# SERVICE HTTPCLIENT/CORS

INF1013, Hiver 2022

DMI, UQTR

# Angular HttpClient Introduction.

- Les navigateurs modernes supportent 2 APIs pour les appels selon le protocole.
  - XMLHttpRequest
  - fetch() : promise
- Le HttpClient de Angular a été construit sur l'api XMLHttpRequest.
- Il se trouver dans le module HttpClientModule.
- HttpClient est injectable, il s'agit donc d'un service.

# Angular HttpClient Méthode Get

- La méthode get<T>(url , opts) : Observable<***HttpResponse***<T>>
  - Quand une DTO est définie pour les données que l'on est supposé recevoir, on peut l'utiliser à la place du générique T
  - On peut toujours utiliser ***any***
  - La méthode ne s'exécute que si on observe son retour.
- Le paramètre ***url*** contient l'url de destination.
- Les paramètres ***opts*** sont optionnels et peuvent contenir
  - ***observe:*** qui détermine la variable observée dans la réponse: défaut '***body***'
  - ***responseType***: de type de donnée retournée : défaut '***json***'
  - ***Headers:*** Entête de la requête http. Objet de type **HttpHeaders**

# Angular HttpClient Méthode Get

Paramètre par default de observe est body, pas besoin de ps.body dans ce cas

```
retrievePrograms() {
  this.http.get<Program[]>(this.url , { observe: 'response' })
   .subscribe(
                ps => this.programs = ps.body ,
                error => {
                            console.log('erreur de requête');
                             this.programs = [];
                          }
               );
}
```

# Angular HttpClient Méthode Post

- La méthode:  post<T>(*url* , *body*, *opts*): Observable<***HttpResponse***<T>>
  - Comme get, la méthode est lazy et ne s'exécute que si l'on observe son retour.
- Le paramètre ***body*** contient un objet JS qui sera envoyé par défaut en format JSON au serveur.
  Sinon, les ***opts*** headers peuvent toujours être changées pour envoyer d'autres types de données
- Les paramètres ***url*** et ***opts*** ont des rôles identiques à **ceux de la méthode *get***

# Angular HttpClient Méthode Post

```
saveInscription(inscData: any): Observable<number[]> {
  return this.http.post<number[]>(this.url, inscData, { /*no options*/})
    .pipe( retry(3),
          catchError((err, caught: Observable<number[]>) => {
                        console.log('erreur lors de la sauvegarde ');
                        this.ids = [];
                        return caught;
                        })
          );
}
```

- La bibliothèque RxJS fournit l'opérateur retry(N) permettant de venir à bout de certaines erreurs transitoires.

# Angular HttpClient Autres Méthodes

- Le  service HttpClient définit d'autres méthodes conformes au standard HTTP (et REST). Il s'agit entre autres des méthodes
  - ***put(url, body, option)*** remplace la données portant le même id.
  - ***delete(url, option)*** efface la données portant le id en url.
- Toutes les méthodes retournent des observables et sont lazy.
  - L'observation est obligatoire pour déclencher l'appel http.

# Angular HttpClient: Entêtes de requête

- Nous avons vu qu'avec les méthodes **Get** et **Post** (et les autres) qu'il existe un objet d'options (optionnel) pouvant contenir entre autre les headers.
- Certains API sécurisés exigent des headers bien identifiés qui seront dans un objet de type HttpHeaders.

```
const httpOptions = {
  headers: new HttpHeaders({
    'Content-Type':  'application/json',
    'Authorization': 'my-auth-token'
  })
};
```

# Angular HttpClient: INTERCEPTION (advicing)

- Les interceptions sont des concepts de la POA utilisés pour rajouter des informations, modifier les requêtes, exécuter un code comme la connexion avant un appel, ou traiter les retours des calls http avant que la méthode ayant initié l'appel ne reçoivent le retour.
- Globalement, ces fonctionnalités qui se rajouteraient avant et/ou après le code de chaque call Http sont des Aspects.
- HttpClient permet de les circonscrire en un endroit avec le pattern « chaine de responsabilité ».
- La technique est basée sur les intercepteurs (advices) semblables au Gin.
- Il faut pour cela implémenter HttpInterceptor injectable.

# Exemple d’intercepteur

```
import { AuthService } from '../auth.service’;
@Injectable()
export class AuthInterceptor implements HttpInterceptor {
        constructor(private auth: AuthService) {}

        intercept(req: HttpRequest<any>, next: HttpHandler) {
                const authToken = this.auth.getAuthorizationToken();
                const req2 = req.clone({ headers:
                        req.headers.set('Authorization', authToken) });
                return next.handle(req2);
                }
}
```

# Remarques:

- L’objet HttpRequest et ses paramètres sont immuables.
  - Raison: l'application peut réessayer plusieurs fois une requête avant qu'elle ne réussisse, ce qui signifie que la chaîne d'intercepteurs peut retraiter plusieurs fois la même requête.
  - Si un intercepteur pouvait modifier la requête d'origine, l'opération de 'réessayage’ recommencerait à partir de la requête modifiée plutôt que de l'originale.
  - L'immuailité garantit que les intercepteurs ont la même requête à chaque essai
  - On utilise alors ***clone*** pour mettre à jour les params de HttpRequest .
- Dans le cas de plusieurs intercepteurs, il est essentiel de prêter attention à l’ordre dans lequel ils sont injectés donc: 'providés'

# Exemples et Applications:

- Exemples 1: Forcer le https

```
const secureReq = req.clone({ url: req.url.replace('http://', 'https://')});
```

- Exemple 2: L'immutabilité ne prévient pas les modifications profondes:
  - En conséquence on peut directement modifier les champs du body par exemple.
    - Ce qui est quand même dangereux avec le retry.
  - Pour le prévenir:

```
const newBody = {username: req.body.username.trim()};
const newReq = req.clone({ body: newBody });
```

# Exemples et Applications:

- Exemple 3 : Headers par default

```
const authReq = req.clone({
  headers: req.headers.set('Authorization', authToken)
});
```

- Il est possible de faire du ***caching***, du ***dubouncing*** et autres techniques pour améliorer les performances de notre app et prévenir les requêtes indésirables.

## Application:

- Utiliser l'intercepteur pour rajouter l'APIKEY sur l'url  de l'application utilisant l'api météo.

# Prévention des attaques de type XSRF

- La falsification de requêtes intersites (XSRF ou CSRF) est une technique d'attaque par laquelle le hacker incite un utilisateur authentifié à exécuter, sans le savoir, des actions sur votre site Web.
- HttpClient supporte un mécanisme commun utilisé pour prévenir les attaques XSRF.
- Lors de l'exécution de requêtes HTTP, un intercepteur lit un token à partir d'un **cookie**, (par défaut XSRF-TOKEN), et l'insert dans l'entête de la requête HTTP, avec la clé: X-XSRF-TOKEN.
- Étant donné que seul le code qui s'exécute sur votre domaine peut lire le cookie, l'API en backend peut être certain que la demande HTTP provient de votre application cliente et non d'un tiers (Hacker).
- Les requêtes de type ***get*** et ***head*** ne sont pas concernées par cet intercepteur. (Elles ne sont pas supposée modifier des données chez le client.

# Prévention des attaques de type XSRF

```
HttpClientXsrfModule.withOptions({
  cookieName: 'My-Xsrf-Cookie',
  headerName: 'My-Xsrf-Header',
}),
```

- Le client n’est responsable que d’une partie de la protection.
- Pour en profiter, le serveur doit renvoyer le cookie My-Xsrf-Cookie contenant le token My-Xsrf-Header lors de l’ouverture de session.

# LES CORS: RAISONS et DÉFINITION

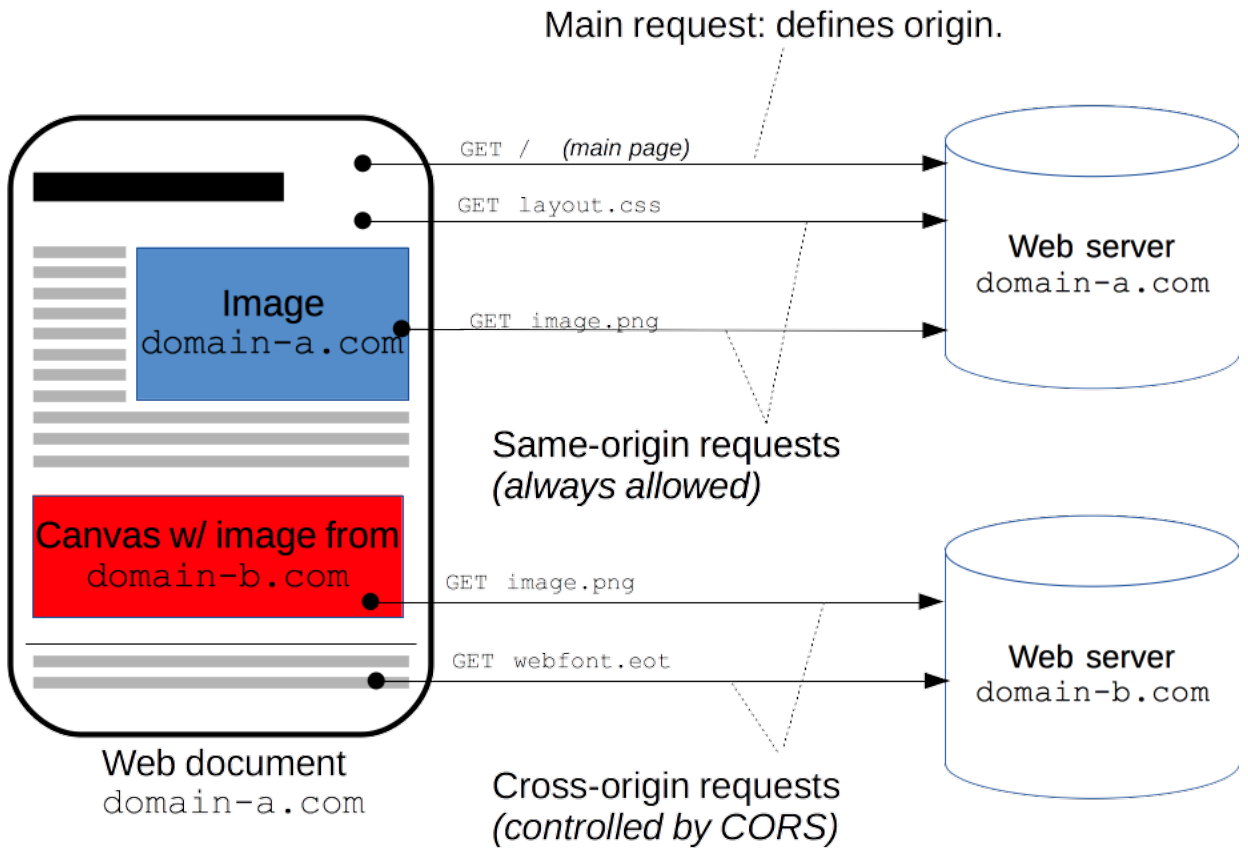
- Le Cross-Origin Resource Sharing (CORS) est un mécanisme utilisant des entêtes HTTP supplémentaires pour indiquer aux navigateurs d’autoriser/restreindre à client issu d’une origine X, l'accès à des ressources qui hébergé chez Y.
- Une application Web exécute une requête HTTP d'origine croisée lorsqu'elle demande une ressource qui a une origine (domaine, protocole ou port) différente de la sienne.
  - Exemple de Requête d'origine croisée:

  le code JavaScript client servi à partir de https://domain-a.com utilise XMLHttpRequest pour faire une demande pour l’origine https://domain-b.com/data.json.

# LES CORS: RAISONS et DÉFINITION

- Pour des raisons de sécurité, les navigateurs restreignent les requêtes HTTP d'origine croisée initiées à partir de scripts.
  - Par exemple, XMLHttpRequest et l'API Fetch suivent la même politique d'origine.
- Donc une app Web utilisant ces API ne peut demander que des ressources de la même origine.
- Origine à partir de laquelle, elle a été chargée, sauf si la réponse d’ origines croisées inclut les entêtes CORS appropriés.

# LES CORS: RAISONS et DÉFINITION



# CORS: FONCTIONNEMENT

- De nouveaux en-têtes HTTP permettent aux serveurs de **décrire les origines autorisées** que les navigateur Web doivent lire.
- De plus, pour les méthodes de requête HTTP qui peuvent modifier les données du serveur (en particulier, les méthodes autres que GET ou POST avec certains types MIME), la spécification impose aux navigateurs de "contrôler en amont" la requête, en sollicitant (appel) la liste des méthodes qui sont autorisées par le serveur.
- Les navigateurs font des pré-requêtes (perflight) avec la méthode OPTIONS (de HTTP).
- Puis après vérification, les navigateurs envoient la requête réelle.

# CORS: FONCTIONNEMENT

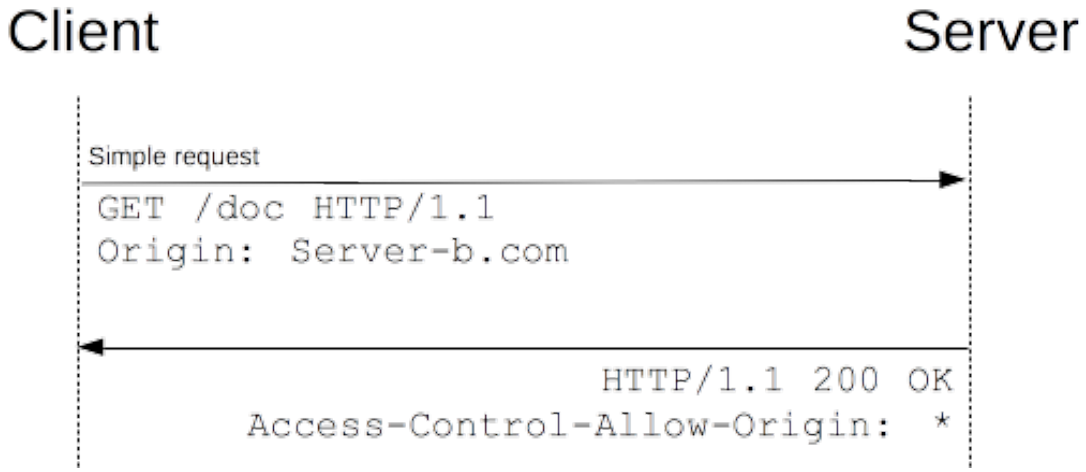
- Les serveurs peuvent également informer les clients si des "informations d'identification" doivent être envoyées avec les requêtes.
- Les échecs CORS entraînent des erreurs. Pour des raisons de sécurité, les détails de l'erreur ne sont pas disponibles pour client JavaScript.
  - Tout ce que le client sait, c'est qu'une erreur s'est produite.
- La seule façon de déterminer ce qui s'est spécifiquement passé est de regarder la console du navigateur pour plus de détails.

# CORS: FONCTIONNEMENT

- Certaines requêtes ne déclenchent pas de contrôle en amont (perflight) CORS.
- Il s’agit de requête remplissant toutes les conditions suivantes:
  - La requête qui a l'une des méthodes autorisées:  GET, HEAD, POST
  - N’ayant que des entêtes (hormis ceux insérés automatiquement par l'agent utilisateur, ceux que la spécification Fetch) :
    - Accept,  Accept-Language,  Content-Language,
    - Et Content-Type: (application/x-www-form-urlencoded, multipart/form-data, text/plain)
- D’autres conditions sont aussi à vérifier: pas de callback, pas de stream. (socket ouverte)

# CORS: FONCTIONNEMENT: SANS PERFLIGHT

```
const xhr = new XMLHttpRequest();
const url = 'https://bar.other/resources/public-data/';
xhr.open('GET', url);
xhr.onreadystatechange = someHandler;
xhr.send();
```

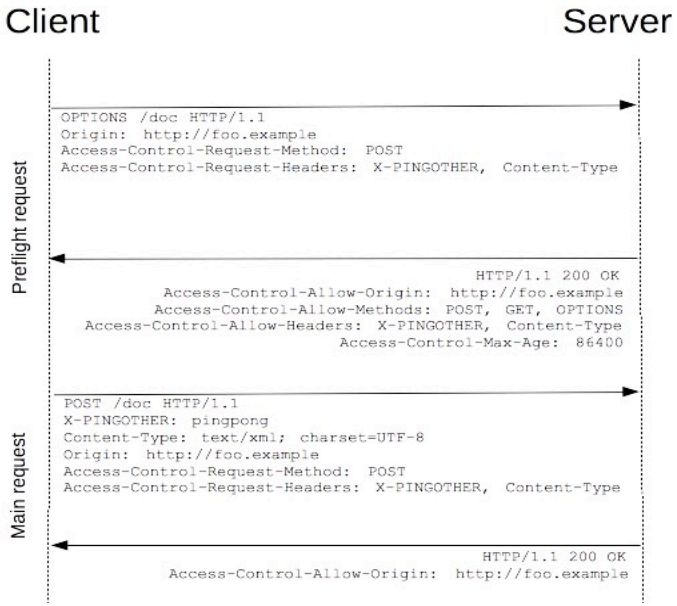



**GET** /resources/public-data/ HTTP/1.1
Host: bar.other
User-Agent: XXX … Gecko/20100101 Firefox/71.0
**Accept**: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
**Accept-Language**: en-us,en;q=0.5
**Accept-Encoding**: gzip,deflate
**Connection**: keep-alive
Origin: **https://foo.example**

HTTP/1.1 200 OK Date: Mon, 01 Dec 2020 00:23:53 GMT
Server: Apache/2
Access-Control-Allow-Origin: *
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive Transfer-Encoding: chunked
Content-Type: application/xml

# CORS: FONCTIONNEMENT: AVEC PERFLIGHT

```
const xhr = new XMLHttpRequest();
xhr.open('POST', 'https://bar.other/resources/post-here/');
xhr.setRequestHeader('X-PINGOTHER', 'pingpong');
xhr.setRequestHeader('Content-Type', 'application/xml');
xhr.onreadystatechange = handler;
xhr.send('<person><name>Arun</name></person>');
```




**OPTIONS** /resources/post-here/ HTTP/1.1
Host: bar.other
User-Agent: XXX Gecko/20100101 Firefox/71.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Connection: keep-alive
Origin: http://foo.example
Access-Control-Request-Method:POST
Access-Control-Request-Headers: X-PINGOTHER, Content-Type

HTTP/1.1 204 No Content
Date: Mon, 01 Dec 2008 01:15:39 GMT
Server: Apache/2
Access-Control-Allow-Origin: https://foo.example
Access-Control-Allow-Methods: POST, GET, OPTIONS
Access-Control-Allow-Headers: X-PINGOTHER, Content-Type
Access-Control-Max-Age: 86400
Vary: Accept-Encoding, Origin
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive

# CORS: PARAMÈTRES DU HEADER SERVER

- Access-Control-Allow-Origin: <origin> | *
  - Spécifie soit une origine fixe <origin> ou accepter toute origine *
  - Exemple:  seules les requêtes venant de https://blabla.org

    Access-Control-Allow-Origin: https://blabla.org

    Vary: Origin  // permet de spécifier que la réponse dépend de l'origine
- Access-Control-Expose-Headers: <header-name>[, <header-name>]*
  - Spécifie les entêtes et leurs valeurs autorisées.
  - Exemple: autoriser X-My-Custom-Header et X-Another-Custom-Header
    - Access-Control-Expose-Headers: X-My-Custom-Header, X-Another-Custom-Header
- Access-Control-Allow-Methods: <method>[, <method>]*
- Access-Control-Allow-Headers: <header-name>[, <header-name>]*
  - Utilisé pour le perflight (pour répondre à OPTION)

# Applications

- Créer une API PHP avec WAMP.
  - Avec des routes save (post) et retrieve (get)
  - Ces  routes reçoivent et retournent des données mocks.
- Exécuter l'API sur une machine différente.
  - Utiliser votre client Angular (Lab précédent) pour lire et envoyer des données.
  - Avez-vous constater un CORS ?
  - Autorisez les requêtes.