

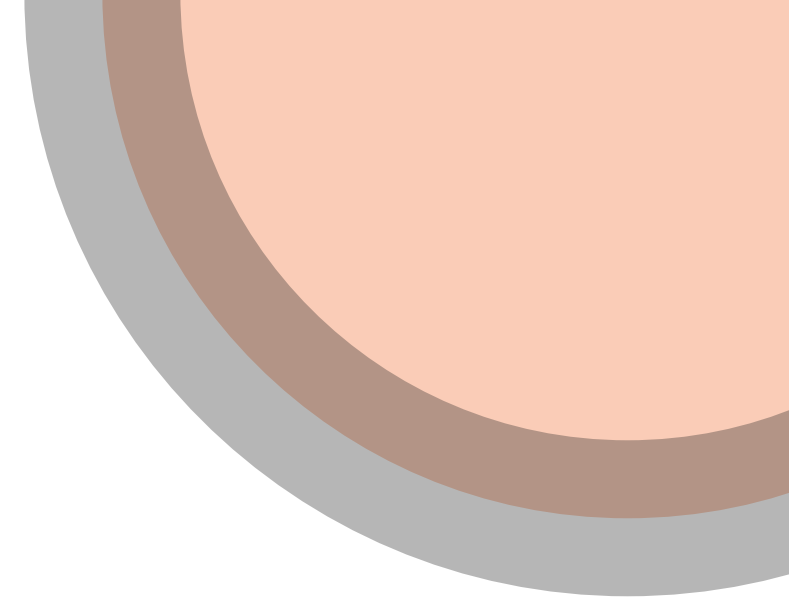
Introduzione a Node.js



Cos'è Node.js?

Node.js è un ambiente di esecuzione per JavaScript costruito sul motore V8 di Google Chrome.

In parole semplici, permette di eseguire codice JavaScript fuori dal browser.



Perché Node.js?

- Event-driven: Funziona con eventi e callback, ideale per gestire molte connessioni contemporaneamente.
- Non bloccante: rende possibile gestire le gestisce in modo asincrono operazioni che altrimenti sarebbero lente.
- JavaScript Ovunque: Con Node.js, gli sviluppatori possono scrivere sia il lato client che il lato server in JavaScript, assicurando una più fluida integrazione tra i due.


Casi d'uso di Node.js

- Applicazioni in Tempo Reale: chat, strumenti di collaborazione simultanea, server di gioco o altre applicazioni in tempo reale.
- Microservizi: Node.js può essere una soluzione efficace per costruire microservizi e API.
- Streaming di Dati: Le funzionalità di streaming di dati di Node.js possono essere utilizzate per elaborare file durante il caricamento, o per la codifica audio o video in tempo reale.



Perché Node.js?

Un certo numero di aziende e servizi popolari utilizzano Node.js nel loro tech stack, tra cui:

- Netflix
 - LinkedIn
 - Uber
 - PayPal
 - Walmart
 - Trello
- 
- 
- 

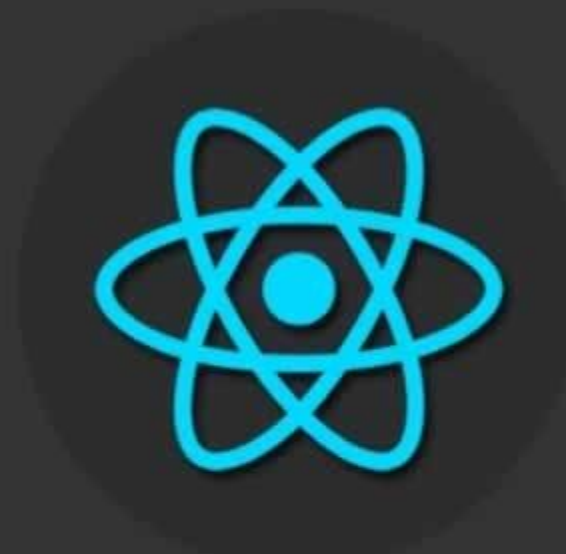


Cos'è MongoDB

- Un database NoSQL basato su documenti JSON-like (formato BSON)
- Archivia dati in collezioni invece che in tabelle.
- È scheme-less: ogni documento può avere una struttura diversa.

Caratteristiche principali:

- Flessibilità: non richiede uno schema rigido come nei DB relazionali.
- Scalabilità: progettato per funzionare facilmente in ambienti distribuiti (sharding, replica).
- Query potente: supporta filtri, proiezioni, aggregazioni, indicizzazione e operazioni geospaziali.

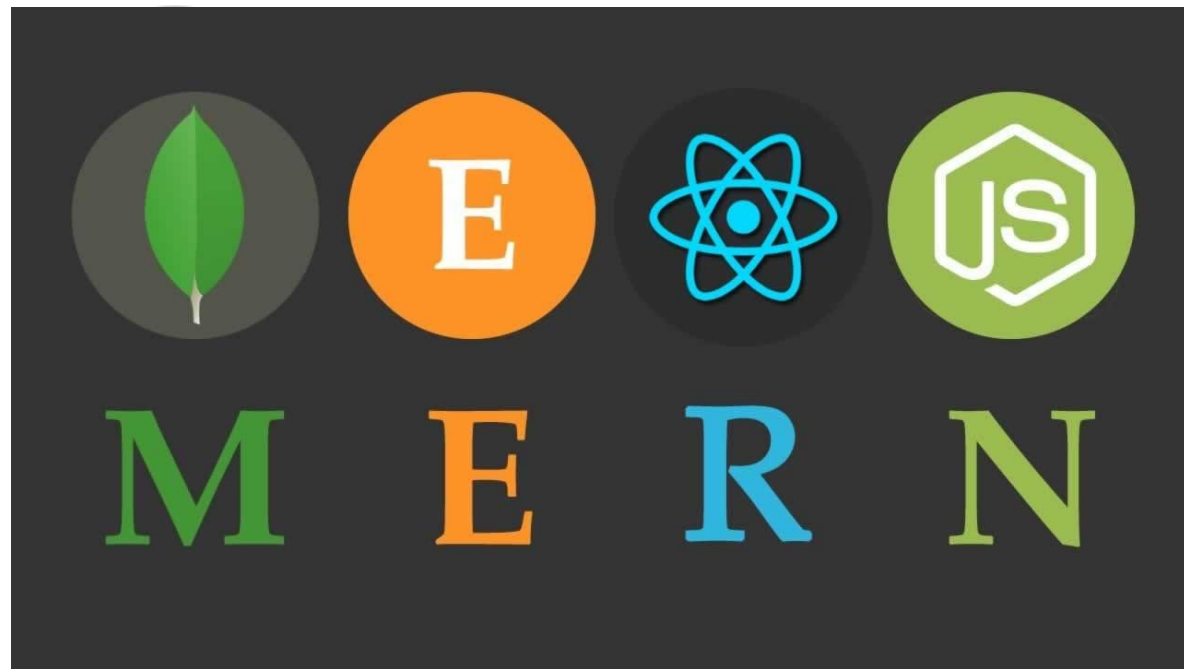


M

E

R

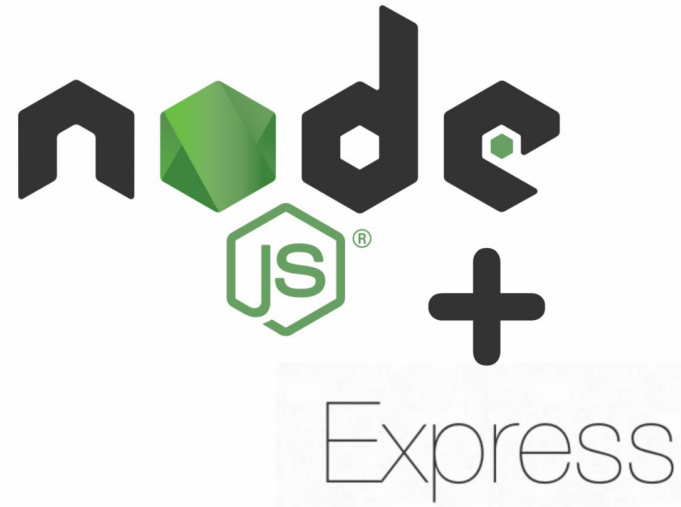
N



Flusso di lavoro tipico:

- React gestisce l'interfaccia utente.
- Invoca API tramite HTTP (es. fetch, axios) verso Express/Node.
- Express gestisce le richieste, elabora i dati e interagisce con MongoDB.
- MongoDB salva o recupera i dati, restituiti al client React.

Unico linguaggio (JS) su tutto lo stack, con accesso ad un ecosistema vasto (librerie, strumenti, comunità ampia) e un'architettura scalabile adatta a REST API e SPA.

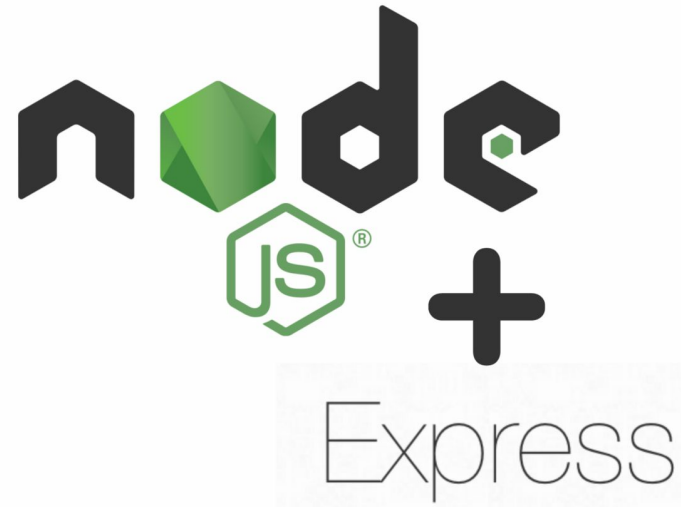


JS ma per il back end

Se provieni da JavaScript per browser, scrivere codice in Node.js ti può sembrare simile, ma in realtà si tratta di due ambienti molto diversi.

JavaScript nativamente è pensato per interagire con l'interfaccia utente: puoi accedere al DOM, manipolare elementi HTML, gestire eventi come click e input, usare l'oggetto window, fare animazioni e dialogare con l'utente.

In Node.js, **invece**, JavaScript viene eseguito **sul server**, di conseguenza non sono disponibili gli oggetti window e document, o il DOM.



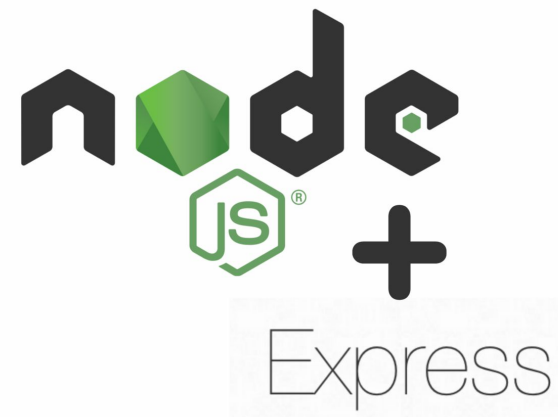
Approccio a moduli

In Node.js il lavoro è reso possibile grazie ai moduli, dovrai conoscere o integrare nuovi moduli per poter sfruttare a pieno le potenzialità.

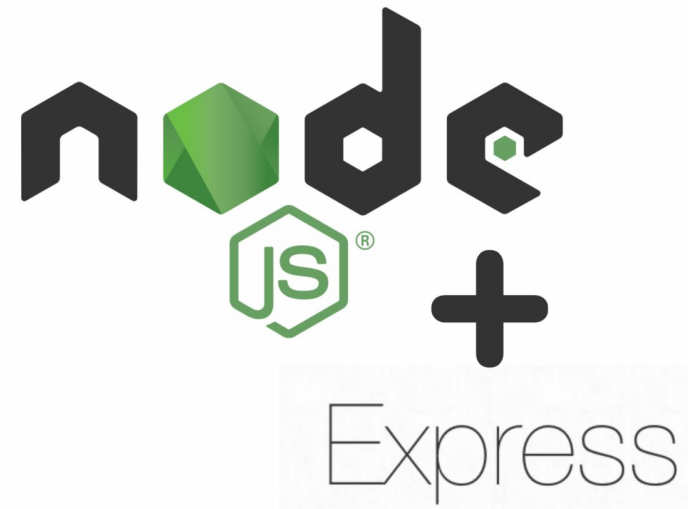
Ecco alcune funzionalità di cui potresti sentire la mancanza:

- fetch -> axios, node-fetch, o moduli HTTP nativi di Node
- localStorage -> dati su files o database

Per inciso, se da browser il lavoro è “visivo”, in Node.js è orientato a logica, elaborazione e gestione di dati.



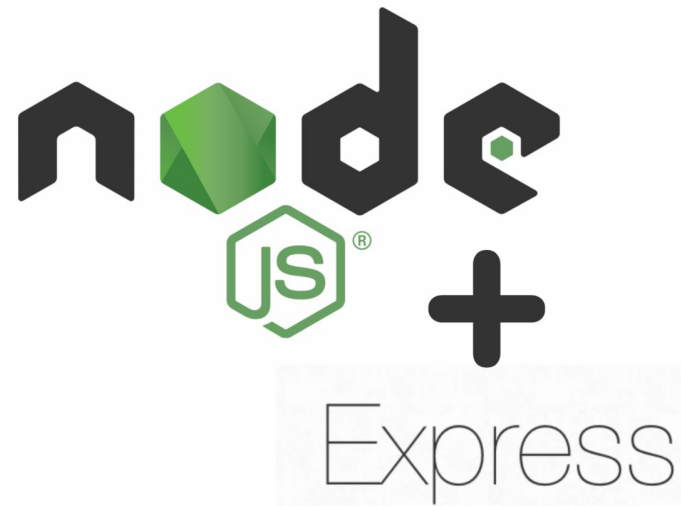
Aspetto	JavaScript nel Browser	JavaScript in Node.js
Ambiente globale	window, document	global, process
Manipolazione DOM	Sì (document.querySelector, ecc.)	No
Accesso al filesystem	No	Sì (fs, path, ecc.)
Moduli	import/export (ES Modules)	require/module.exports o import/export
Caricamento librerie	<script src="..."> o ES Modules via CDN	npm install e require/import
Event loop e async	setTimeout, Promise, async/await	Stesse cose + fs, http, stream, ecc.
API di rete	fetch, XMLHttpRequest, WebSocket	http, https, net, fetch (moderno)
Storage locale	localStorage, sessionStorage	Accesso diretto a file o database
Sicurezza	Limitato e protetto	Pieno accesso al sistema
UI/Interazione utente	Sì (bottoni, form, eventi)	No (nessun supporto per interfacce grafiche)
Debug	DevTools nel browser	Console, debugger da terminale o editor (es. VSCode)
Utilizzo principale	Frontend (interfaccia utente)	Backend (server, API, CLI, script di sistema)



Global

Global è l'equivalente di window nel browser. È l'oggetto che contiene variabili e funzioni disponibili ovunque nel programma, senza bisogno di importarle.

Nota: non è una buona pratica aggiungere variabili a global — rende il codice difficile da capire e testare. È meglio passare dati in modo esplicito o usare moduli.



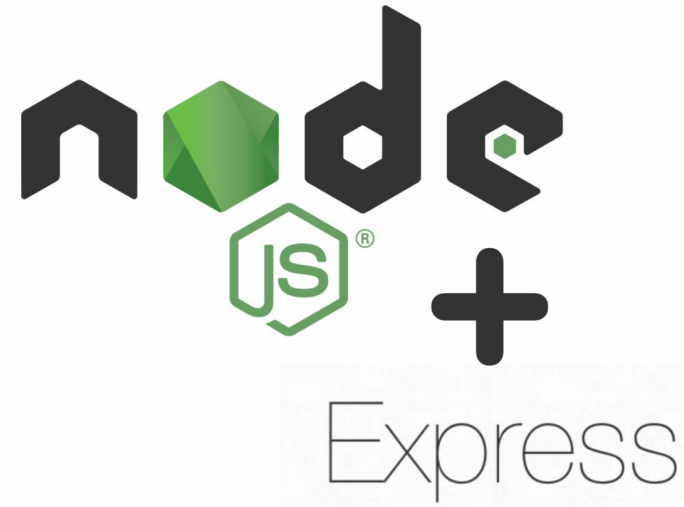
Process

Process è un oggetto globale che rappresenta il processo attivo di Node.js.

Ti permette di:

- Leggere argomenti da riga di comando
- Sapere su quale sistema stai girando
- Gestire variabili d'ambiente
- Uscire manualmente dal programma
- Ricevere eventi (come errori o chiusura)

Funzione	Cosa fa
<code>process.argv</code>	Argomenti da terminale
<code>process.env</code>	Variabili d'ambiente
<code>process.exit()</code>	Termina il programma
<code>process.cwd()</code>	Mostra la cartella di lavoro corrente
<code>process.on('exit', callback)</code>	Gestisce eventi di uscita

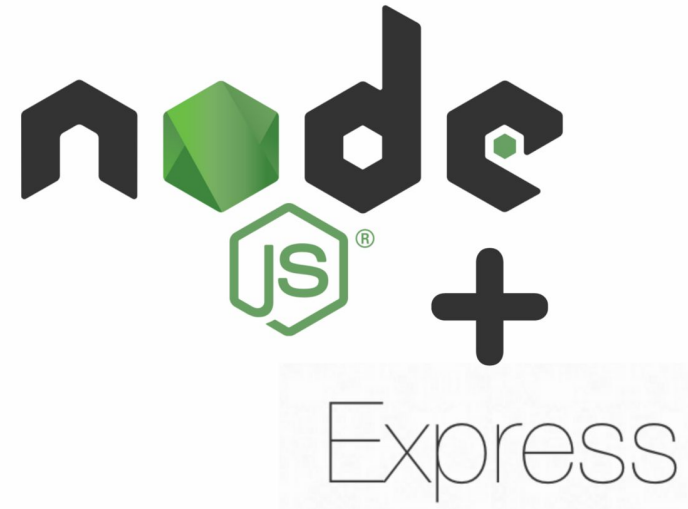


Require vs import

Quando si lavora con Node.js, una delle prime da affrontare è l'importazione di moduli e librerie. Nei tutorial e nei progetti più semplici, si usa spesso la funzione `require()`.

Questo sistema di importazione si chiama **CommonJS** ed è stato il metodo **standard in Node.js** fin dalle sue origini. È semplice, funziona immediatamente senza configurazioni particolari e va bene per la maggior parte degli usi.

Arrivando dal browser però lo sviluppatore JS è abituato alla sintassi degli **ES Modules** che sfrutta **import** ed **export**. A questo punto è bene sapere che anche Node.js ha iniziato a supportare questo sistema, rendendolo una valida alternativa al `require()`.



Require vs import

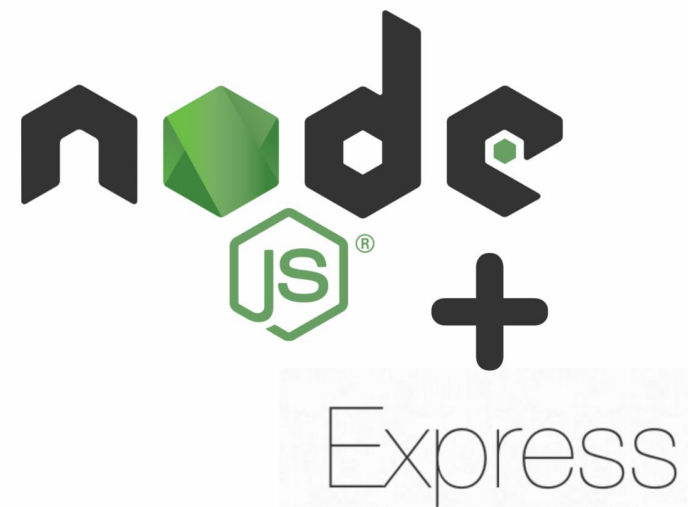
Tuttavia, per usare import in Node.js non basta scrivere il codice in quel modo:

Si deve anche abilitare esplicitamente il supporto agli ES Modules.

Ci sono due modi per farlo:

1. Rinominare i file in .mjs anziché .js.
2. Aggiungere "type": "module" nel file package.json.

Senza questa configurazione, Node.js non riconosce la sintassi import.



Creare un server in Node (puro)

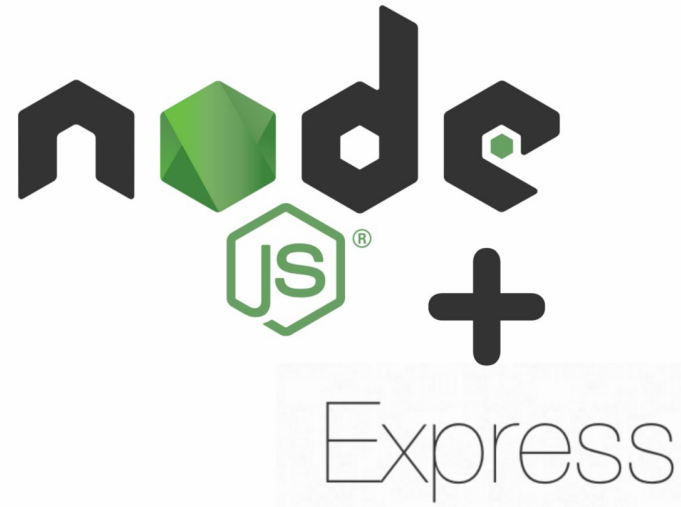
```
const http = require('http');

const server = http.createServer((req, res) => {
  // Imposta intestazioni per risposta JSON
  res.setHeader('Content-Type', 'application/json');

  // Verifica il percorso richiesto
  if (req.url === '/api' && req.method === 'GET') {
    const data = {
      message: 'Ciao dal server!',
      time: new Date().toISOString()
    };

    res.writeHead(200); // OK
    res.end(JSON.stringify(data));
  } else {
    res.writeHead(404);
    res.end(JSON.stringify({ error: 'Not found' }));
  }
});

server.listen(3000, () => {
  console.log('Server attivo su http://localhost:3000');
});
```

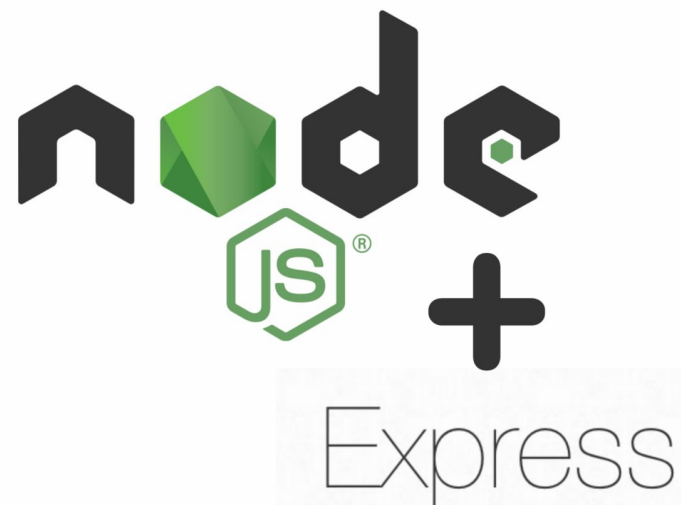


Cos'è Express?

È un framework web per [Node.js](https://nodejs.org/) che facilita la creazione di API e applicazioni web/server. Fornisce un set di funzionalità per gestire routing, middleware, request/response, e molto altro.

Perché usarlo?

- Semplice e leggero: costruisci server con poche righe di codice.
- Estendibile: si integra facilmente con librerie e moduli esterni.
- Popolare: è la base di molti altri framework (es. Nest.js).



Creare un server con Express

```
const express = require('express')
const PORT = 8000;
const server = express();
server.use(express.json());

server.get('/', (req, res) => {
  res.status(200).send('ciao')
});

server.listen(PORT, () => {
  console.log(`Server up and running on port ${PORT}`);
})
```

```
server.get('/users', (req, res) => {
  res
    .status(200)
    .send(users)
});

server.get('/users/:id', (req, res) => {
  const { id } = req.params;
  res
    .status(200)
    .send(users[id])
});
```