

Programmazione per il Web

Alessio Marini, 2122855

Appunti presi durante il corso di **Programmazione per il Web** nell'anno **2025/2026** del professore Mattia Samory.

Gli appunti li scrivo principalmente per rendere il corso più comprensibile **a me** e anche per imparare il linguaggio Typst. Se li usate per studiare verificate sempre le informazioni 🙏 .

Contatti:

📧 alem1105

✉ marini.2122855@studenti.uniroma1.it

September 27, 2025

Indice

1. HTTP (HyperText Transfer Protocol)	3
1.1. Metodi HTTP	4
1.1.1. PUT	4
1.1.2. GET	5
1.1.3. POST	5
1.1.4. DELETE	5
1.1.5. Altri metodi	5
1.2. Codici di Stato	5
1.2.1. 2xx Successful	6
1.2.2. 3xx Redirection	6
1.2.3. 4xx Client Error	6
1.2.4. 5xx Server Error	6
2. API - Application Programming Interface	7
2.1. API Pubbliche	7
3. JSON	9
3.1. YAML (Yet Another Markup Language)	10
4. REST (Representational State Transfer)	12
4.1. Vincoli di un sistema RESTful	13

1. HTTP (HyperText Transfer Protocol)

E' un protocollo a livello applicazione nello stack TCP / IP. La sua variante più sicura si chiama **HTTPS**.

Si basa sul concetto di **client e server**, il client è chi richiede dei servizi o risorse mentre il server è chi le offre.

User Agent (UA)


E' una qualsiasi applicazione del client che avvia una richiesta, ad esempio il browser web, un'app ecc...

Origin Server (O)

Un programma che può originare risposte autorevoli per una data risorsa, ad esempio un sito web.

Esempio di Richiesta


```
1 GET /hello.txt HTTP/1.1
2 User-Agent: curl/7.64.1
3 Host: [www.example.com](https://www.example.com)
4 Accept-Language: en, it
```

 HTTP

- La prima è la **linea di richiesta** dove viene indicato il metodo HTTP, in questo caso GET, l'URI della risorsa e la versione del protocollo.
- Vari campi di intestazione, ad esempio **User-Agent**.
- Corpo del messaggio opzionale.

Esempio di Risposta

```
1 HTTP/1.1 200 OK
2 Date: Mon, 27 Jul 2009 12:28:53 GMT
3 Server: Apache
4 Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
5 ETag: "34aa387-d-1568eb00"
6 Accept-Ranges: bytes
7 Content-Length: 51
8 Vary: Accept-Encoding
9 Content-Type: text/plain
10 Hello World! My content includes a trailing CRLF.
```

 HTTP

- In questo caso abbiamo un codice che indicato lo stato della richiesta.
- Contenuto, opzionale.

Ci sono altri due elementi che possiamo trovare in una connessione HTTP:

Intermediari

Sono altri nodi presenti tra il client e il server, ad esempio dei proxy.

Cache

E' una archivio di vecchi messaggi di risposta. Quindi se ad esempio un nodo che si trova in mezzo ad una comunicazione già conosce la risposta ad una richiesta può subito inviarla.

Per fare in modo che i client memorizzino le risposte il server deve inviare delle risposte con l'header **cacheable**.

Dispositivi come i proxy possono memorizzare risposte mentre i **tunnel** no.

1.1. Metodi HTTP

I metodi che possiamo utilizzare con il protocollo sono:

- GET
- HEAD
- POST
- PUT
- DELETE
- CONNECT
- OPTIONS
- TRACE
- PATCH

I metodi hanno delle proprietà:

- **SAFE**: Un metodo che non ha effetti collaterali sulla risorsa, agisce in sola lettura. Può comunque cambiare lo stato dei server ad esempio creando nuovi file (log). GET, HEAD, OPTIONS e TRACE sono metodi SAFE.
- **IDEMPOTENT**: Richieste identiche multiple con quel metodo hanno lo stesso effetto di una sola richiesta. Ad esempio PUT e DELETE sulle stesse risorse sono idempotenti.
- **CACHEABLE**: Sono i metodi che permettono ad una cache di memorizzare una risposta. GET, HEAD e POST sono CACHEABLE (ma non sempre).


1.1.1. PUT

Serve a creare una nuova risorsa nel server, va specificata nella richiesta. Se questa risorsa già esiste allora la sovrascrive.

Come detto prima è IDEMPOTENTE quindi qualsiasi PUT identica e successiva ad un'altra non modifica la risorsa.

Non è sicuro e non è nemmeno salvabile in cache.


```
1 PUT /course-descriptions/web-and-software-architecture
```

 HTTP

1.1.2. GET

Richiede una risorsa dal server.

```
1 GET /course-descriptions/web-and-software-architecture
```


 HTTP

E' safe dato che non modifica la risorsa, cacheable quindi non serve che il server la rispedisca ad ogni richiesta e questa memorizzazione può avvenire sotto determinate condizioni come ad esempio dei timer. Non è idempotente, quindi richieste successive anche se identiche modificano la risorsa.

1.1.3. POST

Serve ad inviare dati al server o ad aggiornare quelli già presenti.

```
1 POST /announcements/  
2 POST /announcements/{id}/comments/  
3 POST /users/{id}/email
```


 HTTP

La risposta può essere memorizzata in cache, ma non è sicuro nè idempotente.

1.1.4. DELETE

Si invia al server una richiesta per cancellare una determinata risorsa.

```
1 DELETE /courses/web-and-software-architecture
```

 HTTP

Non è safe ma è idempotente.


1.1.5. Altri metodi

- HEAD: Funziona come il GET ma non trasferisce il contenuto della richiesta, soltanto gli header.
- CONNECT: Stabilisce un tunnel verso il server indicato dalla risorsa target.
- OPTIONS: Descrive le opzioni di comunicazione per la risorsa target.
- TRACE: Esegue un test di loop-back del messaggio in lungo tutto il percorso verso la risorsa target.

1.2. Codici di Stato

Negli esempi precedenti abbiamo visto una risposta che iniziava con:

```
1 HTTP/1.1 200 OK
```

 HTTP

Il codice serve a descrivere il risultato della richiesta. Tramite questo possiamo capire, ad esempio:

- Se la richiesta ha avuto successo
- Se ci sono contenuti allegati

I codici sono formati da 3 cifre nell'intervallo 100-599, la prima cifra indica la categoria generale.

- **1xx** (Informational): La richiesta è stata ricevuta ed è in esecuzione.

- **2xx** (Successful): La richiesta è stata ricevuta, compresa ed eseguita con successo.
- **3xx** (Redirection): Sono necessarie ulteriori azioni per completare la richiesta.
- **4xx** (Client error): La richiesta contiene sintassi errata oppure non può essere soddisfatta.
- **5xx** (Server error): Il server non è riuscito a soddisfare una richiesta apparentemente valida.

1.2.1. 2xx Successful

- **200 OK** : In una richiesta GET la risposta conterrà la risorsa richiesta. In una POST la risposta conterrà qualcosa che descrive il risultato dell'azione.
- **201 Created** : La richiesta è stata soddisfatta.
- **204 No Content** : Il server ha elaborato con successo la richiesta ma non sta restituendo dati.

1.2.2. 3xx Redirection

- **301 Moved Permanently** : Questa richiesta ma anche le successive dovrebbe essere reindirizzato all'URI fornito.
- **302 Found** : Visita un'altra URL.

1.2.3. 4xx Client Error

- **400 Bad Request** : Errore del client
- **401 Unauthorized** : Serve un'autenticazione
- **403 Forbidden** : La richiesta è stata compresa dal server e quindi è valida ma l'azione richiesta non è permessa.
- **404 Not Found** : La risorsa non è stata trovata.
- **405 Method not Allowed** : Il metodo richiesto non è supportato.

1.2.4. 5xx Server Error

- **500 Internal Server Error** : Errore del server
- **501 Not Implemented** : Il metodo richiesto non è riconosciuto o il server non è in grado di soddisfare la richiesta.
- **502 Bad Gateway** : Un gateway o un proxy hanno ricevuto una risposta non valida dal server.
- **503 Service Unavailable** : Server sovraccarico o spento.
- **504 Gateway Timeout** : Il server non ha ricevuto una risposta in tempo dal server a monte.

2. API - Application Programming Interface

Un'API è la definizione delle interazioni consentite tra due parti di un software, specifica come un pezzo di codice o servizio può interagire con un altro.

Possiamo vederle come un «contratto» tra il client e il server (consumer e servizio), questa specifica:

- Richieste possibili
- Parametri delle richieste
- Valori di ritorno
- Qualsiasi formato di dato richiesto

Queste portano diversi vantaggi nell'architettura software:

- L'interfaccia da utilizzare è **esplicita**, si conoscono quindi le modalità di interazione
- Stabilisce delle regole che vanno rispettate da entrambe le parti
- La logica interna del software rimane nascosta e viene resa pubblica soltanto l'interfaccia

Esistono diverse categorie di API in base alla loro posizione e funzione:

- **API Locali**: Ad esempio quelle per i linguaggi di programmazione, come le librerie standard di Python, le API del sistema operativo o API hardware
- **API Remote (Web API)**: Interfacce di programmazione basate su protocolli di rete, tipicamente HTTP come ad esempio le API RESTful.

Un'altra distinzione è:

- **API Private**: Destinate solo a determinati utenti
- **API Pubbliche**: Disponibili anche al pubblico, si può comunque limitare o controllare l'accesso attraverso dei **API Tokens** ovvero dei codici univoci che identificano ogni utente.

Una buona API deve essere descritta e spiegata attraverso, ad esempio, una documentazione oppure un linguaggio di descrizione standardizzato.

OAS (OpenAPI Specification) è il linguaggio di descrizione leader del settore per le API moderne basate su HTTP:

- È vendor-neutral (indipendente dal fornitore) per le API remote basate su HTTP
- Rappresenta lo standard industriale per la descrizione di API
- È ampiamente adottato dalla comunità

I file OpenAPI sono spesso scritti in YAML, esempio:

```
1 openapi: 3.0.0
2 info:
3   title: An example OpenAPI document
4   description: |
5     This API allows writing down marks on a Tic Tac Toe board
6     and requesting the state of the board or of individual cells.
7   version: 0.0.1
8 paths: {} # Gli endpoint dell'API verrebbero definiti qui
```

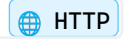
YAML

2.1. API Pubbliche

Ci permettono di capire come inviare delle richieste ad un server.

Tramite il comando `curl` possiamo effettuare richieste tramite il terminale. Se non indichiamo un metodo di default verrà utilizzato il GET.

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



Se vogliamo ad esempio effettuare un POST:

```
1 curl -X POST -H "Content-Type:  
2 application/json" -d '{"title": "Test"}'  
3 https://jsonplaceholder.typicode.com/posts
```



3. JSON

Sta per **JavaScript Object Notation**, è un formato di testo leggero usato per scambiare dati, ad esempio nelle richieste / risposte. Viene utilizzato anche per memorizzare dati in file con estensione `.json`

Esempio

```
1 {
2   "user": {
3     "firstName": "John",
4     "lastName": "Smith",
5     "age": 27
6   }
7 }
```

JSON

È un linguaggio estremamente facile sia da leggere che da scrivere anche per gli umani, inoltre è semplice anche realizzare dei programmi in grado di interpretare dei file json.

Utilizza soltanto due concetti:

- Object
- Array

Un oggetto è una collezione non ordinata di coppie (**nome, valore**) racchiusa tra parentesi graffe:

- Il nome deve essere una stringa unica fra tutte le coppie di un oggetto
- Il valore può essere un numero, stringa, booleano, array, object

```
1 {
2   "WASA": {
3     "name": "Web and Software Architecture",
4     "semester": 1
5   }
6 }
```

JSON

Un array è un insieme di valori separati da virgole e racchiusi tra parentesi quadre:

```
1 {
2   "wasaWeekdays": ["tuesday", "thursday"]
3 }
```

JSON

Esempio più completo

```
1  "anObject": {
2    "aNumber": 42,
3    "aString": "This is a string",
4    "aBoolean": true,
5    "nothing": null,
6    "anArray": [
7      1,
8      {
9        "name": "value",
10       "anotherName": 12
11     },
12     "something else"
13   ]
14 }
```

JSON

3.1. YAML (Yet Another Markup Language)

È un altro linguaggio di serializzazione pensato principalmente per gli esseri umani usato sempre per file di configurazione, archiviazione o scambio di dati. Si basa sull'indentazione (come Python)

```
1  anObject:
2    aNumber: 42
3    aString: This is a string
4    aBoolean: true
5    nothing: null
6    anArray:
7      - 1
8      - anotherObject:
9          someName: some value
10         someOtherName: 1234
11      - something else
```

YAML

YAML superset di JSON

I file JSON sono anche dei validi file YAML, infatti le parentesi graffe sono accettate per gli oggetti e le quadre per gli array. Possiamo commentare delle righe con #, oppure possiamo specificare un documento facendolo iniziare con — e finire con ...

In un file possiamo inserire più documenti.

```
1 anotherArray: [1, 2, 3]
2 anotherObject: { "city": "Rome", "country": "Italy" }
3 aStringWithColon: "COVID-19: procedure di accesso"
4 aNumeriString: "0649911"
5 aLongString: |
6   this string spans
7   more lines
```

YAML

4. REST (Representational State Transfer)

REST è uno stile architetturale pensato per trasferire **la rappresentazione** delle risorse da un componente ad un altro, ad esempio client e server.

Per risorsa intendiamo una qualsiasi informazione che può essere nominata, quindi documenti, immagini, servizi ecc... Una risorsa è un insieme di elementi o valori che possono variare nel tempo e per questo, in un dato momento, due risorse potrebbero mappare gli stessi valori, ad esempio due versioni diverse dello stesso programma.

Per **rappresentazione** della risorsa intendiamo lo stato attuale o previsto ovvero il suo valore in un particolare momento, le rappresentazioni vengono usate dai **componenti REST** per eseguire azioni sulle risorse. Una rappresentazione è composta da **dati e metadati**, il formato dei dati è noto come **media type**.

Per identificare una risorsa utilizziamo gli **URI (Uniform Resource Identifier)** ovvero una sequenza unica di caratteri, ad esempio «*http://example.com/users*»

Per dare dei nomi alle URI esistono delle **best practices**:

- Usare sostantivi per rappresentare le risorse
- Utilizzare sostantivi singolari per una singola risorsa, ad esempio «*http://example.com/users/admin*»
- Utilizzare sostantivi plurali per una collezione di risorse, ad esempio «*http://example.com/users/*»

Inoltre ci sono anche delle convenzioni:

- Utilizzare **forward slash (/)** per esprimere le gerarchie, usare il *trailing slash* solo se la risorsa non è una foglia
- Preferire i trattini agli underscore
- Utilizzare solo lettere minuscole
- Non utilizzare estensioni di file (verranno indicate negli header)
- Utilizzare la componente query per filtrare, ad esempio «*http://example.com/managed-devices/?region=USA*»


Esempi di notazione

Regola	Esempio Positivo	Esempio Negativo
Risorsa Singola (Singolare)	/users/45	/get-user/45 (Contiene un verbo)
Collezione (Plurale)	/invoices	/invoice-list
Gerarchia	/users/45/orders	/orders-from-user/45
Separazione	Preferire i trattini (-)	Evitare gli underscore (_)
Media Type	Non usare estensioni (es. .json)	/products/123.json

Le URI sono usate per identificare in modo univoco le risorse e non le azioni su di esse, infatti azioni diverse possono essere eseguite su una risorsa attraverso i metodo supportati ma sempre attraverso la stessa URI, possiamo usare ad esempio *GET*, *PUT*, *DELETE* sulla URI «<http://example.com/managed-devices/{id}>»

Esempio di richiesta in Python

```
1 import requests
2
3 base_url = " ..."
4 query_params = {
5     "country": "Italy", "min_price": 50.00,
6     "sort_by": "price_asc"
7 }
8 response = requests.Request("GET", base_url, params=query_params).prepare()
9 print(f"URI generato con Query: {response.url}")
```

 Python

requests ci aiuta ad assemblare delle richieste, noi dobbiamo soltanto fornire tutti i dati e lui ci restituisce l'uri.

4.1. Vincoli di un sistema RESTful

1. **Client - Server:** Il client si occupa della UI mentre il server dell'archiviazione dati, serve a migliorare la portabilità, la scalabilità e a separare le responsabilità.
2. **Stateless:** Ogni richiesta che fa il client deve contenere tutte le informazioni per essere compresa, non bisogna utilizzare dati già presenti sul server lasciati da richieste precedenti.

Lo stato della sessione è mantenuto interamente sul client mentre quello delle risorse sul server.

3. **Cacheable:** Il client può memorizzare e riutilizzare rappresentazioni delle risorse, la risorsa deve essere identificata come **cacheable** e nella risposta è indicato anche per quanto tempo può venire memorizzata.
4. **Uniform Interface:** Standardizzare l'interfaccia fra tutti i componenti anche andando a perdere efficienza, le interfacce presentano 4 vincoli:
 - Identificazione delle risorse
 - Manipolazione delle risorse tramite rappresentazione
 - Messaggi auto-descrittivi
 - *Hypermedia come motore dello stato dell'applicazione*, il client ha bisogno solo dell'URI iniziale, poi deve essere in grado di trovare le altre risorse tramite quello.

Ad esempio le API REST basate su HTTP utilizzano metodi standard come GET, POST, PUT e gli URI per identificare le risorse.

5. **Layered System:** In una comunicazione non troviamo soltanto client e server ma anche altri dispositivi come proxy, questi componenti possono agire sia da client che da server infatti possono inoltrare sia risposte che richieste. Ogni componente però può osservare soltanto i dispositivi adiacenti a lui e non cosa c'è oltre.

Metodi HTTP vs REST

Metodo	Funzione	Corrispondenza CRUD
GET	Recupera una risorsa o una collezione.	Read
POST	Crea una nuova risorsa in una collezione.	Create
PUT	Sostituisce completamente una risorsa esistente.	Update/Replace
DELETE	Rimuove una risorsa specifica.	Delete
PATCH	Applica modifiche parziali a una risorsa.	Update/Modify

Gli URI identificano la **risorsa** che vogliamo manipolare mentre i metodi identificano l'**azione** che vogliamo eseguire sulla risorsa.

Per CRUD intendiamo le operazioni di base **Create, Read, Update, Delete**

Codici di Successo:

- **200 OK:** Utilizzato in richieste *PUT, GET, PATCH o DELETE* eseguite con successo.
- **201 Created: MUST** Viene restituito da un *POST* effettuato con successo, dovrebbe restituire anche la risorsa appena creata.
- **204 No Content:** L'azione ha avuto successo ma non ci sono dati nella risposta.

Codici di errore client:


- **400 Bad Request:** Richiesta non soddisfacibile per un errore del client, di solito errori di formattazione.

- **401 Unauthorized:** Manca l'autenticazione.
- **403 Forbidden:** Utente autenticato ma non ha i permessi per svolgere quell'azione.
- **404 Not Found:** Risorsa non esistente.

Esempi


- **Ottenere dati:** Ottenere tutti i dati di prodotti in vendita

1 **GET** /products

 HTTP

- **Creare una nuova risorsa:** Aggiungere un nuovo utente

1 **POST** /users

 HTTP

Il corpo della richiesta conterrà i dati dell'utente. Come risposta ci aspettiamo la nuova risorsa appena creata e l'URI per accedervi.

Se un'azione però non è mappabile nelle operazioni CRUD, come ad esempio «*Pubblica un documento*» o «*Cambia una password*» possiamo seguire due approcci:

- **Modellare l'azione come una risorsa**, ad esempio pubblicare un documento potrebbe diventare:

1 **POST** /documents/{id}/publish

 HTTP

- **Utilizzare PATCH**, serve ad aggiornare un attributo di stato, ad esempio pubblicare un documento aggiornando lo stato:

1 **PATCH** /documents/{id}

 HTTP

E inseriamo come corpo `{'status': 'published'}`

Il secondo approccio è preferito perché mantiene un modello più puro dove si agisce soltanto sullo stato della risorsa.