**NODE JS – 4 SETTEMBRE 2025**

**🔹 Cos’è Node.js**

* **Node.js** è un **runtime JavaScript open-source** basato sul motore **V8 di Chrome**.
* Permette di eseguire JavaScript **fuori dal browser**, ad esempio su un server o sul nostro computer.
* Con Node.js possiamo scrivere non solo il **frontend**, ma anche il **backend** delle applicazioni.
* È progettato per creare applicazioni **performanti e scalabili**, grazie al suo modello **event-driven** e **non bloccante**.

**🔹 Modello tradizionale: Thread per richiesta**

In linguaggi come **Java, PHP, .NET classico** il server gestisce le richieste così:

1. Arriva una richiesta HTTP.
2. Il server crea o assegna un **thread** dedicato.
3. Quel thread esegue tutta la logica (I/O, calcoli, DB).
4. Quando ha finito, libera le risorse.

👉 **Ogni richiesta = un thread**.

**✅ Vantaggi**

* Isolamento: un thread non blocca gli altri.
* Codice lineare e semplice da scrivere.
* Compatibile con molte librerie storiche.

**❌ Svantaggi**

* Creare e gestire thread è costoso (memoria, CPU).
* Scalabilità limitata: con migliaia di richieste simultanee, il server rischia di saturarsi.
* Context switch frequenti → overhead.

**🔹 Modello Node.js: Event loop (asincrono)**

Node.js **non crea un thread per ogni richiesta**.  
Usa un **singolo thread principale** e un **event loop** che gestisce tutto.  
  
L’**event Loop** è il cuore di Node.js: è il meccanismo che controlla l’esecuzione del codice, la gestione degli eventi e le operazioni asincrone.  
Il loop principale verifica continuamente se ci sono eventi da gestire e li esegue in ordine.

**Come funziona:**

1. Il **thread principale** riceve la richiesta.
2. Se è un’operazione **leggera** (es. parsing JSON, calcolo veloce) → la gestisce subito.
3. Se è un’operazione **lenta** e prevede I/O (es. query DB, accesso file) → la delega a worker interni o al sistema operativo, senza bloccare l’esecuzione del programma.
4. Node.js continua a eseguire altro codice mentre aspetta che l’I/O finisca.
5. Quando l’operazione è pronta → viene messo un **evento in coda**.
6. L’**event loop** prende l’evento in coda e richiama il **callback** o la **promise**.

👉 **Tutte le richieste passano dallo stesso thread**, che resta sempre libero di processare altro mentre le operazioni lente vanno avanti in parallelo.

**✅ Vantaggi**

* Gestione di **migliaia di connessioni contemporanee** con poche risorse, aumentando responsività ed efficienza.
* Nessun overhead da creazione/gestione di thread multipli.
* Perfetto per applicazioni **real-time** (chat, streaming, API ad alto traffico).

**❌ Svantaggi**

* Se il codice contiene un’operazione **sincrona e pesante** (es. calcoli intensivi), l’event loop si blocca → tutto il server si ferma.
* Richiede programmazione **asincrona** (callback, promise, async/await), meno immediata.
* Non ideale per applicazioni **CPU-bound** (molti calcoli complessi), dove servono **worker threads** o microservizi dedicati.

**🔹 Tabella comparativa**

| **Caratteristica** | **Thread per richiesta** | **Node.js (event loop)** |
| --- | --- | --- |
| **Gestione richieste** | 1 thread per ogni richiesta | 1 solo thread + delega |
| **Scalabilità** | Limitata  (molti thread = molte risorse) | Altissima  (migliaia di connessioni con poche risorse) |
| **Modello di programmazione** | Lineare e semplice (sincrono) | Asincrono (callback, promise, async/await) |
| **Efficienza memoria/CPU** | Bassa con molte richieste | Alta |
| **Adatto a** | App con poche richieste ma logica pesante | App real-time, API ad alto traffico, I/O intensivo |
| **Problema principale** | Overhead di thread, context switch | Event loop bloccato da codice sincrono pesante |

📌 **In sintesi**:

* Se l’applicazione è **I/O-bound** (API, DB, file system, streaming) → **Node.js è ideale**.
* Se è **CPU-bound** (molti calcoli complessi) → meglio un modello multi-thread o integrare Node.js con worker threads/microservizi.

**Event driven:** significa che c’è un singolo thread che continua a leggere tutto il codice, mentre alcune azioni più onerose vengono demandate ad altri attori che fanno le operazioni e riportano al thread principale il risultato di queste operazioni.

**🔹 NVM**

Per gestire più versioni di NodeJS si usa NVM  
  
Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

Per vedere tutte le versioni di NodeJS gestire da NVM:

**nvm list**

**🔹 NPM**

**NPM** è il **gestore di pacchetti predefinito** per Node.js

Permette di **installare e aggiornare pacchetti** di terze parti, **condividere moduli** sviluppati da te e **gestire le dipendenze** di un progetto attraverso un file di configurazione (package.json)

Per creare un nuovo progetto Node.js, esegui:

**npm init**

Quando installi un pacchetto, puoi specificare se è solo per sviluppo:

**npm install <pkg> --save-dev**

Questo lo aggiunge alla sezione devDependencies del package.json e si assicura che venga tolto dalla produzione.

Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

**Cartella progetto**  
**npm init**  
Per aprire il server:  
  
**node nomefile.js**

(es. index.js, node.js a seconda di come lo chiamiamo)

**🖥️ Cosa vuol dire “costruire un server”**

Un **server** non è altro che un **programma in esecuzione su un computer** (un PC locale o un computer remoto) che:

1. **Sta in ascolto** su una porta di rete (es. 3000).
2. **Riceve richieste** da altri dispositivi (es. il browser).
3. **Risponde** con dei dati (es. HTML, JSON, testo).

In Node.js “costruire un server” significa **scrivere del codice che fa questo lavoro**.

Esempio di base:  
  
Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

👉 Da qui in poi, se apri il browser e vai su http://localhost:3000, stai **parlando col tuo server**.

Qual è quindi il problema?  
  
Un server:

* **ascolta** le richieste degli utenti,
* **gestisce** i dati della richiesta (req),
* **invia** una risposta (res).

👉 In Node.js puro, **tutto avviene dentro** **una sola funzione** passata dentro http.createServer.

Se vuoi:

* controllare l’utente,
* loggare i dati,
* validare l’input,
* servire pagine diverse,

devi mettere tutto dentro quella funzione. Diventa **caotico**.

**🔹 Express**

**Express:** framework leggero ma potente per creare applicazioni web e API con Node.js.

Permette di definire facilmente rotte, usare middleware personalizzati e integrare template engine.

Immagine che contiene testo, schermata, Carattere, documento

Il contenuto generato dall'IA potrebbe non essere corretto.È lo standard per il backend JavaScript.

**Express e gestione del routing**

**In Express, ogni rotta corrisponde a un metodo HTTP e a un percorso.**

app.get('/', (req, res) **=>** {

res.send('Benvenuto nella homepage');

});

app.post('/contatti', (req, res) **=>** {

res.send('Richiesta inviata con successo');

});

**Possiamo anche definire rotte dinamiche:**

app.get('/utente/:id', (req, res) **=>** {

res.send(`Profilo utente con ID ${req.params.id}`);

});

**Per mantenere il codice organizzato, possiamo suddividere le rotte in moduli separati.**

*// app.js*

**const** utentiRouter **=** require('./routes/utenti');

app.use('/utenti', utentiRouter);

**Gestione delle richieste di routing:**

*// routes/utenti.js*

**const** express **=** require('express');

**const** router **=** express.Router();

router.get('/', (req, res) **=>** {

res.send('Lista utenti');

});

router.get('/:id', (req, res) **=>** {

res.send(`Dettagli utente ${req.params.id}`);

});

module.exports **=** router;

**🔹 Codici di risposta HTTP**

**✅ Successo (2xx)**

* **200 OK** → La richiesta è andata a buon fine. (Esempio: una GET che restituisce i dati richiesti)
* **201 Created** → Risorsa creata con successo. (Esempio: una POST che crea un nuovo utente)
* **202 Accepted** → La richiesta è stata accettata, ma non ancora completata.
* **204 No Content** → Richiesta riuscita, ma nessun contenuto da restituire.

**🔄 Reindirizzamenti (3xx)**

* **301 Moved Permanently** → La risorsa è stata spostata in modo permanente.
* **302 Found** → Reindirizzamento temporaneo.
* **304 Not Modified** → I dati non sono cambiati (utile per la cache).

**⚠️ Errori Client (4xx)**

* **400 Bad Request** → La richiesta è errata o mancano parametri.
* **401 Unauthorized** → Autenticazione necessaria o fallita.
* **403 Forbidden** → Accesso negato, anche se autenticato.
* **404 Not Found** → Risorsa non trovata.
* **409 Conflict** → Conflitto nella richiesta (es. doppio inserimento di un dato unico).
* **422 Unprocessable Entity** → I dati inviati sono corretti a livello di sintassi, ma non validi a livello logico (es: email non valida).

**🚨 Errori Server (5xx)**

* **500 Internal Server Error** → Errore generico del server.
* **502 Bad Gateway** → Il server ha ricevuto una risposta non valida da un altro server.
* **503 Service Unavailable** → Servizio non disponibile (sovraccarico o in manutenzione).
* **504 Gateway Timeout** → Timeout nella comunicazione tra server.

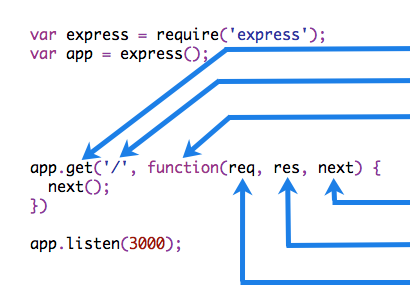
**🔹 Metodi HTTP principali**

* **GET** → Recupera informazioni dal server.  
  (Esempio: GET /users → ottieni la lista degli utenti).
* **POST** → Invia dati al server per creare una nuova risorsa.  
  (Esempio: POST /users → crea un nuovo utente).
* **PUT** → Aggiorna una risorsa esistente **sostituendola completamente**.  
  (Esempio: PUT /users/1 → aggiorna l’utente con ID=1 sovrascrivendo tutti i suoi dati).
* **PATCH** → Aggiorna **parzialmente** una risorsa esistente.  
  (Esempio: PATCH /users/1 → aggiorna solo il campo email dell’utente con ID=1).
* **DELETE** → Elimina una risorsa.  
  (Esempio: DELETE /users/1 → elimina l’utente con ID=1).

**Middleware:**

Un middleware è una funzione che viene eseguita tra la **richiesta del client** e la **risposta finale** del server.  
  
Invece di avere **una sola funzione gigante** come in Node JS puro, fa spezzare la logica in **più funzioni più piccole** → i **middleware**.  
  
Ogni middleware:

1. Riceve la richiesta (req) e la risposta (res).
2. Fa una parte del lavoro.
3. Decide se:
   * Rispondere subito (res.send(...)),
   * Oppure passare il controllo al **prossimo middleware** (next()).



Metodo HTTP per cui si applica la funzione middleware.

Percorso (route) per cui si applica la funzione middleware.

La funzione middleware.

Argomento di callback nella funzione middleware, denominata per convenzione "next".

Argomento [risposta](https://expressjs.com/it/4x/api.html#res) HTTP nella funzione middleware, denominato "res" per convenzione.

Argomento [richiesta](https://expressjs.com/it/4x/api.html#req) HTTP nella funzione middleware, denominato "req" per convenzione.

**Creazione di un server con Express**  
  
Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

app.use((req, res, next) **=>** {

console.log(`Metodo: ${req.method} | Percorso: ${req.url}`);

next(); *// Passa al middleware successivo*

});

app.use(express.json());

app.use(express.urlencoded({ extended: true }));

app.use(express.static('public'));

* ***express.json()****:* parse del corpo delle richieste JSON, disponibile in req.body
* ***express.urlencoded({ extended: true })***: parsifica dati application/x-www-form-urlencoded, utile per i form HTML
* ***express.static('public')***: serve risorse statiche come immagini, CSS o JS da una cartella indicata

**BEST PRACTICES**

1. Mantieni i middleware **leggeri** e con **singola responsabilità**
2. **Ordine importante**: ad esempio, prima logging o security, poi parsing e routing
3. **Gestione errori**: se qualcosa va storto, usa next(err) o middleware dedicati
4. **Memo per next():** se un middleware non chiama next() o non risponde, la richiesta rimane **sospesa**

**Middleware personalizzati:**

Possiamo creare middleware per proteggere aree riservate:

**function** isAuthenticated(req, res, next) {

**if** (req.session?.utente) **return** next();

res.redirect('/login');

}

Per usarlo in una route possiamo fare come segue:

app.get('/dashboard', isAuhenticated, (req, res) **=>** {

res.send('Benvenuto nella tua dashboard');

});

**Gestione delle eccezioni**

Per gestire utenti loggati e mantenere lo stato, possiamo usare express-session.

**const** session **=** require('express-session');

app.use(session({

secret: 'chiave-segreta',

resave: false,

saveUninitialized: false

}));

**Login e Logout**

Esempio di autenticazione base:  
  
app.post('/login', (req, res) **=>** {

**const** { username, password } **=** req.body;

**if** (username **===** 'admin' **&&** password **===** '1234') {

req.session.utente **=** username;

res.redirect('/dashboard');

} **else** {

res.send('Credenziali non valide');

}

});  
  
Per il logout:

app.post('/logout', (req, res) **=>** {

req.session.destroy(() **=>** {

res.redirect('/');

});

});

**Template engine**  
Un **template engine** consente di **generare HTML dinamico** in modo più strutturato rispetto alla concatenazione di stringhe.

In Express.js, i template engine vengono utilizzati per:

* Separare la **logica** di business dalla **presentazione**
* Inserire facilmente dati all’interno del markup
* Gestire layout comuni (es. header, footer)
* Riutilizzare componenti visuali (es. partials)

Esempio di setup con Express

// Backend  
  
**const** express **=** require('express');

**const** app **=** express();

*// imposta EJS come motore*

app.set('view engine', 'ejs');

*// directory delle view*

app.set('views', './views');

app.get('/', (req, res) **=>** {

res.render('index', { title: 'Benvenuto', user: 'Luca' });

});

app.listen(3000);

// index.hml

<!DOCTYPE html>

<html>

<head>

<title><%= title %></title>

</head>

<body>

<h1>Ciao, <%= user %>!</h1>

</body>

</html>

**Utilizzo del template egine in Express.js**

✅ **Layout riutilizzabili:** con template engine come EJS o Handlebars, puoi creare layout base (layout.ejs) con header/footer condivisi e includerli nelle pagine con include.

✅ **Partial:** porzioni di codice HTML riutilizzabili, come card, menu o sidebar.

✅ **Controlli logici** (if, for):

I template engine supportano iterazioni e condizioni per rendere il contenuto dinamico in base ai dati.

**Best practice:**

* **Mantieni la logica fuori dai template**: calcola tutto prima e passa i dati via res.render
* **Organizza le view in sottocartelle**: ad esempio views/pages, views/partials.
* **Rendi i template puliti**: usa i partial per separare header, footer, navbar, ecc.
* **È molto importante l’ordine in cui sono scritti i middleware** (perché next passa al middleware successivo. Es. prima ci deve essere un utente loggato, poi l’utente loggato può leggere cose).

**Esercizi**

**Creare un router per un sito con 3 pagine**  
**Obiettivo**: Crea una piccola app Express con tre pagine.

🛠️ **Istruzioni**:

1. Inizializza un nuovo progetto Node.js
2. Installa Express
3. Crea tre rotte:
   * / → “Homepage”
   * /chi-siamo → “Chi siamo”
   * /contatti → “Contattaci!”

**Middleware personalizzato**  
**Obiettivo:** Loggare data, ora e URL di ogni richiesta.

🛠️ **Istruzioni**:

1. Crea un middleware globale che stampi in console:
   * Metodo
   * URL
   * Data/ora

**Utilizzo template engine**

**Obiettivo:** Implementa un login base con express-session.

🛠️ **Istruzioni**:

1. Installa express-session
2. Crea una rotta /register che registra un utente (get e post)
3. Crea una rotta /login che salva un utente in sessione (get e post)
4. Crea una rotta /dashboard accessibile solo se l’utente è loggato
5. Crea una rotta /logout che distrugge la sessione

**Lista dinamica**  
**Obiettivo:** Visualizza una lista dinamica di prodotti.

🛠️ **Istruzioni**:

1. Crea una rotta /prodotti
2. Passa un array di prodotti a EJS
3. Mostra i prodotti in una lista HTML

**Area riservata con Passport.js**  
🎯 **Obiettivo**

Sviluppare un'applicazione Express.js con:

* Registrazione di nuovi utenti
* Login con autenticazione locale usando Passport
* Accesso a una dashboard privata
* Logout
* Persistenza dei dati in memoria (opzionalmente con un file JSON)

**Database e Sicurezza in NodeJS**

Quando sviluppiamo in Node.js, possiamo interagire sia con database SQL (relazionali) che con NoSQL (non relazionali).

La sicurezza è cruciale perché un database vulnerabile può portare a:

* Furto di dati sensibili (e-mail, password, carte di credito)
* Corruzione dei dati tramite query malevole
* Down dell’applicazione con attacchi di tipo DoS

Principali rischi:

* SQL Injection
* NoSQL Injection
* Accesso non autorizzato
* Esposizione di credenziali nel codice

La **SQL Injection** è una delle vulnerabilità più comuni e pericolose nelle applicazioni web.

Si verifica quando **un attaccante inserisce del codice SQL malevolo** nei campi di input (form, URL, API) per **manipolare le query al database**.

🔴 **Obiettivo dell’attacco**: accedere, modificare o cancellare dati sensibili senza autorizzazione.

**const** username **=** req.body.username;

**const** password **=** req.body.password;

**const** query **=** `SELECT \* FROM users WHERE username = '${username}' AND password = '${password}'`;

connection.query(query, (err, result) **=>** {

**if** (result.length **>** 0) {

res.send("Login riuscito!");

} **else** {

res.send("Credenziali errate!");

}

});

**Problema**: L’input dell’utente viene concatenato direttamente alla query SQL → porta a vulnerabilità.

**SELECT \* FROM users WHERE username = 'admin' AND password = '' OR '1'='1';**

Risultato: la condizione 1=1 è sempre vera → l’attaccante ottiene l’accesso senza credenziali valide.

**const** query **=** "SELECT \* FROM users WHERE username = ? AND password = ?";

connection.query(query, [username, password], (err, result) **=>** {

**if** (result.length **>** 0) {

res.send("Login riuscito!");

} **else** {

res.send("Credenziali errate!");

}

});

**Soluzione:** Query Parametrizzate

La **NoSQL Injection** è un tipo di attacco che colpisce le applicazioni che usano **database NoSQL** (es. MongoDB).

Accade quando l’applicazione **non valida correttamente l’input** e l’attaccante riesce a **iniettare operatori NoSQL** nelle query, ottenendo **accesso non autorizzato ai dati**.

🔴 **Obiettivo dell’attacco**: bypassare l’autenticazione, estrarre o modificare dati sensibili.

**const** user **=** **await** User.findOne({

username: req.body.username,

password: req.body.password

});

**if** (user) {

res.send("Login riuscito!");

} **else** {

res.send("Credenziali errate!");

}

**Problema**: L’input dell’utente viene usato direttamente nella query, senza validazione → vulnerabile a NoSQL Injection.

**{**

**"username": { "$ne": null },**

**"password": { "$ne": null }**

**}**

User.findOne({ username: { $ne: null }, password: { $ne: null } });

**Soluzione:** Usiamo librerie come validator o Joi per accettare solo stringhe valide

L’**accesso non autorizzato** si verifica quando un utente riesce a **entrare in un sistema, un’API o un database** senza avere le **credenziali corrette** o i **permessi necessari**.

Può accadere per **errori di configurazione**, **mancanza di controlli di autenticazione** o **falle di sicurezza**.

🔴 **Obiettivo dell’attacco**: ottenere dati sensibili o compiere azioni riservate senza avere i privilegi necessari.

L’**esposizione di credenziali** si verifica quando **informazioni sensibili** come:

* **Username e password**
* **API keys**
* **Token di accesso**
* **Certificati e secrets**

vengono **inserite direttamente nel codice sorgente** o in **file pubblici** (ad esempio su GitHub).

Questa è una delle vulnerabilità più comuni e pericolose nello sviluppo di applicazioni Node.js.

🔴 Conseguenze possibili:

* Accesso non autorizzato ai database
* Compromissione di API e servizi esterni
* Violazione dei dati degli utenti
* Potenziale perdita di reputazione e sanzioni GDPR

**const** db **=** mysql.createConnection({

host: 'localhost',

user: 'admin',

password: 'SuperPassword123!',

database: 'myapp'

});

**const** apiKey **=** "sk\_live\_51Lm..."; *// API key Stripe*

Se questo file viene caricato su GitHub o condiviso, **chiunque** può usare le tue credenziali per accedere ai servizi.

**L’XSS** è un attacco che permette a un hacker di **iniettare codice JavaScript malevolo** in una pagina web visualizzata da altri utenti.

🔹 Obiettivo: rubare cookie, token di sessione o manipolare l’interfaccia utente.

<input type="text" value="<script>alert('Hacked!')</script>">

Se l’input viene mostrato senza sanificazione, il browser eseguirà lo script → vulnerabilità XSS

1. **Sanificazione degli Input**: usiamo librerie come validator o DOMPurify.
2. Impostare **Header di Sicurezza** con Helmet
3. **Usare res.json() o res.send() in modo sicuro**: non concatenare mai HTML dinamico con dati degli utenti.

**Il CSRF** è un attacco che forza un utente autenticato a **inviare richieste non autorizzate al server.**

🔹 L’attaccante sfrutta i **cookie di sessione** salvati nel browser.

<img src="https://banca.com/transfer?to=attacker&amount=1000">

Se l’utente è loggato, il browser invia automaticamente i cookie → trasferimento completato.

1. **Usare Token Anti-CSRF**: usiamo librerie come csurf, ogni form include un token univoco → l’attacco fallisce..
2. Usare **SameSite** per i Cookie: i cookie non vengono inviati da domini esterni.
3. **Usare Header Personalizzati**: Richiedi un header X-Requested-With per le API → i browser non lo inviano automaticamente.

Il **Rate Limiting** serve a **limitare il numero di richieste** che un utente o IP può fare in un certo intervallo di tempo. Protegge contro:

* **Attacchi brute-force** (es. login)
* **DDoS** e abusi di API
* **Consumatori eccessivi** di risorse

**import** rateLimit **from** "express-rate-limit";

**const** apiLimiter **=** rateLimit({

windowMs: 15 **\*** 60 **\*** 1000, *// 15 minuti*

max: 100, *// massimo 100 richieste per IP*

message: "Troppe richieste! Riprova più tardi."

});

app.use("/api", apiLimiter);

Ogni IP può fare **100 richieste ogni 15 minuti**.

Per collegarsi a un database SQL, possiamo usare librerie come mysql2, pg o mssql.

**import** mysql **from** 'mysql2';

**const** connection **=** mysql.createConnection({

host: 'localhost',

user: 'root',

password: 'mypassword',

database: 'miodb'

});

connection.connect(err **=>** {

**if** (err) **throw** err;

console.log("Connesso al database!");

});

Per database **NoSQL** come MongoDB, la libreria più usata è **Mongoose**.

**import** mongoose **from** 'mongoose';

mongoose.connect('mongodb://localhost:27017/miodb', {

useNewUrlParser: true,

useUnifiedTopology: true

})

.then(() **=>** console.log("Connesso a MongoDB"))

.catch(err **=>** console.error(err));

**ORM**

Un **ORM** è una libreria o framework che permette di **interagire con un database** (SQL o NoSQL) **usando oggetti e metodi**, invece di scrivere query SQL manualmente.

🔹 **Obiettivo**: semplificare la gestione dei dati e rendere il codice più leggibile e manutenibile.

**Come funziona.**

**L’ORM mappa le tabelle del database su modelli/oggetti nel codice.**

Invece di scrivere query SQL complesse, si usano **metodi JavaScript** per **creare**, **leggere**, **aggiornare** ed **eliminare** (CRUD) dati.

**const** query **=** "SELECT \* FROM users WHERE email = ?";

connection.query(query, [email], (err, result) **=>** {

console.log(result);

});

**const** user **=** **await** User.findOne({ where: { email } });

console.log(user);

🔹 Risultato: meno codice, più leggibile, gestione dei dati più semplice.

Vaantaggi  
Semplifica lo sviluppo: non serve conoscere a fondo SQL → usi metodi JavaScript.

* Migliora la leggibilità del codice: query più brevi e intuitive.
* Portabilità: stesso codice → database diversi (MySQL, PostgreSQL, ecc.).
* Gestione delle migrazioni: gli ORM permettono di aggiornare la struttura del database in modo semplice.
* Validazione e modellazione dei dati: definizione di schemi per garantire che i dati siano sempre coerenti.
* Protezione da SQL Injection: gli ORM usano query parametrizzate di default.
* Performance inferiore: le query generate dall’ORM non sempre sono ottimizzate come quelle scritte a mano.
* Curva di apprendimento: gli ORM complessi (come Sequelize) hanno una sintassi articolata.
* Overhead inutile per progetti piccoli: per un’app semplice, un ORM può introdurre complessità non necessaria.
* Debug più complicato: in caso di errori, è più difficile capire che query stia generando l’ORM.
* Limitazioni nelle query avanzate: se servono query molto complesse, a volte si torna a scriverle a mano.

✅ Buona scelta se:

* Vuoi sviluppare velocemente
* Hai bisogno di portabilità tra database
* Vuoi ridurre il rischio di vulnerabilità