Il problema: Express non impone alcuna struttura

Express è volutamente minimale.
Di default, lascia a te la libertà (o il peso) di decidere come strutturare il codice.
All’aumentare della complessità, l’assenza di una struttura porta a problemi evidenti:

- Codice duplicato o accavallato
- Responsabilità non separate
- File monolitici difficili da testare o estendere
- Scarsa manutenibilità

Una struttura modulare

Una struttura chiara ed esplicita consente di:

- Separare le responsabilità: routing, controller, logica applicativa, accesso ai dati
- Facilitare l’estensione: aggiungere nuove feature senza rifattorizzare tutto
- Favorire il testing: unit test per servizi e controller
- Ridurre il coupling tra livelli dell'applicazione

Costruiamo una struttura

index.js – Bootstrap dell'applicazione

Responsabilità:

- Inizializzazione dell’app Express
- Registrazione dei middleware
- Routing verso i moduli funzionali

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

const userRoutes = require('./routes/users.route');
app.use('/users', userRoutes);

app.listen(3000);
```

services/ – Logica applicativa

I servizi incapsulano la logica di business.

Responsabilità:

- Astrazione sul livello dati (DB o altro)
- Composizione di operazioni
- Eventuale accesso a modelli, provider esterni, ecc.

```
exports.getUsers = async () => {
  return [ { name: 'Mario' }, { name: 'Lucia' } ];
};
```

controllers/ – Gestione della richiesta HTTP

I controller sono adapter layer:

- Validano e interpretano i dati della request
- Coordinano i servizi da chiamare
- Producono la response

```
const service = require('../services/users.service');

exports.getAllUsers = async (req, res) => {
  const users = await service.getUsers();
  res.status(200).json(users);
};
```

routes/ – Definizione degli entry point HTTP

Le route definiscono gli entry point dell’API e
delegano la gestione al controller.

```
const router = require('express').Router();
const controller = require('../controllers/users.controller');

router.get('/', controller.getAllUsers);
module.exports = router;
```

I middleware

In Express, i middleware sono funzioni che ricevono i seguenti argomenti:
(req, res, next)

Il parametro next è una funzione che dice al server di passare al prossimo middleware.
Se next() non viene chiamato e non viene inviata una risposta(res.send, res.json, ecc.), il server resterà bloccato in attesa.

```
app.use((req, res, next) => {
  console.log('Richiesta ricevuta');
  next(); // Passa al prossimo middleware
});
```

EventEmitter

La classe EventEmitter, è lo strumento di node.js per la gestione di eventi asincroni.
Ogni istanza di EventEmitter può emettere eventi e registrare listener associati.

Questo pattern permette di separare le responsabilità del codice e reagire a determinati eventi in modo ordinato.

- I listener si registrano tramite .on() o .once().
- Gli eventi si emettono con .emit().
- È possibile rimuovere listener con .off() o .removeListener().

Con questo meccanismo potremo implementare un flusso di lavoro reattivo senza introdurre direttamente callback o promise in ogni punto del codice.

EventEmitter

```
const EventEmitter = require('events');

const emitter = new EventEmitter();

emitter.on('saluto', (nome) => {
  console.log(`Ciao, ${nome}!`);
});

emitter.emit('saluto', 'Alice'); //Ciao, Alice!
```

Eventi Predefiniti

In molte delle sue classi core, soprattutto quelle che estendono EventEmitter, node fornisce eventi predefiniti.

Questi eventi sono già definiti e generati internamente da Node.js, e noi possiamo semplicemente registrarci per ascoltarli (usando on/once).