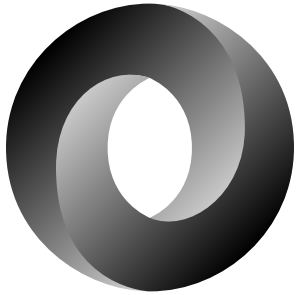
HTTP, JSON & XMLHttpRequest

Niccolò Maltoni
niccolo.maltoni@diennea.com

Cos’è JSON



JSON (*JavaScript Object Notation*) è un formato **testuale** per lo scambio di dati.

- La sintassi è ispirata alla notazione letterale degli oggetti in JavaScript (di qui il nome), ma è di fatto un formato completamente **indipendente** dal linguaggio.
- Molto più **leggibile** e **compatto** rispetto a formati più vecchi come XML.
- Molto facile da generare e consumare in quasi tutti i linguaggi di programmazione.
- Si basa su **strutture dati universali** e facilmente interscambiabili.

Per tutti questi motivi, è diventato lo standard de facto per lo scambio di dati su Internet, soppiantando XML.

Strutture dati in JSON

Dichiaratamente da specifica, JSON è basato solamente su due strutture:

- Un **insieme di coppie nome/valore**.
  - In diversi linguaggi, questo è realizzato come un *oggetto*, un *record*, uno *struct*, un *dizionario*, una *tabella hash*, un *elenco di chiavi* o un *array associativo*.
- Un **elenco ordinato di valori**.
  - Nella maggior parte dei linguaggi questo si realizza con un *array*, un *vettore*, un *elenco* o una *sequenza*.

I **valori** possono essere *stringhe*, *numeri*, *booleani*, *null*, oppure altri *oggetti* o *array*.

Non sono ammessi altri tipi.

> ⚠ **Attenzione**
>
> JSON è un **formato di dati**, non un linguaggio di programmazione!
>
> Non supporta funzioni, variabili, commenti o qualsiasi altra funzionalità di un linguaggio di programmazione.

Strutture dati in JSON: Oggetti

Un **oggetto** è una struttura dati che descrive una sequenza **non ordinata** di coppie nome-valore.

- Un oggetto inizia con parentesi graffa sinistra `{` e finisce con parentesi graffa destra `}`.
- Ogni nome è seguito da due punti `:`, che separano il nome dal valore.
  - Il nome è sempre una stringa!
- Ogni coppia nome-valore è separata da virgola `,`.

```
{
  "id": 1,
  "nome": "Alice",
  "email": "alice@example.com",
  "attivo": true
}
```

A differenza di JavaScript, non possono possedere metodi.

Strutture dati in JSON: Array

Un **array** è una struttura dati che descrive una sequenza **ordinata** di valori.

- Un array comincia con parentesi quadra sinistra `[` e finisce con parentesi quadra destra `]`.
- I valori sono separati da virgola `,`.

```
[
  "lunedi",
  "martedi",
  "mercoledi",
  "giovedi",
  "venerdi"
]
```

- I valori non devono essere dello stesso tipo!
- Array e oggetti possono essere **annidati** l'uno dentro l'altro:

```
{
  "utente": { "id": 1, "nome": "Alice" },
  "tasks": [
    { "id": 1, "titolo": "Task 1" },
    { "id": 2, "titolo": "Task 2" }
  ]
}
```

Valori JSON

Un valore può essere:

- una **stringa** tra virgolette `"`

```
"hello world"
```

- un numero (in formato intero o decimale)

```
42
3.14
```

- un valore booleano `true` o `false`
- un valore `null`,
- un **oggetto** (coppie chiave-valore racchiuse tra `{}`),
- un **array** (valori ordinati racchiusi tra `[]`).

Non sono supportati i commenti (se non nell'estensione JSONC, poco comune).

Formattazione

Gli spazi bianchi (*spazi*, *tabulazioni*, *newline*) sono ignorati e possono essere usati per formattare il JSON.

```
{"id":"0001","type":"donut","name":"Cake","ppu":0.55,"batters":{"batter":[{"id":"1001","type":"Regular"},{"id":"1002","type":"Chocolate"},{"id":"1003","type":"Blueberry"},{"id":"1004","type":"Devil's Food"}]},"topping":[{"id":"5001","type":"None"},{"id":"5002","type":"Glazed"},{"id":"5005","type":"Sugar"},{"id":"5007","type":"Powdered Sugar"},{"id":"5006","type":"Chocolate with Sprinkles"},{"id":"5003","type":"Chocolate"},{"id":"5004","type":"Maple"}]}
```

```
{
  "id": "0001",
  "type": "donut",
  "name": "Cake",
  "ppu": 0.55,
  "batters": {
    "batter": [
      { "id": "1001", "type": "Regular" },
      { "id": "1002", "type": "Chocolate" },
      { "id": "1003", "type": "Blueberry" },
      { "id": "1004", "type": "Devil's Food" }
    ]
  },
  "topping": [
    { "id": "5001", "type": "None" },
    { "id": "5002", "type": "Glazed" },
    { "id": "5005", "type": "Sugar" },
    { "id": "5007", "type": "Powdered Sugar" },
    { "id": "5006", "type": "Chocolate with Sprinkles" },
    { "id": "5003", "type": "Chocolate" },
    { "id": "5004", "type": "Maple" }
  ]
}
```

Essendo un linguaggio molto sintetico, la leggibilità sta nella capacità di identare il file in modo coerente.

JSON in JavaScript

Quando JavaScript si trova a dover **scambiare dati** (con un server, con localStorage, etc.), è comune usare JSON come formato di *serializzazione*.

In questi casi, JSON non sarà altro che una stringa.

Abbiamo due metodi globali per convertire tra oggetti JavaScript e stringhe JSON:

- `JSON.stringify()`: converte un oggetto JavaScript in una stringa JSON (*serializzazione*)
- `JSON.parse`: converte una stringa JSON in un oggetto JavaScript (*deserializzazione*)

`JSON.stringify`: oggetto → stringa

`JSON.stringify` **serializza** un oggetto JavaScript in una stringa JSON:

```
const task = {
  id: 1,
  text: 'Imparare JSON',
  done: false,
  dueDate: '2025-12-25'
};

const jsonString = JSON.stringify(task);
console.log(jsonString);
// '{"id":1,"text":"Imparare JSON","done":false,"dueDate":"2025-12-25"}'

console.log(typeof jsonString);  // 'string'
```

`JSON.stringify` **esclude** automaticamente *proprietà con valore* `undefined` e *funzioni*:

```
const obj = {
  nome: 'Alice',
  age: 30,
  sayHi() { alert("Hello"); }, // ignorato
  something: undefined         // ignorato
};

console.log(JSON.stringify(obj)); // '{"nome":"Alice","age":30}'
```

`JSON.stringify`: opzioni

La funzione `JSON.stringify` ha la seguente firma:

`JSON.stringify(value, replacer, space)`

- `value`: l'oggetto da serializzare
- `replacer`: (opzionale) array di chiavi da includere — es. `JSON.stringify(user, ['id', 'nome'])`
- `space`: (opzionale) numero di spazi per l'indentazione — es. `JSON.stringify(user, null, 2)`

```
const user = { id: 1, nome: 'Alice', email: 'alice@example.com', password: 'secret' };

// Solo le chiavi nell'array replacer
JSON.stringify(user, ['id', 'nome']);
// '{"id":1,"nome":"Alice"}'

// Indentazione con 2 spazi (utile per debug e file salvati)
JSON.stringify(user, null, 2);
// {
//   "id": 1,
//   "nome": "Alice",
//   ...
// }
```

`JSON.parse`: stringa → oggetto

`JSON.parse` **deserializza** una stringa JSON in un oggetto JavaScript:

```
// Stringa JSON (da API, localStorage, file)
const jsonString = '{"id":1,"text":"Imparare JSON","done":false}';

const task = JSON.parse(jsonString);
console.log(typeof task);      // 'object'
console.log(task.id);          // 1
console.log(task.text);        // 'Imparare JSON'
console.log(task.done);        // false
```

Attenzione: se il JSON non è valido, `JSON.parse()` genera un errore (`SyntaxError`)!

Errori comuni di parsing JSON

```
// Errore 1: apici singoli (JSON richiede doppi)
JSON.parse("{'id': 1}");  // SyntaxError

// Errore 2: key senza virgolette
JSON.parse('{"id": 1, name: "Alice"}');  // SyntaxError

// Errore 3: undefined non è valido
JSON.parse('{"id": 1, value: undefined}');  // SyntaxError

// Errore 4: stringa non è JSON valido
JSON.parse('{"id": 1, "testo": "ciao');  // SyntaxError (virgola finale manca)

// Corretto
JSON.parse('{"id": 1, "name": "Alice"}');  // OK
```

Esercizio

Ricettario persistente con JSON

Estendi il ricettario delle lezioni precedenti per renderlo **persistente** usando JSON:

1. Ogni volta che il ricettario viene modificato (nuova ricetta, ingrediente o passo), serializza `ricettario.ricette` con `JSON.stringify` e salva in `localStorage` con chiave `"ricettario"`.
2. Al caricamento della pagina, leggi `localStorage`, deserializza con `JSON.parse` e, se esiste un valore salvato, sostituisci le ricette predefinite con quelle caricate.
3. Stampa in console la stringa JSON e l'oggetto ricostruito per verificare che il ciclo di serializzazione e deserializzazione sia corretto.
4. **Attenzione**: `JSON.stringify` **omette automaticamente** le proprietà con valore `undefined`. Questo significa che dopo il round-trip localStorage, le quantità opzionali (che nel codice sono `undefined`) diventeranno `null` anziché `undefined`. Modifica `mostraRicetta()` per accettare sia `null` che `undefined` come "nessuna quantità".

JSON in rete

Fin qui abbiamo usato JSON per **serializzare dati localmente** (es. in `localStorage`).

Ma JSON non è nato per questo: è uno standard pensato per lo **scambio di dati tra sistemi diversi** su rete.

Oggi JSON è il formato dominante nelle API web: ogni volta che un'app carica dati da un server (e.g. una lista di prodotti, il meteo, i tuoi messaggi) molto probabilmente sta ricevendo una risposta in JSON.

Questo scambio avviene su **HTTP**, il protocollo fondamentale del web. Per capire come funziona davvero, dobbiamo prima capire su quale infrastruttura viaggiano questi dati.

Internet e il web

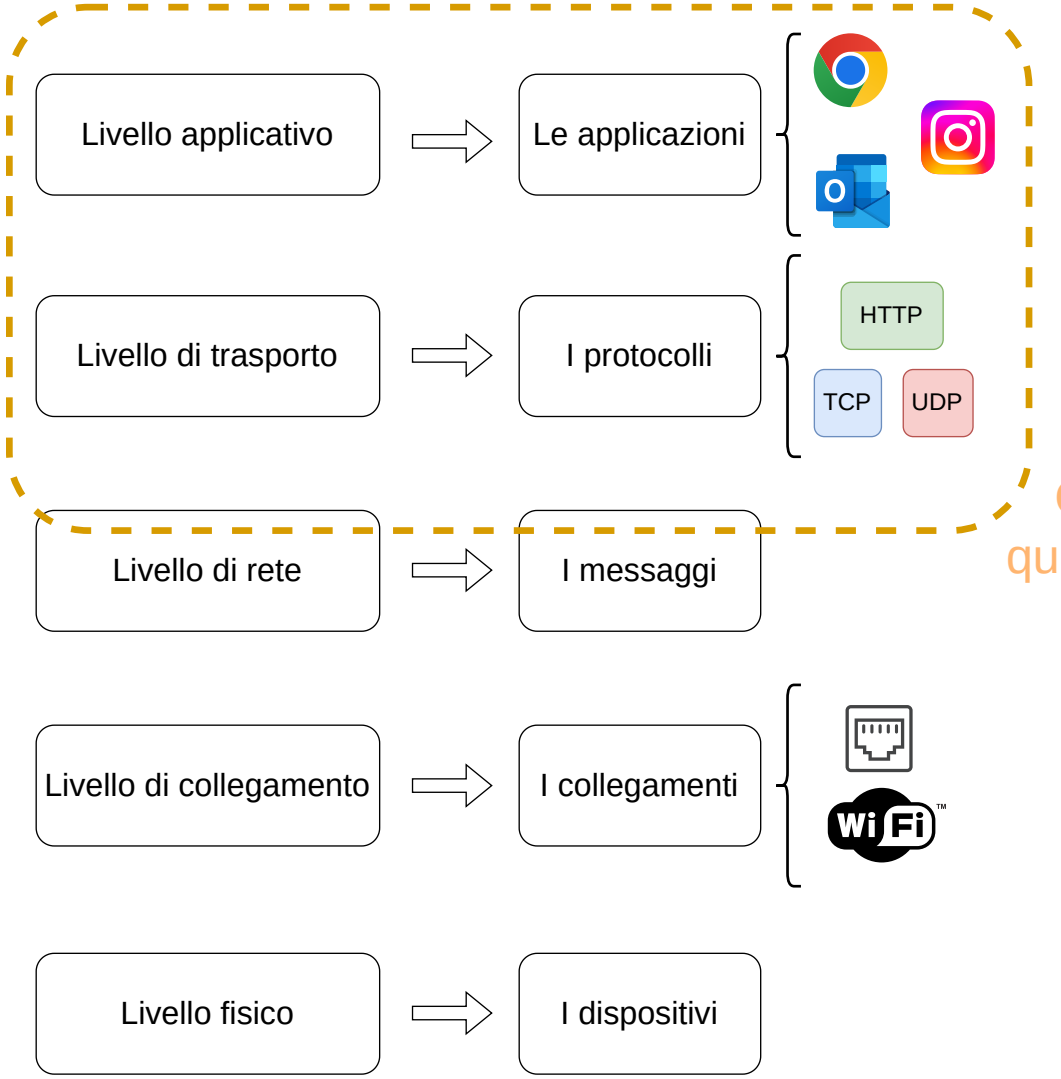
Internet è una rete globale di computer interconnessi che comunicano tramite **protocolli standard**.

- **Client**: dispositivi degli utenti (browser, app, smartphone).
- **Server**: macchine che forniscono risorse e servizi.
- **Router**: dispositivi che instradano i pacchetti tra i nodi della rete.
- **ISP** (*Internet Service Provider*): fornitori che connettono gli utenti alla rete globale.

Funzionamento di base:

1. I dati vengono suddivisi in piccoli **pacchetti**.
2. I pacchetti viaggiano attraverso una serie di router.
3. Vengono ricomposti a destinazione.

Che cos’è Internet?



Ci concentriamo su questa parte dello stack

Cos’è un protocollo

Un **protocollo** definisce il formato e l’ordine dei messaggi scambiati tra due entità in comunicazione, e le azioni da intraprendere alla ricezione.

Protocolli umani:

- “Che ore sono?”
- “Ho una domanda.”
- Presentazioni formali

Prevedono messaggi specifici e azioni determinate al ricevimento.

Protocolli di rete:

- Coinvolgono hardware e software
- Tutta la comunicazione su Internet è governata da protocolli
- Esempi: **UDP**, **TCP**, **IP**, **HTTP**

I protocolli Internet sono pubblicati dall’**IETF** tramite documenti **RFC** (*Request For Comments*).

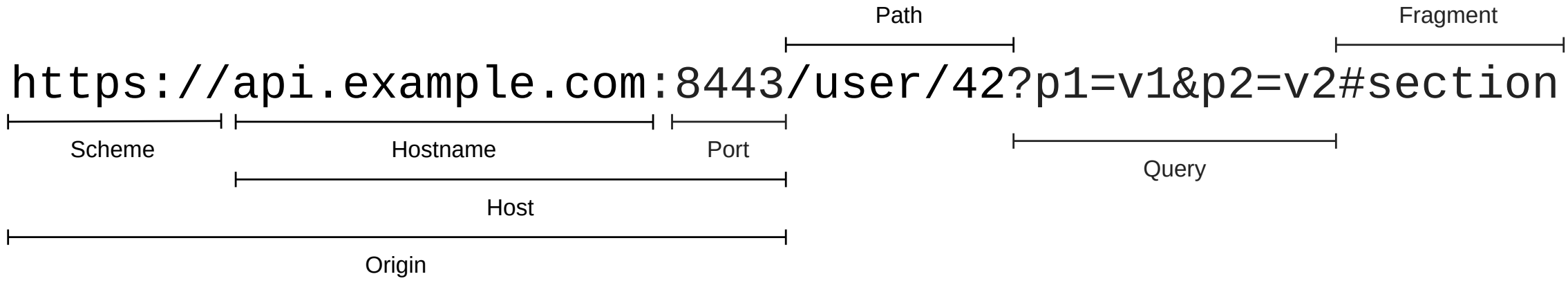
HTTP

HTTP (*HyperText Transfer Protocol*) è il protocollo a **livello di applicazione** usato nel web.

- Nato per trasferire pagine di **ipertesto** (testo con link), oggi trasporta qualsiasi tipo di dato.
- Adotta un **modello client/server**:
  - **Client**: attiva la connessione e richiede una risorsa.
  - **Server**: accetta la connessione, elabora la richiesta, risponde. Poi chiude la connessione.
- È **indipendente dal formato** dei dati trasmessi: funziona per HTML, JSON, immagini, file binari, etc.
- È **stateless**: il server non mantiene memoria delle richieste precedenti. Ogni richiesta parte da zero.

Anatomia di un URL

Ogni risorsa su web è identificata da un **URL** (*Uniform Resource Locator*).



Parte	Esempio	Significato
schema	`https://`	Protocollo usato
host	`api.example.com:8443`	Nome del server (**hostname** + **porta**)
path	`/user/42`	Percorso della risorsa
query	`?p1=v1&p2=v2`	Parametri aggiuntivi (coppie `chiave=valore`)
fragment	`#section`	Àncora nella pagina (gestita solo dal browser)

Oggetto URL

Prima di costruire richieste, è utile sapere che JavaScript ha una classe built-in **URL** per lavorare con gli indirizzi in modo sicuro.

```
const url = new URL('https://api.example.com/utenti');

console.log(url.protocol);  // 'https:'
console.log(url.host);      // 'api.example.com'
console.log(url.pathname);  // '/utenti'
```

Questo ci è molto utile per costruire URL dinamicamente, ad esempio aggiungendo query parameters:

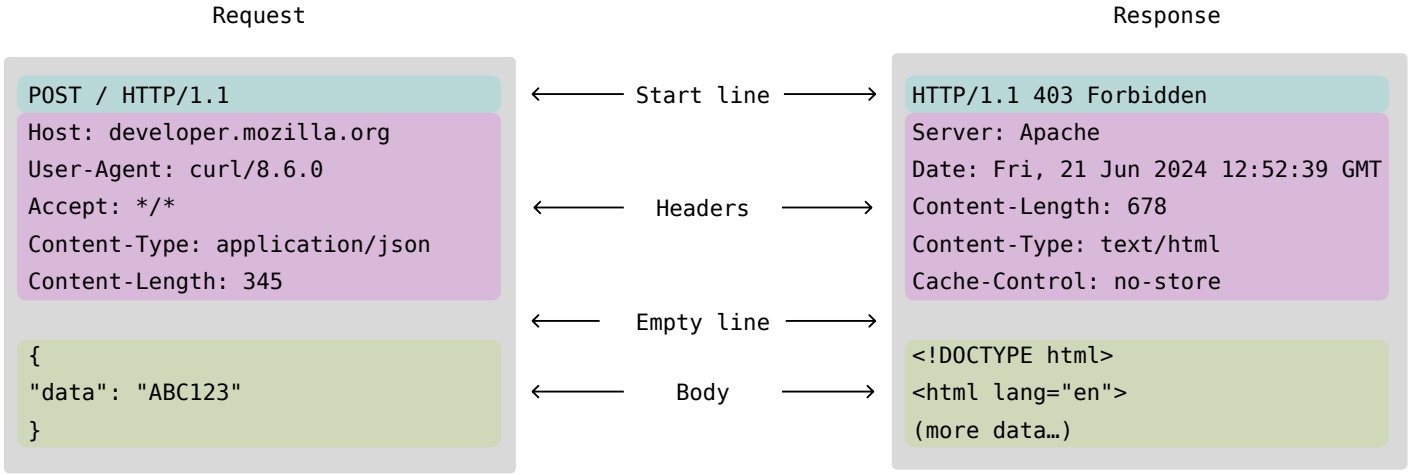
```
const url = new URL('https://api.example.com/ricette');

url.searchParams.set('categoria', 'dolci');
url.searchParams.set('q', 'torta al cioccolato');

console.log(url.toString());
// https://api.example.com/ricette?categoria=dolci&q=torta+al+cioccolato

// L'encoding dei caratteri speciali è automatico!
xhr.open('GET', url);  // oppure: fetch(url)
```

Richiesta e risposte HTTP



Entrambe la richiesta e la risposta HTTP condividono una struttura simile:

- una riga iniziale che descrive la **versione** del protocollo e:
  - per la richiesta: il **metodo** e il **path**
  - per la risposta: il **codice di stato** e il **messaggio**
- una serie di **header** che forniscono informazioni aggiuntive sulla richiesta o sulla risposta
- un **body** opzionale che contiene i dati (es. JSON, HTML, immagini, etc.)

Metodi HTTP

Il protocollo HTTP implementa diversi metodi, di cui i più utilizzati sono `GET` e `POST`.

I **metodi HTTP** indicano l'**azione** che il client vuole compiere sulla risorsa identificata dall'URL.

Nonostante siano talvolta usati in modo improprio, i metodi HTTP hanno significati ben definiti e implicano certe proprietà che dovrebbero generalmente essere rispettate:

- **sicurezza**: un metodo è sicuro (*safe*) se non modifica lo stato del server.
- **idempotenza**: un metodo è idempotente (*idempotent*) se eseguirlo più volte ha lo stesso effetto di eseguirlo una sola volta.
- **cacheability**: un metodo è compatibile con la cache (*cacheable*) se le risposte possono essere memorizzate e riutilizzate.

Inoltre, solamente `PUT`, `POST` e `PATCH` prevedono un **body** nella richiesta.

Metodi HTTP

Metodo	Operazione	*Safe*	*Idempotent*	*Cacheable*	Body
`GET`	Legge una risorsa	Sì	Sì	Sì	No
`HEAD`	Legge solo gli header	Sì	Sì	Sì	No
`OPTIONS`	Interroga il server	Sì	Sì	No	No
`TRACE`	Esegue loopback	Sì	Sì	No	No
`DELETE`	Elimina una risorsa	No	Sì	No	No
`PUT`	Aggiorna una risorsa	No	Sì	No	Sì
`POST`	Crea una risorsa	No	No	A volte	Sì
`PATCH`	Aggiorna parziale	No	No	A volte	Sì
`CONNECT`	Apre un tunnel col server	No	No	No	No

HTTP status code

I codici di stato in una risposta HTTP comunicano l'esito della richiesta. Sono numeri a tre cifre, dove la prima cifra indica la categoria:

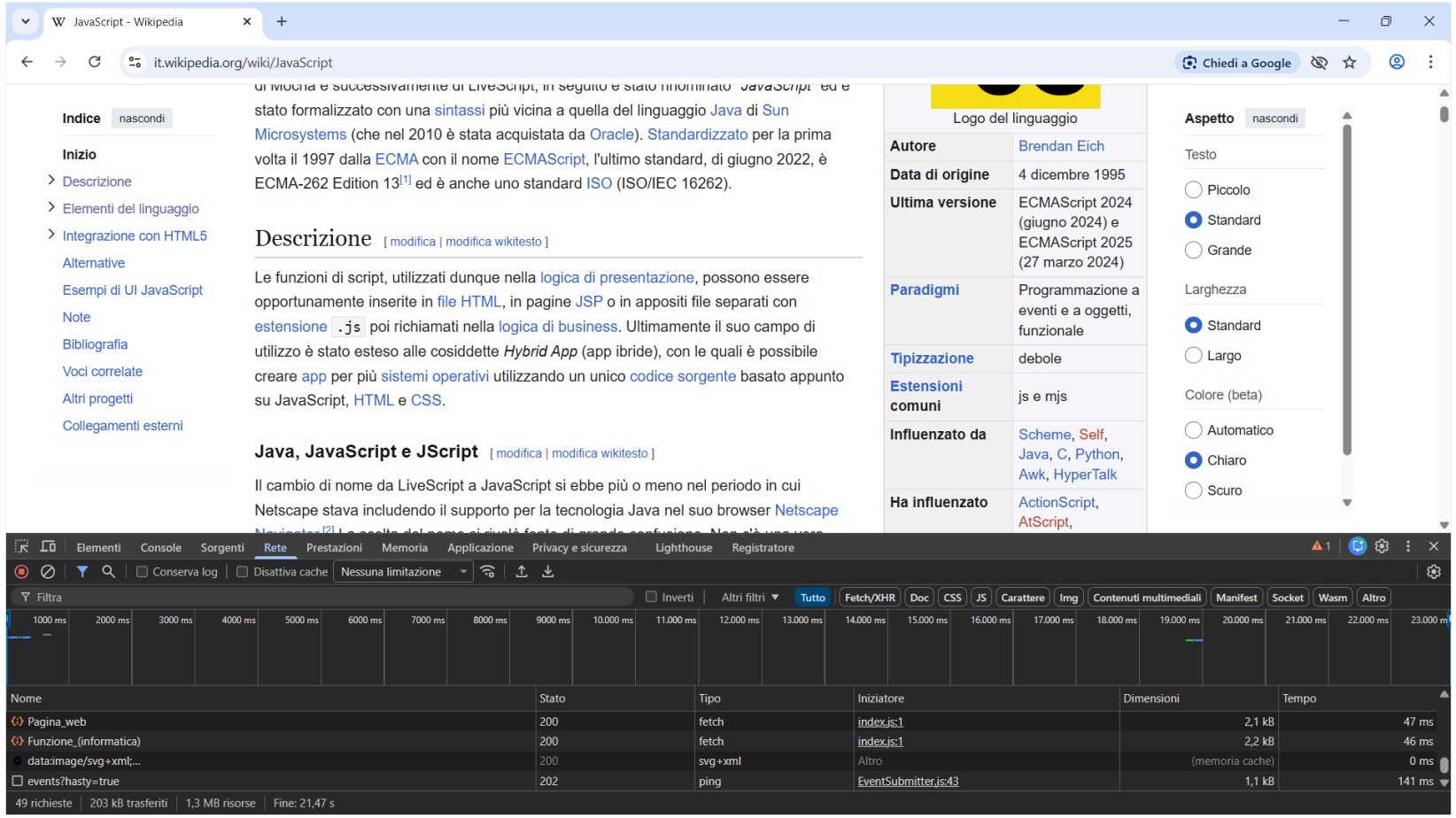
Categoria	Range	Significato
Informazione	1xx	Richiesta ricevuta, elaborazione
Successo	2xx	Richiesta OK
Redirezione	3xx	Risorsa spostata
Errore client	4xx	Errore nella richiesta
Errore server	5xx	Errore del server

HTTP status code comuni

Status	Messaggio	Significato	Azione tipica
`200`	OK	Successo, risorsa restituita	Processa dati
`201`	Created	Risorsa creata (POST)	Conferma creazione
`204`	No Content	Successo, nessun body	DELETE riuscito
`400`	Bad Request	Dati malformati nel body	Valida input
`401`	Unauthorized	Non autenticato (login richiesto)	Richiedi credenziali
`403`	Forbidden	Autenticato ma non autorizzato	Mostra accesso negato
`404`	Not Found	Risorsa non esiste	Mostra errore 404
`500`	Internal Server Error	Errore server	Riprova più tardi
`503`	Service Unavailable	Server offline	Riprova più tardi

DevTools: vedere HTTP in azione

Il browser raccoglie tutte le richieste HTTP che fa per caricare una pagina e le mostra in tempo reale. Lo possiamo osservare dai DevTools:



Cos'è `XMLHttpRequest`

`XMLHttpRequest` (abbreviato **XHR**) è l'API originaria per fare richieste HTTP da JavaScript.

- Nata oltre **20 anni fa**, è stata la base di tutto quello che chiamiamo *AJAX*.
  - Rivoluzionò il web permettendo di caricare dati in background **senza ricaricare la pagina**.
- Nonostante il nome contenga "XML", funziona con qualsiasi formato: JSON, testo, binario.
- Oggi è largamente sostituita dalle `fetch` API, più moderne e leggibili.
- Ancora supportata da tutti i browser, ma **sconsigliata per codice nuovo**.

`XMLHttpRequest`: API

Configurazione e invio:

```
const xhr = new XMLHttpRequest();

xhr.open(method, url);       // Configura richiesta (GET, POST, etc.)
xhr.send(body);              // Invia la richiesta (body opzionale)
xhr.abort();                 // Cancella richiesta in corso
xhr.setRequestHeader(k, v);  // Aggiunge un header custom
xhr.timeout = 5000;          // Timeout in ms (genera evento 'timeout' se scade)
```

Proprietà della risposta e eventi:

```
xhr.status;                // Codice HTTP (200, 404, 500, ...)
xhr.statusText;            // Testo dello status ("OK", "Not Found", ...)
xhr.response;              // Body della risposta (nel formato di responseType)
xhr.responseType = 'json'; // Parsing automatico: xhr.response sarà già un oggetto

xhr.onload;                // Risposta arrivata (anche con status 4xx/5xx!)
xhr.onerror;               // Errore di rete (non status HTTP)
xhr.onprogress;            // Durante il download (utile per progress bar)
```

`XMLHttpRequest`: GET request

Ecco come fare una **GET request con XHR**:

```
1  const xhr = new XMLHttpRequest();                                          ①
2  xhr.open('GET', 'https://jsonplaceholder.typicode.com/todos/1');
3  xhr.responseType = 'json';
4
5  xhr.onload = () => {                                                       ②
6    if (xhr.status === 200) {
7      console.log('Todo:', xhr.response.title);
8    } else {
9      console.error('Errore HTTP:', xhr.status);
10   }
11 };
12
13 xhr.onerror = () => console.error('Errore di rete');                      ③
14
15 xhr.send();                                                               ④
```

① Creiamo l'istanza XHR e configuriamo il GET (impostando `responseType` **prima** di inviare).

② `onload`: gestisce la risposta, indipendentemente dallo status.

③ `onerror`: gestisce errori di rete (non è uno status HTTP).

④ `send()`: invia la richiesta (body vuoto).

`XMLHttpRequest`: POST request

Ecco come fare una **POST request con XHR** per inviare JSON:

```
1  const xhr = new XMLHttpRequest();                                          ①
2  xhr.open('POST', 'https://jsonplaceholder.typicode.com/todos');
3  xhr.setRequestHeader('Content-Type', 'application/json');
4  xhr.responseType = 'json';
5
6  xhr.onload = () => {                                                       ②
7    if (xhr.status === 201) {
8      console.log('Nuova todo creata con ID:', xhr.response.id);
9    } else {
10     console.error('Errore HTTP:', xhr.status);
11   }
12 };
13
14 const body = JSON.stringify({                                            ③
15   title: 'Nuovo task',
16   completed: false,
17   userId: 1
18 });
19 xhr.send(body);                                                           ④
```

① Creiamo l'istanza XHR, richiesta POST, header e `responseType` (dopo `open`, prima di `send`).

② `onload`: verifica status 201 (Created).

③ Prepariamo il body: `JSON.stringify` serializza i dati.

④ `send(body)`: invia il JSON al server.