(): number;
  abstract envoyer(): void;
}

@Component({
  selector: 'app-deux-noms',
  template: `
    <p>Notifications n° {{ nouveau.numero }} : {{ nouveau.nombre() }}
      · AncienNotifService n° {{ ancien.numero }} : {{ ancien.nombre() }}</p>
    <button (click)="ancien.envoyer()">Envoyer par l'ancien nom</button>
  `,
})
export class DeuxNoms {
  protected readonly nouveau = inject(Notifications);
  protected readonly ancien = inject(AncienNotifService);
}

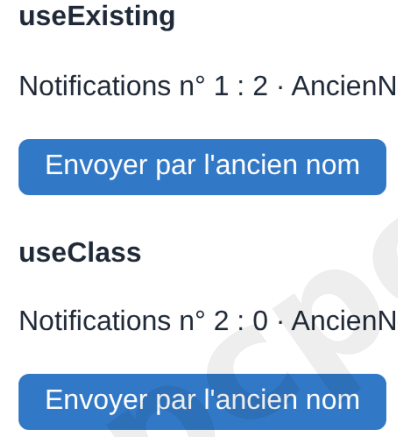
@Component({
  selector: 'app-avec-use-existing',
  imports: [DeuxNoms],
```

apcpedagogie.com

```
  providers: [Notifications, { provide: AncienNotifService, useExisting: Notifications }],
  template: `<h4>useExisting</h4><app-deux-noms />`,
})
export class AvecUseExisting {}

@Component({
  selector: 'app-avec-use-class',
  imports: [DeuxNoms],
  providers: [Notifications, { provide: AncienNotifService, useClass: Notifications }],
  template: `<h4>useClass</h4><app-deux-noms />`,
})
export class AvecUseClass {}

@Component({
  selector: 'app-alias',
  imports: [AvecUseExisting, AvecUseClass],
  template: `<app-avec-use-existing /><app-avec-use-class />`,
})
export class Alias {}
```

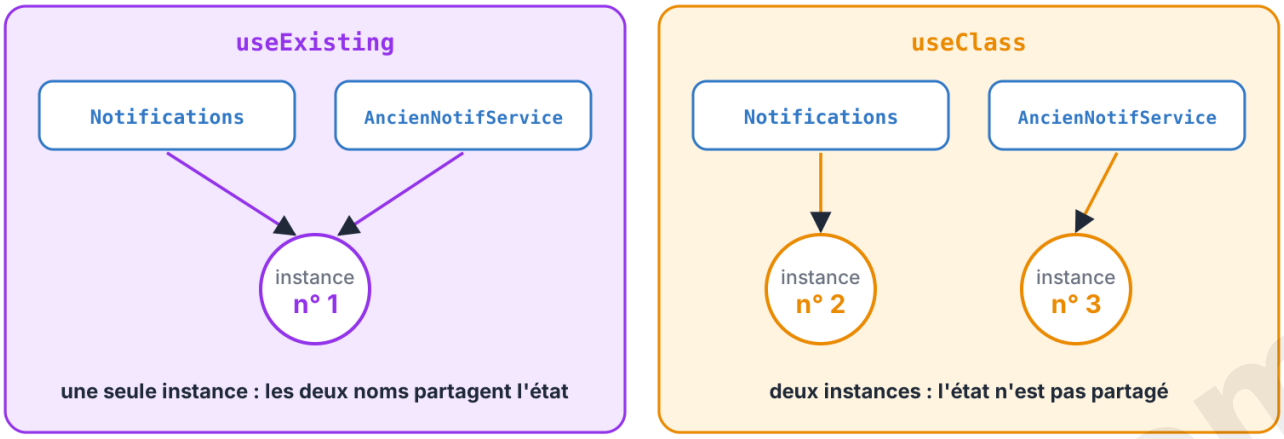


- Avec `useExisting`, les deux noms donnent l'instance **n° 1**: un envoi par l'ancien nom est visible des deux côtés.
- Avec `useClass`, l'ancien nom reçoit une instance **n° 3**, distincte : les compteurs divergent.
- Le jeton cible doit être fourni lui aussi. Sans `Notifications` dans les `providers`: `NG0201: No provider found for `Notifications``.

apcpedagogie.com

useExisting ou useClass ?

Deux noms pour un même objet, ou deux objets différents




```
providers: [Notifications, { provide: AncienNotifService, useExisting: Notifications }]
```

Avec useExisting, Notifications doit être fourni lui aussi ; sinon : NG0201 No provider found for `Notifications`.

Les quatre recettes

useClass, useValue, useFactory, useExisting

useClass

Angular crée une instance de la classe

```
{ provide: Stockage,
  useClass: StockageMemoire }
```

Quand ? changer d'implémentation

useValue

Angular donne l'objet tel quel

```
{ provide: CONFIG_ECOLE,
  useValue: { nom: 'APC' } }
```

Quand ? configuration, constante, faux service

useFactory

Angular appelle la fonction

```
{ provide: Journal, useFactory:
  () => inject(MODE) === …  }
```

Quand ? choisir selon une condition

useExisting

Angular renvoie une instance déjà fournie

```
{ provide: AncienNotifService,
  useExisting: Notifications }
```

Quand ? alias, vue restreinte

Toutes s'écrivent dans providers : app.config.ts, une route, un composant ou TestBed.

apcpedagogie.com

1. ## Configurer un service depuis @Service()

   - Avec Angular 22, le décorateur `@Service()` accepte deux options qui remplacent un provider :
     - `{ autoProvided: false }`: le service n'est **pas** fourni automatiquement ; il faut l'ajouter à un tableau `providers` (vu au chapitre sur les services) ;
     - `{ factory: () => … }`: le service est fourni à la racine avec **cette recette**. Très pratique sur une classe abstraite, qui ne peut pas être instanciée directement.

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

// Le contrat porte sa recette par défaut : aucun provider à écrire pour l'utiliser
@Service({ factory: () => new HorlogeSysteme() })
export abstract class Horloge {
  abstract maintenant(): Date;
}

class HorlogeSysteme extends Horloge {
  maintenant() { return new Date(); }
}

@Component({
  selector: 'app-salutation',
  imports: [DatePipe],
  template: `<p>{{ message }} · il est {{ heure | date: 'HH:mm' }}</p>`,
})
export class Salutation {
  protected readonly heure = inject(Horloge).maintenant();
  protected readonly message = this.heure.getHours() < 18 ? 'Bonjour' : 'Bonsoir';
}

// Une horloge arrêtée à 20 h : pratique pour une démonstration ou un test
@Component({
  selector: 'app-soiree',
  imports: [Salutation],
  providers: [{ provide: Horloge, useValue: { maintenant: () => new Date(2026, 10, 1, 20, 0) } }],
  template: `<app-salutation />`,
})
export class Soiree {}
```

apcpedagogie.com

```
@Component({
  selector: 'app-service-fabrique',
  imports: [Salutation, Soiree],
  template: `
    <h4>Recette par défaut</h4><app-salutation />
    <h4>Horloge remplacée</h4><app-soiree />
  `,
})
export class ServiceFabrique {}
```

Recette par défaut

Bonjour · il est 16:37

Horloge remplacée

Bonsoir · il est 20:00

- Aucun provider n'est écrit pour la première salutation : la recette est sur `Horloge` elle-même.
- Le composant `Soiree` remplace la recette par un objet `useValue`: c'est ce qu'on fera aussi dans un test.
- `HorlogeSysteme` est déclarée **après** `Horloge`: ce n'est pas un problème, car la fabrique n'est appelée qu'à la première demande.

apcpedagogie.com

@Service() : trois façons de se fournir

La configuration peut être écrite sur le service lui-même (Angular 22)

Écriture	Effet	Remarque
`@Service()`	fourni à la racine, créé à la 1re demande	cas courant
`@Service({ autoProvided: false })`	à fournir soi-même (providers, provideXxx)	NG0201 sinon
`@Service({ factory: () => … })`	fourni à la racine avec cette recette	classe abstraite

```
@Service({ factory: () => new HorlogeSysteme() })   // recette par défaut
export abstract class Horloge { abstract maintenant(): Date; }

// remplaçable comme n'importe quel provider :
providers: [{ provide: Horloge, useValue: { maintenant: () => new Date(2026, 10, 1, 20, 0) } }]
```

Le contrat (classe abstraite) porte sa recette : aucun provider à écrire pour l'utiliser normalement.

1. **Plusieurs valeurs pour un jeton : multi: true**
   - Par défaut, un jeton a **une** valeur. Avec `multi: true`, chaque provider **ajoute** sa valeur, et `inject()` renvoie un **tableau**. C'est le principe des « extensions » : chaque partie de l'application peut ajouter une règle sans modifier les autres.
   - Les comptes élèves exigent une règle de mot de passe, les comptes administrateurs trois :

```
import { Component, InjectionToken, Provider, computed, inject, signal } from '@angular/core';

export interface Regle {
  message: string;
  verifier(valeur: string): boolean;
}

// Un jeton qui recevra PLUSIEURS valeurs : Angular les rassemble dans un tableau
export const REGLES = new InjectionToken<Regle[]>('REGLES');

export const longueur: Provider = {
  provide: REGLES,
  multi: true,
  useValue: { message: 'au moins 8 caractères', verifier: (v: string) => v.length >= 8 },
```

```
};
export const chiffre: Provider = {
  provide: REGLES,
  multi: true,
  useValue: { message: 'au moins un chiffre', verifier: (v: string) => /\d/.test(v) },
};
export const majuscule: Provider = {
  provide: REGLES,
  multi: true,
  useValue: { message: 'au moins une majuscule', verifier: (v: string) => /[A-Z]/.test(v) },
};

@Component({
  selector: 'app-champ-mdp',
  template: `
    <input type="password" placeholder="Mot de passe"
(input)="valeur.set($any($event.target).value)" />
    <ul>
      @for (r of etat(); track r.message) {
        <li [class.ok]="r.ok">{{ r.ok ? '✔' : '✘' }} {{ r.message }}</li>
      }
    </ul>
    <p>{{ regles.length }} règle(s) · {{ valide() ? 'mot de passe accepté' : 'mot de passe
refusé' }}</p>
  `,
  styles: `li { color: #c62828; } li.ok { color: #2e7d32; }`,
})
export class ChampMdp {
  protected readonly regles = inject(REGLES);
  protected readonly valeur = signal('');
  protected readonly etat = computed(() =>
    this.regles.map((r) => ({ message: r.message, ok: r.verifier(this.valeur()) })),
  );
  protected readonly valide = computed(() => this.etat().every((r) => r.ok));
}

@Component({
  selector: 'app-mdp-simple',
  imports: [ChampMdp],
  providers: [longueur],
  template: `<h4>Compte élève</h4><app-champ-mdp />`,
})
export class MdpSimple {}
```

apcpedagogie.com

```
@Component({
  selector: 'app-mdp-renforce',
  imports: [ChampMdp],
  providers: [longueur, chiffre, majuscule],
  template: `<h4>Compte administrateur</h4><app-champ-mdp />`,
})
export class MdpRenforce {}

@Component({
  selector: 'app-multi',
  imports: [MdpSimple, MdpRenforce],
  template: `<app-mdp-simple /><app-mdp-renforce />`,
})
export class Multi {}
```

Compte élève

••••••••••

✔ au moins 8 caractères

1 règle(s) · mot de passe accepté

Compte administrateur

••••••••••

✔ au moins 8 caractères

✔ au moins un chiffre

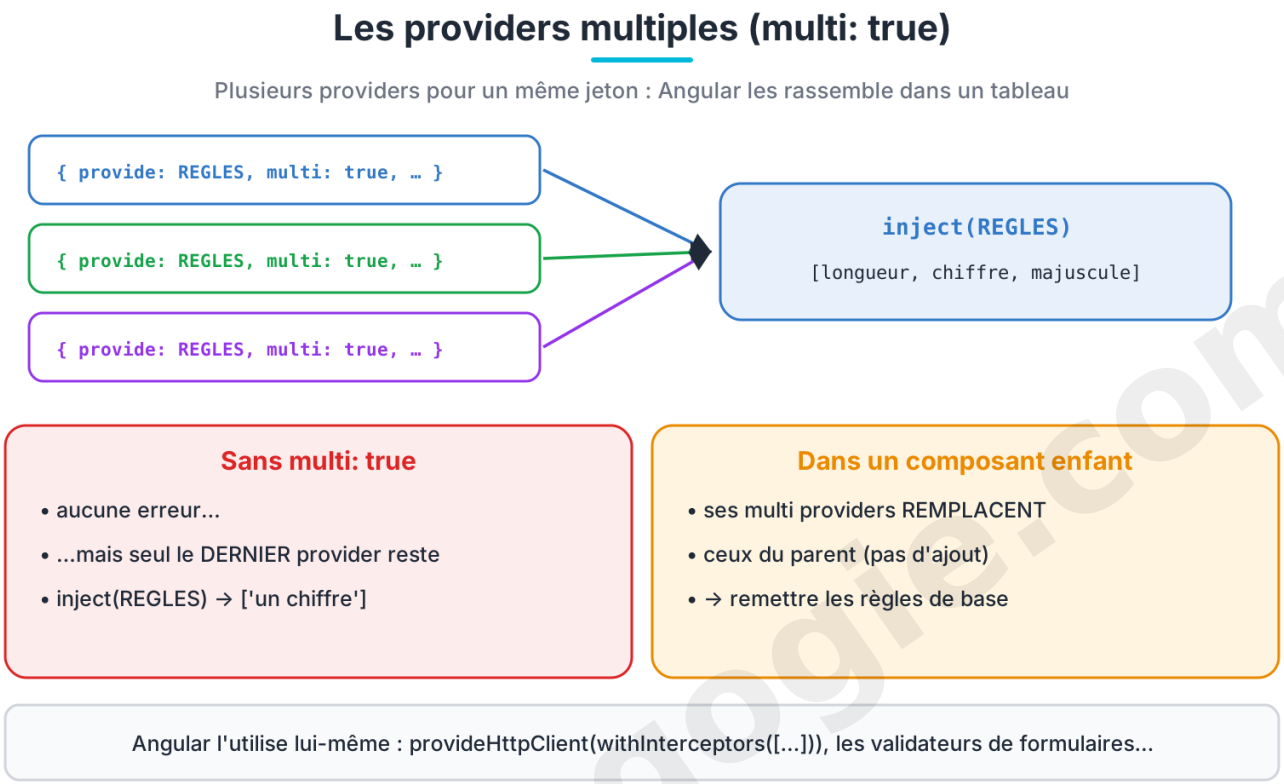
✘ au moins une majuscule

3 règle(s) · mot de passe refusé

- `ChampMdp` ne connaît aucune règle : il parcourt le tableau reçu. Ajouter une règle ne demande pas de toucher au composant.
- **Oubli de `multi: true`**: aucune erreur, mais chaque provider remplace le précédent. Avec deux providers `useValue: ['8 caractères']` puis `['un chiffre']`, `inject(REGLES)` renvoie seulement `['un chiffre']`.
- **Au niveau d'un composant**, les multi providers d'un enfant **remplacent** ceux du parent, ils ne s'y ajoutent pas. Test réalisé : parent `'a'`, enfant `'b'` → l'enfant lit `['b']`. C'est le piège de

l'activité 5.

- Angular utilise ce mécanisme lui-même : les intercepteurs HTTP, les validateurs de formulaires, les initialiseurs...



1. Écrire sa fonction provideXxx()

- `provideRouter()`, `provideHttpClient()`... : Angular regroupe la configuration de chaque fonctionnalité dans une fonction. On peut faire de même avec `makeEnvironmentProviders()`, qui renvoie des `EnvironmentProviders`.
- La fonction `provideSuivi()` fournit en une ligne la configuration, le service, et une fonction exécutée à la création de l'injecteur (`provideEnvironmentInitializer`) :

```
import {
  EnvironmentProviders, InjectionToken, Service, inject, makeEnvironmentProviders,
  provideEnvironmentInitializer, signal,
} from '@angular/core';

export interface OptionsSuivi {
  prefixe: string;
}
export const OPTIONS_SUIVI = new InjectionToken<OptionsSuivi>('OPTIONS_SUIVI');
```

apcpedagogie.com

```
// Pas fourni automatiquement : il faut passer par provideSuivi()
@Service({ autoProvided: false })
export class Suivi {
  private readonly options = inject(OPTIONS_SUIVI);
  private readonly _lignes = signal<string[]>([]);
  readonly lignes = this._lignes.asReadonly();

  noter(message: string) {
    this._lignes.update((l) => [...l, `[${this.options.prefixe}] ${message}`]);
  }
}

// Une seule fonction regroupe tout ce qu'il faut fournir
export function provideSuivi(options: OptionsSuivi): EnvironmentProviders {
  return makeEnvironmentProviders([
    { provide: OPTIONS_SUIVI, useValue: options },
    Suivi,
    provideEnvironmentInitializer(() => inject(Suivi).noter('injecteur prêt')),
  ]);
}
```

- `Suivi` est en `autoProvided: false`: on ne peut l'utiliser qu'après avoir appelé `provideSuivi()`, ce qui garantit que `OPTIONS_SUIVI` existe.
- `provideEnvironmentInitializer()` exécute sa fonction dès que l'injecteur est créé, dans un contexte d'injection.
- Les `EnvironmentProviders` sont **refusés** dans un composant, dès la compilation : `TS2322: Type 'EnvironmentProviders' is not assignable to type 'Provider'`. C'est voulu : ils configurent des services globaux à une zone entière.

apcpedagogie.com

Écrire sa fonction provideXxx()

makeEnvironmentProviders() regroupe tout ce qu'une fonctionnalité doit fournir

```
export function provideSuivi(options: OptionsSuivi): EnvironmentProviders {
  return makeEnvironmentProviders([
    { provide: OPTIONS_SUIVI, useValue: options },   // la configuration
    Suivi,                                           // le service
    provideEnvironmentInitializer(() => inject(Suivi).noter('injecteur prêt')),
  ]);
}
```

✓ Accepté dans un injecteur d'environnement

- app.config.ts : providers
- une route : providers
- createEnvironmentInjector([...])
- TestBed : providers

× Refusé dans un composant

`providers: [provideSuivi({…})]`

TS2322: Type 'EnvironmentProviders' is not assignable to type 'Provider'.

c'est voulu : ces services sont globaux

1. **Configurer le démarrage**
    - Certaines données doivent être là **avant** le premier affichage : des paramètres, une langue, une session. `provideAppInitializer()` enregistre une fonction exécutée au démarrage ; si elle renvoie une `Promise` ou un `Observable`, Angular **attend** sa fin avant d'afficher le premier composant.
    - Placer `parametres.json` dans `public/`:

```
{
  "etablissement": "Centre de formation APC",
  "accueil": "Bienvenue sur la plateforme des formations !",
  "maxInscriptions": 3
}
```

```
import {
  Component, EnvironmentProviders, Service, inject, makeEnvironmentProviders,
  provideAppInitializer, signal,
} from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { firstValueFrom } from 'rxjs';

export interface Reglages {
```

apcpedagogie.com

```
  etablissement: string;
  accueil: string;
  maxInscriptions: number;
}

@Service()
export class Parametres {
  private readonly http = inject(HttpClient);
  private readonly _reglages = signal<Reglages | null>(null);
  readonly reglages = this._reglages.asReadonly();

  async charger() {
    const debut = performance.now();
    this._reglages.set(await firstValueFrom(this.http.get<Reglages>('parametres.json')));
    console.log(`parametres.json chargé en ${Math.round(performance.now() - debut)} ms`);
  }
}

// L'application attend la fin de charger() avant d'afficher le premier composant
export function provideParametres(): EnvironmentProviders {
  return makeEnvironmentProviders([provideAppInitializer(() => inject(Parametres).charger())]);
}

@Component({
  selector: 'app-accueil',
  template: `
    <h4>{{ reglages.etablissement }}</h4>
    <p>{{ reglages.accueil }} (au plus {{ reglages.maxInscriptions }} inscriptions)</p>
  `,
})
export class Accueil {
  // Déjà chargés : pas de null à gérer dans le composant
  protected readonly reglages = inject(Parametres).reglages()!;
}
```

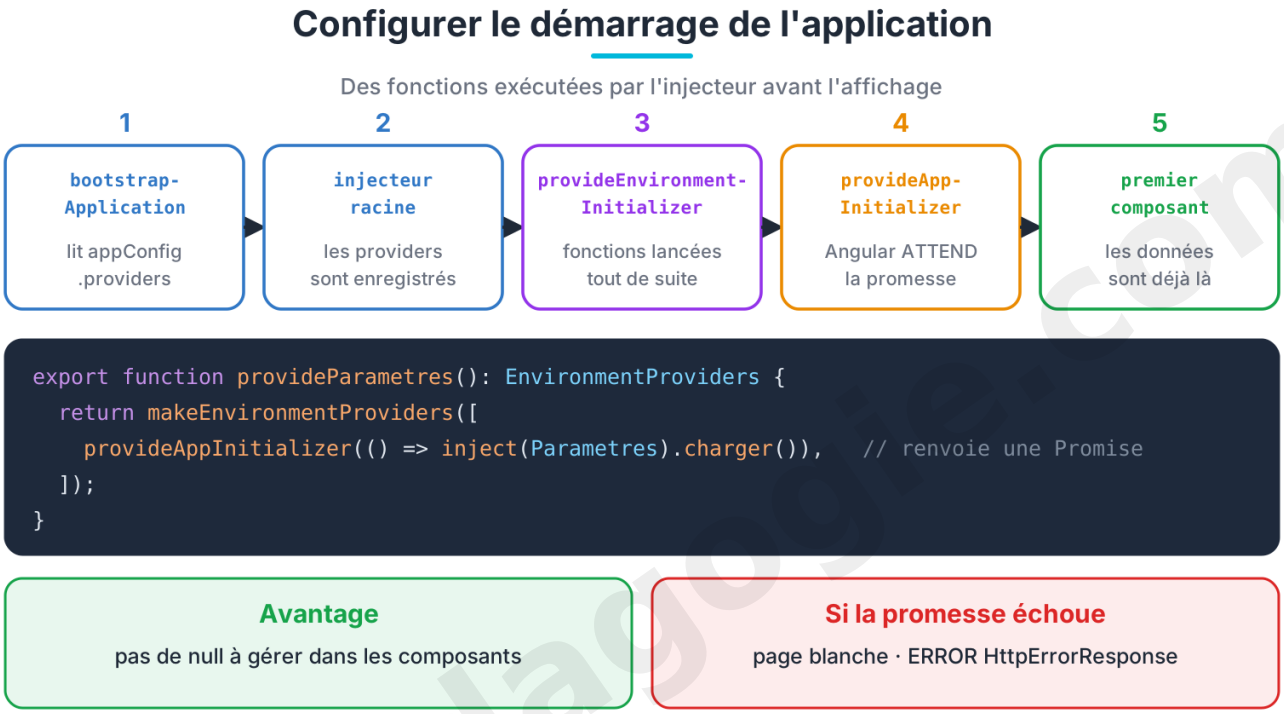
Centre de formation APC

Bienvenue sur la plateforme des formations ! (au plus 3 inscriptions)

```
parametres.json chargé en 48 ms
Angular is running in development mode.     ← le chargement a eu lieu AVANT le démarrage
```

apcpedagogie.com

- Le composant `Accueil` lit les réglages dans l'initialisation d'un champ, avec `!`: ils ne peuvent pas être `null`, car l'application attend la fin de `charger()`.
- Si la promesse échoue (fichier absent, serveur arrêté), l'application **ne démarre pas**: page blanche et `ERROR HttpErrorResponse` dans la console. Pour des données facultatives, il faut attraper l'erreur dans `charger()` et prévoir des valeurs par défaut.
- Un initialiseur retarde l'affichage : il faut le réserver aux données **vraiment indispensables**.



1. ## Les injecteurs d'environnement enfants

   - Un injecteur d'environnement enfant crée une nouvelle « zone » de services, avec ses propres instances. On en obtient un de deux façons :
     - les `providers` d'une **route** (détaillés au chapitre sur le routage) ;
     - la fonction `createEnvironmentInjector(providers, parent)`, pour afficher un composant dans une zone avec `*ngComponentOutlet`.

```
import {
  Component, DestroyRef, EnvironmentInjector, createEnvironmentInjector, inject,
} from '@angular/core';
import { NgComponentOutlet } from '@angular/common';
import { Suivi, provideSuivi } from './suivi';

@Component({
```

```
  selector: 'app-panneau-suivi',
  template: `
    <button (click)="suivi.noter('clic')">Noter un clic</button>
    @for (l of suivi.lignes(); track $index) { <div><code>{{ l }}</code></div> }
  `,
})
export class PanneauSuivi {
  protected readonly suivi = inject(Suivi);
}

@Component({
  selector: 'app-environnement',
  imports: [PanneauSuivi, NgComponentOutlet],
  template: `
    <h4>Injecteur racine (app.config.ts)</h4>
    <app-panneau-suivi />
    <h4>Injecteur d'environnement enfant</h4>
    <ng-container *ngComponentOutlet="panneau; environmentInjector: admin" />
  `,
})
export class Environnement {
  protected readonly panneau = PanneauSuivi;

  // Un nouvel injecteur d'environnement, enfant de celui de l'application
  protected readonly admin = createEnvironmentInjector(
    [provideSuivi({ prefixe: 'ADMIN' })],
    inject(EnvironmentInjector),
    'Administration',
  );

  constructor() {
    inject(DestroyRef).onDestroy(() => this.admin.destroy());
  }
}
```

API : `http://localhost:3000`

Journal détaillé : oui

- Le premier panneau reçoit le `Suivi` de la racine (`provideSuivi({ prefixe: 'APP' })` dans `app.config.ts`), le second celui de la zone `Administration`.

- L'initialiseur de `provideSuivi()` s'est exécuté **une fois par injecteur**: on voit « injecteur prêt » dans chaque panneau.
- Un injecteur créé avec `createEnvironmentInjector()` doit être **détruit** à la main (`destroy()`), ici à la destruction du composant.

- Le même découpage avec des routes, vérifié par un test :

```
import { Routes } from '@angular/router';
import { PanneauSuivi } from './environnement';
import { provideSuivi } from './suivi';

export const routesSuivi: Routes = [
  { path: '', component: PanneauSuivi },
  {
    path: 'admin',
    // Cette route crée son propre injecteur d'environnement
    providers: [provideSuivi({ prefixe: 'ADMIN' })],
    component: PanneauSuivi,
  },
];
```

```
import { TestBed } from '@angular/core/testing';
import { provideRouter } from '@angular/router';
import { RouterTestingHarness } from '@angular/router/testing';
import { routesSuivi } from './routes-suivi';
import { provideSuivi } from './suivi';

describe('Providers de route', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [provideRouter(routesSuivi), provideSuivi({ prefixe: 'APP' })],
    });
  });

  it('la route "" utilise le Suivi de l\'application', async () => {
    const harness = await RouterTestingHarness.create('/');
    expect(harness.routeNativeElement?.textContent).toContain('[APP] injecteur prêt');
  });

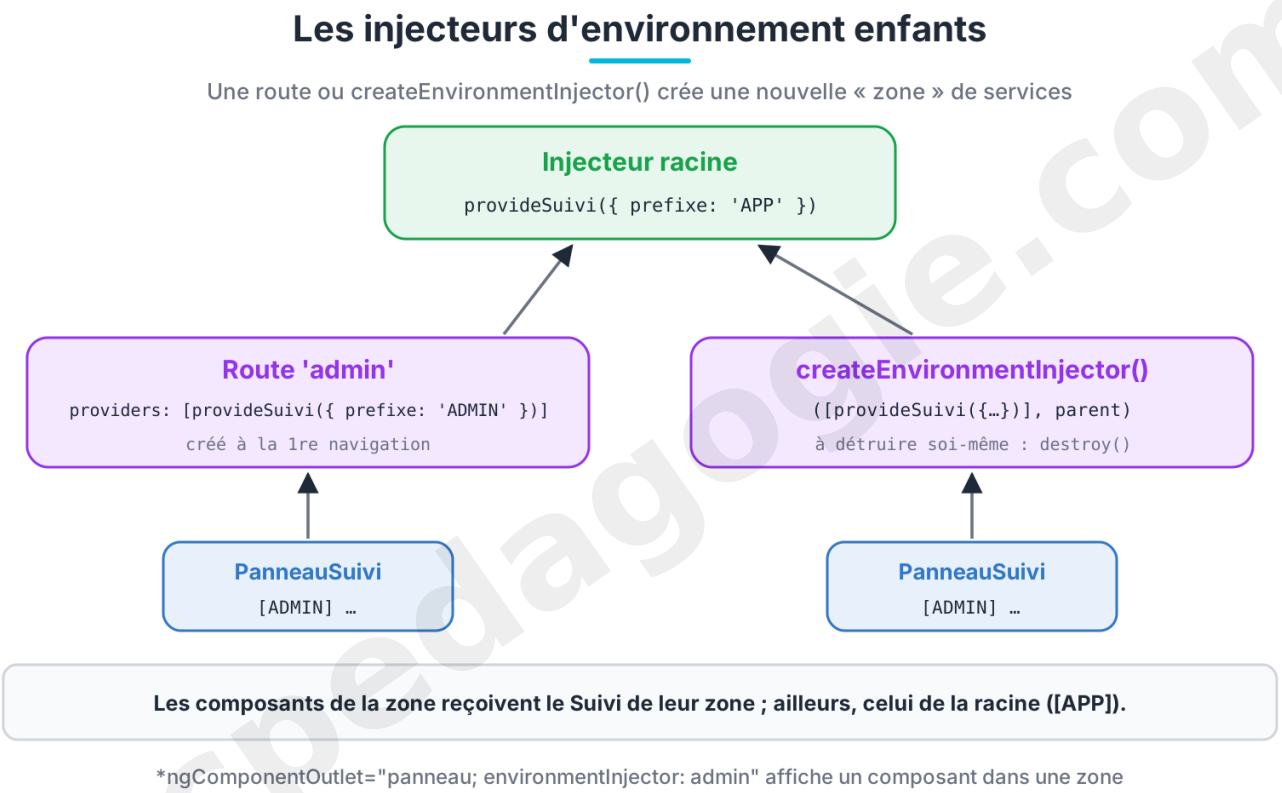
  it('la route "admin" a son propre Suivi', async () => {
    const harness = await RouterTestingHarness.create('/admin');
    expect(harness.routeNativeElement?.textContent).toContain('[ADMIN] injecteur prêt');
```

apcpedagogie.com

```
    expect(harness.routeNativeElement?.textContent).not.toContain('[APP]');
  });
});
```

```
$ ng test --include src/app/configurer/routes-suivi.spec.ts
 ✓ Providers de route > la route "" utilise le Suivi de l'application
 ✓ Providers de route > la route "admin" a son propre Suivi

      Tests  2 passed (2)
```



1. Configurer l'injecteur dans un test

- `TestBed.configureTestingModule({ providers })` configure l'injecteur **racine** du test. Mais un composant qui déclare ses propres `providers` trouve son service **avant** d'atteindre la racine : le faux service est ignoré.
- Pour remplacer un provider déclaré dans un composant, on utilise `TestBed.overrideComponent()` :

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

```
@Injectable()
export class Chrono {
  private readonly _secondes = signal(0);
  readonly secondes = this._secondes.asReadonly();
  private id?: ReturnType<typeof setInterval>;

  constructor() {
    inject(DestroyRef).onDestroy(() => this.arreter());
  }
  demarrer() {
    this.arreter();
    this.id = setInterval(() => this._secondes.update((s) => s + 1), 1000);
  }
  arreter() { clearInterval(this.id); }
}

// Chaque minuteur fournit SON chrono
@Component({
  selector: 'app-minuteur',
  providers: [Chrono],
  template: `
    <span class="ecran">{{ affichage() }}</span>
    <button (click)="chrono.demarrer()">Démarrer</button>
    <button (click)="chrono.arreter()">Arrêter</button>
  `,
  styles: `.ecran { font: 700 1.4rem monospace; margin-right: 8px; }`,
})
export class Minuteur {
  protected readonly chrono = inject(Chrono);
  protected readonly affichage = computed(() => {
    const s = this.chrono.secondes();
    return `${String(Math.floor(s / 60)).padStart(2, '0')}:${String(s % 60).padStart(2, '0')}`;
  });
}
```

```
import { TestBed } from '@angular/core/testing';
import { signal } from '@angular/core';
import { Chrono, Minuteur } from './minuteur';

class FauxChrono {
  readonly secondes = signal(75);
  demarrages = 0;
  demarrer() { this.demarrages++; }
  arreter() {}
```

apcpedagogie.com

```
}

describe('Minuteur', () => {
  it('les providers de TestBed ne remplacent PAS ceux du composant', async () => {
    const faux = new FauxChrono();
    TestBed.configureTestingModule({ providers: [{ provide: Chrono, useValue: faux }] });
    const fixture = TestBed.createComponent(Minuteur);
    await fixture.whenStable();
    expect(fixture.debugElement.injector.get(Chrono)).not.toBe(faux);
    expect(fixture.nativeElement.textContent).toContain('00:00');
  });

  it('overrideComponent remplace le provider du composant', async () => {
    const faux = new FauxChrono();
    TestBed.overrideComponent(Minuteur, {
      set: { providers: [{ provide: Chrono, useValue: faux }] },
    });
    const fixture = TestBed.createComponent(Minuteur);
    await fixture.whenStable();
    expect(fixture.nativeElement.textContent).toContain('01:15');
    fixture.nativeElement.querySelector('button').click();
    expect(faux.demarrages).toBe(1);
  });
});
```

```
$ ng test --include src/app/configurer/minuteur.spec.ts
 ✓ Minuteur > les providers de TestBed ne remplacent PAS ceux du composant
 ✓ Minuteur > overrideComponent remplace le provider du composant

      Tests  2 passed (2)
```

- Le premier test le prouve : `fixture.debugElement.injector.get(Chrono)` n'est pas le faux, et l'écran affiche `00:00`.
- `overrideComponent` doit être appelé **avant** `createComponent()`. `set` remplace tous les providers du composant ; `add` et `remove` existent aussi.
- Une fonction `provideXxx()` s'utilise aussi dans les `providers` de `TestBed`, comme `provideSuivi()` dans le test des routes.

apcpedagogie.com

Configurer l'injecteur dans un test

TestBed.providers ne remplace pas les providers d'un composant

TestBed.configureTestingModule

`providers: [{ provide: Chrono, … }]`

→ va dans l'injecteur RACINE du test

le composant qui a providers: [Chrono]
garde son propre Chrono ×

TestBed.overrideComponent

`(Minuteur, { set: { providers: […] } })`

→ remplace les providers du composant

le composant reçoit le faux service
affiche 01:15 ✓

```
TestBed.overrideComponent(Minuteur, {
  set: { providers: [{ provide: Chrono, useValue: faux }] },
});
const fixture = TestBed.createComponent(Minuteur);   // après la surcharge
```

Autres réglages : TestBed.inject(X) lit un service · providers: [provideSuivi({...})] fonctionne aussi.

1. Aide-mémoire

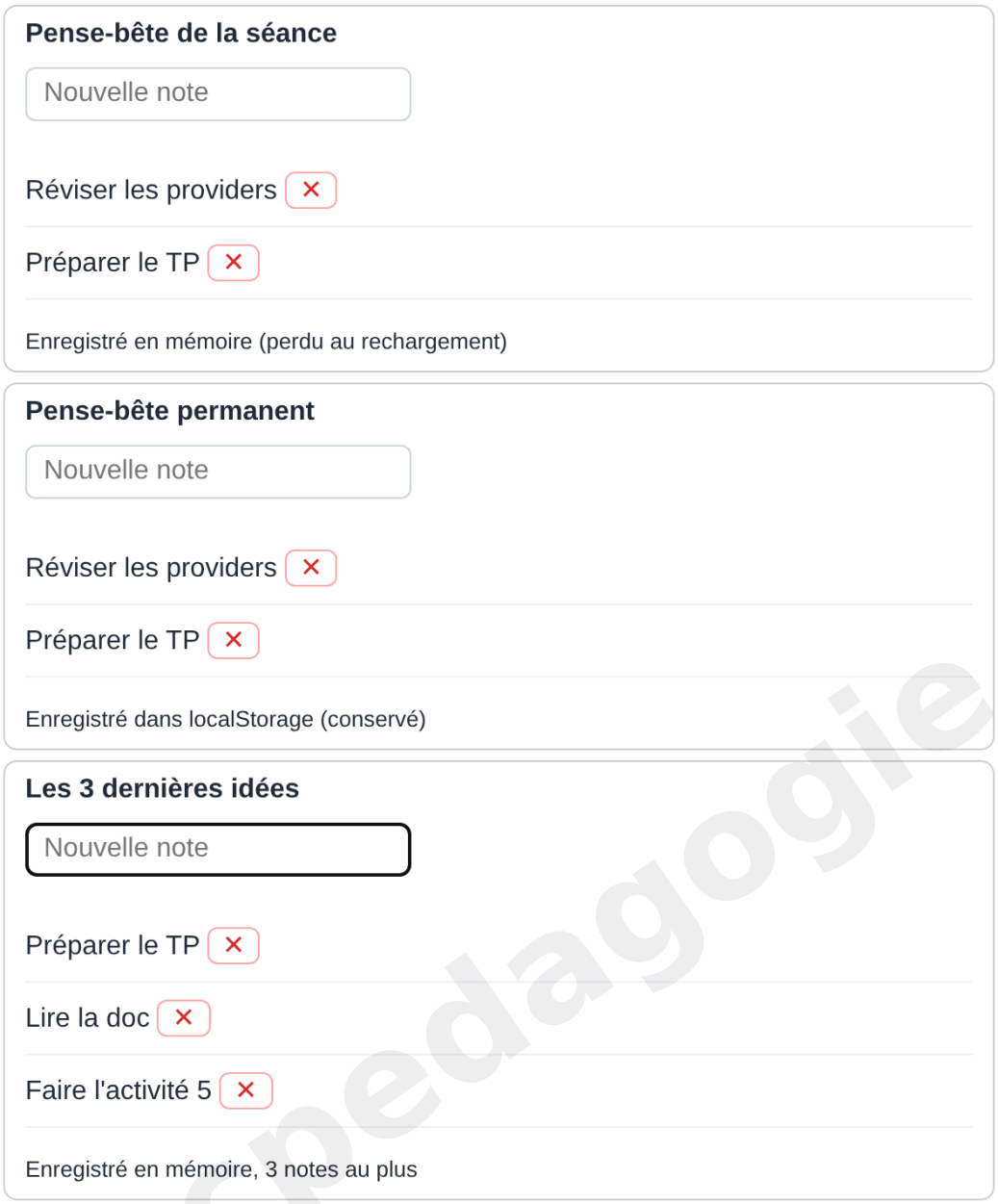
apcpedagogie.com

Aide-mémoire : configurer l'injecteur

Forme courte	`providers: [Panier]`
Changer de classe	`{ provide: Stockage, useClass: StockageLocal }`
Donner une valeur	`{ provide: CONFIG, useValue: { … } }`
Fabriquer selon le contexte	`{ provide: X, useFactory: () => inject(Y) ? … : … }`
Alias / vue restreinte	`{ provide: LECTURE, useExisting: Panier }`
Jeton typé	`new InjectionToken<T>('NOM')`
Jeton avec défaut	`new InjectionToken<T>('NOM', { factory: () => … })`
Recette sur le service	`@Service({ factory: () => … }) · { autoProvided: false }`
Plusieurs valeurs	`{ provide: REGLES, multi: true, useValue: … }`
Regrouper	`makeEnvironmentProviders([...]) → provideXxx()`
Au démarrage	`provideAppInitializer(() => …) · provideEnvironmentInitializer`
Zone de services	`route providers · createEnvironmentInjector([...], parent)`
Test	`TestBed.overrideComponent(C, { set: { providers } })`
Erreurs	`NG0201 rien trouvé · TS2322 provideXxx dans un composant`

1. **Activité 1 : un pense-bête au stockage interchangeable (useClass)**
   - Créer la classe abstraite `Sauvegarde` (`description`, `charger(cle)`, `enregistrer(cle, notes)`) et trois implémentations : `SauvegardeMemoire`, `SauvegardeLocale` (`localStorage`, en JSON) et `SauvegardeLimitee`, qui **hérite** de la première et ne garde que les 3 dernières notes.
   - Créer le composant `PenseBete` (entrée `titre`) : un champ qui ajoute une note avec Entrée, la liste avec un bouton ×, et « Enregistré … » en bas. Il n'injecte que `Sauvegarde`.
   - L'afficher trois fois, chacun dans un composant qui fournit une implémentation différente avec `useClass`.

apcpedagogie.com



- Vérifier : après rechargement, seul le pense-bête permanent a gardé ses notes ; le troisième n'affiche jamais plus de 3 notes.
- Pourquoi le composant relit-il les notes avec `charger()` après chaque `enregistrer()` ?

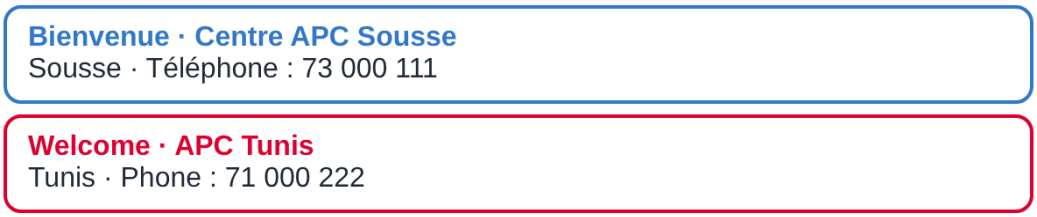
1. Activité 2 : des cartes de visite configurables (InjectionToken)

- Créer le jeton `ETABLISSEMENT` (interface : `nom`, `ville`, `telephone`, `couleur`), **sans** fabrique, et le jeton `LANGUE` (`'fr' | 'en'`) avec la fabrique `() => 'fr'`.
- Le composant `CarteVisite` affiche « Bienvenue · nom » puis « ville · Téléphone : … », encadré de la couleur de l'établissement ; les libellés viennent d'un dictionnaire indexé

apcpedagogie.com

par la langue.

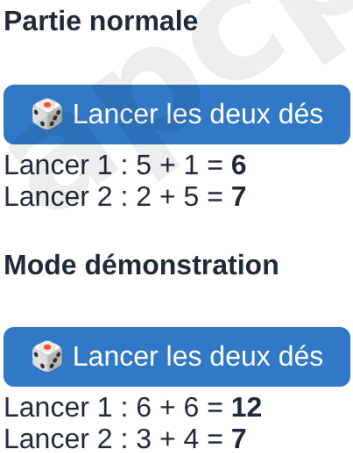
- `SiteSousse` fournit seulement l'établissement ; `SiteTunis` fournit un autre établissement et la langue `'en'`.



- Vérifier : la carte de Tunis est en anglais et en rouge, celle de Sousse en français et en bleu.
- Que se passe-t-il si l'on affiche `<app-carte-visite />` hors des deux sites ? Et pour la langue ?

1. Activité 3 : un jeu de dés en mode démonstration (useFactory)

- Créer la classe abstraite `Hasard` (`entier(min, max)`) et deux implémentations non exportées : `HasardMath` (`Math.random()`) et `HasardRejoue`, qui renvoie toujours la même suite de nombres, en boucle.
- Créer le jeton `SUITE_DEMO` (`readonly number[] | null`, `null` par défaut) et le provider `HASARD_PROVIDER`: sa fabrique choisit `HasardRejoue` si une suite est fournie, `HasardMath` sinon.
- Le composant `JeuDes` fournit `HASARD_PROVIDER`, lance deux dés à chaque clic et affiche l'historique. Une seconde partie est placée dans un composant qui fournit `SUITE_DEMO = [6, 6, 3, 4, 1, 2]`.



- Vérifier : en mode démonstration, les lancers sont toujours 6 + 6 = 12, 3 + 4 = 7, 1 + 2 = 3, puis la suite recommence.

- Pourquoi ce mode est-il précieux pour écrire un test ?

1. **Activité 4 : une vue en lecture seule du panier (useExisting)**
    - Créer le service `Panier` (`@Injectable()`) : articles privés, `nombre` et `total` calculés, `ajouter()` et `vider()`.
    - Créer l'interface `PanierLecture` (seulement `nombre` et `total`) et le jeton `PANIER_LECTURE`.
    - Le composant `Boutique` fournit `Panier` et l'alias `{ provide: PANIER_LECTURE, useExisting: Panier }`. `Rayon` injecte `Panier` (boutons d'ajout) ; `BadgePanier` injecte `PANIER_LECTURE` et affiche « 3 article(s) · 103 DT ».

🛒 3 article(s) · 103 DT | Clé USB (18 DT) | Souris (25 DT) | Casque (60 DT) | Vider

- Vérifier : le badge se met à jour à chaque ajout et revient à 0 avec « Vider ».
- Écrire `this.panier.ajouter(…)` dans `BadgePanier`: quelle erreur donne le compilateur ? Et si l'on remplace `useExisting` par `useClass`, que montre le badge ?

1. **Activité 5 : des contrôles d'inscription extensibles (multi)**
    - Créer l'interface `Controle` (`verifier(candidat)` renvoie un message ou `null`) et le jeton `CONTROLES`.
    - Écrire `provideControlesBase()`, qui renvoie trois providers `multi`: nom d'au moins 2 caractères, âge d'au moins 16 ans, e-mail contenant `@`.
    - Le composant `FormInscription` (nom, âge, e-mail, niveau) affiche les erreurs, ou « ✔ Inscription possible », et le nombre de contrôles appliqués.
    - La formation « Bureautique » applique les contrôles de base ; la formation « Angular avancé » ajoute un contrôle : niveau « Confirmé » obligatoire.

apcpedagogie.com

Bureautique

Amel Ben Ali | 17
amel@exemple.tn | Débutant

✔ **Inscription possible**

3 contrôle(s) appliqué(s)

Angular avancé

Amel Ben Ali | 17
amel@exemple.tn | Débutant

Angular demande le niveau « Confirmé ».

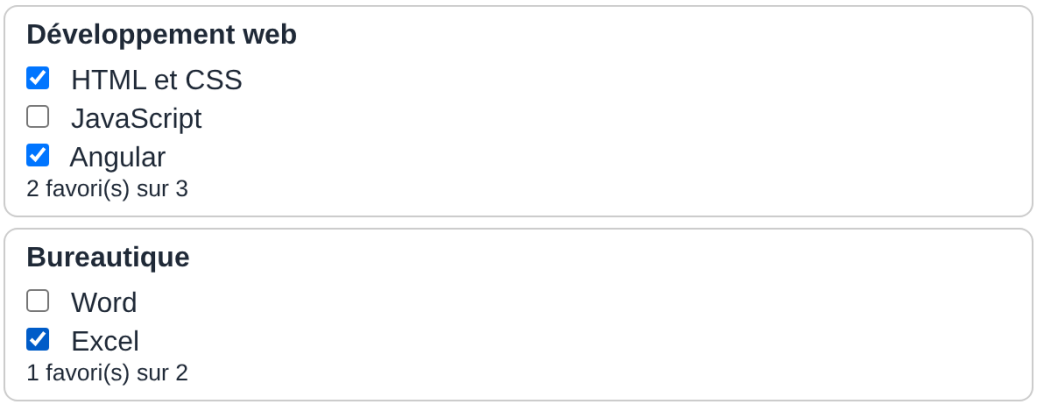
4 contrôle(s) appliqué(s)

- Vérifier : 3 contrôles pour Bureautique, 4 pour Angular avancé ; « Confirmé » rend l'inscription Angular possible.
- Piège : essayer de fournir **seulement** le contrôle du niveau dans le composant Angular, en laissant les contrôles de base dans un composant parent. Combien de contrôles sont appliqués ? Pourquoi ?

1. ## Activité 6 : deux catalogues avec provideCatalogue()

   - Créer le jeton privé `OPTIONS_CATALOGUE` (`titre`, `formations`) et le service `Catalogue` en `@Service({ autoProvided: false })` : il lit les options, gère une liste de favoris et un résumé « 2 favori(s) sur 3 ».
   - Écrire `provideCatalogue(options)` avec `makeEnvironmentProviders()` : les options, le service, et un `provideEnvironmentInitializer()` qui écrit « Catalogue « … » prêt (n formations) » dans la console.
   - Le composant `Vitrine` affiche un catalogue avec des cases à cocher. `DeuxCatalogues` crée deux injecteurs d'environnement (« Développement web » et « Bureautique ») et affiche une vitrine dans chacun avec `*ngComponentOutlet`, puis les détruit à sa destruction.

apcpedagogie.com



- Vérifier : les favoris des deux catalogues sont indépendants ; la console affiche deux fois « … prêt ».
- Comment écrire la même chose avec deux routes `/web` et `/bureautique` ?

1. **Activité 7 : tester un quiz au tirage aléatoire**
    - Le composant `Quiz` fournit le service `Tirage`, qui mélange les questions au hasard, puis affiche une question à la fois, ses choix, et le score :

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

export interface Question { texte: string; choix: string[]; bonne: number; }

const QUESTIONS: Question[] = [
  { texte: 'Quelle clé dit à Angular de rassembler plusieurs valeurs ?', choix: ['multi: true',
'many: true'], bonne: 0 },
  { texte: 'Quelle recette crée un alias ?', choix: ['useClass', 'useExisting'], bonne: 1 },
  { texte: 'Où écrire provideHttpClient() ?', choix: ['app.config.ts', 'providers d\'un
composant'], bonne: 0 },
];

// Mélange les questions : chaque quiz a son propre tirage
@Injectable()
export class Tirage {
  melanger<T>(liste: readonly T[]): T[] {
    return [...liste].sort(() => Math.random() - 0.5);
  }
}

@Component({
```

apcpedagogie.com

```
  selector: 'app-quiz',
  providers: [Tirage],
  template: `
    @if (question(); as q) {
      <p class="question">{{ rang() + 1 }}/{{ questions.length }} · {{ q.texte }}</p>
      @for (c of q.choix; track $index) {
        <button (click)="repondre($index)">{{ c }}</button>
      }
    } @else {
      <p class="fin">Terminé !</p>
    }
    <p class="score">Score : {{ score() }}</p>
  `,
})
export class Quiz {
  protected readonly questions = inject(Tirage).melanger(QUESTIONS);
  protected readonly rang = signal(0);
  protected readonly score = signal(0);
  protected readonly question = computed(() => this.questions[this.rang()]);

  protected repondre(i: number) {
    if (i === this.question().bonne) this.score.update((s) => s + 1);
    this.rang.update((r) => r + 1);
  }
}
```

- Écrire `quiz.spec.ts` avec une classe `TirageFixe` qui ne mélange rien et compte ses appels, et trois tests : le tirage du test est bien utilisé (« 1/3 · Quelle clé… ») ; une bonne réponse donne « Score : 1 » et passe à 2/3 ; trois bonnes réponses affichent « Terminé ! » et « Score : 3 ».

1/3 · Quelle recette crée un alias ?

useClass useExisting

Score : 0

- Vérifier : `ng test` affiche 3 tests réussis, à chaque exécution.
- Pourquoi `providers: [{ provide: Tirage, useValue: tirage }]` dans `configureTestingModule` ne suffit-il pas ?

apcpedagogie.com

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

1. Erreurs fréquentes

Message ou symptôme	Cause	Solution
`NG0201: No provider found for `InjectionToken CONFIG_ECOLE`.`	Jeton sans fabrique et sans provider	Le fournir avec `useValue`, ou lui donner une `factory`
`NG0201: No provider found for `Stockage`.`	Classe abstraite demandée sans provider	`{ provide: Stockage, useClass: … }` ou `@Service({ factory })`
`NG0201: No provider found for `Notifications`.`	`useExisting` vers un jeton qui n'est pas fourni	Fournir aussi le jeton cible
`TS2322: Type 'EnvironmentProviders' is not assignable to type 'Provider'.`	`provideXxx()` dans les `providers` d'un composant	Le placer dans `app.config.ts`, une route ou `createEnvironmentInjector()`
Seule la dernière valeur est reçue	`multi: true` oublié	Ajouter `multi: true` à **chaque** provider du jeton
Les règles du parent ont disparu	Multi providers dans un composant enfant : ils remplacent ceux du parent	Remettre les providers de base dans l'enfant (fonction `provideXxxBase()`)
`TypeError: this.s.lire is not a function`	`useValue: StockageMemoire`: la classe est donnée, pas une instance	`useClass: StockageMemoire`
Deux états différents pour deux noms	`useClass` utilisé pour un alias	`useExisting`
Page blanche, `ERROR HttpErrorResponse`	L'initialiseur de `provideAppInitializer()` a échoué	Vérifier le fichier ; attraper l'erreur et prévoir des valeurs par défaut
Le faux service est ignoré dans le test	Le composant fournit lui-même le service	`TestBed.overrideComponent(C, { set: { providers } })`
Fuite mémoire, initialiseur relancé	Injecteur créé avec `createEnvironmentInjector()` jamais détruit	`destroy()` dans `DestroyRef.onDestroy()`

apcpedagogie.com

1. À retenir

- Un provider associe un **jeton** (`provide`) à une **recette**; `[X]` est l'abréviation de `{ provide: X, useClass: X }`.
- On configure l'injecteur dans `app.config.ts`, dans une route, dans un composant, ou avec `createEnvironmentInjector()`; le provider le plus proche gagne.
- `useClass` crée une instance, `useValue` donne une valeur, `useFactory` fabrique selon le contexte, `useExisting` crée un alias.
- `InjectionToken<T>` représente ce qui n'est pas une classe ; l'option `factory` lui donne une valeur par défaut.
- `@Service({ factory })` met la recette sur le service ; `{ autoProvided: false }` oblige à le fournir.
- `multi: true` rassemble plusieurs valeurs dans un tableau ; dans un composant enfant, elles remplacent celles du parent.
- `makeEnvironmentProviders()` permet d'écrire ses fonctions `provideXxx()`, utilisables partout sauf dans un composant.
- `provideAppInitializer()` retarde l'affichage jusqu'au chargement des données indispensables.
- En test, `overrideComponent` remplace les providers déclarés par un composant.

> *Les composants disent ce dont ils ont besoin ; la configuration de l'injecteur décide comment y répondre. C'est elle qu'on modifie pour changer de comportement, de contexte ou d'environnement, sans réécrire l'application.*

Sommaire du cours Angular

Source : https://apcpedagogie.com/configurer-linjecteur-angular · téléchargé le 07/11/2026 · © APC Pédagogie · apcpedagogie.com