1 Approfondimento: qualità del codice e buone pratiche

L’obiettivo non è solo “far funzionare”, ma costruire un backend **leggibile, prevedibile e mantenibile**. Questo rende il codice più facile da testare, correggere e far crescere.

A) Strutturare il progetto in moduli e cartelle pulite

Una struttura tipica e professionale:

```
src/
├─ server.js              # avvio app, wiring dei middleware e router
├─ app.js                 # istanzia express, monta middleware/rotte
(senza .listen)
├─ config/                # configurazioni (db, carico .env, cors policy)
│  ├─ env.js
│  └─ db.js
├─ routes/                # definisce solo le rotte e “chiama” i controller
│  └─ user.routes.js
├─ controllers/           # logica HTTP (legge req, chiama service, manda res)
│  └─ user.controller.js
├─ services/              # logica di business riusabile (no oggetti req/res)
│  └─ user.service.js
├─ models/                # modelli Mongoose/ORM
│  └─ user.model.js
├─ middleware/            # middleware personalizzati
│  ├─ errorHandler.js
│  ├─ validate.js
│  └─ auth.js
├─ utils/                 # utilità (logger, helper, email, ecc.)
│  └─ logger.js
└─ docs/                  # documentazione API (Swagger/OpenAPI YAML)
   └─ openapi.yaml
```

Perché aiuta?
Separi responsabilità: le **routes** non fanno business, i **controller** orchestrano, i **service** implementano regole/operazioni, i **models** parlano col DB.

B) Variabili d’ambiente e gestione sicura delle chiavi

Tutte le configurazioni sensibili (chiavi JWT, stringhe DB, API keys) **fuori dal codice** in `.env`.

`.env`

```
PORT=3000
MONGO_URI=...
JWT_SECRET=supersegreto
NODE_ENV=development
```

`src/config/env.js`

```
require('dotenv').config();

const env = {
  nodeEnv: process.env.NODE_ENV || 'development',
  port: Number(process.env.PORT) || 3000,
```

```
  mongoUri: process.env.MONGO_URI,
  jwtSecret: process.env.JWT_SECRET
};

// piccola validazione per non avviare l’app senza variabili critiche
['mongoUri', 'jwtSecret'].forEach((k) => {
  if (!env[k]) {
    console.error(` Variabile d'ambiente mancante: ${k.toUpperCase()}`);
    process.exit(1);
  }
});

module.exports = env;
```

Non committare mai `.env` (aggiungi a `.gitignore`). In produzione, imposta le env nel pannello del servizio.

C) Errori chiari e coerenti: un unico `errorHandler.js`

Crea un **middleware di errore globale** per rispondere in modo uniforme e nascondere dettagli interni.

`src/middleware/errorHandler.js`

```
// Classe opzionale per errori applicativi
class AppError extends Error {
  constructor(status, code, message) {
    super(message);
    this.status = status;
    this.code = code;
  }
}

const notFound = (req, res, next) => {
  res.status(404).json({ error: 'NOT_FOUND', message: `Route ${req.originalUrl}
non trovata` });
};

const errorHandler = (err, req, res, next) => {
  // Log tecnico (usa un logger vero nella pratica)
  console.error('', err);

  // Errori "conosciuti"
  if (err instanceof AppError) {
    return res.status(err.status).json({ error: err.code, message:
err.message });
  }

  // Errori di validazione (es. Joi)
  if (err.isJoi) {
    return res.status(400).json({
      error: 'INVALID_INPUT',
      message: 'Dati non validi',
      details: err.details?.map(d => d.message) || []
    });
  }

  // Fallback generico
  res.status(500).json({ error: 'INTERNAL_ERROR', message: 'Si è verificato un
errore inatteso' });
};
```

```
module.exports = { AppError, notFound, errorHandler };
```

Ordine nel server:
middleware globali → routes → `notFound` → `errorHandler`.

D) Separare la logica: controller vs router

Router: definisce gli endpoint e delega.
Controller: gestisce req/res, chiama il **service**, non parla col DB direttamente.

`src/routes/user.routes.js`

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

router.post('/', userController.create);
router.get('/', userController.list);

module.exports = router;
```

`src/controllers/user.controller.js`

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

exports.create = async (req, res, next) => {
  try {
    const user = await userService.create(req.body); // nessun req/res qui
    res.status(201).json(user);
  } catch (err) { next(err); }
};

exports.list = async (req, res, next) => {
  try {
    const users = await userService.list();
    res.json(users);
  } catch (err) { next(err); }
};
```

E) Commentare e documentare le API (JSDoc / Swagger)

JSDoc nei controller per spiegare input/output. **Swagger/OpenAPI** per generare documentazione navigabile.

Esempio **Swagger YAML** minimissimo (`docs/openapi.yaml`):

```
openapi: 3.0.0
info:
  title: Demo API
  version: 1.0.0
paths:
  /api/users:
    get:
      summary: Lista utenti
      responses:
        '200':
          description: OK
    post:
      summary: Crea utente
```

```
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              required: [email, password]
              properties:
                email: { type: string, format: email }
                password: { type: string, minLength: 6 }
      responses:
        '201':
          description: Creato
```

Puoi poi servirlo con `swagger-ui-express` per una UI interattiva.

2 Sicurezza base e best practices per API pubbliche

L’obiettivo è ridurre il rischio di abusi e fughe di dati con poche mosse efficaci.

A) Rate limiting (blocca troppi tentativi)

Previene brute force o abusi su endpoint critici (login, registrazione).

```
const rateLimit = require('express-rate-limit');

const authLimiter = rateLimit({
  windowMs: 15 * 60 * 1000,   // 15 minuti
  max: 100,                   // max 100 richieste/ IP / finestra
  standardHeaders: true,
  legacyHeaders: false
});

// app.use('/api/auth/', authLimiter); // applicalo ad area sensibile
```

B) Helmet (header di sicurezza)

```
const helmet = require('helmet');
app.use(helmet());
```

Imposta header HTTP sicuri (X-Content-Type-Options, X-DNS-Prefetch-Control, ecc.).

C) Sanitizzazione input

- Evita **XSS** e stringhe maliziose nei campi di testo.
- Validazione con Joi + (se necessario) sanificazione con librerie come `validator` o `xss`.

Esempio (idea): rimuovere tag HTML indesiderati prima di salvare.

D) CORS mirato (solo frontend autorizzati)

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

const allowed = ['https://tuo-frontend.it', 'http://localhost:5173'];
app.use(cors({
  origin: (origin, cb) => {
    if (!origin || allowed.includes(origin)) return cb(null, true);
    cb(new Error('Not allowed by CORS'));
  },
  credentials: true
}));
```

E) Timeout e messaggi d’errore generici

- Imposta un timeout di richiesta (es. con proxy/nginx o lib apposita).
- Non rivelare stack interni: l’`errorHandler` deve rispondere in modo **generico** (log interni più dettagliati, risposta utente sobria).

3 Deploy e strumenti professionali (overview operativa)

A) Dev vs Production

- **Development**: logging verboso, errori dettagliati, hot reload.
- **Production**: logging moderato/strutturato, messaggi d’errore generici, `.env` impostate sul server, CORS restringente, rate limit attivo, cache HTTP dove possibile.

B) Punti fermi prima del deploy

- `package.json` con `"start": "node src/server.js"`
- `.env` **non** nel repo
- variabili d’ambiente pronte (DB, segreti)
- test base con Postman/curl locali
- log e error handler funzionanti

C) Testing online

Dopo la pubblicazione:

- Riprova tutte le rotte con **Postman** (URL pubblico).
- Se usi Swagger, pubblica anche la UI per condivisione e test.

4 Introduzione a framework avanzati: Laravel o NestJS (apertura)

Scopo: mostrare che i concetti imparati sono **trasferibili**.

A) Cos’è un framework full-stack

Un framework come **Laravel (PHP)** o **NestJS (Node/TS)** offre:

- **struttura** decisa (cartelle, convenzioni)
- **tooling integrato** (CLI, generatori di moduli)
- **pattern** formali (MVC, moduli, dependency injection)
- **ecosistema** (autenticazione, code, mail, queue, cache, pipeline)

B) Differenze e affinità

Concetto	Express	NestJS	Laravel
Routing	manuale, libero	modulare, decorators	controller/route + middleware
Controller	sì (a mano)	sì (decorators)	sì (MVC classico)
Service layer	consigliato	nativo (providers)	nativo (Services, Repositories)
Middleware/Guards	sì	sì (guards, interceptors)	sì
Validazione	a mano/Joi	class-validator integrato	form request / validation rules
Doc API	swagger-ui-express	@nestjs/swagger	laravel-swagger / scrittura

Quello che avete imparato (routing, controller, service, middleware, validazione, auth) **vale ovunque**. Cambia il “dialetto”, non i concetti.

Conclusione

- Struttura pulita → progetti che crescono bene.
- Config sicure → niente segreti nel codice.
- Errori coerenti → debugging più rapido e UX migliore per i client.
- Sicurezza minima → riduce rischi con poche mosse efficaci.
- Mentalità “production-ready” → anche se restate in locale, pensate come se il vostro codice dovesse andare online domani.
- I concetti imparati in Express vi preparano a **framework strutturati** (NestJS, Laravel).