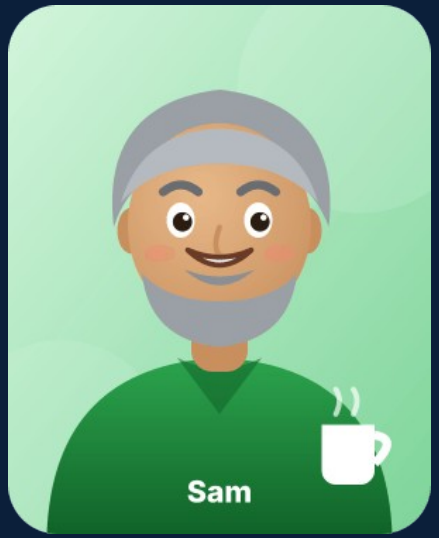
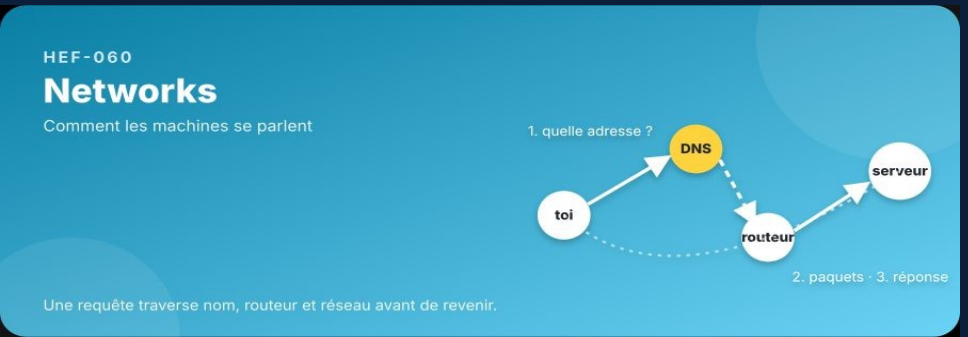
HEF-060 · MISSION 07 / 10

HTTP et les API

Le web, c'est une longue conversation de requêtes et de réponses. Apprenons sa langue.





Badge : Facteur du web

85 min · intermédiaire

L'histoire

Kodi ouvre les outils de développement du navigateur et voit défiler des dizaines de lignes : `GET`, `POST`, `200`, `404`, `301`. Il demande : « Que se passe-t-il quand je clique sur un lien ? »



> *« Ton navigateur envoie une demande textuelle à un serveur, qui répond par un autre texte, dit Sam. Ça ressemble à une conversation de bureau : poli, structuré, avec des codes pour dire si ça s'est bien passé. »*

Tu apprends à lire, écrire et concevoir des échanges HTTP, et à bâtir une petite API.

Ta mission

À la fin, je serai capable de…

1. décrire la structure d'une requête et d'une réponse HTTP
2. utiliser correctement les méthodes (GET, POST, PUT, DELETE) et savoir ce qu'est l'idempotence
3. interpréter les familles de codes de statut (2xx, 3xx, 4xx, 5xx)
4. construire une petite API JSON avec `http.server` et l'interroger avec `urllib`
5. expliquer l'absence d'état de HTTP, les cookies et le cache

Défi d'échauffement

Défi d'échauffement

Tu commandes dans un restaurant par écrit. Tu remets un papier avec : **ce que tu veux faire** (consulter la carte, commander, annuler), **quelle table**, et des **précisions** (sans oignon). Le serveur répond avec un papier : un **code** (« ok », « plat épuisé », « vous n'avez pas le droit »), des précisions, et le plat. Écris la structure de ces deux papiers.

Le concept

HTTP, le protocole du web

HTTP (*HyperText Transfer Protocol*) est un protocole d'**application** de type **requête-réponse**, au-dessus de TCP (et de TLS pour HTTPS). Le client envoie une **requête**, le serveur renvoie une **réponse**.

Une requête

```
GET /cours/hef-000?langue=fr HTTP/1.1
Host: hashbox.example
Accept: application/json
Authorization: Bearer abc123
```

- **Méthode** : ce qu'on veut faire.
- **Chemin** (et paramètres) : sur quelle ressource.
- **En-têtes** : informations complémentaires.

Le concept

Une requête (suite)

- **Corps** (facultatif) : les données envoyées (pour `POST`, `PUT`).

Une réponse

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 41

{"id": "HEF-000", "titre": "Mentalité"}
```

- **Ligne de statut** : version, code, texte.
- **En-têtes**, ligne vide, puis **corps**.

Le concept

Les méthodes

Méthode	Sens	Sûre ?	Idempotente ?
GET	lire	oui	oui
POST	créer / déclencher	non	**non**
PUT	remplacer entièrement	non	oui
PATCH	modifier en partie	non	pas forcément
DELETE	supprimer	non	oui

- **Sûre** : ne modifie rien sur le serveur.
- **Idempotente** : l'appliquer 1 fois ou 10 fois donne le **même résultat**. Important quand un client **réessaie** après une erreur réseau : refaire un `PUT` est inoffensif, refaire un `POST` peut créer un doublon.

Le concept

Les codes de statut

Famille	Sens	Exemples
1xx	information	101 Switching Protocols
2xx	succès	200 OK, 201 Created, 204 No Content
3xx	redirection	301 Moved Permanently, 304 Not Modified
4xx	erreur du **client**	400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, 429 Too Many Requests
5xx	erreur du **serveur**	500 Internal Server Error, 502 Bad Gateway, 503 Service Unavailable

Règle : 4xx = « tu as mal demandé », 5xx = « j'ai mal répondu ».

Le concept

HTTP n'a pas de mémoire

Chaque requête est **indépendante** : le serveur ne se souvient pas de la précédente (*sans état*). Pour garder un contexte (utilisateur connecté), on utilise :

- des **cookies** (le serveur donne un identifiant que le client renvoie) ;
- des **jetons** (`Authorization: Bearer …`).

Le cache

Des en-têtes (`Cache-Control`, `ETag`) permettent de réutiliser une réponse sans la redemander. Le code `304 Not Modified` signifie « ta copie est toujours valable ».

Le concept

Les API REST

Une **API** (*Application Programming Interface*) permet à un programme d'en utiliser un autre. Le style **REST** modélise des **ressources** avec des URL et utilise les méthodes HTTP :

Action	Requête
Lister les cours	`GET /cours`
Lire un cours	`GET /cours/HEF-000`
Créer un cours	`POST /cours`
Remplacer un cours	`PUT /cours/HEF-000`
Supprimer un cours	`DELETE /cours/HEF-000`

Les données circulent souvent en **JSON**.

Le concept

HTTP/1.1, 2, 3

HTTP/2 et HTTP/3 changent la façon de **transporter** (multiplexage, QUIC sur UDP) mais gardent la **même sémantique** : méthodes, codes, en-têtes.

Exemple pas à pas

Problème : écrire un mini serveur d'API JSON pour des cours, et l'interroger.

```
import json, threading, urllib.request, urllib.error
from http.server import BaseHTTPRequestHandler, HTTPServer

COURS = {"HEF-000": {"titre": "Mentalité d'ingénieur"}, "HEF-010": {"titre": "Algo
   rithmes"}}

class Gestionnaire(BaseHTTPRequestHandler):
    def _repondre(self, code, donnees=None):
        corps = json.dumps(donnees).encode() if donnees is not None else b""
        self.send_response(code)
        self.send_header("Content-Type", "application/json")
        self.send_header("Content-Length", str(len(corps)))
        self.end_headers()
        self.wfile.write(corps)

    def do_GET(self):
```

Exemple pas à pas

Exemple pas à pas (suite)

```
        if self.path == "/cours":
            self._repondre(200, sorted(COURS))
        elif self.path.startswith("/cours/") and self.path[7:] in COURS:
            self._repondre(200, COURS[self.path[7:]])
        else:
            self._repondre(404, {"erreur": "introuvable"})

    def do_POST(self):
        longueur = int(self.headers.get("Content-Length", 0))
        try:
            donnees = json.loads(self.rfile.read(longueur))
            COURS[donnees["id"]] = {"titre": donnees["titre"]}
        except (ValueError, KeyError):
            return self._repondre(400, {"erreur": "JSON invalide"})
        self._repondre(201, {"id": donnees["id"]})

    def log_message(self, *args):
        pass
```

Exemple pas à pas (suite)

```
serveur = HTTPServer(("127.0.0.1", 0), Gestionnaire)
port = serveur.server_address[1]
threading.Thread(target=serveur.serve_forever, daemon=True).start()
base = f"http://127.0.0.1:{port}"

def appeler(methode, chemin, donnees=None):
    corps = json.dumps(donnees).encode() if donnees is not None else None
    requete = urllib.request.Request(base + chemin, data=corps, method=methode,
                                     headers={"Content-Type": "application/json"})
    try:
        with urllib.request.urlopen(requete, timeout=2) as r:
            return r.status, json.loads(r.read() or b"null")
    except urllib.error.HTTPError as e:
        return e.code, json.loads(e.read() or b"null")

print(appeler("GET", "/cours"))                                  # 200, liste
print(appeler("GET", "/cours/HEF-000"))                          # 200, objet
print(appeler("GET", "/cours/XXX"))                              # 404
print(appeler("POST", "/cours", {"id": "HEF-020", "titre": "Structures"}))   # 201
```

Exemple pas à pas (suite)

```
print(appeler("POST", "/cours", {"titre": "sans id"}))          # 400
serveur.shutdown()
```

Lecture : le serveur choisit le **code de statut** selon la situation (404 introuvable, 201 créé, 400 requête invalide). Le client lit le code **avant** le corps : c'est ce qui permet de gérer les erreurs proprement.

À toi, avec aide

À toi, avec aide

Pour chaque situation, choisis la méthode et le code de statut les plus adaptés :

Situation	Méthode	Code
Lire la liste des membres	?	?
Un membre s'inscrit (création)	?	?
Un membre demande une page qui n'existe pas	?	?
Un membre non connecté demande son profil	?	?
Le serveur de base de données est en panne	?	?
Un membre envoie un JSON mal formé	?	?

À toi, avec aide · réponse

Indices progressifs

Indice 1

401 = non authentifié ; 403 = authentifié mais pas autorisé.

Indice 2

La création réussie renvoie souvent 201.

Indice 3

Une panne côté serveur est une erreur 5xx.

À toi, avec aide · réponse

Réponse

Situation	Méthode	Code
Lire la liste des membres	GET	200
S'inscrire	POST	201
Page inexistante	GET	404
Profil sans être connecté	GET	401
Base de données en panne	(toute)	500 ou 503
JSON mal formé	POST	400

À toi, en solo

Idempotence et réessai. Étends le serveur de l'exemple avec `PUT /cours/<id>` (remplace le cours) et `DELETE /cours/<id>`.

1. Écris un client `appeler_avec_reessai(methode, chemin, donnees, essais=3)` qui réessaie en cas d'erreur 503 ou de coupure.
2. Simule un serveur instable qui renvoie 503 une fois sur deux.
3. Montre qu'un `PUT` réessayé donne le bon résultat final, alors qu'un `POST` réessayé peut créer des doublons (ajoute `POST /commandes` qui génère un numéro).
4. Propose une clé d'idempotence (en-tête `Idempotency-Key`) pour sécuriser le `POST`.

À toi, en solo · réponse

Pistes de correction

```
import json, threading, urllib.request, urllib.error
from http.server import BaseHTTPRequestHandler, HTTPServer

etat = {"cours": {}, "commandes": [], "requetes": 0, "cles": {}}

class H(BaseHTTPRequestHandler):
    def _r(self, code, donnees=None):
        corps = json.dumps(donnees).encode() if donnees is not None else b""
        self.send_response(code); self.send_header("Content-Length", str(len(corps
   ))); self.end_headers()
        self.wfile.write(corps)

    def _instable(self):
        etat["requetes"] += 1
        if etat["requetes"] % 2 == 1:
            self._r(503, {"erreur": "indisponible"})
            return True
        return False
```

À toi, en solo · réponse

Pistes de correction (suite)

```
    def do_PUT(self):
        n = int(self.headers.get("Content-Length", 0)); donnees = json.loads(self.
    rfile.read(n))
        if self._instable(): return
        etat["cours"][self.path.split("/")[-1]] = donnees
        self._r(200, donnees)

    def do_POST(self):
        n = int(self.headers.get("Content-Length", 0)); self.rfile.read(n)
        cle = self.headers.get("Idempotency-Key")
        if self._instable(): return
        if cle and cle in etat["cles"]:
            return self._r(200, etat["cles"][cle])
        numero = len(etat["commandes"]) + 1
        etat["commandes"].append(numero)
        if cle: etat["cles"][cle] = {"numero": numero}
        self._r(201, {"numero": numero})

    def log_message(self, *a): pass
```

À toi, en solo · réponse

Pistes de correction (suite)

```
serveur = HTTPServer(("127.0.0.1", 0), H)
port = serveur.server_address[1]
threading.Thread(target=serveur.serve_forever, daemon=True).start()

def appeler(methode, chemin, donnees, entetes=None, essais=3):
    for _ in range(essais):
        req = urllib.request.Request(f"http://127.0.0.1:{port}{chemin}", data=json
   .dumps(donnees).encode(),
                                     method=methode, headers=entetes or {})
        try:
            with urllib.request.urlopen(req, timeout=2) as r:
                return r.status
        except urllib.error.HTTPError as e:
            if e.code != 503:
                return e.code
    return "échec"

print(appeler("PUT", "/cours/HEF-000", {"titre": "X"}), etat["cours"])
```

À toi, en solo · réponse

Pistes de correction (suite)

```
etat["requetes"] = 1                    # force un 503 puis un succès pour le PO
   ST sans clé
# POST dont la réponse 201 est "perdue" côté client : simulons un réessai manuel
appeler("POST", "/commandes", {}); appeler("POST", "/commandes", {})
print("sans clé  :", etat["commandes"])
etat.update(commandes=[], cles={}, requetes=0)
for _ in range(2):
    appeler("POST", "/commandes", {}, {"Idempotency-Key": "abc"}, essais=3)
print("avec clé  :", etat["commandes"])
serveur.shutdown()
```

Sans clé, deux envois du même ordre créent deux commandes. Avec une clé d'idempotence, le serveur reconnaît la répétition et renvoie la **même** réponse : c'est le mécanisme utilisé par les API de paiement.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
`POST` réessayé aveuglément	Doublons (double paiement)	Clé d'idempotence ou PUT
Confondre 401 et 403	Mauvais message à l'utilisateur	401 : qui es-tu ? 403 : tu n'as pas le droit
Tout renvoyer en 200	Les clients ne détectent pas les erreurs	Utiliser les codes de statut
Mettre des secrets dans l'URL	Journaux et historique les conservent	En-têtes ou corps, jamais l'URL

Mini-boss

Mini-boss

Mini API de tâches. Conçois et implémente (avec `http.server`) une API REST pour une liste de tâches :

- `GET /taches` (avec filtre `?fait=true`), `GET /taches/<id>` ;
- `POST /taches` (crée, 201 + en-tête `Location`) ;
- `PUT /taches/<id>` (remplace), `PATCH /taches/<id>` (modifie un champ), `DELETE /taches/<id>` (204).

Exigences : JSON valide, codes de statut corrects (400, 404, 405 pour une méthode non supportée), messages d'erreur clairs. Écris une suite de tests avec `urllib` qui vérifie chaque route, y compris les cas d'erreur.

Mini-boss · réponse

Pistes de correction

Un bon travail : un dictionnaire de tâches et un compteur d'identifiants ; un routage par expression régulière sur le chemin ; une fonction `_repondre(code, donnees, entetes)` ; `Location: /taches/<id>` sur les créations ; `204` sans corps pour `DELETE` ; `405` avec en-tête `Allow` quand la méthode ne convient pas ; `400` quand le JSON est invalide ou qu'un champ obligatoire manque ; `404` pour une tâche inconnue. Les tests couvrent : liste vide, création, lecture, remplacement, modification partielle, suppression, deuxième suppression (404), JSON invalide (400), méthode inconnue (405).

Quiz éclair

Quiz éclair

1. Un code 404 signifie :

A. la ressource demandée est introuvable
B. une erreur du serveur
C. une redirection

2. Une méthode idempotente :

A. est toujours sûre
B. donne le même résultat si on la répète
C. ne renvoie jamais de corps

3. HTTP est dit « sans état » car :

A. il ne stocke aucun fichier

Quiz éclair

Quiz éclair (suite)

B. il n'utilise pas TCP
C. le serveur ne se souvient pas des requêtes précédentes

4. 🔁 *Révision mission 06* : Un enregistrement MX indique :

A. le serveur de mail d'un domaine
B. l'adresse IPv6
C. un alias

5. 🔁 *Révision mission 06* : Le TTL d'une réponse DNS détermine :

A. la vitesse de réponse
B. combien de temps elle peut être gardée en cache
C. la taille du paquet

Quiz éclair · réponse

Réponses

1. **A**.
2. **B**.
3. **C**.
4. **A**.
5. **B**.

Je réfléchis

1. Quand tu relances une action (clic sur « payer »), qu'est-ce qui te protège d'un doublon ?
2. Pourquoi les codes de statut sont-ils un contrat utile entre client et serveur ?

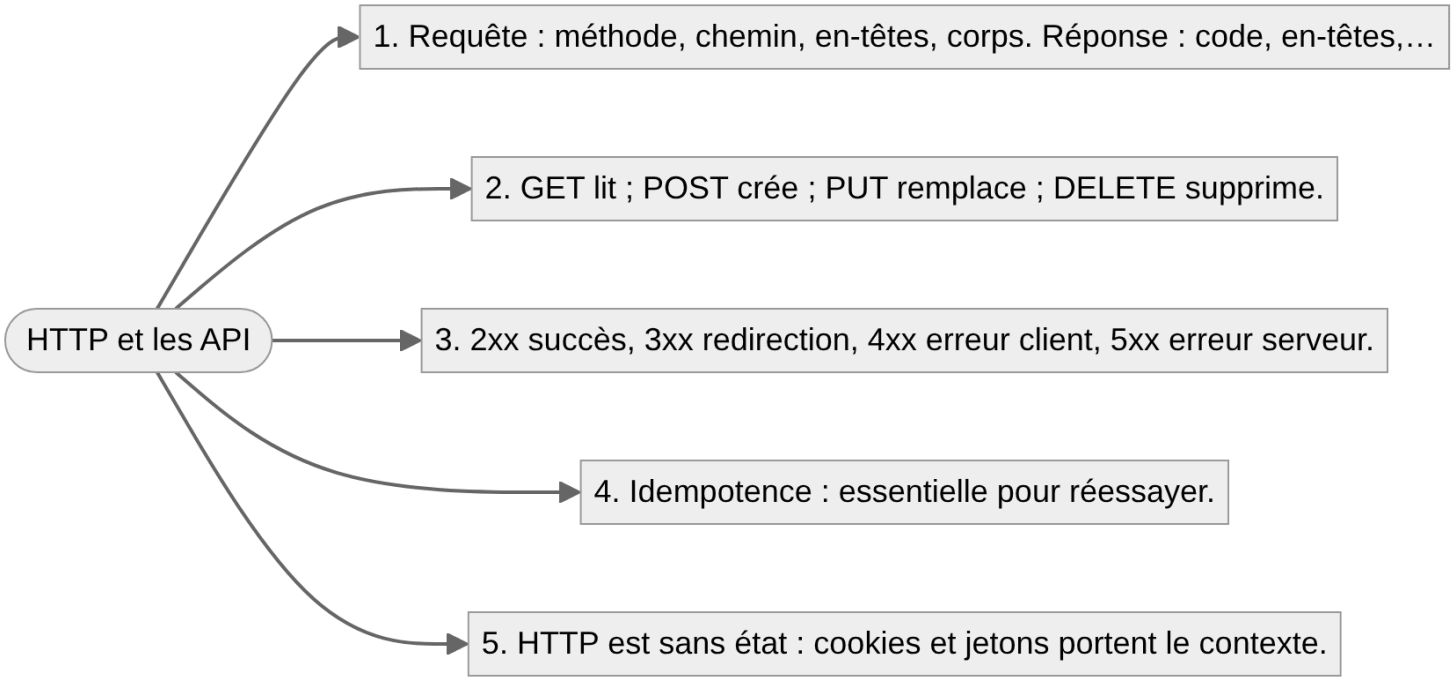
À retenir

À retenir

1. Requête : méthode, chemin, en-têtes, corps. Réponse : code, en-têtes, corps.
2. GET lit ; POST crée ; PUT remplace ; DELETE supprime.
3. 2xx succès, 3xx redirection, 4xx erreur client, 5xx erreur serveur.
4. Idempotence : essentielle pour réessayer.
5. HTTP est sans état : cookies et jetons portent le contexte.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Facteur du web

Prochaine mission : TLS et HTTPS : la confiance

