5. Injection

Introduction

Le framework Angular met à disposition une implémentation du pattern d'injection de dépendance, permettant de mettre en oeuvre le principe d'inversion de contrôle (IOC). Il consiste à créer dynamiquement (à injecter) les dépendances entre différents objets, la mise en oeuvre de ce mécanisme étant délégué au framework.

Dans Angular, les objets à injecter sont décrits dans l'attribut `providers` (et `viewProviders` pour un composant) d'un `module` ou d'un `composant` . Ces objets peuvent être récupérés dans la classe qui souhaite exploiter l'objet, depuis l'injection dans le constructeur.

```
//dans le module
providers: [MonService];

//dans le composant
@Component()
export class MonComposant {
  constructor(private monService: MonService) {}
}
```

Un Objet injecté dans un `module` pourra être disponible dans n'importe quel Objet (via le constructeur) décrit dans le `module` . A l'inverse, un objet injecté dans un `composant` sera uniquement disponible pour ce composant (et ses enfants si l'injection est déclarée dans `viewProviders`).

Définition de l'injection

Il possible de personnaliser la façon dont l'on souhaite injecter un objet dans l'application :

- Directement par la classe : Le service est injecté sous forme d'une classe, l'objet associé sera instancié sous forme de singleton uniquement lorsqu'une demande d'injection sera identifié par le framework. La récupération de l'objet se fera au travers d'un objet typé `AuthService` dans la classe qui souhaite l'exploiter.

```
providers: [AuthService];
```

équivaut à

```
[{ provide: AuthService, useClass: AuthService }];
```

- Injection d'une implémentation ou d'une classe étendue : Idem au précédent, sauf que la récupération de l'objet se fera au travers d'un objet typé `AuthServiceBase` dans la classe qui souhaite l'exploiter, permettant de switcher facilement d'implémentation.

```
[{ provide: AuthServiceBase, useClass: AuthServiceMockImpl }];
```

- Injection d'un objet directement instancié : Le service est injecté sous forme d'un objet directement instancié lors de la création du module. Cela peut être interessant si l'on souhaite injecter une implémentation en fonction d'une configuration de build.

```
let impl =
  env.prod === true ? new AuthServiceWSImpl() : new AuthServiceMockImpl();

[{ provide: AuthServiceBase, useValue: impl }];
```

- Injection par une factory : le service est instancié au runtime par une factory. La factory peut avoir besoin de dépendances injectées également, qu'elle prend en entrée de la méthode de génération de l'instance. Cela peut être utile si vous avez besoin d'injecter un service, ayant besoin d'informations uniquement disponible au runtime.

```
{
    provide: AuthServiceBase,
    useFactory: (initService: InitService) => {
        return new AuthServiceWSImpl(initService.isReady);
    };
    deps: [InitService]
};
```

- Injection de plusieurs implémentation : dans le cas d'une injection de plusieurs objets implémentant une même classe de base, vous devrez avoir recourt aux `InjectionToken` , sinon dans la récupération de l'injection, l'erreur suivante sera remontée : 'Can't inject using the interface as the parameter type'.

```
export const AUTH_SERVICE_MOCK =
  new InjectionToken() < IAuthSerice > "auth.service.mock";
export const AUTH_SERVICE_WS =
  new InjectionToken() < IAuthSerice > "auth.service.ws";

providers: [
  { provide: AUTH_SERVICE_MOCK, useValue: AuthServiceMockImpl },
  { provide: AUTH_SERVICE_WS, useValue: AuthServiceWSImpl },
];
```

La récupération de l'injection se fera alors au travers du nom du token (voir chapitre suivant).

- Injection d'un objet non basé sur une classe : de la même façon, vous pourrez utiliser les `InjectionToken` pour injecter une donnée.

```
export const APP_CONF = new InjectionToken() < string > "app.config";

providers: [{ provide: APP_CONF, useValue: "hello" }];
```

La récupération de l'injection se fera également au travers du nom du token (voir chapitre suivant).

Récupération de l'injection

Décorateur @Injectable

Pour qu'une classe soit en capacité de récupérer un objet Angular Injecté, celle-ci doit être annotée `@Injectable`. Lorsqu'Angular va créer une instance de la classe, Angular va savoir au travers de cette annotation que des objets doivent être injecté dans son constructeur. Le décorateur `@Components` inclu de base la possibilité d'injecter des objets.

```
import { Injectable } from "@angular/core";
import { InitAppService } from "./init-app.service";

@Injectable()
export class MailingListSubscriptionService {
  constructor(private initAppService: InitAppService) {}
}
```

Dans l'exemple ici, le service `MailingListSubscriptionService` attend en entrée un service injecté, typé `InitAppService`.

Il est a noter que si vous annotez la variable injectée dans le constructeur comme `private`, `public` ou `protected`, celle-ci est défini comme étant un attribut de votre classe, et sera donc exploitable en déhors du constructeur. Dans le cas contraire, la portée de la variable est uniquement valable dans le constructeur.

Décoracteur @Inject

Dans le cas de l'InjectionToken, on utilisera en plus le décorateur `@Inject` en entête des objets à récupérer.

```
constructor(@Inject(AUTH_SERVICE_MOCK) authServiceM: AuthServiceBase,
            @Inject(AUTH_SERVICE_WS) authServiceWS: AuthServiceBase) {

}
```

Documentation officielle

Exercice

Présentation

Application Météo.

L'objectif de cet excercice est d'exploiter un service dans un composant Angular.

Etapes de création d'un service à injecter

1. Créer un dossier `service` dans le dossier app/meteo.
2. Créer un service `meteo`

> ng generate service meteo

3. Ajouter les appels nécessaires pour appeler l'API météo via la méthode publique `search(ville:string)` dans le service `meteo.service.ts`. Vous pouvez utiliser ce qui a été produit lors du TP JS/TS. Continuez à utiliser `fetch` pour l'appel, nous allons voir prochainement qu'il y a des alternatives.
4. Injecter ce service dans le composant `meteo-view.component.ts` & référencer le dans le provider du module `meteo`.

5. Dans le composant de recherche `meteo-search.component.ts`, utiliser un `EventEmitter` afin de propager la demande de recherche au parent `meteo-view.component.ts`, pour lancer la recherche.

Documentation sur l'EventEmitter

6. Associer l'EventEmitter dans la template HTML `meteo-view.component.htlm`, à un callback `searchHandler` que vous définierez dans le contrôleur `meteo-view.component.ts`.

7. Dans la méthode `searchHandler` définie dans le contrôleur `meteo-view.component.ts`, récupérer la ville dans le callback, et lancer l'appel API via `meteo.service.ts`. Récupérer le résultat en retour du service (promesse résolue via async / await).

8. Avec le Flux JSON, afficher l'entête de la ville dans le composant `meteo-view.component.html`.

   Pour cela nous allons créer des interfaces en Typescript à partir du Flux JSON, qui vont nous permettre de définir des ValueObject typés. Documentation ici

   i. Créer un dossier `vo` pour ValueObject.
   ii. Créer une interface `meteo-result.vo` avec comme propriété `city` & `list`
   iii. Créer une interface `meteo-city.vo` contenant les informations de la ville (voir infos disponibles dans flux json du retour de l'API météo).
   iv. Créer une interface `meteo-item.vo` contenant les informations d'un item de météo (voir infos disponibles dans flux json du retour de l'API météo).

Ces VO ne sont rien d'autres que la représentation de la structure JSON renvoyée par le service. Pour plus de simplicité, nous ne parserons pas / mapperons pas le JSON, donc utilisez les propriétés définis dans le JSON dans la définition de vos VO.

9. Une fois vos `vo` construit, vous pouvez afficher la propriété `city` (`meteo-city.vo`) de `meteo-result.vo` en entête de `meteo-view.component.html`. Utilisez pour cela la notion de databinding entre le controleur TS et la template HTML.

Regarder ici comment fonctionne le data binding : https://angular.io/guide/interpolation

10. Ajouter un loader sur la vue pendant le chargement. Ce loader est affiché au lancement du service, et caché à la fin de l'appel (appel en succès ou en erreur). Vous utiliserez pour cela le databinding sur un boolean défini dans votre contrôleur, et référencé dans votre template; ainsi que la directive nglf (nous y reviendrons dans un chapitre dédié).

Bravo, vous avez créé un service, vous l'avez injecté, vous l'avez exploité, vous avez généré un modèle de données associé, et vous avez affiché une donnée du résultat.

Prochaine étape, nous afficherons les données météo dans un prochain chapitre. Mais nous allons d'abord revenir sur le service `meteo.service.ts`, et regarder du côté de HTTPModule.

Correction disponible ici