

Cours de Sécurité Web ZZ2F5

HTTP

Le protocole HTTP

- ▶ les acteurs et la couche application
- ▶ historique : HTTP/1.1, HTTP/2, HTTP/3 (QUIC)
- ▶ URI et URLs, URLs trompeuses
- ▶ requêtes : verbes, paramètres, idempotence

Réponses, REST et état

- ▶ réponses et codes HTTP (1xx – 5xx)
- ▶ REST : la sémantique HTTP (maturité de Richardson)
- ▶ HTTP est *stateless*
- ▶ cookies, sessions, *local storage*, tracking

HTTP ne transporte **que** des requêtes et des réponses : l'**état** (cookies, sessions) est construit côté applicatif et constitue une **surface d'attaque**.

Le Web : concepts et distinctions

Internet vs. Web

Internet est l'infrastructure mondiale constituée de réseaux interconnectés :

- ▶ câbles, routeurs, serveurs
- ▶ protocoles permettant la communication

Le Web est un service qui fonctionne sur Internet, au même titre que la messagerie électronique, FTP, VoIP, etc.

Site Web

Un ensemble de **pages Web** liées :

- ▶ par un même nom de domaine
- ▶ hébergées sur un serveur Web

Page Web

Un document unique, généralement écrit en **HTML**, qui peut contenir :

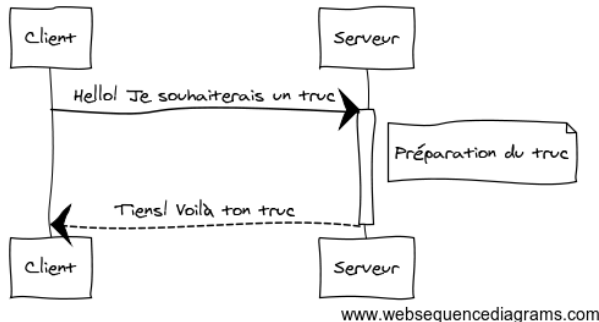
- ▶ du texte et des images
- ▶ des vidéos et autres médias
- ▶ des hyperliens vers d'autres ressources

Navigation

Le Web est relié par des **hyperliens** :

Page Web $\xrightarrow{\text{hyperlien}}$ Page Web

Les utilisateurs naviguent entre les ressources, formant ainsi un vaste **réseau décentralisé d'informations**.



Le **client** (navigateur, ...) émet une **requête** ; le **serveur** répond par une **réponse** HTTP.

Le serveur

est typiquement un serveur web

La ressource

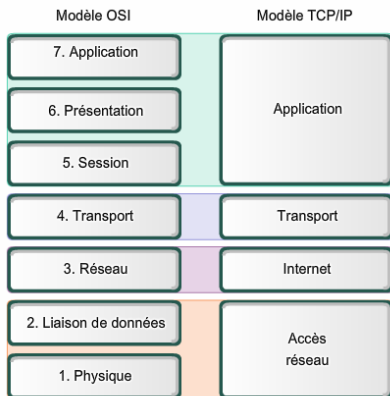
est typiquement une ressource associée à une URI

Le client

est typiquement

- ▶ un navigateur web
- ▶ une web app
- ▶ un objet connecté
- ▶ n'importe quoi qui consomme de l'API...

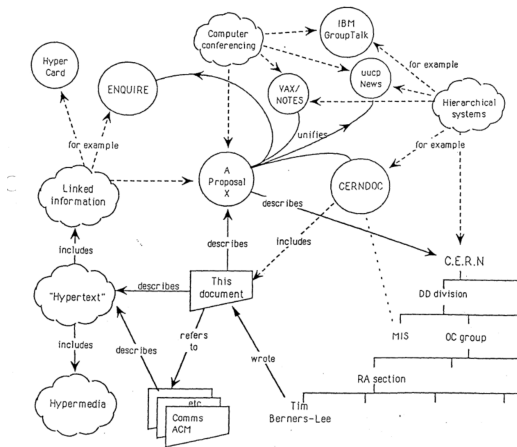
HTTP : un protocole de la couche application



Définition (RFC 2616)

"HTTP is an application-level protocol for distributed, collaborative, hypermedia information systems."

Inventé par Tim Berners-Lee en 1989.





HTTP : historique et versions

Inventé par Tim Berners-Lee en 1989.

HTTP/1.1 (RFC 2616) jusqu'à 1999.

HTTP/2 (RFC 7540) basé sur SPDY de Google.

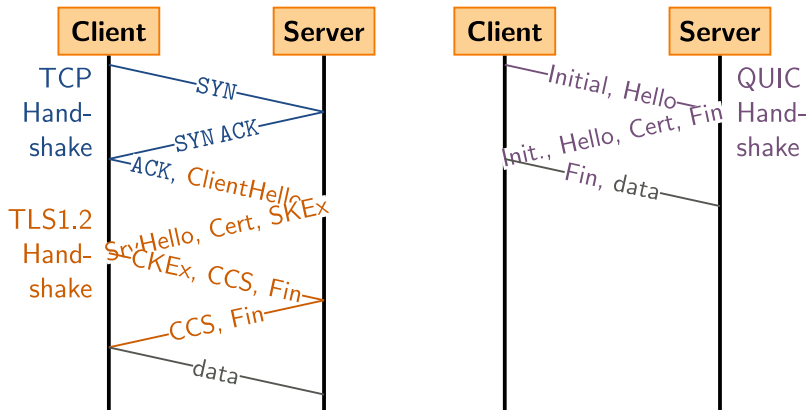
- ▶ utilisation efficace des ressources réseau
- ▶ réduction de la latence perçue (compression et échanges multiples simultanés)
- ▶ envoi spontané de données des serveurs vers les clients

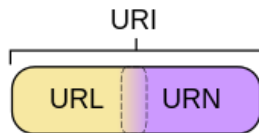
HTTP/3 (RFC 9114) basé sur QUIC.

Qu'est-ce que HTTP/3 – Informations sur le nouveau protocole UDP rapide ?

- ▶ multiplexage des flux
- ▶ connexion à faible latence

Comparaison entre HTTP/2+TLS et HTTP/3





URN : Uniform Resource Name

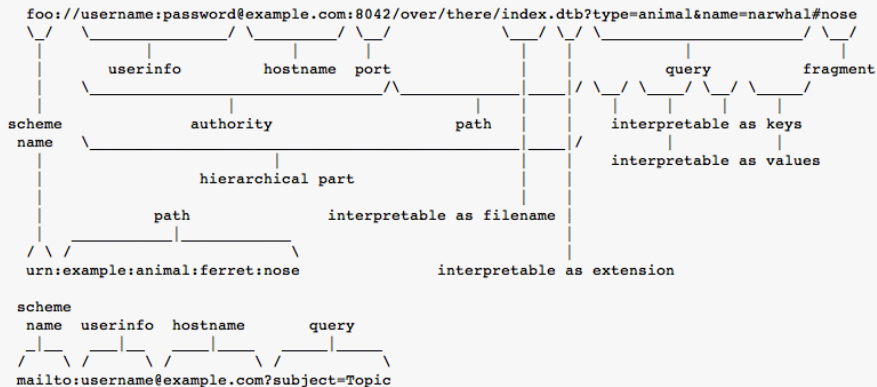
ex. : le nom d'une personne

URL : Uniform Resource Locator

ex. : l'adresse d'une personne

URI : Uniform Resource Identifier

des URLs, des URNs, ou les deux



Les mots de passe peuvent transiter **en clair** via les schémas d'URL

Des identifiants dans le champ userinfo

scheme://[userinfo@]host[:port]/path?query#fragment

exemple : `http://www.visa.com:1337@33.32.323.22:8080/cart?add=1345`

Des identifiants de session dans l'URL

userinfo		hôte (IP non valide)
↓		↓
<code>http://visa.com:UserSession=56858&useroption=879 @ 42.216.64.464</code>		

L'hôte est ce qui suit le dernier @

`http://www.visa.com@33.32.323.22` — serveur invalide : 33.32.323.22

Des domaines qui se ressemblent

`https://www.v-i-s-a.com` $\neq$ `https://www.visa.com`

```
GET / HTTP/2
Host: perdu.com
Accept-Language: en-US,en;q=0.9
User-Agent: Mozilla/5.0
Sec-Fetch-Site: none
Sec-Fetch-Mode: navigate
Sec-Fetch-Dest: document
Accept-Encoding: gzip, deflate, br
Priority: u=0, i
Connection: keep-alive
```



Requête HTTP

- ▶ **un verbe HTTP** décrivant l'action
- ▶ **une URI** (Uniform Resource Identifier)
- ▶ **des en-têtes / headers** :
 - ▶ contraintes sur le contenu demandé
 - ▶ informations contextuelles
 - ▶ sous la forme clé: valeur
- ▶ une ligne vide
- ▶ un corps servant à éventuellement envoyer des données



GET :

récupérer une ressource ou une collection de ressources

POST :

créer une nouvelle ressource

PUT :

mettre à jour une ressource existante ou créer une nouvelle ressource à une URI donnée

DELETE :

supprimer une ressource donnée

PATCH :

mettre à jour partiellement une ressource existante

	GET	POST
Visibilité	Visible pour l'utilisateur dans le champ d'adresse	Invisible pour l'utilisateur
Marque-page et historique de navigation	Les paramètres de l'URL sont stockés en même temps que l'URL.	L'URL est enregistrée sans paramètres URL.
Cache et fichier log du serveur	Les paramètres de l'URL sont stockés sans chiffrement.	Les paramètres de l'URL ne sont pas enregistrés automatiquement.
Comportement lors de l'actualisation du navigateur / Bouton « précédent »	Les paramètres de l'URL ne sont pas envoyés à nouveau.	Le navigateur avertit que les données du formulaire doivent être renvoyées.
Type de données	Caractères ASCII uniquement.	Caractères ASCII mais également données binaires.
Longueur des données	Limitée – longueur maximale de l'URL à 2 048 caractères.	Illimitée.

L'envoi de formulaires

“Seuls GET et POST sont implémentés dans les navigateurs”

- Les formulaires HTML natifs ne permettent directement que GET et POST comme méthodes HTTP.

Les autres sont utilisés dans un contexte d'exécution, par exemple via javascript, XML...
cf. REST, plus loin

La liste n'est pas exhaustive

- ▶ **HEAD** : comme GET mais sans le corps (test rapide, *headers* seulement)
- ▶ **OPTIONS** : liste des méthodes autorisées sur la ressource
- ▶ **TRACE** : renvoie la requête telle qu'elle a été reçue (généralement désactivé)
- ▶ **CONNECT** : établit un tunnel (ex. proxy HTTPS)

Idempotence

- ▶ **GET, HEAD, PUT, DELETE, OPTIONS** : *idempotents* (répéter ne change pas le résultat)
- ▶ **POST** ne l'est **pas** : un *replay* peut créer des doublons
- ▶ **PATCH** : idempotence *non garantie* par le protocole

Deux emplacements possibles

- ▶ la **query string** (dans l'URL)
- ▶ le **corps** de la requête

Précisions

- ▶ la query string est **visible** : *logs* serveur, historique, header Referer $\Rightarrow$ n'y mettre jamais de secrets
- ▶ le corps n'est pas plus sûr qu'une query string *sans* **TLS**

Une réponse HTTP est composée

- ▶ d'une **ligne de statut** : le code HTTP et son texte associé (HTTP/1.1 200 OK, HTTP/1.1 304 Not Modified)
- ▶ de quelques **en-têtes / headers** :
 - ▶ informations sur le contenu servi
 - ▶ sous la forme clé: valeur
- ▶ d'un **corps** : le contenu renvoyé (éventuellement vide)

Exemple : une réponse HTTP

```
HTTP/1.1 304 Not Modified
Date:    Fri, 20 Feb 2015 15:15:01 GMT
Server:  Apache
Connection:  Keep-Alive
Keep-Alive:  timeout=2, max=100
Etag:     "cc-480d5dd98a340"
Vary:     Accept-Encoding
```

```
<html>
  <head><title>Vous Etes Perdu ?</title></head>
  <body>
    <h1>Perdu sur l&#x27;Internet ?</h1>
    <h2>Pas de panique, on va vous aider</h2>
    <strong>* <----- vous êtes ici </strong>
  </body>
</html>
```

<httpstatus.es> : la liste détaillée des codes ; en voici les plus courants.

- ▶ **1xx** – Informational : la requête est en cours (100 Continue)
- ▶ **2xx** – Successful :
 - ▶ 200 OK
 - ▶ 201 Created – typique après un **POST**
 - ▶ 204 No Content – corps vide (ex. après un **DELETE**)
 - ▶ 206 Partial Content – téléchargement partiel / reprise
- ▶ **3xx** – Redirections :
 - ▶ 301 Moved Permanently – l'URI a changé (*définitif*)
 - ▶ 302 Found – redirection *temporaire*, souvent transformée en GET
 - ▶ 304 Not Modified – cache : contenu inchangé, **corps vide**
 - ▶ 308 Permanent Redirect – comme 301, mais méthode et corps *préservés*

Nota bene

un 304 doit avoir un corps **vide** : l'exemple de réponse *perdu.com* (corps affiché) est en réalité non conforme à la RFC.

Les codes d'erreur : 4xx – Client Error

le client a envoyé une requête que le serveur ne peut pas satisfaire.

400	Bad Request	409	Conflict
401	Unauthorized	410	Gone
403	Forbidden	415	Unsupported Media Type
404	Not Found	418	I'm a Teapot (clin d'œil, RFC 2324)
405	Method Not Allowed	429	Too Many Requests
406	Not Acceptable	451	Unavailable for Legal Reasons
408	Request Timeout		

401 vs 403 : la nuance

401 : pas (ou plus) **authentifié** ; le header WWW-Authenticate indique comment s'authentifier

403 : **authentifié**, mais accès *interdit* (droits insuffisants)

le serveur a échoué à traiter une requête *pourtant* valide.

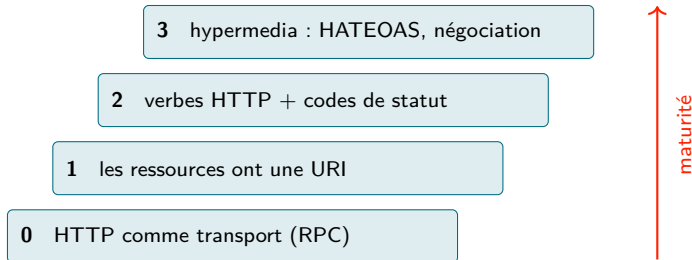
- ▶ 500 Internal Server Error – erreur interne, à détecter mais **jamais** à détailler dans la réponse
- ▶ 501 Not Implemented
- ▶ 502 Bad Gateway – passerelle / reverse proxy défaillant
- ▶ 503 Service Unavailable – surcharge, maintenance
- ▶ 504 Gateway Timeout

httpstatus.es : le réflexe à avoir pour la signification exacte d'un code

REpresentational State Transfer

Une architecture pour les API web (R. Fielding, 2000)

le **modèle de maturité de Richardson** gradue la mise en œuvre d'une API REST :



En résumé

plus le niveau est élevé, plus l'API exploite la **sémantique HTTP** (verbes, codes, liens) au lieu de se servir d'HTTP comme d'un simple tuyau.

Must read

- ▶ [HATEOAS](#) – Wikipedia
- ▶ [Architectural Styles and the Design of Network-based Software Architectures](#) (R. Fielding, 2000)
- ▶ [Swagger Petstore](#) : un exemple d'API REST à manipuler

Attention

une API REST bien conçue reste une application web : elle est exposée aux **mêmes vulnérabilités** (injections, failles d'authentification et d'autorisation, ...)

Un protocole amnésique

chaque requête est traitée comme une requête **indépendante**, sans mémoire des requêtes précédentes (aucun état *implicite*).

Conséquence

pour maintenir un état (panier, connexion, préférences), le client et le serveur doivent **coopérer** :

- ▶ cookies, sessions, jetons (*tokens*)
- ▶ chaque mécanisme d'état ajoute une **surface d'attaque**

Historique

introduits par **Netscape** en **1994** pour fiabiliser l'implémentation du panier d'achat.

Le serveur les pose (header Set-Cookie)

```
Set-Cookie: name=value  
            [; Max-Age=age] [; Expires=date]  
            [; Domain=domain] [; Path=path]  
            [; Secure] [; HttpOnly] [; SameSite=Lax]
```

Le client les renvoie (à chaque requête)

```
Cookie: name=value
```

about:profiles

Dans firefox : `cookies.sqlite`

Périmètre

- ▶ **cloisonnés** par domaine : envoyés à l'hôte (et éventuellement à ses sous-domaines)
- ▶ les cookies *tiers* (posés par des ressources embarquées) servent souvent au *tracking*

Attributs de sécurité

- ▶ Path – chemin URL pour lequel le cookie est valide.
- ▶ Secure – transmis *uniquement* en HTTPS
- ▶ HttpOnly – invisible pour JavaScript (anti-**XSS**)
- ▶ SameSite=Lax|Strict – restreint l'envoi en contexte cross-site (anti-**CSRF**)

Conformité

avez-vous le consentement de l'utilisateur pour les cookies non essentiels ? [CNIL](#) / [RGPD](#)

Côté serveur

les **données** sont stockées côté serveur ; seul **l'identifiant de session** est transmis au client :

Cookie: PHPSESSID=hr0ms75gs6f7vlph0hhct2bjj3

Précautions

- ▶ régénérer l'identifiant après connexion (`session_regenerate_id`) : *anti-session fixation*
- ▶ protéger le cookie de session : `HttpOnly` + `Secure` + `SameSite`
- ▶ un cookie de session volé $\Rightarrow$ **usurpation complète** de l'identité de l'utilisateur

Un peu comme un cookie, mais côté client

- ▶ données gérées **côté client**, manipulées *uniquement* par JavaScript (API `localStorage`)
- ▶ **5 Mo** par domaine (vs **4 Ko** pour un cookie)
- ▶ non envoyées automatiquement avec les requêtes HTTP
- ▶ supprimables uniquement via JavaScript

Sécurité

- ▶ lisible par **tout script** exécuté sur la page ⇒ **exfiltration** dès qu'une faille **XSS** existe
- ▶ n'y stocker **jamais** de secrets : tokens, mots de passe, données personnelles
- ▶ pour les données sensibles, préférer un cookie `HttpOnly` (inaccessible au JavaScript)

Cookie de tracking

suivre votre comportement en ligne

Pour creuser

- ▶ willdurand-edu/cookie-playground
- ▶ samyk/evercookie : des cookies "zombis"
- ▶ enregistrer les mouvements de souris en JavaScript
- ▶ HTML5 local storage vs cookies pour le tracking

Et au-delà : amiunique.org

le *fingerprinting* permet de suivre un utilisateur **sans aucun cookie** : bien plus difficile à contre-attaquer que le tracking classique.

Tout ce qui vient du client doit être considéré comme **non fiable**.



La surface d'attaque du Web

- ▶ **query string** : visible (logs, historique, Referer) et modifiable
- ▶ **corps** : modifiable
- ▶ **cookies** : stockés côté client, donc modifiables (et lisibles par le serveur s'ils ne sont pas `HttpOnly`)
- ▶ **headers** : modifiables
- ▶ **script** : s'exécute côté client, à isoler du reste (SOP, CORS)

Tout ce qui vient du client est **non fiable** : query string, corps, headers, cookies, script...

Le protocole HTTP :

protocole de la **couche application** (RFC 2616, T. Berners-Lee, 1989) ; **HTTP/1.1**, **HTTP/2**, **HTTP/3** (QUIC) ; ressource désignée par une **URI** – URLs **trompeuses**, query string **visible**.

Requêtes, réponses et codes :

verbes **GET/POST/PUT/DELETE/PATCH**, idempotence ; réponse = ligne de statut + **headers** + corps ; codes **2xx/3xx/4xx/5xx** (401 vs 403).

REST :

exploiter la **sémantique HTTP** (URLs, verbes, codes, hypermédia).

L'état :

HTTP est **stateless** ; **cookies** (Set-Cookie/Cookie), **sessions**, localStorage ; Secure/HttpOnly/SameSite