boolean

Emoji Stories

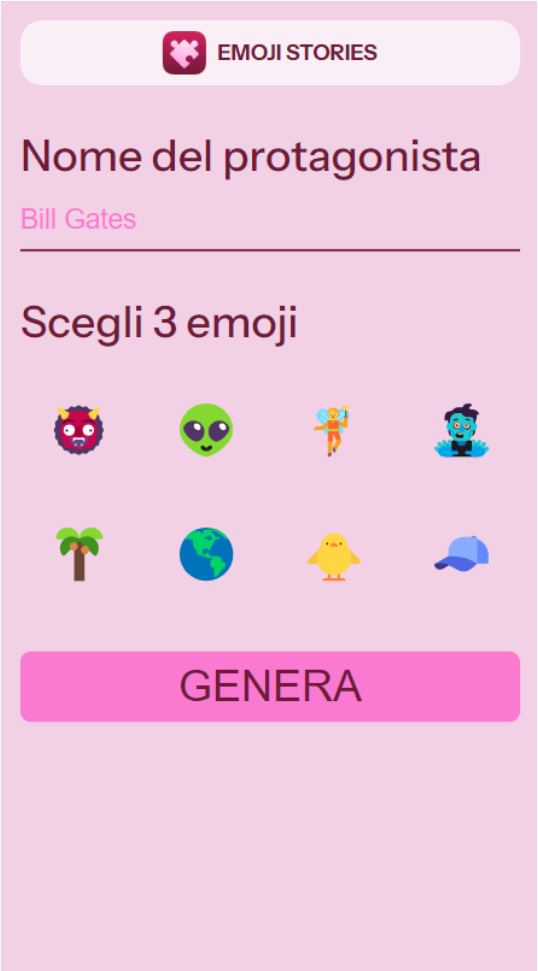
"👩‍💼 Imparara a programmare"

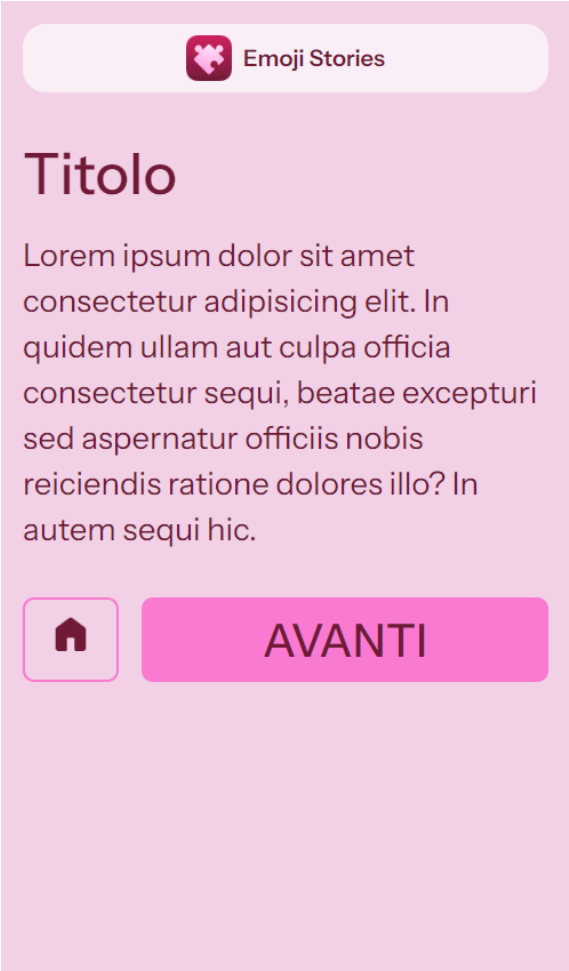
Emoji Stories

Impariamo i concetti base della programmazione e applichiamoli in JavaScript per sviluppare la logica della nostra app.

Contenuti:

- intro alla programmazione
- variabili e dati
- le funzioni
- strutture condizionali e iterative
- eventi
- API e Chiamate AJAX





JS

JavaScript

intro alla programmazione

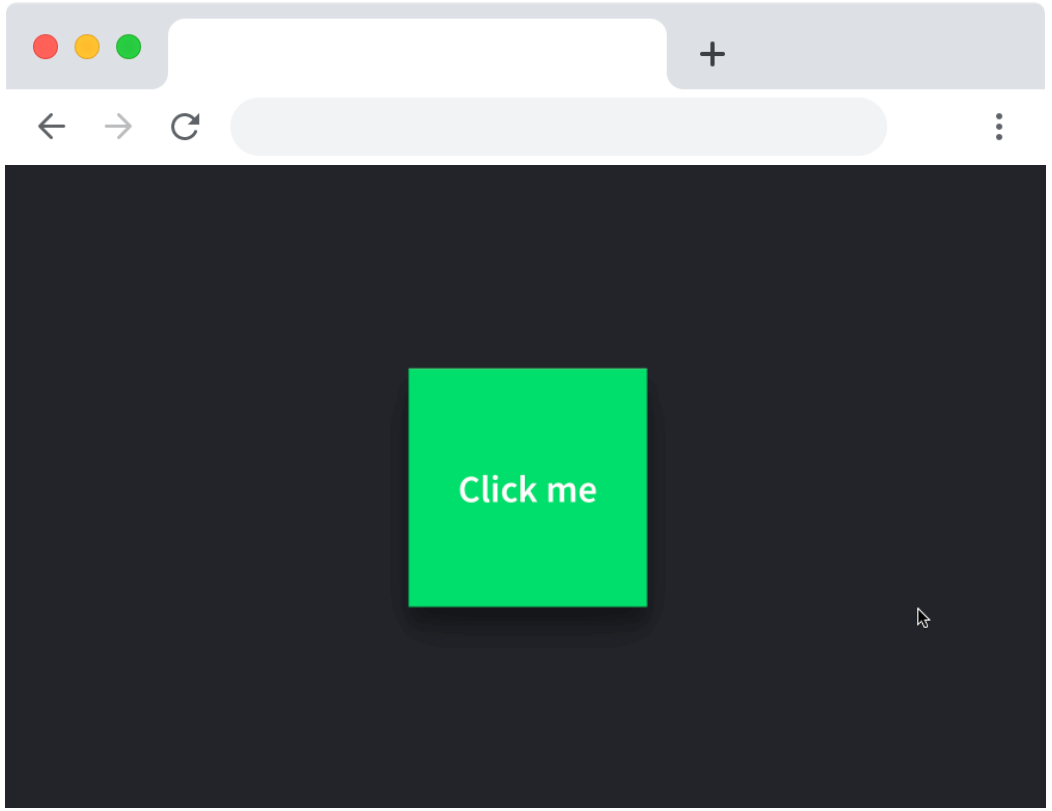
JavaScript

Per dare dinamicità alle pagine web

JavaScript consente di:

- accedere e modificare il contenuto di una pagina HTML
- reagire ad eventi generati dall'utente (ad es. il click del mouse)
- richiamare dati da fonti esterne (API)

... e molto altro!



JavaScript

JS

Hello JS!

Per aggiungere del codice JavaScript ad una pagina web è necessario creare **un file con estensione .js** e collegarlo al file html inserendo un tag **<script>** alla fine del <body>.

index.html

```html
<html>
<head>
</head>
<body>

  <!-- Dopo tutti gli altri contenuti -->

  <script src="script.js"></script>

</body>
</html>
```

script.js

```javascript
// qui va il codice Javascript
console.log('Hello World!');
```

JS

JavaScript

le variabili

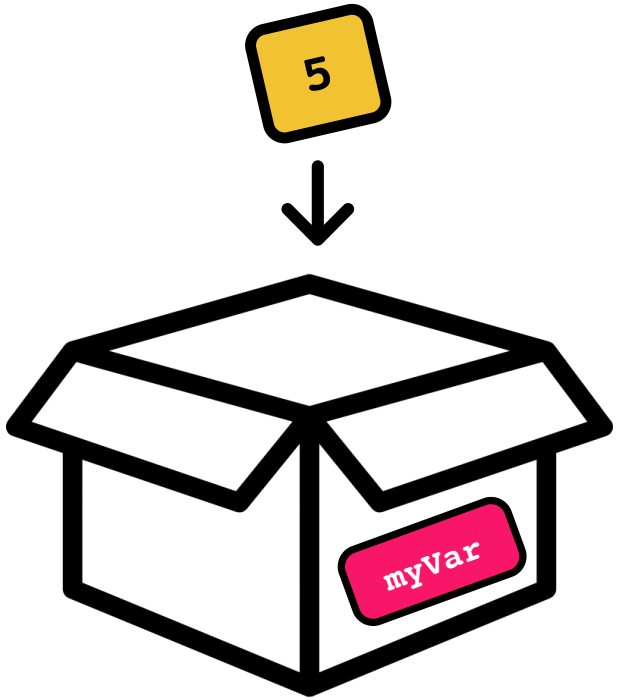
Variabili

Come possiamo conservare dei dati?

Una variabile è uno spazio di memoria riservato per contenere un valore.

Quando vogliamo richiamare una variabile si fa riferimento al suo nome per accedere al valore memorizzato all'interno.

È come una scatola con una etichetta sopra!




```
// dichiarazione di una variabile
const myVar = 5;

// usi successivi (senza 'const' davanti)
console.log(myVar)
```

Variabili

Dichiarazione

Le variabili in **JavaScript** sono dichiarate attraverso una specifica **parola chiave** seguita dal **nome della variabile**.

Queste parole chiave sono:

- ***const*** per dichiarare variabili che non possono essere modificate una volta inizializzate
- ***let*** per variabili riassegnabili

```
// dichiarazione variabile costante
const myVar1 = 5;

// dichiarazione variabile riassegnabile
let myVar2 = 5;
```

Naming delle variabili
Il nome di una variabile e completamente arbitrario, ma ci sono alcune regole da seguire: il nome deve contenere solo lettere, numeri, simboli $ e _ (underscore), il primo carattere non può essere un numero, le variabili in JavaScript sono case sensitive.

I Tipi Variabili

Il codice JavaScript manipola dei valori ed ogni valore appartiene ad un Tipo.

Alcune tipologie di dati:

Number: Numero

String: Stringa, sequenza di caratteri

Boolean: true|false

Array: liste di valori

Objects: struttura fatta di proprietà e valori che rappresentano oggetti della realtà

Nodes: elementi HTML presi dalla pagina

JS

JavaScript

Interpolazione

La via tradizionale

Concatenazione con operatore "+"

Spesso si presenta l'esigenza di mescolare dei pezzi di testo (stringhe) statici ad alcuni pezzi dinamici.

L'operatore "+" permette di farlo ma con qualche seccatura, tra cui la gestione degli spazi vuoti:

```
1 const firstName = 'Marco';
2 const lastName = 'Lancellotti';
3 const job = 'Web Developer'
4
5 const presentazione = 'Ciao, mi chiamo ' + firstName + ' ' + lastName + ' e lavoro come ' + job;
6
7 // RISULTATO: "Ciao mi chiamo Marco Lancellotti e lavoro come Web Developer";
```

Interpolazione

Stringhe e variabili

Le specifiche ES6 ci permettono di mescolare testo fisso e variabili con una sintassi molto comoda.

Basterà scrivere la nostra frase tra backticks (`) e racchiudere le variabili in ${}!

```
1 const firstName = 'Marco';
2 const lastName = 'Lancellotti';
3 const job = 'Web Developer'
4
5 const presentazione = `Ciao, mi chiamo ${firstName} ${lastName} e lavoro come ${job}`;
6
7 // RISULTATO : "Ciao mi chiamo Marco Lancellotti e lavoro come Web Developer";
```

Template Literals

HTML e variabili!

Inoltre, ci permettono di inserire anche dei veri e propri "template" di HTML

```
const firstName = 'Marco';
const lastName = 'Lancellotti';
const job = 'Web Developer';


const myHTML = `
    <div class="miaClasse">
        <h1>Titolo</h1>
        <p>`Ciao, mi chiamo ${firstName} ${lastName} e lavoro come ${job}`</p>
    </div>
`;
```

Dove trovo il backtick nella tastiera? 🤔

Su Mac:

- digitare **Option** (⌥) **+ ** (backspace)

Su Windows:

- digitare **Alt + 96** (dal tastierino numerico)

JS

JavaScript

JS + HTML

JavaScript e l'HTML

Accedere agli elementi della pagina

Ci sono diversi metodi per ritrovare specifici elementi all'interno del documento HTML, in particolare è possibile recuperare un elemento e salvare il suo riferimento in una variabile.

Alcuni metodi per accedere agli elementi:

- **document.querySelector():** recupera un singolo elemento sulla base di un selettore CSS
- **document.querySelectorAll():** recupera una lista di elementi sulla base di un selettore CSS

Naturalmente, possiamo conservare gli elementi HTML nelle variabili!

```
1 const title = document.querySelector('h1');
```

DOM Element Object

Cosa si può fare su un Element?

Più di 80 operazioni, vediamone alcune:

- element.innerText: permette di leggere o modificare il contenuto testuale di un elemento
- element.className: permette di leggere o stabilire la classe css di un elemento
- element.classList.add: permette di aggiungere dinamicamente classi css a un elemento
- element.classList.remove: permette di rimuovere dinamicamente classi css da un elemento
- element.classList.contains: permette di sapere se una classe è presente in un elemento

JS

JavaScript

Gli array

Array

Salviamo più dati assieme

L'array ci permette di salvare liste di valori, invece che un singolo valore!

Gli elementi di un array
si elencano tra parentesi quadre
e si separano con una virgola.

```
// Questo array contiene tre elementi di tipo stringa
const iscritti = ["Luca", "Marco", "Paolo"];

```

Nell'array possiamo conservare
ogni tipo di dato!

Array

Accesso ai dati e lunghezza

Come si accede agli elementi di un array?

nomeArray[IndiceElemento]

Come so quanti elementi ci sono?

nomeArray.length

🧐 **Attenzione: gli array contano da 0!**

La prima **posizione/indice** quindi sarà 0 e non 1.

```
1  const iscritti = ["Luca", "Marco", "Paolo"];
2
3
4  iscritti[0] // Luca
5
6  iscritti[1] // Marco
7
8  iscritti[2] // Paolo
9
10
11 iscritti.length // 3
```

Array

E se voglio aggiungere/rimuovere un elemento?

Dopo la creazione di un array possiamo modificare il suo contenuto:

.push() *per aggiungere un elemento alla fine*

.shift() *per rimuovere l'elemento iniziale*

```
1 // creazione dell'array iniziale
2 const iscritti = ["Luca", "Marco", "Paolo"];
3
4 // aggiunta di un elemento all'array
5 iscritti.push('Michele');
6
7  // output dell'array modificato
8  ["Luca", "Marco", "Paolo", "Michele"];
```

```
1 // creazione dell'array iniziale
2 const iscritti = ["Luca", "Marco", "Paolo"];
3
4 // rimozione del primo elemento dell'array
5 iscritti.shift();
6
7  // output dell'array modificato
8  ["Marco", "Paolo"];
```

Array

Altre operazioni

Esistono tantissimi altri metodi tra cui:

.join() *per trasformare l'elenco in una stringa*

.includes() *per verificare che un elemento sia presente*

```
// creazione dell'array iniziale
const iscritti = ["Luca", "Marco", "Paolo"];

// indichiamo un separatore (es: la virgola)
iscritti.join(', ');

 // output dell'array che diventa stringa
 'Luca, Marco, Paolo';

```

```
// creazione dell'array iniziale
const iscritti = ["Luca", "Marco", "Paolo"];

// rimozione del primo elemento dell'array
iscritti.includes("Marco");

 // output dell'includes ( Marco c'è)
 true

```

JS

JavaScript

Ripetere alcune operazioni

Costrutti Iterativi

Ripetere un'operazione più volte

Molto spesso abbiamo la necessità di far eseguire un blocco di codice più volte

I **costrutti iterativi** ci permettono di scrivere il codice una volta e poi ripeterlo quante volte vogliamo, a seconda delle nostra istruzioni.

Quando vogliamo ripetere un'operazione per tutti gli elementi della lista, una delle soluzioni più comode è il **for of.**

per ognuno degli elementi della lista {

// esegui questa istruzione

// e anche questa!

}

```
1 // dato un array
2 const iscritti = ["Luca", "Marco", "Paolo"];
3
4 // Per ogni iscritto della lista iscritti...
5
6 for (const iscritto of iscritti){
7   // Stampa il nome per ognuno degli iscritti
8   console.log(iscritto);
9 }
```

JS

JavaScript

Gli eventi

Gli Eventi

Nella pagina ci sono molti eventi che possiamo intercettare.

Tra i più comuni, ci sono le interazioni con la pagina da parte dell'utente: click, submit, scroll, resize ...

Rimaniamo in **ascolto di un evento** su quell'elemento.
Quale evento? '**click**'

In questo esempio vediamo una **funzione** che viene attivata solo quando avviene l'evento '**click**'

```
1  <div class="saluto">CIAO</div>
```

```
1 const element = document.querySelector(".saluto");
2
3 element.addEventListener('click', function() {
4    // codice
5  });
```

JavaScript

Istruzioni condizionali

Istruzioni Condizionali

E se?

Se volessimo far “accadere una cosa” solo in un caso specifico?

Le **istruzioni condizionali** eseguono un certo blocco di codice **se si verifica una precisa condizione.**

La parte “altrimenti” non è obbligatoria.

Se si verifica la (condizione) {

// allora viene eseguito questo blocco di istruzioni 1 (notare le graffe)

} **altrimenti** {

// viene eseguito questo blocco di istruzioni 2 (notare le graffe)

}

```
1 if (condizione) {
2
3  // blocco di istruzioni 1
4
5 } else {
6
7  // blocco di istruzioni 3
8
9 }
```

Operatori relazionali

Come confrontiamo due valori

Vengono utilizzati per confrontare due valori e restituire un valore booleano che indica se il confronto è vero o falso.

==	*uguale a*	`3 == 3`	✅
!=	*diverso da*	`'a' != 'b'`	✅
>	*maggiore di*	`3 > 2`	✅
<	*minore di*	`3 < 2`	❌

>= *maggiore o uguale*

<= *minore o uguale*

Operatori Logici

Uso avanzato delle condizioni

Gli operatori logici consentono di costruire **condizioni più complesse**.

&&	(AND) tutte le condizioni concatenate devono risultare VERE
||	(OR) almeno una delle condizioni concatenate deve risultare VERA
!	(NOT) inverte un valore booleano, utile ad esempio, verificare se un certo valore è falso

JS

JavaScript

Un accenno alle funzioni

Le Funzioni

Le funzioni sono **blocchi di istruzioni** raggruppati insieme che possono essere eseguiti a piacimento.

Non vengono eseguiti non appena il browser arriva al loro rigo, ma in un secondo momento, deciso da noi.

Sono utilissime per vari motivi:

- Il loro codice viene eseguito **quando decidiamo noi,** non nell'esatto momento in cui browser le vede.

- Possiamo **riutilizzare più volte le istruzioni** al loro interno più volte nel nostro codice, senza riscriverlo.

- Permettono di **organizzare il codice** in maniera più ordinata, **spezzettando le parti complesse**.

Funzioni Anonime

Come si scrive una funzione?

E' un blocco di codice che rappresenta una serie di operazioni in blocco che possono essere eseguite più volte.

Un esempio classico è una funzione che reagisce al click di un bottone!

Per dichiarare una funzione possiamo usare la keyword **function**

```
1 const button = document.querySelector('button')
2
3 button.addEventListner('click', function(){
4     // Istruzioni da eseguire
5     // OGNI volta che clicco il bottone
6 })
```

Funzioni Nominali

Come si scrive una funzione?

Possiamo anche dare un nome ad una funzione, per rendere più chiaro il codice ed isolare la sua logica, per poi richiamarla più agevolmente in un secondo momento!

- **keyword:**
  function
- **Nome funzione:**
  si imposta il nome della funzione
  con la quale potremo poi richiamarla.
- **Codice da eseguire:**
  Tra le parentesi graffe inseriamo
  il codice che vogliamo eseguire.

```
1  // DICHIARAZIONE
2
3  function miaFunzione() {
4
5    // blocco di codice
6    console.log('Ciao, ho eseguito la funzione')
7  }
8
9
10 //INVOCAZIONE
11 miaFunzione()
```

Funzioni Nominali

Come si scrive una funzione completa?

- **keyword:**
  function
- **Nome funzione:**
  si imposta il nome della funzione
  con la quale potremo poi richiamarla.
- **Codice da eseguire:**
  Tra le parentesi graffe inseriamo
  il codice che vogliamo eseguire.
- **Parametri:** dati in ingresso che possono essere elaborati per restituire un risultato.
- **Valore restituito:** il risultato di una operazione.

```
1  function miaFunzione(num1, num2) {
2
3    // blocco di codice
4    const risultato = num1 + num2;
5
6    return risultato;
7
8    // eventuali altre istruzioni dopo il return
9    // non verranno eseguite
10
11 }
```

🤔 **Non è obbligatorio avere dei parametri, né tantomeno restituire un risultato**.

API

Application Programming Interface

API

Concetto di Interfaccia

Un' **interfaccia** è un insieme di regole che permettono di "comunicare" con qualcosa. Ne siamo circondati nella vita di tutti i giorni:

- Interruttore della luce
- Telecomando del televisore
- Elettrodomestici vari

In tutti questi casi ci sono delle regole che vanno rispettate, e delle modalità precise per ottenere il risultato desiderato!

Ad esempio non posso cambiare il canale della TV premendo il tasto off, né spegnerla premendo il tasto 5.

API

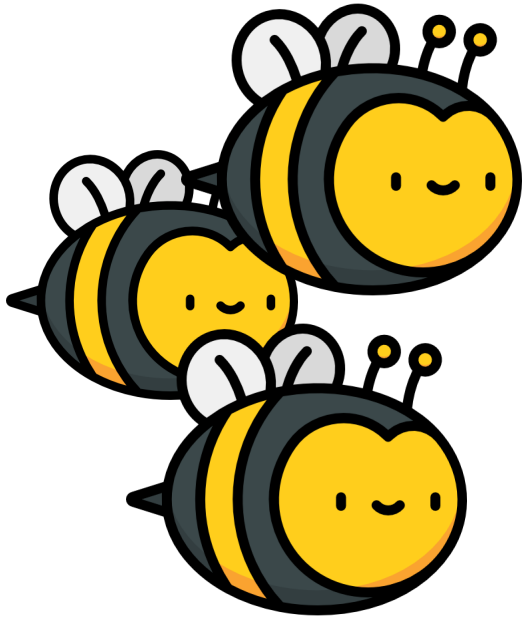
Application Programming Interface

Quando parliamo di **API** parliamo di un'interfaccia per la comunicazione tra sistemi (programma/software/servizio ecc.)

Con API si intende un insieme di funzionalità rese disponibili (ad un programmatore) per svolgere un insieme coerente di operazioni.

È un termine generico che indica delle operazioni rese disponibili nascondendo parte della complessità sottostante.

Ciascuna API è diversa: se ne studia la documentazione e si apprendono le regole per ottenere il risultato desiderato, secondo le indicazioni fornite.



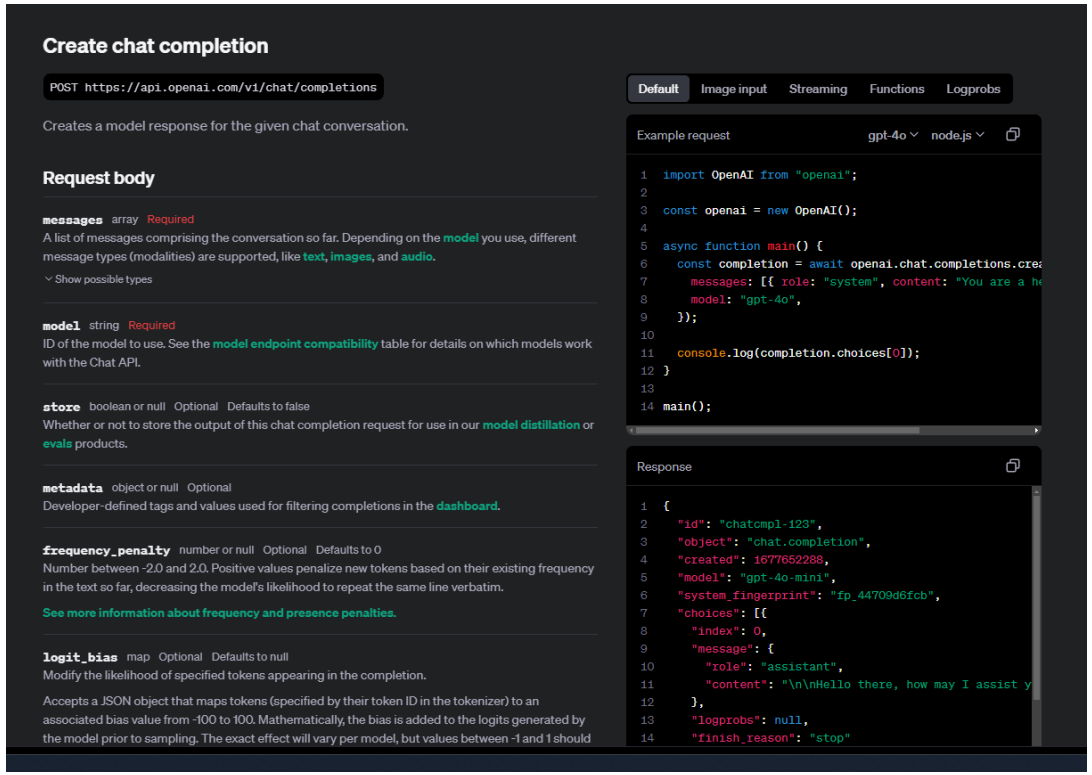
ChatGPT Web API

Servizio di chat con intelligenza artificiale

Una chat con l'intelligenza artificiale che possiamo usare nelle nostre App!

Ha una interfaccia web, ma mette a disposizione anche una API con la sua documentazione.

https://platform.openai.com/docs/api-reference/making-requests



Create chat completion

POST https://api.openai.com/v1/chat/completions

Creates a model response for the given chat conversation.

Request body

messages array Required
A list of messages comprising the conversation so far. Depending on the model you use, different message types (modalities) are supported, like text, images, and audio.
Show possible types

model string Required
ID of the model to use. See the model endpoint compatibility table for details on which models work with the Chat API.

store boolean or null Optional Defaults to false
Whether or not to store the output of this chat completion request for use in our model distillation or evals products.

metadata object or null Optional
Developer-defined tags and values used for filtering completions in the dashboard.

frequency_penalty number or null Optional Defaults to 0
Number between -2.0 and 2.0. Positive values penalize new tokens based on their existing frequency in the text so far, decreasing the model's likelihood to repeat the same line verbatim.
See more information about frequency and presence penalties.

logit_bias map Optional Defaults to null
Modify the likelihood of specified tokens appearing in the completion.
Accepts a JSON object that maps tokens (specified by their token ID in the tokenizer) to an associated bias value from -100 to 100. Mathematically, the bias is added to the logits generated by the model prior to sampling. The exact effect will vary per model, but values between -1 and 1 should

Default | Image input | Streaming | Functions | Logprobs

Example request — gpt-4o — node.js

```
import OpenAI from "openai";

const openai = new OpenAI();

async function main() {
  const completion = await openai.chat.completions.crea
    messages: [{ role: "system", content: "You are a he
    model: "gpt-4o",
  });

  console.log(completion.choices[0]);
}

main();
```

Response

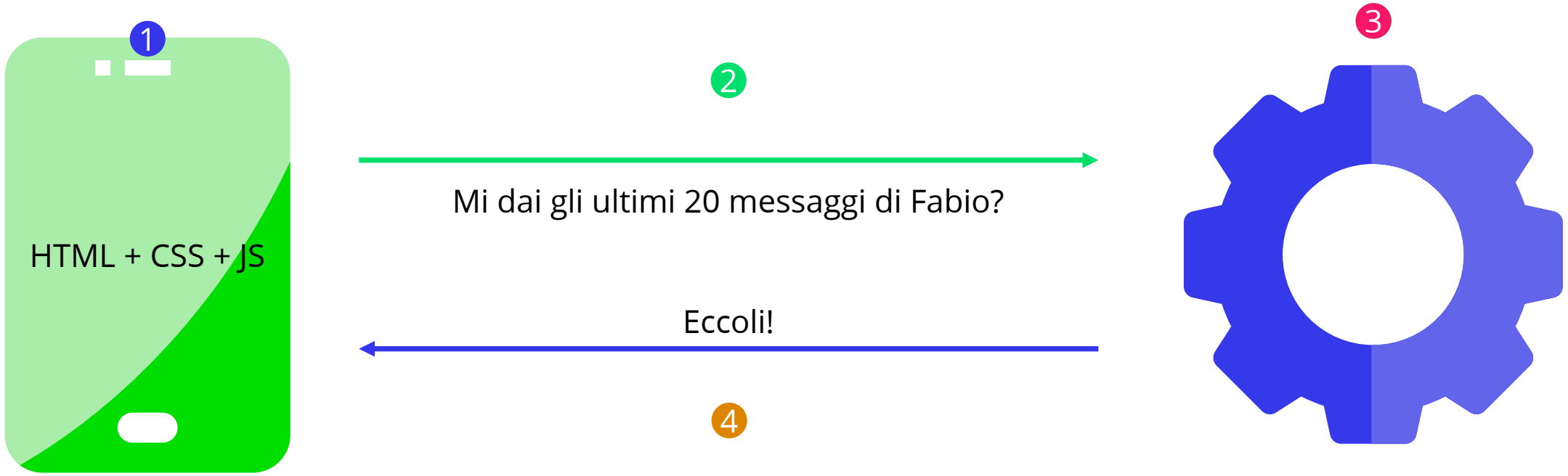
```
{
  "id": "chatcmpl-123",
  "object": "chat.completion",
  "created": 1677652288,
  "model": "gpt-4o-mini",
  "system_fingerprint": "fp_44709d6fcb",
  "choices": [{
    "index": 0,
    "message": {
      "role": "assistant",
      "content": "\n\nHello there, how may I assist y
    },
    "logprobs": null,
    "finish_reason": "stop"
```

Chiamate HTTP

Come far dialogare la nostra pagina con l’esterno

Un visione d’insieme

Immaginiamo un'app di messagistica:



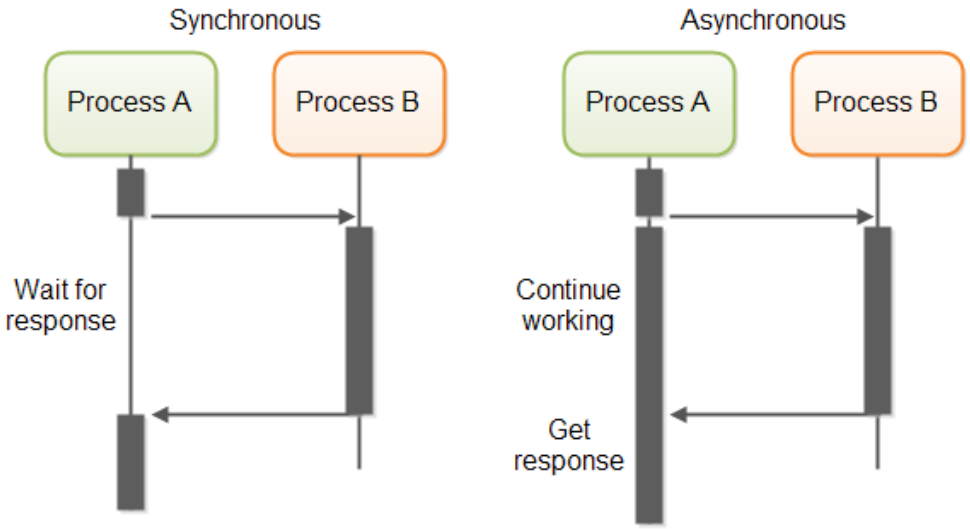
JS + HTTP = Ajax!

chiamate asincrone

La tecnica con cui effettuiamo chiamate HTTP con JavaScript è detta **Ajax.**

Le chiamate Ajax sono chiamate **asincrone:** il browser continua a funzionare e poi esegue il nostro codice non appena avrà ricevuto una risposta!

Nel 2015 arrivò il metodo **fetch**, che ci permette di eseguire una chiamata AJAX con una sintassi molto comoda.




```
1 async function getData() {
2     const response = await fetch('https://api.example.com/data');
3     const data = await response.json();
4     console.log(data);
5 }
6
```

Fetch

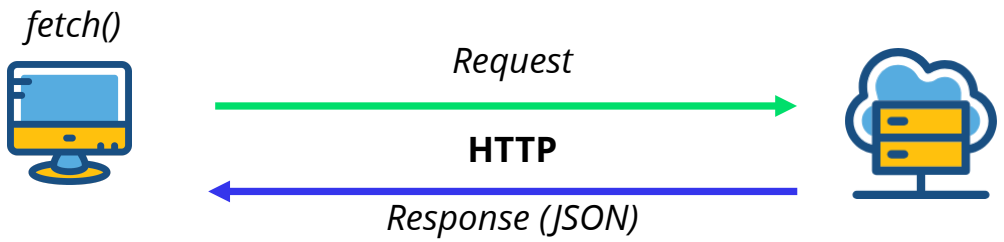
Eseguire chiamate asincrone a un server

Il metodo **Fetch** consente di effettuare **richieste HTTP asincrone** verso server web e di gestire le risposte ottenute.

La chiamata andrà rivolta a un indirizzo speccifico detto **endpoint**.

Una volta ricevuta la risposta, dovremo rielaborarla per raccoglierne i dati.

```
1 const endpoint = 'https://api.example.com/data'
2
3 async function getData() {
4     const response = await fetch(endpoint);
5     const data = await response.json();
6     console.log(data);
7 }
8
```



JS + HTTP = Ajax!

Async/ await

Se vogliamo che il browser attenda la risposta prima di andare avanti, possiamo usare **`await`** davanti alla chiamata asincrona.

Per farlo,però, dobbiamo indicare che la funzione è asincrona, premettendo la keyword **`async`**

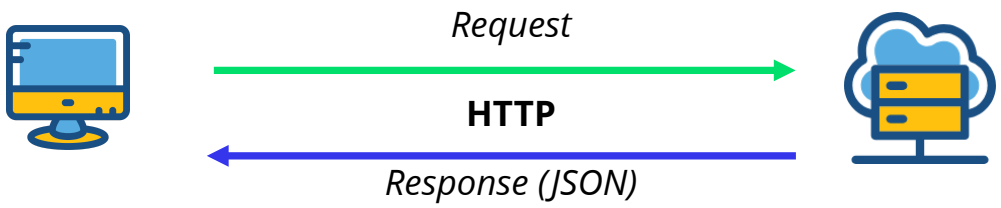
```
1 const endpoint = 'https://api.example.com/data'
2
3 async function getData() {
4     const response = await fetch(endpoint);
5     const data = await response.json();
6     console.log(data);
7 }
8
```

JSON

JavaScript Object Notation

Come comunicano client e server?
Esistono alcuni standard, ma il più comune resta il JSON.

JSON (JavaScript Object Notation) utilizza la notazione che conosciamo per definire gli oggetti in JavaScript




```
1  {
2   "users": [
3     {
4     "name": "Ema",
5     "cognome": "Princi",
6     "indirizzo": "Torino",
7     "anni": 20
8     },
9     {
10      "name": "Max",
11      "cognome": "Rossi",
12      "indirizzo": "Roma",
13      "anni": 54
14    }
15  ]
16 }
17
```

JS Objects

Un "raggruppatore" di informazioni

Spesso abbiamo bisogno di rappresentare nel codice un qualcosa che ha pù caratteristiche, per cui **una semplice stringa o un numero non bastano**.

In questi casi, possiamo utilizzare gli *object literals* , più comunemente conosciuti come **Oggetti**

Questi ci permettono di rappresentare entità più complesse e le loro caratteristiche tramite un sistema di **chiave - valore**

Potremo poi accedere al valore delle chiavi tramite l'uso del punto (.), detta **dot notation**

```
1 const person = {
2   firstName: 'Marco',
3   lastName: 'Lancellotti',
4   job: 'Web Developer',
5   age: 38
6 }
7
8
9 person.age; // 38
```

JS + JSON = ❤️

Come far parlare browser e server

Quando da JavaScript vogliamo mandare un oggetto al server, possiamo convertirlo in una stringa JSON con **`JSON.stringify()`**

Quando riceviamo una stringa JSON dal server, possiamo riconvertirla in oggetto Javascript con **`JSON.parse()`**

```
1 const person = {
2   firstName: 'Marco',
3   lastName: 'Lancellotti',
4   job: 'Web Developer',
5   age: 38
6 }
```

```
1 '{"firstName":"Marco","lastName":"Lancellotti","job":"Web Developer","Age":38}'
```

HTTP - Facciamo una richiesta

Cosa dobbiamo specificare?

Endpoint

Un'indirizzo da chiamare (sotto forma di *stringa)*

Metodo

Il metodo HTTP con cui effettuare la chiamata (come stringa)

Headers (intestazioni)

Ad esempio il tipo di applicazione, il linguaggio, credenziali, (sotto forma di *oggetto)*

Body

Le informazioni che vogliamo inviare al server (come stringa JSON)

```
1  const endpoint = 'http://example.com/endpoint';
2
3  const response = await fetch(
4      endpoint,
5    {
6      method: '...',
7      headers: { ... },
8      body: JSON.stringify({ ... })
9    }
10 )
```