TD - *Same Origin Policy* et *CORS*

> Ce TD est complètement et *ouvertement* inspiré de l'excellente présentation de la SOP par Kirk Jackson (2017)

Introduction

La *Same Origin Policy* (SOP) est un ensemble de règles plus ou moins bien défini lorsque Javascript a été introduit en 1995 dans les navigateurs web. Nous allons voir ce qu'est une origine, comment la SOP fonctionne et pourquoi elle est fondamentale du point de vue de la sécurité côté client, et protège contre de nombreuses attaques sur le web.

Pour cette démonstration, nous simulons deux origines différentes, deux sites web, `site A` et `site B`. Lire le manuel pour savoir comment utiliser la démo.

Qu'est-ce que l'*origine* ?

Origine = scheme, domain, port

Exercice: same *origin* ?

- http://goodsite.com/a et http://goodsite.com/b ? Yes
- http://goodsite.com et https://goodsite.com ? No (diff scheme)
- http://goodsite.com et http://www.goodsite.com ? No (diff domain)
- http://goodsite.com et http://goodsite.com:80 ? No (dif port)
- http://goodsite.com/user et http://goodsite.com/admin ? Yes
- http://goodsite.com et http://api.goodsite.com ? No (diff domain!)

  C'est parce que le port fait partie de l'origine que notre démo locale fonctionne pour tester la SOP !

La SOP est là pour répondre à ces questions:

- Est-ce que le JS d'une origine peut accéder au DOM (l'état de l'application: html, js, cs, cookie, local Storage, session Storage) d'une *autre* origine (autre site) ?
- Est ce qu'un agent peut, à partir d'un site, accéder aux données d'un autre site et les manipuler ?

Implémentée par les navigateurs web. Le navigateur l'applique, le serveur l'influence. C'est au navigateur de faire ce job, car c'est lui qui gère la *navigation*, c.-à-d., les états applicatifs de différents sites en même temps du côté de l'agent.

Communications entre les fenêtres d'un navigateur (onglets) ?

Impossible en JS, sauf si on ouvre une fenêtre nous-mêmes en JS, avec `window.open`. On garde alors une référence vers elle.

```
//Ouvrir un nouvel onglet depuis `site A`
var newwin = window.open('http://localhost:5001', 'right')
//Acceder au body
newwin.document.body
//Changer le body
newwin.document.body.innerHTML="Hello!"
```

On garde une référence vers cette nouvelle fenêtre. On peut accéder au DOM et modifier le contenu de la page. Pour l'instant, on a ouvert `site A` depuis `site A`. Mais est ce que ca marcherait si on ouvrait `site B` de la même manière ?

```
//Ouvrir un nouvel onglet depuis `site A` vers siteB
var newwin = window.open('http://localhost:5002', 'right')
//Acceder au body
newwin.document.body
Uncaught DOMException: Permission denied to access property "document" on
        cross-origin object
//Changer le body
newwin.document.body.innerHTML="Hello!"
//On récupere une erreur
Uncaught DOMException: Permission denied to access property "document" on
        cross-origin object
```

On peut ouvrir une fenêtre et garder une ref vers elle, mais on ne peut plus accéder au document ! On ne peut donc pas manipuler l'état de l'application (page sur laquelle on est) depuis une autre application (sécurité)

`site A` ne peut pas accéder à `site B`, restrictif. C'est la SOP: `siteA` est seulement autorisé a accéder aux pages issues de `site A` (même origine)

Chaque onglet est une fenêtre (*window*) et **chaque *window* a son environnement: thread JS et DOM dédié et *isolé***.

Donc si vous êtes sur la page de votre banque, aucun autre site ouvert dans un autre onglet ne peut y accéder, tant mieux ! Mais, si deux onglets sont sur le même site (même origine), chaque onglet peut accéder au contenu de l'autre. (Par exemple le *Local Storage*)

La même, mais avec `frames` et `iframes`

C’est pareil pour les iframe: isolés, dispose de son propre environnement (JS THread et dom). L’iframe ne peut pas accéder au DOM de son site parent (celui qui inclut la frame), car différente origin.

Pourquoi la *SOP* est-elle importante ?

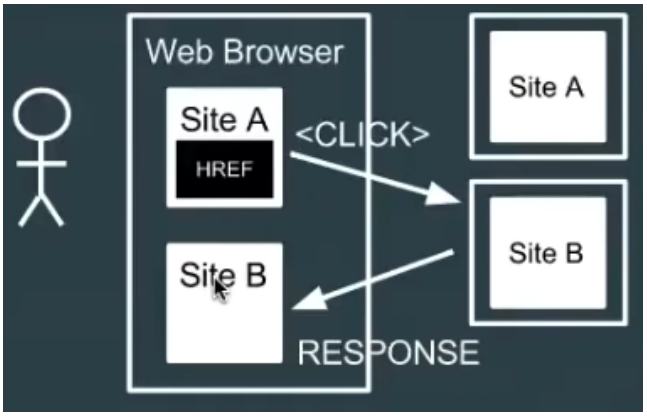
- **protège l’état d’une application des autres états d’application dans le même navigateur**. Il protège une page web d’une autre (chaque état est bien isolé *par origine* *)
- empêche données d’un site d’être lues par un autre site
- vous permet de naviguer en sécurité (si votre navigateur l’implémente bien)

  *Sauf pour les cookies qui fonctionnent un peut différemment.

Comment la SOP s’applique à différentes technos du web ?

Pour la suite on a `site A` qui essaie d’accéder au `site B`.

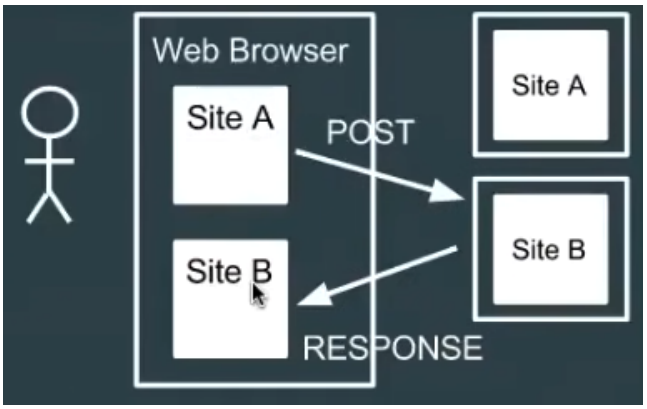
Anchors (links) balise `<a>`



Les sites web peuvent créer des hyperliens vers d’autres sites.

Ex: Depuis `site A` `site B`. La réponse s’ouvre dans un nouveau contexte. N’importe quel site peut lire, mais **ne peut pas lire la réponse**.

Formulaires (balise `<form>`)



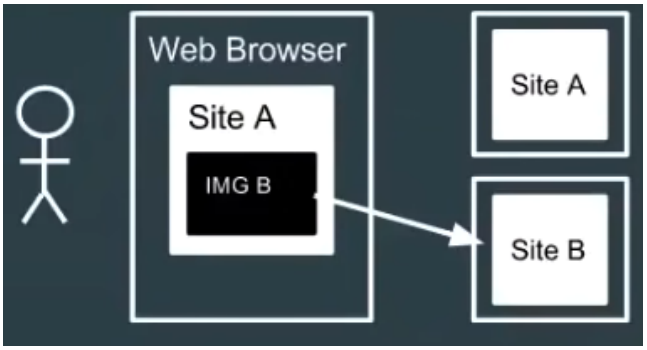
Un formulaire peut `POST` vers n'importe quel site.

> Ex: Depuis `site A` `<form action="localhost:5002" method="POST"></form>`.
> La réponse s'ouvre dans un nouveau contexte. N'importe quel site peut POST vers un autre site, mais **ne peut pas lire la réponse**.

Démo `form`

- Se rendre à l'url `http://localhost:5001/post.html` (`site A`)
- Montrer l'HTML
- poster vers `site B`: on voit que siteA est remplacé par `site B`
- `site B` a recu les données postées, mais `site A` est remplacé et ne peut pas lire la réponse de `site B`
- Regarder dans devtools la requête. On voit la requête POST vers `site B`, et la réponse est

Images (balise `img`)



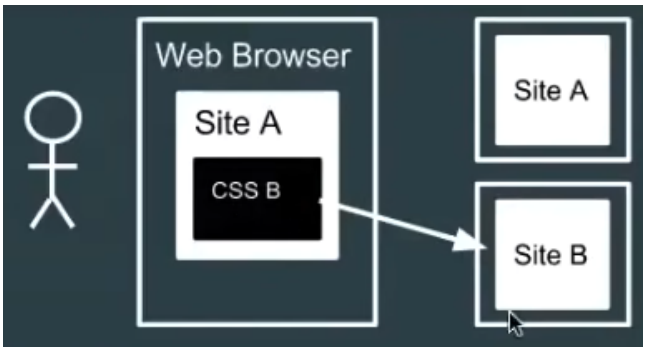
> Une balise a effectue automatiquement une requête GET de l'url indiquée dans `href`.

N'importe quel site peut afficher les images d'un autre site.

Ex: Depuis `site A`, balise a vers l'image de `site B` `<img src="http://localhost:5002/LetterB.svg/>`

Cependant, **les sites ne peuvent pas lire les données de l'image d'autres sites**. S'applique aussi aux autres médias statiques (audio, vidéo).

Feuilles de style CSS



N'importe quel site peut inclure CSS d'un autre site.

Ex: Depuis `site A`, `<link rel="stylesheet" type="text/css" href="http://localhost:5002/site-b.css">`

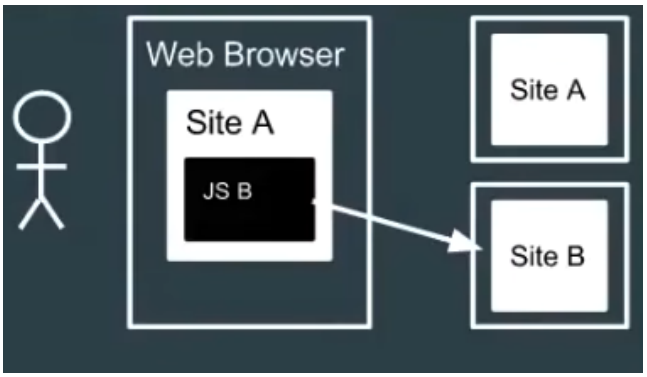
Cependant, **les sites ne peuvent pas lire les données du CSS d'autres sites**. Cad le CSS s'applique, mais vous ne pouvez pas inspecter le CSS.

Démo CSS

- Ouvrir `site A`
- En JS, accéder aux feuilles de style: `document.styleSheets[0]`, voir sous la clef `rules`
- Changer le HTML pour charger le style de `site B` (`href="http://localhost:5002/site-b.css"`)
- En JS, `document.styleSheets[0]`, plus possible de voir les `rules=null`

  C'est fait ainsi si la feuille de style contient des infos sensibles (clés API par exemple)

JS (balise `script`)



N’importe quel site peut inclure du JS d’un autre site. (Comme ça que fonctionne le tracking)

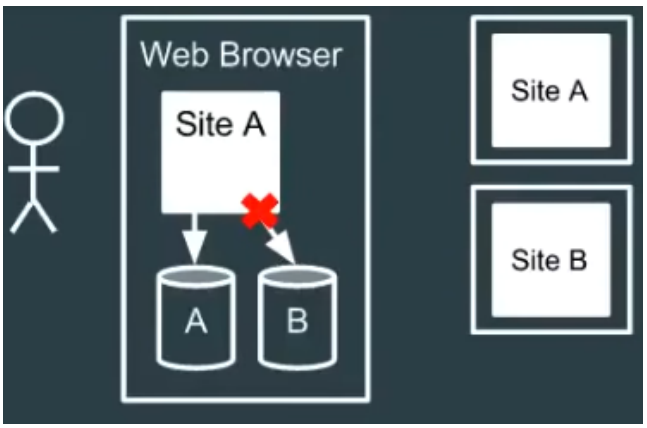
Ex: Depuis `site A`, `<script src="http://localhost:5002/site-b.js"/>`

Ce script est exécuté dans le contexte du site d’origine (`site A`) ! Porte ouverte aux attaques XSS (*Cross Site Scripting*). Le code s’exécutera dans le contexte de `site A` et fera *ce qu’il veut* (manipuler le DOM, envoyer des requêtes, déposer des cookies).

Démo JS

- Sur `site A`, remplacer le script par celui de `site B` `<script src="http://localhost:5002/site-b.js"></script>`
- Le DOM a bien été manipulé par le JS et affiche `B`
- Le cookie déposé par le JS de `site B` est bien déposé

Web Storage



La SOP s’applique aux *web storage*. Ces storage permettent à du JS de stocker des données dans le navigateur. Petite base de données cle/valeur. Il y’en a 2: local storage et session storage.

Local Storage

- partagé entre les fenêtres (applications) **avec la même origine**
- survit quand la fenêtre est fermée

Session Storage

- non partagé, unique à la fenêtre en cours
- lifetime de la window (quand elle est fermée, donnée supprimée)

Démo Local Storage

- Se rendre sur `site A`
- Dans les *devtools* de votre navigateur, onglet `Application`, inspecter `Local Storage` et `Session storage`
- Entrée une clef/valeur à la main `foo bar` ou avec JS `window.localStorage['foo']="bar"`, puis inspecter le local storage
- Acceder au même storage depuis une autre fenêtre: ouvrir `site A` dans une autre fenêtre. Inspecter le local Storage: on retrouve nos clefs/valeurs. Il est bien partagé entre fenêtres (entre *états applicatifs* comme on dirait de manière REST).
- Mais si je vais sur `site B` et regarde le local storage, je vois bien que je ne peux pas accéder à celui de `site A` : `window.localStorage['foo']` retourne `undefined`

Cookies

Les cookies se comportent de manière différente du reste. *C'est l'enfant pauvre de la sécurité sur le web*. Inventé par Netscape en 1994, au départ un outil en interne. Pas un standard, **crée avant la SOP**.

> L'introduction des cookies n'a pas été largement connue du grand public pour autant. En particulier, les cookies étaient acceptés par défaut dans les paramètres des navigateurs, et les utilisateurs n'étaient pas informés de leur présence. Certaines personnes étaient au courant de l'existence des cookies au début de l'année 1995, mais le grand public n'apprit leur existence qu'après la publication par le Financial Times d'un article le 12 février 1996.

Fonctionnement des cookies (rappel):

- L'utilisateur demande a accéder au `site A`
- `site A` (serveur) répond et le navigateur stocke un cookie lors de la réception requête HTTP dans une *cookie jar* de `site A`. Le HTML arrive dans le web browser sans le cookie, il est rendu.
- L'utilisateur demande une nouvelle ressource au serveur, la requête HTTP emporte avec elle tous les cookies déposés et les envoie au serveur, puis le serveur répond.
- Si j'ouvre une nouvelle fenêtre vers `site B`, il n'a pas acces à la *cookie jar* de `site A`.

Que se passe-t-il si `site B` propose un formulaire `form` vers `site A` `<form action="http://localhost:5001" method="POST"></form>` ? **Est-ce que la requête `POST`**

embarque avec elle les cookies de `site A` ? Oui, c'est la base des attaques CSRF (*Cross Site Request Forgery*)

La sécurité des cookies est un peu différente de la SOP. Quand un cookie est set (déposé) il a **un nom, un domaine et un path**, différent de la SOP (scheme, domain, port). Ce qui fait qu'un cookie déposé par un domaine est accessible à tous les sous-domaines ! Par ex, `monsite.com` dépose un cookie, accessible à `api.monsite.com` ou `blog.monsite.com`. Un cookie est donc restreint à *une famille de sites* (et non à une origine).

Donc un cookie posé par un domaine **est visible et utilisable par tous les sous-domaines**.

> Par exemple, quand vous êtes loggé sur Google (via des cookies), vous êtes *de facto* logé sur les autres applis Google (sous-domaine de Google `google.com`). En pratique c'est plus complexe, car il y a en place tout un système de redirections.

Third-party cookies

Si vous incluez une ressource externe, comme du JS (on l'a vu on peut inclure du JS d'un autre site). Ce JS peut donc set un cookie et manipuler votre DOM comme il le souhaite. Le cookie installé par des ressources externes (provenant d'une autre origine) est appelé un *third-party cookie*. Ils servent généralement à tracker l'utilisateur à des fin de collectes de données et de marketting publicitaire.

AJAX, `XMLHttpRequest`

> `XMLHttpRequest` crée en 2006. Fetch est la nouvelle Web API mis a disposition de JS vers les threads de requête asynchrone, version plus moderne de `XMLHttpRequest` (utilise notamment les promesses), mais ca reste le même principe.

Avec ce construct, vous pouvez GET ou POST n'importe quel site depuis JS (un domaine). `AJAX` c'est un client qui s'execute dans le navigateur. Le navigateur va auto ajouter les cookies ou auth aux requêtes envoyées (conformément à leurs règles de comportement).

```
// Depuis `site B`, requete vers `site A`
var xhr = new XMLHttpRequest();
xhr.open('GET','http://localhost:5001', false);
xhr.send();
```

Seulement des requêtes vers le même site d'origine peuvent lire la réponse. Le navigateur web vous autorise a envoyer n'importe quelle donnée (requête) vers n'importe quel site, comme on l'a vu avec le form et POST, mais ne vous laissera pas lire la réponse.

Démo `XMLHttpRequest`

Depuis `site A`, j'envoie une requête `GET` vers `site A` avec `XMLHttpRequest`

```
var xhr = new XMLHttpRequest();
xhr.open('GET','http://localhost:5001', false);
xhr.send();
//Lire réponse: on voit le HTML
xhr.responseText
```

Si j'inspecte la réponse j'ai bien le HTML, le network affiche ma réponse HTTP avec le code status `200`. Tout va bien. Les requêtes depuis la même origine ont le droit de lecture.

Maintenant la même en requêtant `site B` depuis `site A`

```
xhr.open('GET','http://localhost:5002', false);
xhr.send();
xhr.responseText
```

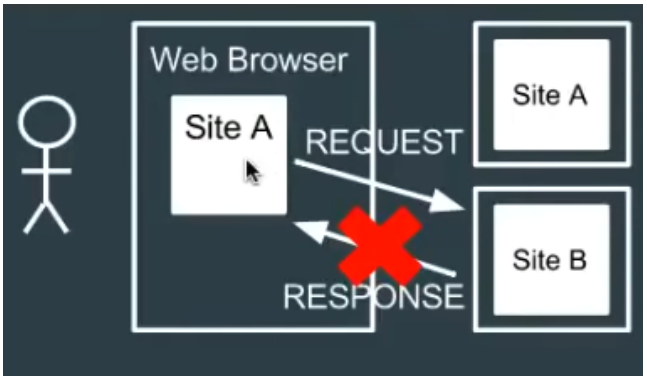
Je reçois une erreur

```
Blocage d'une requête multiorigines (Cross-Origin Request) : la politique « Same Origin » ne permet pas de consulter la ressource distante située sur http://localhost:5002/. Raison : l'en-tête CORS « Access-Control-Allow-Origin » est manquant. Code d'état : 200.
```

C'est bien ce qu'on veut. `site A` ne peut pas accéder aux données de `site B` (la *SOP* à l'œuvre). On parlera de l'en-tête `CORS « Access-Control-Allow-Origin »` dans un instant;

> Si on regarde dans le Network, **les requêtes sont bien envoyées et fonctionnent**. J'*ai* et je *peux* inspecter la réponse dans le `devtools` (page HTML de `site B`), mais le navigateur n'autorise pas le JS à lire la réponse.

CORS



On l'a vu avec la démo `XMLHttpRequest`, **les requêtes issues d'une origine (d'un site) ne peuvent pas *lire* les réponses d'un autre site**. Rappelez-vous l'erreur `(Cross-Origin Request) : la politique « Same Origin » ne permet pas de consulter la ressource distante située sur http://localhost:5002/. Raison : l'en-tête CORS « Access-Control-Allow-Origin » est manquant.`

Pour autoriser de telles requêtes (par exemple une API), le serveur doit l'autoriser avec un header HTTP, et ainsi diminuer la SOP (l'affaiblir) afin de permettre a d'autres sites (par exemple une appli front end sur un autre sous-domaine ou carrément n'importe quel site). C'est au site de décider est ce que mes ressources sont suffisamment sécurisées pour autoriser des agents d'autres sites à les lire et les manipuler ?

Donc en pratique, le client fait une requête HTTP standard (via `XMLHttpRequest` par exemple), et le serveur voie une nouvelle requête arriver. Il a sa politique de réponse dans les header, et le header `Access-Control-Allow-Origin: http://localhost:5001` pour dire au navigateur "autorise le `site A` a lire la réponse que je renvoie".

> Attention, la CORS n'est appliqué que côté client ! Comme on l'a vu, la requête fonctionne même sans CORS, la réponse est renvoyée c'est **seulement le navigateur** qui empêche le client (et le JS) de lire la réponse et de la manipuler. Mais les données sont transmises. Donc ce n'est pas un mécanisme suffisant pour protégérer les ressources de votre serveur.

Démo `CORS`: d'un site à l'autre

CORS permet de moduler la SOP côté client, dans le navigateur (autoriser à lire la réponse d'une requête issue d'un autre site).

Décommenter la ligne suivante dans `site-a/index.php`

```
header("Access-Control-Allow-Origin: http://localhost:5002");
```

Elle fixe un header qui autorise la lecture de la réponse HTTP pour une requête issue de l'origine `http://localhost:5002`.

Démo CORS: d'un serveur à l'autre

Un serveur *n'a pas d'origine* (pas ce concept, uniquement dans le *navigateur*). On peut donc requêter n'importe quel serveur depuis un script client (par exemple ave `curl`) ou un autre serveur. Le serveur ne sait pas quand il reçoit une requête si elle vient d'un navigateur web ou d'un client HTTP quelconque.

```
curl http://localhost:5002
```

va bien retourner la représentation de la ressource (page HTML) même si le header CORS n'autorise que `site A` a lire ses données. **Il n'y a pas de concept d'*origine* en dehors du navigateur**.

Limitations

- **la SOP ne s'applique qu'aux navigateurs**, les autres agents peuvent accéder à vos ressources **sans restrictions**

- Pour protéger votre API, il faudra donc **compter sur d'autres mesures** en fonction de vos besoins et des ressources exposées:
  - Authentification
  - Autorisations (JWT)
  - Rate Limitation
  - Captchas
  - White list
  - etc.

Conclusions

- Nous avons appris le concept d'*origine*
- Nous avons appris ce qu'était la *Same Origin Policy*: une politique de sécurité pour naviguer sur le web en sécurité dans le navigateur. Elle est appliquée par le navigateur, modulée par le serveur.
- Pourquoi la SOP est importante
- Comment elle s'applique en pratique: hypermedia (`a`, `img`, `form`, etc.), JS, AJAX (`XLMHttpRequest` ou `Fetch`), etc.
- Comment la contourner pour partager des données entre sites. Le serveur peut la moduler avec sa politique CORS et l'en-tête `Access-Control-Allow-Origin`

Références

- MDN: Same Origin Policy (SOP), les contraintes *cross-origin* implémentées côté *client*
- MDN: Cross Origin Resource Sharing (CORS), les contraintes *cross-origin* implémentées côté *serveur*
- Same-origin policy: The core of web security @ OWASP Wellington, excellente présentation de Kirk Jackson de la Same Origin Policy avec démonstrations à l'appui. **À regarder**.