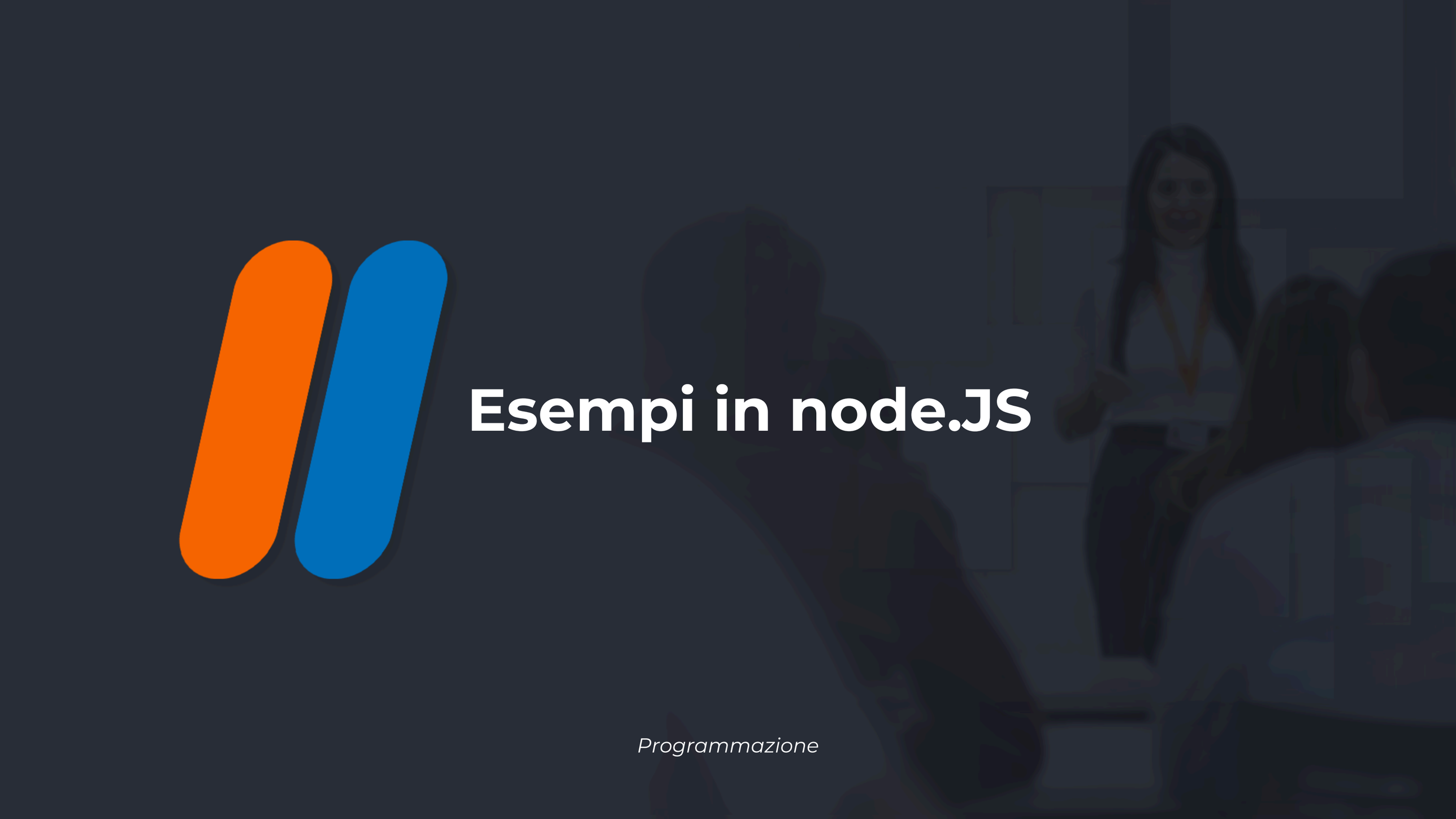


Esempi in node.JS

Programmazione

Node.js è un runtime JavaScript lato server costruito sul motore V8 di Chrome.

Permette di eseguire codice JavaScript fuori dal browser, tipicamente su un server, rendendo possibile creare applicazioni web complete solo con JavaScript.

1. Server HTTP base

La creazione di un server HTTP tramite il modulo nativo http rappresenta l'approccio più essenziale per comprendere come Node.js gestisce le richieste e le risposte in modo asincrono.

A livello concettuale, questo esempio mostra la filosofia "event-driven" di Node: quando arriva una richiesta, viene attivata una callback che costruisce la risposta manualmente, impostando gli header e inviando il contenuto finale al client.

Questo modello permette di capire come Node operi a basso livello, senza astrazioni aggiuntive, simulando ciò che avviene internamente anche in framework più evoluti.

È un punto di partenza utile per chi vuole comprendere i meccanismi alla base di server performanti e non bloccanti.

1. Server HTTP base

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

const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello from Node.js!');
});

server.listen(3000);
```

2. Lettura file con fs

L’esempio sulla lettura dei file introduce uno dei moduli centrali dell’ecosistema Node.js: fs, dedicato alle operazioni sul file system.

Qui è evidente la natura asincrona dell’ambiente: la richiesta di lettura viene avviata, ma il programma non rimane fermo ad aspettare il risultato; quando il file è pronto, la callback viene eseguita con eventuali errori e il contenuto testuale.

Questo pattern è fondamentale per lavori intensivi in I/O, come parsing log, gestione upload, automazioni e pipeline di dati.

A livello didattico, evidenzia perché Node sia particolarmente efficiente in scenari con molte operazioni I/O contemporanee.

2. Lettura file con fs

```
1.const fs = require('fs');
2.
3.fs.readFile('testo.txt', 'utf8', (err, data) => {
4.  if (err) return console.error(err);
5.  console.log(data);
6.});
```

3. Scrittura file con fs

Lo snippet dedicato alla scrittura dei file completa la comprensione dell'I/O asincrono introducendo il concetto di creazione o sovrascrittura di risorse nel file system.

Il metodo writeFile consente di generare file di output per log, configurazioni, esportazioni di dati o cache temporanee.

L'operazione, come tutte quelle del modulo fs, è non bloccante: il server rimane libero di gestire altre richieste mentre la scrittura prosegue in background.

L'esempio aiuta a mettere a fuoco il ruolo delle callback nella gestione degli errori e conferma il perché Node sia un ambiente adatto a script di automazione e servizi backend che manipolano file.

3. Scrittura file con fs

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

fs.writeFile('output.txt', 'Ciao!', (err) => {
  if (err) return console.error(err);
  console.log("File scritto!");
});

```

4. Server Express semplice

Express è una delle librerie più diffuse per la costruzione di applicazioni web con Node.js perché introduce un modello a middleware semplice ma potentissimo.

A differenza del modulo http, qui la gestione delle rotte, dei parametri e dei formati di risposta viene semplificata da un'interfaccia più intuitiva.

L'esempio mostra una rotta GET che risponde con un testo, ma dietro le quinte Express gestisce parsing della richiesta, formattazione della risposta e molto altro.

Questo snippet è fondamentale per comprendere come nascono API REST, microservizi e applicazioni full-stack basate su Node.

4. Server Express semplice

```
1. const express = require('express');
2. const app = express();
3.
4. app.get('/', (req, res) => {
5.   res.send('Hello Express!');
6. });
7.
8. app.listen(3000);
9.
```

5. Richiesta HTTP esterna con Axios

L’utilizzo di Axios dimostra come un’applicazione Node.js possa interagire facilmente con API esterne, recuperando dati da servizi remoti come GitHub, OpenWeather, database headless e altri microservizi.

Axios implementa internamente le Promesse, offrendo un’interfaccia più pulita rispetto ai moduli nativi e integrando funzionalità essenziali come gestione degli header, risposta in JSON, gestione degli errori e timeout.

Comprendere questo esempio significa vedere Node non solo come server, ma anche come “client” di altri sistemi, un aspetto cruciale nelle architetture moderne basate su integrazioni multiple.

5. Richiesta HTTP esterna con axios

```
1. const axios = require('axios');
2.
3. axios.get('https://api.github.com/users/octocat')
4.   .then(res => console.log(res.data))
5.   .catch(err => console.error(err));
6.
```

6. Uso dei moduli (import/export)

La gestione dei moduli è un elemento chiave per organizzare il codice in Node.js, permettendo di separare funzioni, classi o configurazioni in file distinti per migliorarne la manutenibilità e la riusabilità.

L’esempio con module.exports e require utilizza lo standard CommonJS, che è stato per anni il modello principale di Node.

Questo approccio consente di strutturare applicazioni anche molto complesse suddividendole in moduli autonomi, come controller, servizi, repository o utility.

È un concetto fondamentale in qualsiasi applicazione reale, soprattutto quando si inizia a costruire API o progetti modulari.

Uso di moduli (import/export)

math.js:

```
1. module.exports.somma = (a, b) => a + b;
```

index.js:

```
1. const { somma } = require('./math');
2. console.log(somma(5, 3));
```

7. Uso di Async/Await

L’esempio sull’uso di async/await mette in evidenza il modello asincrono moderno introdotto per rendere il codice più leggibile rispetto alle callback annidate e alle catene di Promesse.

In questo snippet, la funzione fetchData simula un’operazione asincrona (come una richiesta HTTP o una lettura da database), mentre la funzione run attende il risultato in maniera sequenziale e più chiara.

Questo paradigma è ormai lo standard nello sviluppo backend con Node perché unisce la leggibilità del codice sincrono ai vantaggi prestazionali dell’asincronia, rendendo più semplice scrivere flussi di lavoro complessi.

7. Async/Await

```
async function fetchData() {
  return new Promise(resolve => setTimeout(() => resolve("OK"), 1500));
}

async function run() {
  const result = await fetchData();
  console.log(result);
}

run();
```

8. Creazione di una semplice REST API

L’esempio che mostra la creazione di una REST API con Express introduce un caso d’uso fondamentale: ricevere dati dal client sotto forma di JSON e restituire una risposta strutturata.

L’uso di express.json() abilita il parsing automatico del body della richiesta, permettendo al server di accettare input come oggetti, form data o payload provenienti da frontend e servizi esterni.

Questo è uno dei pattern più comuni nel mondo del backend moderno: creare endpoint che permettono la registrazione, modifica, eliminazione e consultazione di dati.

È la base di qualsiasi architettura microservizi, applicazione mobile o piattaforma web basata su Node.js.

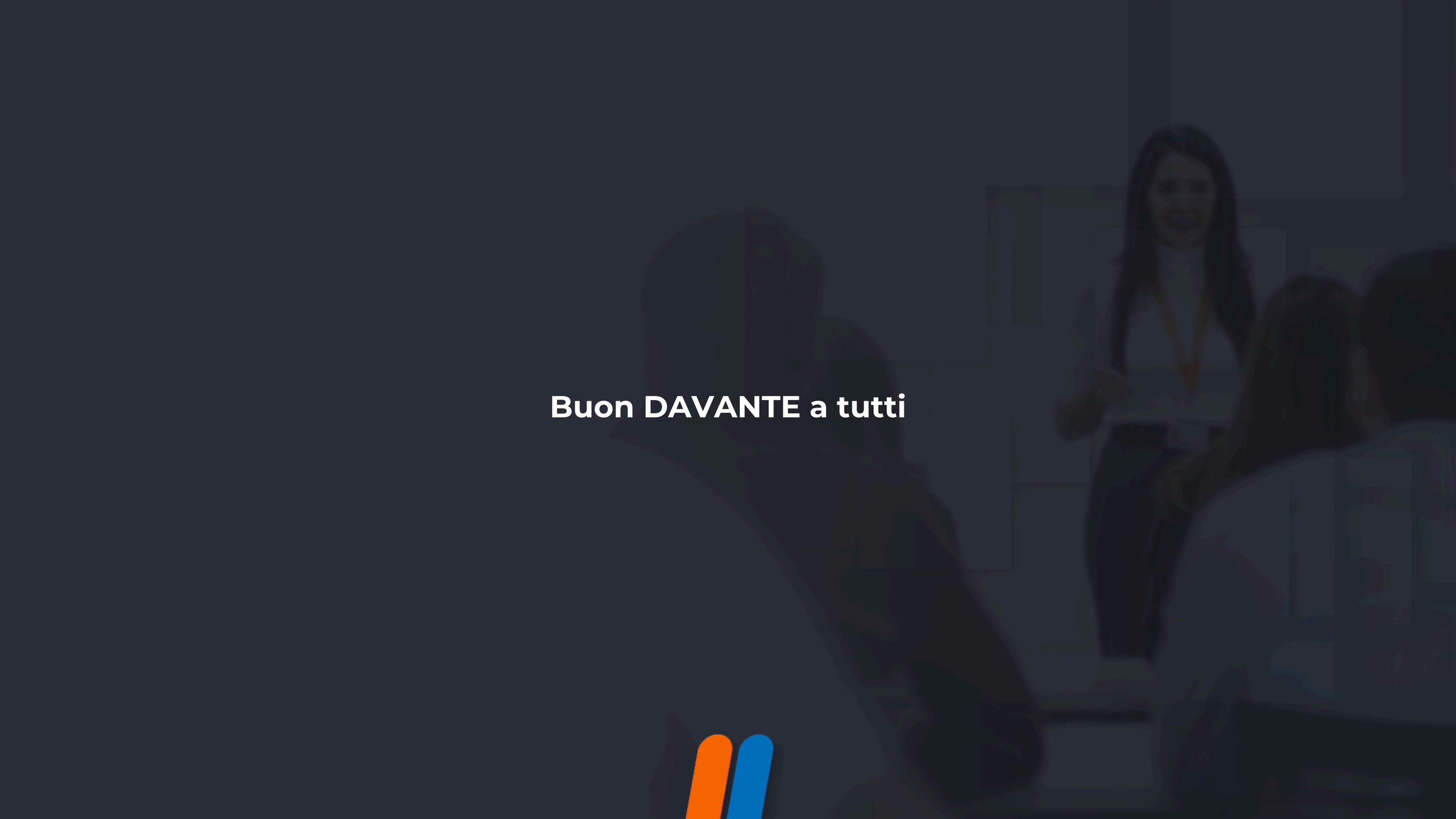
8. Creazione di un semplice REST API

```
const express = require('express');
const app = express();

app.use(express.json());

app.post('/user', (req, res) => {
  const { name } = req.body;
  res.json({ message: `Utente ${name} creato` });
});

app.listen(3000);
```


Buon DAVANTE a tutti