

Chapitre 1 - Le web et HTTP

- [Petite histoire des API](#)
- [Anatomie d'une requête HTTP](#)
- [Qu'est-ce qu'une API distante ?](#)
- [Panorama : REST, SOAP, GraphQL et les autres](#)
- [TP : Trouver l'API cachée d'un site](#)

Petite histoire des API

Aujourd'hui, la plupart des usages numériques (lire ses mails, poster du contenu, utiliser une IA, etc.) déclenchent des dizaines d'appels API. Ça n'a pas toujours été le cas : petit retour en arrière...

Le web 1.0 : des pages et des formulaires

Au début des années 90, le web sert à afficher des **documents**. Le navigateur demande une page, le serveur la renvoie. Et c'est tout !

- Pas de JavaScript : il arrive en 1995 et sert surtout à faire clignoter des images.
 - Retour dans le passé : <https://spacejam.com/1996/>
- Seulement deux verbes HTTP utilisés en pratique : `GET` pour afficher une page et `POST` pour envoyer un formulaire.
- Chaque action recharge **toute** la page.

```
<form action="/livre-dor.php" method="POST">
  <input name="message" value="Super site, bravo au webmaster !">
  <button>Signer le livre d'or</button>
</form>
```

Les autres verbes (`PUT`, `DELETE` ...) existent dans HTTP depuis 1996, mais un formulaire HTML `<form>` ne sait toujours envoyer que `GET` ou `POST`, même en 2026.

Faire parler les machines entre elles

À la fin des années 90, les entreprises veulent que leurs programmes s'échangent des données, pas seulement des pages pour humains.

- **1998 - XML-RPC**, puis **SOAP** : on appelle une fonction à distance en envoyant une enveloppe XML. Puissant, mais très verbeux.
- **2000 - REST** : dans sa thèse, [Roy Fielding \(un des auteurs de HTTP\)](#) décrit un style d'architecture qui utilise le web tel qu'il est : des ressources, des URL et les verbes HTTP.

- **2002 - [Le mémo Bezos](#)** : chez Amazon, toutes les équipes doivent exposer leurs données uniquement via des interfaces de service. Ceux qui ne le font pas seront licenciés. Ambiance. C'est l'ancêtre d'AWS.

Le web devient une application

- **2005 - AJAX** : Gmail et Google Maps chargent des données en arrière-plan sans recharger la page. Le navigateur devient un **client d'API**.
- **2007 - l'iPhone** : les applications mobiles n'affichent pas de HTML, elles ont besoin de données brutes. Il faut des API partout.
- **Années 2010 - l'ère des API publiques** : Twitter, Google Maps, Stripe, OpenWeather... Des entreprises vendent même **uniquement** une API. JSON remplace XML.
- **2015 - OpenAPI** (ex-Swagger) : un format standard pour décrire une API.

Aujourd'hui : les machines consomment les API

Les applications « SPA » (React, Vue...) sont des coquilles vides qui récupèrent tout leur contenu par API.

Une API bien conçue et bien documentée sert aussi bien un développeur humain qu'une IA. C'est le fil conducteur de ce cours.

Année	Étape
1991	Le web : pages et formulaires
1998	XML-RPC, puis SOAP
2000	Thèse de Fielding : REST
2005	AJAX
2007	iPhone : les API deviennent indispensables

Année	Étape
2010+	API publiques, JSON
2015	OpenAPI
2024	MCP : les IA consomment les API

Anatomie d'une requête HTTP

HTTP (*HyperText Transfer Protocol*) est le protocole du web. Le principe est simple : un **client** envoie une **requête**, le **serveur** renvoie une **réponse**. Toujours dans ce sens, toujours une réponse par requête.

Voir HTTP en vrai

L'option `-v` (verbose) de `curl` affiche tout l'échange :

```
curl -v https://formation.tfrichet.fr
```

Les lignes qui commencent par `>` sont envoyées par le client, celles qui commencent par `<` sont reçues du serveur.

La requête

Exemple avec [JSONPlaceholder](#), une fausse API publique faite pour s'entraîner :

```
curl -v https://jsonplaceholder.typicode.com/posts \
  -H "Content-Type: application/json" \
  -d '{"title": "Hello", "body": "Mon premier article", "userId": 1}'
```

```
POST /posts HTTP/1.1
Host: jsonplaceholder.typicode.com
User-Agent: curl/8.10.1
Accept: */*
Content-Type: application/json
```

```
{"title": "Hello", "body": "Mon premier article", "userId": 1}
```

- **Ligne de requête** : la méthode (`POST`), le chemin (`/posts`) et la version de HTTP.
- **Headers** : des paires `Nom: valeur` qui donnent des informations sur la requête.

- **Ligne vide**, puis le **body** (facultatif) : les données envoyées.

La réponse

```
HTTP/1.1 201 Created
Content-Type: application/json; charset=utf-8
Location: https://jsonplaceholder.typicode.com/posts/101

{"title": "Hello", "body": "Mon premier article", "userId": 1, "id": 101}
```

- **Ligne de statut** : la version, le **code de statut** et sa description.
- **Headers**, ligne vide, puis le **body**.

Les headers à connaître

Header	Sens	Rôle
Host	Requête	Le nom de domaine visé (plusieurs sites peuvent partager un serveur)
Content-Type	Les deux	Le format du body envoyé : application/json, text/html...
Accept	Requête	Le format que le client souhaite recevoir
Authorization	Requête	Les informations d'authentification
User-Agent	Requête	Le logiciel client
Set-Cookie / Cookie	Réponse / Requête	Le serveur dépose un cookie, le client le renvoie ensuite
Location	Réponse	L'adresse d'une ressource créée ou d'une redirection

Oublier `Content-Type: application/json` en envoyant du JSON est l'erreur numéro 1. Le serveur ne sait pas comment lire le body et répond souvent par une erreur peu claire.

Un protocole sans état

HTTP est **sans état** (*stateless*) : le serveur ne se souvient de rien entre deux requêtes. Chaque requête doit contenir tout ce qu'il faut pour être comprise.

Alors comment un site sait-il que vous êtes connecté ? Le client renvoie à chaque requête une preuve d'identité : un **cookie** pour un site web, un **token** ou une **clé d'API** pour une API.

HTTP n'a pas de mémoire persistante. Ce n'est pas un défaut : c'est ce qui permet de répartir les requêtes sur plusieurs serveurs.

Les familles de codes de statut

Famille	Signification	Exemples
1xx	Information, patientez	101 Switching Protocols
2xx	Succès	200 OK, 201 Created, 204 No Content
3xx	Redirection, c'est ailleurs	301 Moved Permanently, 304 Not Modified
4xx	Erreur du client : c'est ta faute	400 Bad Request, 401, 404 Not Found
5xx	Erreur du serveur : c'est la mienne	500 Internal Server Error, 503 Service Unavailable

Le détail des codes est vu au chapitre 2.

Et HTTPS, HTTP/2, HTTP/3 ?

- **HTTPS** : le même HTTP, mais chiffré par TLS.
- **HTTP/2** et **HTTP/3** : le transport est plus rapide (multiplexage, QUIC), mais les méthodes, headers et codes restent les mêmes. Tout ce cours s'applique.

Qu'est-ce qu'une API distante ?

API signifie **Application Programming Interface** : une interface qui permet à un programme d'utiliser les fonctionnalités d'un autre, sans connaître son fonctionnement interne.

L'analogie du restaurant : vous (le client) ne rentrez pas en cuisine. Vous passez commande au serveur avec un vocabulaire défini (la carte), et il vous rapporte le plat. Le serveur, c'est l'API.

Une API distante

Une API **distante** (ou API web) est exposée sur le réseau, en général via HTTP. Le client et le serveur sont deux programmes séparés, souvent sur deux machines différentes, écrits dans des langages différents.

```
curl https://api.github.com/users/thibaudfrichet
```

```
{
  "login": "thibaudfrichet",
  "id": 277130160,,
  "name": "Thibaud Frichet",
  "public_repos": 5,
  ...
}
```

Un programme Python, une application mobile ou un script bash peuvent tous obtenir ces informations, sans savoir comment GitHub les stocke.

Son rôle : faire communiquer des **services** ou des **applications** entre eux.

- un front (SPA, application mobile) avec son back ;
- deux services d'une même entreprise ;
- votre application avec un service tiers (paiement, météo, cartographie...) ;

- un assistant IA avec vos outils.

Ce qu'on appelle aussi « API »... mais qui n'est pas distant

Exemple	API ?	Distante ?
La bibliothèque <code>math</code> de Python, <code>lodash</code> en JS	Oui, API d'une bibliothèque	Non : appel de fonction dans le même programme
<code>document.querySelector()</code>	Oui, API DOM du navigateur	Non : fournie par le navigateur
Les appels système Linux (<code>open</code> , <code>read</code>)	Oui, API du noyau	Non
Une base de données MariaDB / MySQL	Elle a un protocole, mais on ne parle pas d'API web	Elle est sur le réseau, mais ce n'est pas du HTTP
<code>https://api.github.com/users/t</code> <code>hibaudfrichet</code>	Oui	Oui

SSH, MySQL, RDP, etc. : Ce sont des protocoles, comme HTTP. On ne parle pas d'API dans ce cas.

API, webservice, microservice

- **API** : le contrat, l'interface (les routes, les formats, les règles).
- **Webservice** : un service accessible par le réseau qui expose une API. Le terme est surtout utilisé pour l'époque SOAP.
- **Microservice** : une manière de découper une application en petits services indépendants, qui communiquent entre eux... par API.

Panorama : REST, SOAP, GraphQL et les autres

REST n'est pas la seule façon de construire une API. Voici les principales, avec le même besoin à chaque fois : **récupérer le livre n°3** d'une bibliothèque en ligne.

REST

Des **ressources** identifiées par des URL, manipulées avec les verbes HTTP. Le format est libre, JSON dans la grande majorité des cas.

```
GET /v1/livres/3 HTTP/1.1
Accept: application/json
```

```
{"id": 3, "titre": "Le Petit Prince", "auteur": "Antoine de Saint-Exupéry"}
```

Simple, lisible, compatible avec tout ce qui parle HTTP. C'est le sujet de ce cours.

SOAP

Un appel de fonction à distance, emballé dans une enveloppe XML, toujours en **POST**. Le contrat est décrit dans un fichier WSDL.

```
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Body>
    <GetLivre>
      <Id>3</Id>
    </GetLivre>
  </soap:Body>
</soap:Envelope>
```

Encore très présent dans les banques, les assurances et les administrations.

Si un jour on vous confie une API SOAP, ne paniquez pas : c'est verbeux, mais c'est bien documenté par nature.

GraphQL

Une seule URL, et le client décrit **exactement** les champs qu'il veut. Créé par Facebook en 2015.

```
query {  
  livre(id: 3) {  
    titre  
    auteur  
  }  
}
```

Pratique quand les besoins du front varient beaucoup. Plus complexe à mettre en cache et à sécuriser.

gRPC

Appels de fonctions à distance très performants, en binaire (Protocol Buffers) sur HTTP/2. Créé par Google.

```
service Bibliotheque {  
  rpc GetLivre (LivreRequest) returns (Livre);  
}
```

Surtout utilisé entre microservices, rarement exposé au navigateur.

WebSocket

Une connexion qui reste **ouverte** dans les deux sens. Le serveur peut envoyer des messages quand il veut : chat, jeux, cours de bourse, notifications en direct.

Webhooks

L'inverse d'une API classique : « ne nous appelez pas, on vous appellera ». Vous donnez une URL au service, et c'est lui qui vous envoie une requête HTTP quand un événement se produit (paiement reçu, commit poussé...).

Comparatif

	Format	Transport	Cas d'usage typique
REST	JSON (souvent)	HTTP	API publiques, back de SPA et d'apps mobiles
SOAP	XML	HTTP (POST)	Systèmes d'entreprise historiques
GraphQL	JSON	HTTP (une seule URL)	Fronts complexes, données très liées
gRPC	Binaire (Protobuf)	HTTP/2	Communication entre microservices
WebSocket	Libre	Connexion persistante	Temps réel
Webhooks	JSON (souvent)	HTTP, dans l'autre sens	Notification d'événements

TP : Trouver l'API cachée d'un site

Objectif

Beaucoup de sites modernes sont des **SPA** : le navigateur charge une page quasi vide, puis récupère tout son contenu en appelant une API. Cette API n'est pas documentée publiquement, mais elle est sous vos yeux.

En groupe, vous allez **découvrir l'API cachée d'un site réel**, puis **rejouer une de ses requêtes hors du navigateur** avec `curl`.

On observe, on ne casse rien : uniquement des requêtes de lecture, à la main, sans boucle ni script. Les conditions d'utilisation de certains sites interdisent l'usage automatisé de leurs API internes.

Consignes

1 - Choisir un site

Choisissez un site parmi la liste, ou proposez-en un autre au formateur.

- **Plutôt faciles** : Wikipédia (barre de recherche), Open Food Facts, Steam (suggestions de recherche), Deezer (recherche), adresse.data.gouv.fr
- **Plus coriaces** : Leboncoin, SNCF Connect

2 - Observer le trafic

- Ouvrez les DevTools (`F12`), onglet **Réseau** (*Network*).
- Filtrez sur **Fetch/XHR** pour ne garder que les appels d'API.
- Utilisez le site normalement (recherche, filtres, pagination...) et regardez quelles requêtes apparaissent.

Cochez **Preserve log** pour ne pas perdre les requêtes quand la page change.

3 - Analyser une requête

Pour une requête intéressante, relevez :

- l'URL et la méthode HTTP ;
- les paramètres (dans l'URL ou dans le body) ;
- les headers qui semblent importants ;
- le code de statut et le format de la réponse (onglets **Aperçu** et **Réponse**).

4 - Rejouer la requête avec curl

- Clic droit sur la requête → **Copier** → **Copier en tant que cURL**.
- Collez la commande dans un terminal et exécutez-la : vous devez obtenir la même réponse.
- Supprimez les headers **un par un** et relancez à chaque fois. Gardez la **commande la plus courte qui fonctionne encore**.
- Modifiez un paramètre (le terme recherché, la page...) et vérifiez que la réponse change.

Si la requête rejouée est refusée (403, captcha...), le site est sans doute protégé par un anti-bot. Ce n'est pas un échec : notez quels headers ou cookies semblent nécessaires et pourquoi.

Résultat attendu

Une fiche par groupe, présentée à l'oral en quelques minutes :

- le site choisi et ce que fait la requête étudiée ;
- l'URL, la méthode et les paramètres ;
- un extrait de la réponse (format, structure) ;
- la **commande curl minimale** qui fonctionne, ou l'explication de ce qui bloque ;
- votre avis : cette API est-elle « propre » ? Les URL sont-elles lisibles ?