esprit
Se former autrement
ONORIS UNITED UNIVERSITIES

Le service HttpClient

United Nations Educational, Scientific and Cultural Organization
UNESCO Chair "Project-based learning" ESPRIT School of engineering, Tunisia
uniTwin

EUR-ACE®
Délivrée par la Commission des Titres d'Ingénieur

CONFERENCE DES GRANDES ÉCOLES

cdio
CONCEIVE DESIGN IMPLEMENT OPERATE

- Présentation du service HttpClient
- Envoie de la requête « HttpRequest »
- HttpResponse
- Transformation des données avec les opérateurs RxJS
- La méthode subscribe()
- Gestion des erreurs « HttpErrorResponse »
- Exploitation des méthodes de HttpClient (CRUD)
- Problème de CORS

Présentation du service HttpClient

HttpClient - Introduction

- La plupart des applications front-end doivent communiquer avec un serveur via le protocole HTTP, pour télécharger ou envoyer des données et accéder à d'autres services back-end. Angular fournit une **API HTTP** côté client pour les applications Angular.
- Cette API est la classe de service HttpClient,

HttpClient - Caractéristiques

- **HttpClient** est un **service Angular** inclus dans le package @angular/common/http qui permet à ton application **de communiquer avec un serveur via HTTP**.
- Pour l'utiliser, il faut l'injecter (dans un composant ou un service).
- Il supporte toutes les méthodes principales du protocole http
- HttpClient retourne toujours des **Observables** (Observable<T>).
  Tu peux donc :
  - réagir de façon asynchrone,
  - chaîner les appels (map, catchError, etc.),
  - typer tes réponses.

Fournisseur de HttpClient

HttpClient est fourni en utilisant la fonction provideHttpClient()

1) Dans un projet standalone, app.config.ts

```
export const appConfig: ApplicationConfig = {
  providers: [ provideHttpClient(),] };
```

2) Dans un projet modulaire (notre cas):

```
@NgModule({
  providers: [
    provideHttpClient(),
  ],
  // ... other application configuration
})
export class AppModule {}
```

HttpClient - Méthodes

Méthode	Description	Exemple
get()	Lire / récupérer des données	this.http.get('api/users')
post()	Envoyer des données	this.http.post('api/users', user)
put()	Remplacer complètement une ressource	this.http.put('api/users/1', user)
patch()	Modifier partiellement une ressource	this.http.patch('api/users/1', { name: ‘foulen' })
delete()	Supprimer une ressource	this.http.delete('api/users/1')
head() / options()	Métadonnées de la requête	Rarement utilisé

- Ces méthodes retournent un Observable qui émet les données de la réponse. On doit s'abonner à l'Observable pour traiter la réponse.

Envoie de la requête « HttpRequest »

HttpClient - HttpRequest

- HttpRequest est **la classe Angular** qui représente **une requête HTTP sortante**, c'est-à-dire *ce que tu envoies au serveur*.
- Un objet HttpRequest contient toutes les informations de la requête:
  ✓ URL
  ✓ méthode (GET, POST, etc.)
  ✓ headers
  ✓ params
  ✓ body
  ✓ options
- Cet objet est utilisé par **HttpClient** et **les interceptors**.
- Cet objet est créé par Angular

HttpClient - HttpRequest

- Les propriétés clés :

```
HttpRequest<T> {
  method: string;
  url: string;
  body: T | null;
  headers: HttpHeaders;
  params: HttpParams;
  withCredentials: boolean;
  responseType: 'json' | 'text' | 'blob' | ...
}
```

- On ne peut pas modifier un objet HttpRequest, il faut le cloner.

```
const clone = req.clone({
  setHeaders: { Authorization: `Bearer ${token}` }
});
```

req.headers = ... ✗ interdit

HttpClient – params: Les paramètres d'URL

- Vous pouvez spécifier les paramètres de la requête (query params) qui doivent être inclus dans l'URL de la requête en utilisant l'option **params**.

```
this.http.get('https://api.example.com/users', {
  params: { active: 'true', sort: 'name' }
});
```

- Ils servent à **envoyer des informations visibles dans la requête**, souvent pour filtrer ou trier des données.
- Angular construit l'objet httpRequest avec la valeur de params fournie.

HttpClient – headers: Les en-têtes de requête

- Vous pouvez spécifier les en-têtes de requête qui doivent être inclus dans la requête en utilisant l'option **headers**.

```
this.http.post('https://api.example.com/data', body, {
  headers: {
    'Content-Type': 'application/json',
    'Authorization': 'Bearer token123'
  }
});
```

- Les **request headers** sont des **métadonnées** envoyées avec la requête, dans le corps HTTP, **non visibles dans l'URL**.
  Ils servent à **informer le serveur** sur la requête ou à **transmettre des informations sensibles ou techniques**.
- Angular construit l'objet httpRequest avec la valeur de headers fournie.

HttpClient - Options internes de Angular

Ces options modifient comment Angular gère la requête et la réponse

Elles n'aparaissent ni dans l'URL ni dans les headers et elle ne sont pas visibles côté serveur

Option	Type	Valeurs possibles	Rôle
observe	string	'body' (défaut), 'response', 'events'	Définit **ce que HttpClient renvoie** dans l'observable
reportProgress	boolean	true / false	Active les **événements de progression** (upload/download)
responseType	string	'json' (défaut), 'text', 'blob', 'arraybuffer'	Indique le **type de contenu attendu** dans la réponse
withCredentials	boolean	true / false	Envoie les **cookies ou tokens de session** (CORS)
context	HttpContext	objet	Permet de **passer des infos personnalisées** aux interceptors
transferCache (Angular SSR)	boolean	true / false	Utilisé pour la **mise en cache côté serveur (SSR)**

HttpClient – L'Option interne responseType

- Par défaut, HttpClient suppose que les serveurs renverront des données JSON.
- Lorsqu'on interagit avec une API non JSON, on peut indiquer à HttpClient le type de réponse attendu et à renvoyer lors de la requête à travers l'option **responseType**.
- Exemple :

```
http.get('/images/dog.jpg',
{responseType: 'arraybuffer'})
```

responseType **value**	**Returned response type**
'json' (default)	JSON
'text'	string
'arraybuffer'	ArrayBuffer contenant les octets de réponse bruts
'blob'	Blob instance

HttpClient – L’Option interne observe

Mode	Ce que tu reçois dans .subscribe()	Exemple
'body' (défaut)	Le corps de la réponse (response.body)	observe: 'body'
'response'	Objet complet HttpResponse (headers + status + body)	observe: 'response'
'events'	Tous les événements du cycle HTTP	observe: 'events'

```
getUsers(): Observable<HttpResponse<any>> {
  return this.http.get<any>(this.apiUrl, {
    observe: 'response'
  });
}
```

```
this.usersService.getUsers().subscribe({
    next: (response: HttpResponse<any>) => {
      console.log(response.body);
      console.log(response.status.toString());
      console.log(response.headers.get('Content-Type') ?? 'N/A');
    },
    error: (err) => {
      console.error('Erreur:', err);
    }
  });
}
```

HttpClient – Autres options internes

```
this.http.get('https://api.example.com/users', {
  observe: 'response',          // Renvoie tout l'objet HttpResponse
  reportProgress: true,         // Permet de suivre la progression
  responseType: 'json',         // Attends une réponse JSON
  withCredentials: true         // Envoie les cookies/session
})
```

Angular construit l'objet httpRequest avec les valeurs fournies.

HttpClient - HttpOptions

- On peut regrouper les différentes options d'une requête dans un objet appelé HttpOptions afin de garder le code plus propre et réutiliser la configuration
- Cet objet est appelé httpOptions par convention mais ce n'est pas un mot réservé on peut l'appeler comme on veut.

```
const httpOptions = {
  headers: {
    Authorization: 'Bearer token123'
  },
  params: {
    page: '1',
    limit: '10'
  },
  observe: 'response',
  responseType: 'json',
  withCredentials: true
};
```

```
this.http.get('https://api.example.com/users',
httpOptions)
  .subscribe(res => console.log(res));
```

HttpResponse

HttpClient - HttpResponse

- HttpResponse est **la classe Angular** qui représente une **réponse HTTP réussie** envoyée par le serveur **à ton application**.
- C’est ce que tu reçois **quand tout va bien** (status 200–299).
- **C’est une réponse complète** du serveur contenant : le body, le status, les headers, l’URL…
- Importé depuis @angular/common/http
- Envoyé seulement si tu demandes explicitement la réponse complète à travers l’option { observe: 'response' } sinon tu reçois par défaut le body de la réponse.

HttpClient - HttpResponse

- Les propriétés les plus importantes:

```
HttpResponse<T> {
  body: T;
  status: number;
  statusText: string;
  headers: HttpHeaders;
  url: string | null;
  ok: boolean;        // true si 200-299
  type: HttpEventType.Response;
}
```

- On utilise HttpResponse pour lire des informations importantes comme :
- ✓ les **headers** (ex : tokens, pagination, rate limits)
  ✓ le **status code**
  ✓ les **cookies** (si envoyés)
  ✓ la **réponse complète**

HttpClient - HttpResponse

- Exemple:

```
this.http.get('/api/users', { observe: 'response' })
  .subscribe((res: HttpResponse<any>) => {
    console.log('Status:', res.status);
    console.log('Headers:', res.headers);
    console.log('Body:', res.body);
  });
```

- Résultat:

```
Status: 200
Headers: HttpHeaders { ... }
Body: [{ id: 1, name: 'Ameni' }]
```

Transformation des données avec RxJS

pipe()

- pipe() est **une méthode des Observables** qui sert à **enchaîner des opérateurs RxJS** pour transformer les données, gérer les erreurs, filtrer…
- un "pipeline" où tu fais passer les données et appliques des transformations **dans l'ordre**.

```
this.http.get('/api/users')
  .pipe(
    map(users => users.filter(u => u.active)),   // transformation
    tap(() => console.log('Request done')),      // effet secondaire
  )
  .subscribe(result => console.log(result));
```

HttpClient → pipe(map → filter → tap) → subscribe()

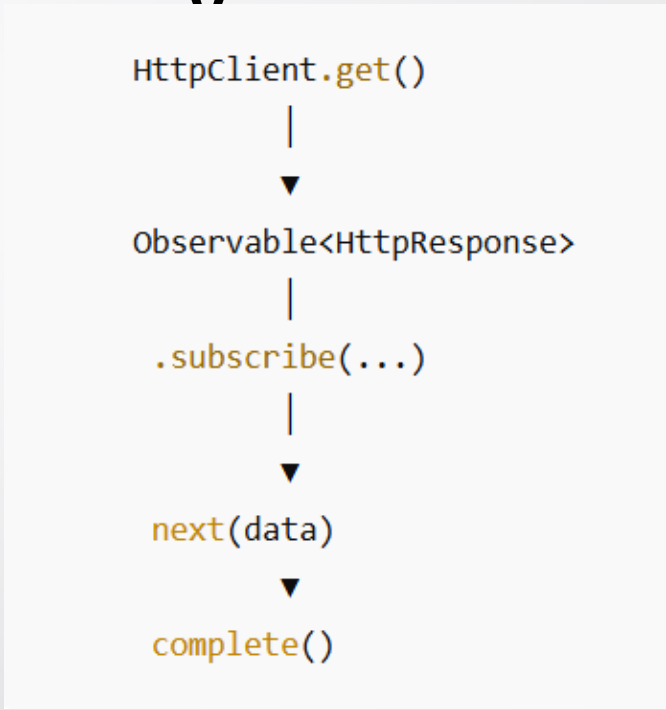
Les opérateurs RxJS utiles avec HttpClient

Opérateur	Rôle	Utilisé dans pipe() ?
map()	Transformer les données	✓
tap()	Effets secondaires (logs, loader…)	✓
switchMap()	Changer d'observable (requêtes dépendantes)	✓
mergeMap()	Chaîner en parallèle	✓
forkJoin()	Combiner plusieurs requêtes	✗
catchError()	Intercepter les erreurs	✓
finalize()	Code toujours exécuté (ex: désactiver loader)	✓

La méthode subscribe()

HttpClient – subscribe()

- Dans le cas de HttpClient, subscribe() reçoit **toujours une seule valeur** qui est la réponse puis **complete** automatiquement.
- Le flux se ferme tout seul, **pas besoin de unsubscribe()**
- Donc :
  1. **next()** : reçoit la réponse
  2. **complete()** : appelé juste après next()
  3. **error()** : si un problème serveur survient



Gestion des erreurs « HttpErrorResponse »

HttpClient – HttpErrorResponse

- **HttpErrorResponse** est la classe Angular qui représente **une erreur HTTP** renvoyée par HttpClient.
- Quand une requête HttpClient échoue (404, 401, 500, réseau, timeout…), Angular crée un objet **HttpErrorResponse** et l'envoie dans catchError() ou dans subscribe({ error }).

```
HttpErrorResponse {
  message: string;      // Le message d'erreur
  status: number;       // Le code HTTP (ex: 404, 500, 0)
  statusText: string;   // Le texte du statut (ex: "Not Found")
  url: string | null;   // URL appelée
  error: any;           // Ce que le backend renvoie
}
```

```
{
  "status": 404,
  "statusText": "Not Found",
  "message": "Http failure response for /api/user: 404 Not Found",
  "url": "/api/user",
  "error": { "message": "User not found" }
}
```

- status = 0 => erreur réseau et non pas une erreur serveur (internet coupé, mauvais domaine, CORS bloqué, serveur down)

HttpClient – HttpErrorResponse

- La propriété « **message** » de HttpErrorResponse contient un message généré par ANGULAR pas par le serveur. Ce message ressemble souvent à :

  Http failure response for https://api.com/users: 404 Not Found

  Ou bien pour une erreur réseau

  Http failure at connecting to https://api.com/users

- La propriété « **error.message** » est un message d'erreur renvoyé par ton API. Il dépend entièrement du **serveur**. Si le backend ne renvoie pas message, alors **error.error.message = undefined.**

HttpClient – Gestion des erreurs

Pour gérer les erreurs on peut :

- Utiliser catchError() dans pipe()
- Centraliser les erreurs dans un service

On ne gère pas les erreurs dans les components

On les traite dans un **service** dédié

HttpClient - catchError()

- catchError() est **un opérateur RxJS** qui permet d'**intercepter et traiter les erreurs** qui se produisent dans un observable, y compris les erreurs HTTP dans Angular.
- Dans le code normal on utilise try/catch, pour les observables on utilise catchError()
- Fonctionnement catchError() :

1. attrape les erreurs dans un flux RxJS
2. te permet de faire une action (logguer, afficher un message, renvoyer une alternative…)
3. renvoie un **nouvel observable** (souvent via throwError() ou of(), EMPTY,…)

HttpClient - catchError()

catchError() est **un opérateur RxJS** qui permet d'**intercepter et traiter les erreurs** qui se produisent dans un observable, y compris les erreurs HTTP dans Angular.

1) Exemple 1 :

```
this.http.get('/api/users').pipe(
  catchError((error: HttpErrorResponse) => {
    console.error('Erreur HTTP :', error);
    return throwError(() => error);
  })
);
```

2) Exemple 2:

```
getUser() {
  return this.http.get('/api/user/1').pipe(
    catchError((err: HttpErrorResponse) => {
      if (err.status === 404) {
        return throwError(() => 'Utilisateur introuvable');
      }
      return throwError(() => 'Erreur inconnue');
    })
  );
}
```

HttpClient – ErrorService

```
import { Injectable } from '@angular/core';
import { HttpErrorResponse } from '@angular/common/http';
import { throwError } from 'rxjs';

@Injectable({ providedIn: 'root' })
export class ErrorService {

  handleError(error: HttpErrorResponse) {
    let message = '';

    if (error.error instanceof ErrorEvent) {
     message = `Erreur réseau : ${error.error.message}`;
    } else {
      message = `Erreur serveur (${error.status}) : ${error.message}`;
    }
    console.error(message);

    return throwError(() => message);
  }
}
```

Exploitation des méthodes de HttpClient

Etapes

- Pour appliquer les défférentes méthodes : GET, POST, PUT et DELETE du service HttpClient de Angular, il faut:
- 1- Créer un service pour appeler ces différentes méthodes.
- 2- Injecter le service HttpClient dans le service créé
- 3- Avoir les urls des différentes « api » de CRUD. Angular consomme les méthodes de CRUD créés dans la partie backend.

Affichage - READ

Au niveau du service : Définition de méthode

```
apiUrl : string = "http://.....";
getData(): Observable<Type[]>{
    return this.http.get<Type[]>(this.apiUrl);
  }
```

Au niveau du composant : consommation de la méthode du service

```
constructor(private _service : CRUDService) { }
getDataFormService(){
    this._service.getData().subscribe(res=>traitement);

  }
```

Ajout - Create

Au niveau du service : Définition de méthode

```
apiUrl : string = "http://.....";
httpOptions = { headers: new HttpHeaders({
      'Content-Type': 'application/json'})}

addData(myObject:Type): Observable<Type>{
    return this.http.post<Type>(this. apiUrl, myObject, this.httpOptions);  }
```

Au niveau du composant : consommation de la méthode du service

```
constructor(private _service : CRUDService) { }
addMyObject(obj:Type){
    this._service.addData(obj).subscribe();
  }
```

Modification - Update

Au niveau du service : Définition de méthode

```
apiUrl : string = "http://.....";
httpOptions = { headers: new HttpHeaders({
      'Content-Type': 'application/json'})}

updateData(id:number,myObject:Type): Observable<Type>{
    return this.http.put<Type>(this.apiUrl+'/'+ id, myObject, this.httpOptions);  }
```

Au niveau du composant : consommation de la méthode du service

```
constructor(private _service : CRUDService) { }
updateMyObject(id:number,obj:Type){
    this._service.updateData(id,obj).subscribe();
  }
```

Suppression - Delete

Au niveau du service : Définition de méthode

```
apiUrl : string = "http://.....";
deleteData (myObject: Type | number): Observable<Type> {
    const id = typeof myObject === 'number' ? myObject : myObject.id;
    return this.http.delete<Type>(this.apiUrl+'/'+id);
  }
```

Au niveau du composant : consommation de la méthode du service

```
constructor(private _service : CRUDService) { }
deleteMyObject(objToDelete){
    this._service. deleteData(objToDelete).subscribe();

  }
```

Problème de CORS

Définition

Le « *Cross-origin resource sharing* » (CORS) est un mécanisme qui consiste à ajouter des en-têtes HTTP afin de permettre à un agent utilisateur d'accéder à des ressources d'un serveur situé sur une autre origine que le site courant. Un agent utilisateur réalise une requête HTTP **multi-origine (*cross-origin*)** lorsqu'il demande une ressource provenant d'un domaine, d'un protocole ou d'un port différent de ceux utilisés pour la page courante

Définition

- Pour des raisons de sécurité, les requêtes HTTP multi-origine émises depuis les scripts sont restreintes.
- Ainsi XMLHttpRequest et l’API Fetch respecte la règle d’origine unique.
- Cela signifie qu'une application web qui utilise ces API peut uniquement émettre des requêtes vers la même origine que celle à partir de laquelle l'application a été chargée, sauf si des en-têtes CORS sont utilisés.

Définition

- Lorsque nous développons une application Angular qui a besoin d'un back-end pour conserver les données, le back-end est souvent servi sur un autre port de localhost.
- Par exemple, l'URL de l'application angular frontale est http://localhost:4200, tandis que l'URL du serveur principal est http://localhost:3000.
- Dans ce cas, si nous effectuons une requête HTTP de l'application frontale vers le serveur principal, il s'agit d'une requête interdomaine et nous devons effectuer un travail supplémentaire pour y parvenir

Solutions

Il y a deux solutions, nous pouvons utiliser

- CORS (Solution coté backend)
- un proxy côté serveur (Solution coté Angular)

Néthographie

- https://angular.io/guide/
- https://guide-angular.wishtack.io/angular
- https://developer.mozilla.org/fr/docs/Web/HTTP/CORS
- https://betterprogramming.pub/setup-a-proxy-for-api-calls-for-your-angular-cli-app-6566c02a8c4d

Merci de votre attention