

Token d'injection

```
providers: [  
  CounterService  
]
```

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

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

Token d'injection

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

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

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

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

Token d'injection

- Pour utiliser le token :

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

```
firstCounterService = inject(FIRST_TOKEN)
```

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

useClass

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

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

useExisting

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

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

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

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

useValue

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

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

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

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

useFactory

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

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

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

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

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

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

Modification de la recherche

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

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

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

Modification de la recherche

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

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

Modification de la recherche

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

```
@Self() public firstCounterService: CounterService,
```

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

Modification de la recherche

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

```
@Host() public firstCounterService: CounterService,
```

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

Exercices

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

Exercices

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

Style

Encapsulation de style

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

ViewEncapsulation.Emulated

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

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

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

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

Pseudo-class selecteurs

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

`:host`

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

`:host-context`

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