

12/12/2022

Version 3

ANGULAR

JÉRÉMY PERROUULT



SERVICES

DISTRIBUER DES SERVICES

SERVICES

- Un service fournit un ensemble de fonctionnalités hors affichage
 - Intégré à un module ou à un composant
 - Réutilisable
- Un service permet de créer un objet TypeScript ordinaire
 - Qui va fournir un ensemble de tâches (fonctionnalités)
 - La création est encapsulée et isolée du reste du code
- Instance gérées par Angular et injectables par l'injection de dépendances
- Certaines sont natives à Angular
 - **Title, Meta**

SERVICES

- Utilisation de Angular CLI pour générer un composant

```
ng generate service monService
```

```
ng g s monService
```

- Angular CLI va
 - Générer les fichiers TS et SPEC.TS (pour les tests unitaires)
 - Ajouter dans le module principal la déclaration du nouveau service

<https://angular.io/cli/generate#service>

SERVICES

- Chaque service doit être dans un module (ou un composant), il faut donc
 - Importer ce service dans le module (ou dans le composant)
 - Le déclarer dans la liste des providers
 - Ceci est nécessaire lorsque l'option `providedIn` n'est pas spécifiée
 - Sinon, Angular le fait pour nous

SERVICES

- Dans le fichier *mon-service.service.ts*
 - On déclare le service, avec une classe, en le décorant de **@Injectable**
 - On y ajoute tous les attributs et méthodes dont le service a besoin pour faire son travail
- Dans le décorateur **@Injectable**, on peut préciser l'option `providedIn`
 - `root` Singleton Application
 - `any` Une instance par demande
 - `platform` Singleton partagé entre plusieurs app Angular (sur la même page HTML ...)

SERVICES

- Si on ne spécifie pas l'option `providedIn`
 - On peut choisir de fournir la dépendance au niveau d'un module ou d'un composant
 - Dans la liste des `providers`
- En l'ajoutant directement dans la liste

```
providers: [ MonServiceService ]
```

- Ou en spécifiant un objet littéral ...

```
providers: [  
  {  
    provide: MonServiceService  
  }  
]
```

SERVICES

- ... mais le plus souvent, en utilisant `provide` comme nom de « token »
- Et `useClass` pour utiliser une classe de service en particulier ...

```
providers: [  
  {  
    provide: 'LeTokenDuService',  
    useClass: MonServiceService  
  }  
]
```


SERVICES

- ... ou `useValue` comme valeur du service ...

```
providers: [  
  {  
    provide: 'LeTokenDuService',  
    useValue: {  
      methode: () => console.log("démo")  
    }  
  }  
]
```

SERVICES

- ... ou `useFactory` comme fonction de fabrique du service
 - Avec éventuellement `deps` si une dépendance est requise

```
providers: [  
  {  
    provide: 'LeTokenDuService',  
    useFactory: (service: AutreService) => ({  
      methode: () => console.log(service)  
    }),  
    deps: [  
      AutreService  
    ]  
  }  
]
```

SERVICES

- Une dernière option, `multi`
 - Permet d'indiquer si un même « token » peut fournir une liste de services

```
providers: [  
  {  
    provide: 'LeTokenDuService',  
    multi: true,  
    useValue: {  
      methode: () => console.log("Première démo")  
    }  
  },  
  {  
    provide: 'LeTokenDuService',  
    multi: true,  
    useValue: {  
      methode: () => console.log("Deuxième démo")  
    }  
  }  
]
```

Donc l'injection attendra un tableau

SERVICES

- Pour injecter ce service (dans un composant par exemple)
 - On utilise l'injection de dépendance

```
constructor(private monService: MonServiceService) { }
```

```
constructor(@Inject(MonServiceService) private monService: MonServiceService) { }
```

```
constructor(@Inject('LeTokenDuService') private monService: MonServiceService) { }
```

- Dans le cas d'un « token » multiple, on réceptionne un tableau

```
constructor(@Inject('LeTokenDuService') private mesServices: MonServiceService[]) { }
```

EXERCICE

- Récupérer le service **Title** pour modifier le titre de la page
 - L'utiliser sur chaque composant contrôleur

EXERCICE

- Créer un service **TodoService** qui permet
 - De gérer une liste de todos
 - De retourner tous les todos (méthode `findAll()`)
 - De retourner des todos par leur nom (méthode `findAllByNom()`)
 - De retourner un todo avec son id (méthode `findById(id)`)
 - D'ajouter un todo à sa liste (méthode `save(todo)`)
 - De modifier un todo de sa liste (méthode `save(todo)`)
 - De supprimer un todo par son identifiant (méthode `deleteById(id)`)
- Modifier les composants pour utiliser ce service (récupérer l'instance du service)
 - **TodoComponent**
 - **TodoDetailComponent**



OBSERVABLE

LA PROMESSE D'UN MONDE MEILLEUR

OBSERVABLE

- Observable est issu de la bibliothèque *RxJS*
 - Développement réactif, ou *Stream*
- Même objectif que les promesses, mais syntaxe et manipulation bien différente
 - On s'abonne au flux de données
 - On se désabonne du flux de données

OBSERVABLE

- Pour créer un **Observable**, utilisation du constructeur ...

```
let obs: Observable<number> = new Observable((observer)
=> {
  observer.next(1);
  observer.next(2);
  observer.next(3);
  observer.next(4);
  observer.next(5);
  observer.complete();
});
```

OBSERVABLE

- ... ou utilisation de la fonction `of` ou encore la fonction `from`

```
let obs: Observable<number> = of(1, 2, 3, 4, 5);
```

```
let obs: Observable<number> = from([ 1, 2, 3, 4, 5 ]);
```

OBSERVABLE

- La fonction `pipe` permet de modifier le flux, en filtrant, en le transformant par exemple
 - On peut lui fournir autant de fonctions de transformation que nécessaire, appelées « opérateur »
 - Ces opérateurs sont à importer depuis `rxjs` ou `rxjs/operators`

```
obs = obs.pipe(  
  filter(val => (val as number) > 2),  
  map(val => val as number * 2),  
  tap(val => console.log(val))  
);
```

OBSERVABLE

- Une fois qu'on a notre **Observable**, il faut s'abonner au flux pour manipuler les données

```
let sub: Subscription = obs.subscribe(val => console.log(val));
```

- Lorsqu'on s'abonne à un flux, il faut penser à se désabonner

- Sauf dans un cas particulier, avec Angular, et le pipe *async*, qui fera ces actions pour nous

```
sub.unsubscribe();
```

EXERCICE

- Créer un flux de liste de posts
 - Filtrer sur le titre
 - Afficher la liste sur un template, avec le pipe *async*



HTTP

COMMUNIQUER AVEC UNE API

HTTP

- A l'instar de \$.ajax (en *jQuery*), ou de *fetch*, **HttpClient** nous permet d'interroger une ressource
 - En précisant la commande HTTP
 - En ajoutant des données dans le corps de la requête (*body*)
- Pour l'utiliser
 - Il faut déclarer l'utilisation du module **HttpClientModule** (dans le module principal)
 - Importer et injecter **HttpClient** là où on a besoin de l'utiliser (dans le service par exemple)

```
import { HttpClientModule } from '@angular/common/http';
```

```
import { HttpClient } from '@angular/common/http';
```

HTTP

- Disponible depuis la version 4.3 de Angular
 - C'est une évolution de **Http**
 - Il convertit automatiquement en JSON si nécessaire
 - Il retourne un **Observable** (RxJS)
 - Pour le reste, il s'utilise de la même façon que **Http** (get, post, put, delete, patch)

HTTP

- **HttpClient** met à disposition ces méthodes

| Nom de la méthode | Paramètres |
|-------------------|---------------------|
| get | url, options? |
| post | url, body, options? |
| put | url, body, options? |
| patch | url, body, options? |
| delete | url, options? |

```
http.get('/api/produit');  
http.post('/api/produit', produit);  
http.put(`/api/produit/${ produit.id }`, {  
    nom: "GoPRO HERO 11",  
    prix: 420  
});  
http.delete(`/api/produit/${ produit.id }`);
```

- Le nom des méthodes correspond aux commandes HTTP

HTTP

- Chacune des méthodes retournent un objet de type **Observable**
 - La réponse reçue est automatiquement traitée en *JSON*
 - La requête envoyée est automatiquement traitée en *JSON*
 - On peut préciser la nature de l'objet retourné par l'appel au service

```
http.get<Produit>('...');
```

```
http.get<Produit[]>('...');
```

```
http.get<Array<Produit>>('...');
```

HTTP

- Chaque méthode retourne un objet de type **Observable**
 - A la main, on va écouter la réception d'une réponse en s'inscrivant à l'**Observable**
 - Avec la méthode *subscribe* (Si cette méthode n'est pas appelée, l'appel **HTTP** ne se fait pas !)
 - En s'abonnant à la main, il faut se désabonner à la main aussi avec la méthode *unsubscribe*

```
let sub: Subscription = this.http.get<Array<Produit>>('...')  
  .subscribe((produits) => {  
    console.log(produits);  
  });
```

```
// ...
```

```
sub.unsubscribe();
```

HTTP

- Exemple complet avec une souscription manuelle, à éviter ...

```
export class AppComponent implements OnInit, OnDestroy {
  produits!: Produit[];
  produitsSubscription!: Subscription;

  constructor(private srvProduit: ProduitService) { }

  ngOnInit(): void {
    this.produitsSubscription = this.srvProduit.findAll().subscribe((produits) => {
      this.produits = produits;
    });
  }

  ngOnDestroy(): void {
    this.produitsSubscription.unsubscribe();
  }
}
```

```
<ul>
  <li *ngFor="let produit of produits">
    {{ produit.nom }}
  </li>
</ul>
```

HTTP

- ... puisqu'on va préférer utiliser le *Stream* directement (**Observable**)

```
export class AppComponent implements OnInit {  
  produits$: Observable<Array<Produit>>;  
  
  constructor(private srvProduit: ProduitService) { }  
  
  ngOnInit(): void {  
    this.produits$ = this.srvProduit.findAll();  
  }  
}
```

On utilise \$ en suffixe de variable pour signifier qu'il s'agit d'un *Stream*

```
<ul>  
  <li *ngFor="let produit of produits$ | async">  
    {{ produit.nom }}  
  </li>  
</ul>
```

Le pipe *async* va s'abonner et se désabonner automatiquement

HTTP

- Grâce aux **Observable**, en cas d'erreur ou de paramètres manquants
 - On peut retourner un **Observable** par défaut avec les fonctions `of` ou `from`

EXERCICE

- Modifier le service **TodoService**
 - Utiliser **HttpClient**
 - `findAll()` ira chercher la liste
 - <https://jsonplaceholder.typicode.com/todos>
 - En cas d'erreur, retourner un tableau vide

EXERCICE

- Modifier le service **TodoService**
 - Implémenter les méthodes restantes pour le CRUD
 - La méthode *findAll* appelle le service qui retourne tous les todos
 - La méthode *findById* appelle le service qui retourne un todo par son identifiant
 - La méthode *save* appelle le service qui ajoute ou sauvegarde un todo
 - La méthode *deleteById* appelle le service qui supprime un todo par son identifiant

HTTP

- On peut utiliser un **HttpInterceptor**

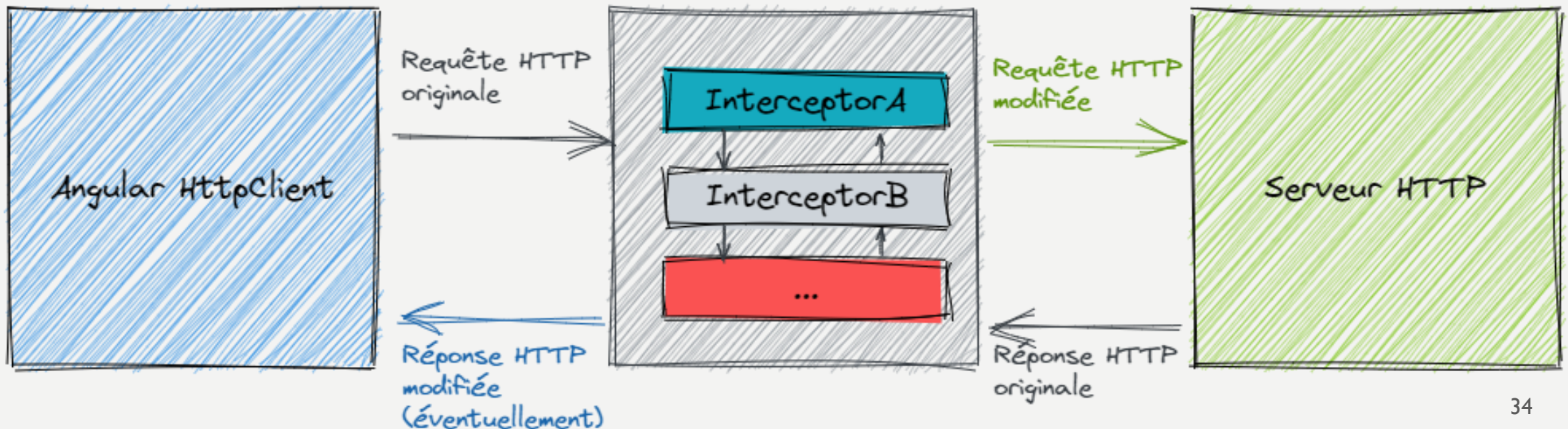
```
ng generate interceptor monInterceptorHttp
```

```
ng g interceptor monInterceptorHttp
```

- Utilisé(s) en les déclarant dans le provider
 - Permettent d'ajouter des informations dans l'en-tête, de journaliser, etc. chaque requête avec **HttpClient**

HTTP

- Utilisé(s) en les déclarant dans le provider, permettent pour chaque requête **HttpClient**
 - De modifier l'en-tête HTTP (pour l'authentification par exemple)
 - De journaliser
 - etc.



HTTP

- Il faut ensuite fournir ces intercepteurs en tant que fournisseurs, dans le module
 - Avec le « token » **HTTP_INTERCEPTORS** (@angular/common/http)
 - Et l'option « multi » à vrai

```
providers: [  
  {  
    provide: HTTP_INTERCEPTORS,  
    useClass: MonInterceptor,  
    multi: true  
  }  
]
```

A decorative wavy line in light blue and white, flowing vertically along the left side of the slide.

ENVIRONNEMENT

LES VARIABLES D'ENVIRONNEMENT

ENVIRONNEMENT

- Il est possible d'ajouter des variables d'environnement
 - Conditionnées à l'environnement d'exécution (dev, test, production, etc.)
- Créer des fichiers
 - *src/environments/environment.ts*
 - *src/environments/environment.prod.ts*

```
export const environment = {  
  apiUrl: 'http://localhost:8080'  
}
```

ENVIRONNEMENT

- On import ce fichier comme n'importe quel autre pour le manipuler

```
import { environment } from 'src/environments/environment';  
console.log(environment.apiUrl);
```

ENVIRONNEMENT

- On configure Angular pour lui spécifier de modifier les fichiers pour une configuration
 - *angular.json*

```
"configurations": {  
  "production": {  
    "fileReplacements": [  
      {  
        "replace": "src/environments/environment.ts",  
        "with": "src/environments/environment.prod.ts"  
      }  
    ]  
  }  
}
```

ENVIRONNEMENT

- Pour tester en développement, on peut spécifier l'option de configuration

```
ng serve --configuration production
```

```
ng s -c production
```


EXERCICE

- Ajouter une variable d'environnement « apiUrl »
 - Qui aura pour valeur « <https://jsonplaceholder.typicode.com> »
- Modifier le service pour utiliser la variable



FORMBUILDER

CREATION DE FORMULAIRES

FORMBUILDER

- Comme vu précédemment, le module **FormsModule** a introduit **ngModel**
- Le module **ReactiveFormsModule** apportera
 - **FormBuilder**, **FormGroup**, **FormControl**, et **Validators**
 - **FormBuilder** nous aide à fabriquer un groupe et des contrôles
 - **FormGroup** nous permet de regrouper un ensemble de contrôles
 - **FormControl** nous permet d'avoir un contrôle de saisi
 - **Validators** nous permet de valider une saisie

FORMBUILDER

- Il faut récupérer **FormBuilder** par l'injection de dépendance
 - On peut l'utiliser pour fabriquer des contrôles, qu'on ajoutera dans un groupe

```
userForm!: FormGroup;

ngOnInit(): void {
  this.userForm = this.formBuilder.group({
    username: this.formBuilder.control('valeur par défaut'),
    password: this.formBuilder.control('')
  });
}
```

FORMBUILDER

- Côté Template

```
<form (ngSubmit)="onSubmit()" [formGroup]="userForm">
  <div>
    <label>Nom d'utilisateur</label>
    <input formControlName="username" />
  </div>

  <div>
    <label>Mot de passe</label>
    <input type="password" formControlName="password" />
  </div>

  <input type="submit" value="Se connecter !" />
</form>
```

Nom du groupe

Nom du contrôle

EXERCICE

- Créer un formulaire pour ajouter un utilisateur avec **ReactiveForms**
 - Nom d'utilisateur
 - Adresse e-mail
 - Mot de passe
 - Vérification du mot de passe

FORMBUILDER

- On peut ajouter des validations sur les contrôles

```
this.userForm = this.formBuilder.group({  
  username: this.formBuilder.control('Valeur', Validators.required),  
  password: this.formBuilder.control('', [ Validators.required, Validators.minLength(8) ])  
});
```

FORMBUILDER

```
<form (ngSubmit)="onSubmit()" [formGroup]="userForm">
  <div>
    <label>Nom d'utilisateur</label>
    <input formControlName="username" />

    <div *ngIf="userForm.get('username')?.hasError('required')">
      Le nom d'utilisateur est obligatoire
    </div>
  </div>

  <div>
    <label>Mot de passe</label>
    <input type="password" formControlName="password" />

    <div *ngIf="userForm.get('password')?.dirty && userForm.get('password')?.hasError('required')">
      Le mot de passe est obligatoire
    </div>

    <div *ngIf="userForm.get('password')?.hasError('minlength')">
      Le mot de passe doit contenir au moins 8 caractères
    </div>
  </div>

  <input type="submit" [disabled]="userForm.invalid" value="Se connecter !" />
</form>
```


FORMBUILDER

- Pour alléger un peu le Template, on peut stocker les contrôles

```
userForm!: FormGroup;
usernameCtrl!: FormControl;
passwordCtrl!: FormControl;

ngOnInit(): void {
  this.usernameCtrl = this.formBuilder.control('Valeur', Validators.required);
  this.passwordCtrl = this.formBuilder.control('', [ Validators.required, Validators.minLength(8) ]);

  this.userForm = this.formBuilder.group({
    username: this.usernameCtrl,
    password: this.passwordCtrl
  });
}
```

FORMBUILDER

```
<form (ngSubmit)="onSubmit()" [formGroup]="userForm">
  <div>
    <label>Nom d'utilisateur</label>
    <input formControlName="username" />

    <div *ngIf="usernameCtrl.hasError('required')">
      Le nom d'utilisateur est obligatoire
    </div>
  </div>

  <div>
    <label>Mot de passe</label>
    <input type="password" formControlName="password" />

    <div *ngIf="passwordCtrl.dirty && passwordCtrl.hasError('required')">
      Le mot de passe est obligatoire
    </div>

    <div *ngIf="passwordCtrl.hasError('minlength')">
      Le mot de passe doit contenir au moins 8 caractères
    </div>
  </div>

  <input type="submit" [disabled]="userForm.invalid" value="Se connecter !" />
</form>
```

FORMBUILDER

- Il est également possible de créer ses propres validateurs

```
const customValidator = (arg1: any): ValidatorFn => {  
  return (control: AbstractControl): ValidationErrors | null => {  
    if (control.value !== arg1) {  
      return { nomErreur: true }  
    }  
  
    return null;  
  };  
};  
  
this.username = this.formBuilder.control('', customValidator("démo"));
```

```
<div *ngIf="usernameCtrl.hasError('nomErreur')">  
  Erreur personnalisée !  
</div>
```

EXERCICE

- Ajouter des validateurs

- Nom d'utilisateur

Obligatoire

- Adresse e-mail

Obligatoire, format e-mail

- Mot de passe

Obligatoire, minimum 8 caractères, 1 majuscule, 1 caractère spécial

- Vérif mot de passe

Vérifier qu'il correspond au mot de passe