Remarques

- La commuication entre le père et le fils se fait en deux étapes :
  - Le décorateur @Input coté fils, lie la valeur de l'attribut du composant et une propriété de son élément
  - Un property binding coté père

- On peut définir les propriétés de l'élément fils en mettant des attributs dans la balise HTML du composant

```
<app-selector [topics]="topics" key="title" (topicSelected)="selectedTopic=$event" />
```

- Le type des attributs étant forcément des strings, la méthode de transformation peut être un bon moyen de les convertir au type souhaité

- Par exemple, depuis Angular 17, une fonction est disponible pour transformer un input un boolean :

```
@Input({ transform: booleanAttribute }) expanded = false
```

Smart vs dumb components

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

Services et injection de dépendances

Services

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

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

Utilisaton

- Dans le constructeur

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

- Depuis Angular 14, avec la fonction inject()

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

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

Injection de dépendances

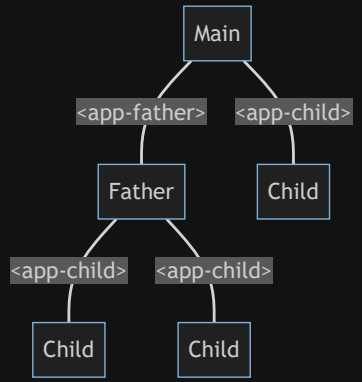
- L’injection de dépendances est un des principes fondamentaux d’Angular, qui mets en relation des providers et des consumers

- Les consumers peuvent être des composants, pipes, directives ou d’autres services

- L’injecteur gère une liste de providers, sous forme de singletons, et les fournit aux consumers qui en ont besoin

Définition des providers

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



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

Définition des providers

- Une application Angular comporte plusieurs injecteurs, avec chaqun une liste de providers
  - root : unique pour l'application
  - ElementInjector : spécifique pour un composant (ou une directive)

- En réalité, l'injecteur root est un EnvironmentInjector (anciennement ModuleInjector), et il y a plusieurs EnvironmentInjector dans une application Angular

root

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

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

- On peut également les définir dans les providers des modules

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

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

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

Commentaires

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

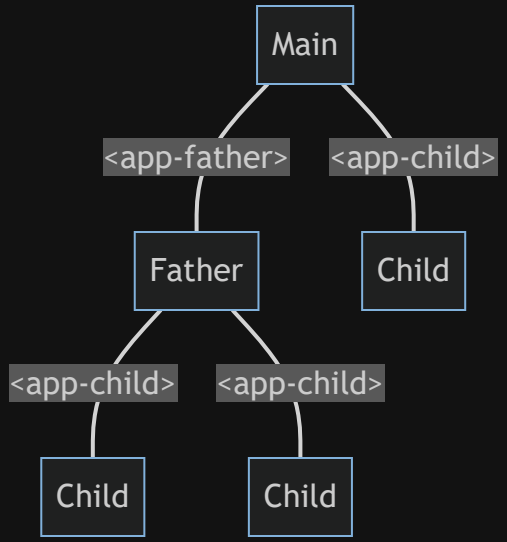
Element Injector

- Les composants et les directives ont également leur propre injecteur

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

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

Priorité de résolution



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

Token d’injection

```
providers: [
  CounterService
]
```

```
constructor(
  public firstCounterService: CounterService
)
```

- L’injecteur d’Angular assigne un token (une clé) à chaque provider pour l’indentifier

Token d’injection

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

```
constructor(
  @Inject(CounterService) public firstCounterService: CounterService
)
```

- L’injecteur d’Angular assigne un token (une clé) à chaque provider pour l’indentifier
- Pour une classe, par défaut le token est la classe elle-même

- 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 classé dédiée : InjectionToken<T>

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

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

- 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

useClass

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

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

- 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 a 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 déja
- 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 un valeur fixe à un token
- useValue peut être utiliser 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