Introduzione

HTTP (HyperText Transfer Protocol) è un protocollo di **livello applicativo** all'interno dello **stack di protocolli Internet**, progettato per il trasferimento di informazioni ipertestuali (come pagine web, immagini, file o dati JSON).

È il linguaggio di comunicazione tra **client** (ad esempio un browser web o un'app mobile) e **server** (ad esempio un sito web o un servizio API).

Versione	Anno di introduzione	Stato attuale
HTTP/0.9	1991	Obsoleto
HTTP/1.0	1996	Obsoleto
HTTP/1.1	1997	Standard (ancora molto diffuso)
HTTP/2	2015	Standard
HTTP/3	2022	Standard più recente

HTTP è stato ideato da **Tim Berners-Lee** tra il **1989 e il 1991** al **CERN**, come parte della nascita del **World Wide Web**.
La sua variante sicura, che utilizza la crittografia **TLS**, è nota come **HTTPS** (HTTP Secure).

Architettura Client–Server

HTTP segue un modello di comunicazione **client/server**:

- Il **client** invia una **richiesta (request)**.
- Il **server** elabora la richiesta e invia una **risposta (response)**.

Client (User Agent)

Un **User Agent (UA)** è qualunque programma in grado di inviare richieste HTTP, come:

- Browser web (Chrome, Firefox, Safari…)
- App mobili (Instagram, YouTube…)
- Dispositivi IoT (telecamere, smart TV…)
- Script automatici (spider, bot, crawler, ecc.)

Server (Origin Server)

Un **Origin Server (O)** è il sistema che riceve e gestisce le richieste, producendo risposte autorevoli.
Esempi:

- Un sito web (es. `www.example.com`)
- Una piattaforma di streaming
- Un server di videosorveglianza
- Un gestionale aziendale online

Ciclo Richiesta–Risposta

Il funzionamento di base è:

```
CLIENT (UA)  --->  RICHIESTA HTTP  --->  SERVER (O)
CLIENT (UA)  <---  RISPOSTA HTTP   <---  SERVER (O)
```

Esempio di Richiesta HTTP

```
GET /hello.txt HTTP/1.1
Host: www.example.com
User-Agent: curl/7.64.1
Accept-Language: en, it
```

- **Linea di richiesta:** `GET /hello.txt HTTP/1.1`
  - **Metodo:** `GET`
  - **URI:** `/hello.txt`
  - **Versione protocollo:** `HTTP/1.1`
- **Intestazioni (Header Fields):**
  - `Host` indica il dominio del server.
  - `User-Agent` specifica il client che invia la richiesta.
  - Accept-Language` indica le lingue preferite per la risposta.
- **Corpo del messaggio:** opzionale (in `GET` normalmente assente).

Esempio di Risposta HTTP

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
ETag: "34aa387-d-1568eb00"
Content-Length: 51
Content-Type: text/plain

Hello World! My content includes a trailing CRLF.
```

- **Linea di stato:** `HTTP/1.1 200 OK`
  - `200` è il **codice di stato** → successo.
  - `OK` è il **messaggio testuale** associato.
- **Header di risposta:** forniscono metadati sul contenuto:
  - `Server`: indica il software server (es. Apache, Nginx, ecc.)
  - `Last-Modified`: data ultima modifica
  - `ETag`: identificativo univoco del contenuto

    - `Content-Type`: tipo di contenuto (`text/plain`, `application/json`, ecc.)

- **Corpo della risposta:** contiene i dati richiesti (in questo caso, testo semplice).

Intermediari e Cache nel protocollo HTTP

Nel modello client-server, tra l'*User Agent (UA)* e l'*Origin Server (O)* possono trovarsi uno o più **intermediari**: componenti che partecipano alla trasmissione delle richieste o delle risposte.

Tipi di Intermediari

- **Proxy** – ricevono richieste dal client e le inoltrano al server, eventualmente memorizzando le risposte (cache proxy).
- **Gateway** – traducono le richieste da un protocollo a un altro (es. da HTTP a FTP o a un sistema interno).
- **Tunnel** – creano un canale trasparente (es. HTTPS tramite il metodo `CONNECT`).

```
UA → A → B → C → O
```

Dove:

- `UA` è il client,
- `A`, `B`, `C` sono intermediari,
- `O` è il server di origine.

Cache

La **cache** è un archivio di risposte HTTP memorizzate per velocizzare richieste future.
Quando un client richiede una risorsa già presente in cache e ancora valida, la risposta può essere servita immediatamente, **senza contattare il server**.

Requisiti per essere memorizzabile

- Il server deve dichiarare la risposta come **cacheable**.
- I **proxy** possono memorizzare risposte.
- I **tunnel** invece **non** possono farlo.

Esempio di cache in azione

1. Il client richiede una pagina:

```
GET /index.html HTTP/1.1
Host: www.example.com
```

Il server risponde:

```
HTTP/1.1 200 OK Cache-Control: max-age=3600
Content-Type: text/html
```

2. Se il client ripete la richiesta entro un'ora, la cache risponde direttamente senza ricontattare il server.

Metodi HTTP

I **metodi HTTP** (detti anche *verbi HTTP*) specificano l'azione che il client desidera eseguire su una risorsa.

Metodo	Descrizione breve
GET	Recupera una rappresentazione di una risorsa
HEAD	Come GET, ma senza il corpo della risposta
POST	Invia dati al server per creare o aggiornare risorse subordinate
PUT	Crea o sostituisce una risorsa specifica
DELETE	Rimuove una risorsa
CONNECT	Crea un tunnel (spesso usato per HTTPS)
OPTIONS	Elenca i metodi supportati dal server per una risorsa
TRACE	Esegue un test di loop-back per verificare il percorso
PATCH	Applica modifiche parziali a una risorsa

Proprietà dei metodi

Ogni metodo HTTP può essere:

- **Safe (sicuro)** – non altera lo stato del server.
  Esempi: `GET`, `HEAD`, `OPTIONS`, `TRACE`.
- **Idempotent (idempotente)** – richieste ripetute producono lo stesso effetto.
  Esempi: `PUT`, `DELETE`, `GET`, `HEAD`, `OPTIONS`.
- **Cacheable (memorizzabile nella cache)** – la risposta può essere salvata.
  Esempi: `GET`, `HEAD`, `POST` (in alcuni casi).

GET — Recuperare informazioni

Richiede una rappresentazione della risorsa indicata dall'URI.

```
GET /api/users/1 HTTP/1.1
Host: example.com
Accept: application/json
```

Risposta:

```
HTTP/1.1 200 OK
Content-Type: application/json
{
    "id": 1,
    "name": "Alice"
}
```

- **Safe** – non modifica nulla sul server.
- **Cacheable** – può essere memorizzata.
- **Non idempotente** in senso stretto, perché il server può comunque generare effetti collaterali (es. logging o conteggio visite).

POST — Inviare dati o creare risorse

Usato per **creare una nuova risorsa subordinata** o **attivare un'azione**.

```
POST /api/users HTTP/1.1
Host: example.com
Content-Type: application/json
{
    "name": "Alice",
    "email": "alice@example.com"
}
```

Risposta:

```
HTTP/1.1 201
Created Location: /api/users/123
```

- **Non safe** – crea o modifica risorse.
- **Non idempotente** – inviare la stessa richiesta due volte crea due risorse.
- **Cacheable solo se specificato** dal server (raro).

PUT — Creare o sostituire una risorsa

Crea una risorsa all'URI specificato o la sostituisce completamente se già esiste.

```
PUT /api/users/123 HTTP/1.1
Host: example.com
Content-Type: application/json
{
    "name": "Alice Updated",
    "email": "alice@newmail.com"
}
```

Risposta:

```
HTTP/1.1 200 OK
```

- **Idempotente** – ripetere la stessa richiesta produce sempre lo stesso stato finale.
- **Non safe**, poiché modifica dati.
- **Non cacheable**.

DELETE — Rimuovere una risorsa

Richiede la cancellazione di una risorsa dal server.

```
DELETE /api/users/123 HTTP/1.1 Host: example.com
```

Risposta:

```
HTTP/1.1 204 No Content
```

- **Idempotente** – eliminare una risorsa già rimossa non ha ulteriori effetti.
- **Non safe**, **non cacheable**.

PATCH — Aggiornamento parziale

Permette di modificare **solo parte** di una risorsa, invece di sostituirla interamente come fa `PUT`.

```
PATCH /api/users/123 HTTP/1.1
Host: example.com
Content-Type: application/json
{
    "email": "new_email@example.com"
}
```

Risposta:

```
HTTP/1.1 200 OK
```

- **Idempotente in alcuni casi**, dipende dal tipo di modifica.
- **Non safe**, **non cacheable**.
- Definito in **RFC 5789**.

Metodi meno usati

Metodo	Descrizione	Esempio pratico
HEAD	Come `GET`, ma senza corpo di risposta (utile per verificare metadati)	`curl -I https://www.example.com`
CONNECT	Apre un tunnel (usato per HTTPS tramite proxy)	`CONNECT www.example.com:443`
OPTIONS	Restituisce i metodi supportati	`Allow: GET, POST, OPTIONS`
TRACE	Ritorna la richiesta originale per test diagnostici	utile per debug, ma raramente usato in produzione

Codici di Stato HTTP

Ogni **risposta HTTP** contiene un **codice di stato** a tre cifre, che indica il risultato della richiesta e la semantica della risposta.

Formato generale:

```
HTTP/<versione> <codice> <messaggio>
```

Esempio:

```
HTTP/1.1 200 OK
```

Il codice (es. `200`) fornisce un'informazione standardizzata sul risultato, mentre il messaggio testuale (`OK`) è solo descrittivo.

Classi di codici di stato

Classe	Intervallo	Significato generale
1xx	100–199	Informazioni: richiesta ricevuta, in elaborazione
2xx	200–299	Successo: richiesta completata correttamente
3xx	300–399	Reindirizzamento: serve un'ulteriore azione (nuovo URL, cache, ecc.)
4xx	400–499	Errore del client: richiesta non valida o non autorizzata
5xx	500–599	Errore del server: fallimento durante l'elaborazione valida

1xx – Informational (Informativo)

Questi codici indicano che la richiesta è stata ricevuta e il server è in attesa di ulteriori azioni.

Codice	Significato	Esempio
100 Continue	Il client può continuare a inviare il corpo della richiesta	Usato da `POST` o `PUT` di grandi dimensioni
101 Switching Protocols	Il server accetta di cambiare protocollo (es. a WebSocket)	Utilizzato quando si passa da HTTP a WebSocket
102 Processing	Il server sta ancora elaborando (tipico in WebDAV)	

Questi codici sono **rari** nelle applicazioni web comuni, ma importanti nei sistemi complessi (es. streaming, upload, WebSocket).

2xx – Successful (Richiesta completata con successo)

Indicano che la richiesta è stata **ricevuta, compresa e gestita correttamente**.

Codice	Significato	Esempio
200 OK	Richiesta completata con successo	GET di una pagina o risorsa
201 Created	Nuova risorsa creata con successo	Risposta a un POST su `/users`
202 Accepted	Richiesta accettata, ma non ancora completata	Richieste asincrone o batch
204 No Content	Richiesta eseguita ma senza contenuto da restituire	DELETE o PUT completati
206 Partial Content	Risposta parziale (download parziale, streaming)	Video in streaming o resume di file

3xx – Redirection (Reindirizzamento)

Il client deve eseguire **ulteriori azioni**, spesso una nuova richiesta a un altro indirizzo (URI).

Codice	Significato	Esempio
301 Moved Permanently	La risorsa è stata spostata in modo permanente	Usato per redirect SEO o cambi di dominio
302 Found	Spostamento temporaneo (redirect temporaneo)	Usato in moduli o login
303 See Other	Redireziona a un'altra risorsa per ottenere il risultato	Usato dopo POST per visualizzare pagina di conferma
304 Not Modified	Risorsa non modificata, usare versione in cache	Usato da browser per risparmiare banda
307 Temporary Redirect	Come 302 ma con metodo invariato	POST rimane POST
308 Permanent Redirect	Come 301 ma preserva il metodo	PUT rimane PUT

4xx – Client Error (Errore del client)

Il server **non può o non vuole** elaborare la richiesta a causa di errori del client.

Codice	Significato	Esempio
400 Bad Request	Sintassi della richiesta errata	JSON malformato in POST
401 Unauthorized	Autenticazione richiesta o token mancante	Mancanza header `Authorization`
403 Forbidden	Accesso vietato anche se autenticato	Utente senza permessi
404 Not Found	Risorsa non trovata	URL errato
405 Method Not Allowed	Metodo HTTP non supportato	PUT su risorsa di sola lettura
409 Conflict	Conflitto con lo stato attuale della risorsa	Tentativo di doppia registrazione
429 Too Many Requests	Troppe richieste in poco tempo (rate limit)	API con limitazioni
418 I'm a teapot	Easter egg dell'RFC 2324	Non serio ma famoso

5xx – Server Error (Errore del server)

Indicano che il server ha riscontrato un problema interno durante l'elaborazione di una richiesta apparentemente valida.

Codice	Significato	Esempio
500 Internal Server Error	Errore generico del server	Eccezione imprevista nel codice
501 Not Implemented	Metodo non supportato	Metodo HTTP non riconosciuto
502 Bad Gateway	Risposta non valida da un server a monte	Proxy che riceve errore
503 Service Unavailable	Server sovraccarico o in manutenzione	Picco di traffico o update
504 Gateway Timeout	Timeout del server intermedio	API troppo lente

API Pubbliche e utilizzo di cURL

HTTP non serve solo per navigare sul Web: è la base della comunicazione tra **client e server** anche per le **API (Application Programming Interface)**, ossia servizi che permettono di ottenere o inviare dati tramite Internet.

Esempi di API Pubbliche
Le API pubbliche sono ottime per imparare e testare le richieste HTTP, poiché non richiedono autenticazione o token

API	Descrizione	Esempio URL
SWAPI	API di Star Wars, restituisce dati sui personaggi e film	`https://swapi.dev/api/people/1/`
JSONPlaceholder	API di test per simulare post, utenti, commenti	`https://jsonplaceholder.typicode.com/posts/1`
Chuck Norris API	Restituisce battute casuali	`https://api.chucknorris.io/jokes/random`
PokéAPI	Dati su Pokémon	`https://pokeapi.co/api/v2/pokemon/bulbasaur`
Wayback Machine	Mostra versioni archiviate di un sito web	`https://web.archive.org/web/20140707075340/http://www.di.unirom.it`
HTTP Cat	Restituisce immagini per ogni codice HTTP	`https://http.cat/status/404.jpeg`

cURL – Command Line URL tool

cURL è uno strumento da riga di comando molto potente che consente di **inviare richieste HTTP** e visualizzare le risposte.
È disponibile su **Linux**, **macOS**, **Windows** e come libreria (`libcurl`) nei linguaggi di programmazione.

Sintassi generale:

```
curl [OPZIONI] [URL]
```

Esempio base – GET (default)

Se non si specifica nulla, `curl` effettua una **richiesta GET**:

```
curl https://swapi.dev/api/people/1/
```

Risultato:

```
{
    "name": "Luke Skywalker",
    "height": "172",
    "mass": "77",
    "gender": "male"
}
```

Suggerimento:
Puoi formattare il JSON in modo leggibile con:
`curl https://swapi.dev/api/people/1/ | jq`

HEAD – Ottenere solo le intestazioni (header)

L'opzione `-I` o `--head` mostra solo gli header della risposta, non il corpo.

```
curl -I https://swapi.dev/api/people/1/
```

Esempio di output:

```
HTTP/1.1 200 OK
Content-Type: application/json
Server: cloudflare
Date: Wed, 08 Oct 2025 15:32:00 GMT
```

POST – Inviare dati per creare una nuova risorsa

Per inviare dati JSON:

- Usa `-X POST` per specificare il metodo.
- Usa `-H` per impostare gli header (es. `Content-Type`).
- Usa `-d` per fornire il corpo della richiesta.

```
curl -X POST \
-H "Content-Type: application/json" \
-d '{"title": "Test", "body": "Ciao mondo!", "userId": 1}' \
https://jsonplaceholder.typicode.com/posts
```

Esempio di risposta:

```
{
    "title": "Test",
    "body": "Ciao mondo!",
    "userId": 1,
    "id": 101
}
```

Nota: anche se questa API non salva realmente i dati (è di test), risponde come se li avesse creati.

PUT – Aggiornare o sostituire una risorsa

```
curl -X PUT \
-H "Content-Type: application/json" \
-d '{"id": 1, "title": "Nuovo titolo", "body": "Contenuto aggiornato"}' \
https://jsonplaceholder.typicode.com/posts/1
```

Risposta:

```
{
    "id": 1,
    "title": "Nuovo titolo",
    "body": "Contenuto aggiornato"
}
```

- Idempotente: inviare la stessa richiesta due volte produce lo stesso risultato.

DELETE – Eliminare una risorsa

```
curl -X DELETE https://jsonplaceholder.typicode.com/posts/1
```

Risposta:

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

HEADERS – Visualizzare tutte le intestazioni

L'opzione `-i` mostra **sia le intestazioni che il corpo** della risposta:

```
curl -i https://swapi.dev/api/people/1/
```

Output:

```
HTTP/1.1 200 OK
Content-Type: application/json
...
{
    "name": "Luke Skywalker",
    ...
}
```

Altri esempi utili con cURL

Obiettivo	Comando
Salvare il risultato su file	`curl -o output.json https://api.chucknorris.io/jokes/random`
Seguire i redirect automaticamente	`curl -L https://example.com`
Specificare un header personalizzato	`curl -H "Authorization: Bearer <TOKEN>" https://api.example.com/data`
Limitare il numero di redirect	`curl --max-redirs 3 -L https://example.com`
Visualizzare tempi e statistiche	`curl -w "@curl-format.txt" -o /dev/null -s https://example.com`

Esempio pratico – Chiamata API con filtro

API: `https://api.nationalize.io/?name=pietro`

```
ccurl "https://api.nationalize.io/?name=pietro"
```

Risposta:

```
"name": "pietro",
"country": [
    {
        "country_id": "IT",
        "probability": 0.85
    },
    {
        "country_id": "ES",
        "probability": 0.07
    },
    {
        "country_id": "PT",
        "probability": 0.04
    }
    ]
}
```

Debug delle richieste HTTP

Puoi usare `-v` (verbose) per visualizzare il dettaglio della comunicazione:
`curl -v https://api.chucknorris.io/jokes/random`
Questo mostrerà intestazioni di richiesta e risposta, handshake TLS, e altri dettagli utili per capire cosa accade “dietro le quinte”.