1. Validazione dei dati con Joi o Validator.js

Obiettivo

Validare i dati è il primo passo per rendere un'app sicura e stabile.
Invece di scrivere manualmente controlli (`if (!email || !password)`), si usa una **libreria dedicata** come `Joi` (più diffusa) o `Validator.js`.

Installazione

```
npm install joi
```

Concetto base

Con Joi si definisce uno **schema di validazione** che descrive la "forma" dei dati accettati.

Esempio:

```
const Joi = require('joi');

const schema = Joi.object({
  email: Joi.string().email().required(),
  password: Joi.string().min(6).required()
});
```

Questo significa:

- `email` deve essere una stringa e un indirizzo email valido;
- `password` deve essere una stringa di almeno 6 caratteri;
- entrambi sono obbligatori.

Middleware di validazione

Creiamo un middleware riutilizzabile che usa Joi per controllare i dati **prima** che arrivino al controller.

`middleware/validate.js`

```
const Joi = require('joi');

// Riceve uno schema e restituisce un middleware Express
module.exports = function validate(schema) {
  return (req, res, next) => {
    // Valida il corpo della richiesta (req.body)
    const { error, value } = schema.validate(req.body, { abortEarly: false });
```

```
    // Se i dati non sono validi
    if (error) {
      // Estraggo i messaggi di errore
      const details = error.details.map(d => d.message);
      return res.status(400).json({
        error: 'INVALID_INPUT',
        message: 'Dati non validi',
        details
      });
    }

    // Sovrascrivo req.body con la versione "pulita" (senza campi extra)
    req.body = value;
    next();
  };
};
```

Esempio pratico: registrazione utente

`routes/auth.js`

```
const express = require('express');
const Joi = require('joi');
const validate = require('../middleware/validate');

const router = express.Router();

// Definisco lo schema di validazione per la registrazione
const registerSchema = Joi.object({
  email: Joi.string().email().required(),
  password: Joi.string().min(6).required()
});

// Rotta di registrazione che usa il middleware di validazione
router.post('/register', validate(registerSchema), (req, res) => {
  // Se arriviamo qui, i dati sono già validi
  const { email, password } = req.body;
  res.status(201).json({ message: `Utente ${email} registrato con successo!` });
});

module.exports = router;
```

In pratica:

1. Joi controlla i dati prima del controller;
2. se qualcosa non va, la richiesta viene bloccata;
3. se tutto è ok, `req.body` contiene solo dati “puliti”.

Esempio di risposta a un errore

```
{
  "error": "INVALID_INPUT",
  "message": "Dati non validi",
  "details": [
    "\"email\" must be a valid email",
    "\"password\" length must be at least 6 characters long"
  ]
}
```

2. Modello “Service Layer”

Obiettivo

Separare la **logica di business** dalla **logica di routing**.
Le rotte devono solo gestire richieste e risposte, mentre le operazioni vere (creare, aggiornare, cancellare) stanno nei **service**.

Struttura del progetto

```
backend/
├── routes/
│   └── userRoutes.js
├── services/
│   └── userService.js
└── models/
    └── user.js
```

`services/userService.js`

```
// services/userService.js — contiene la logica "vera" del backend
const bcrypt = require('bcrypt');
const User = require('../models/user');

async function register({ email, password }) {
  // 1️⃣ Controlla se l'utente esiste
  const existing = await User.findOne({ email });
  if (existing) throw new Error('Email già registrata');

  // 2️⃣ Cifra la password
  const passwordHash = await bcrypt.hash(password, 10);

  // 3️⃣ Crea e salva l'utente
  const user = await User.create({ email, passwordHash });
  return user;
}
```

```
async function getAll() {
  return User.find().select('-passwordHash');
}

module.exports = { register, getAll };
```

routes/userRoutes.js

```
const express = require('express');
const userService = require('../services/userService');
const router = express.Router();

//  Rotta di registrazione
router.post('/register', async (req, res, next) => {
  try {
    const user = await userService.register(req.body);
    res.status(201).json(user);
  } catch (err) {
    next(err);
  }
});

//  Rotta per elenco utenti
router.get('/', async (req, res, next) => {
  try {
    const users = await userService.getAll();
    res.json(users);
  } catch (err) {
    next(err);
  }
});

module.exports = router;
```

- `routes/` → solo input/output
- `services/` → logica riutilizzabile (business logic)
- `models/` → comunicano con il database

➡ È il primo passo verso **MVC** e **Clean Architecture**.

3. Logging e monitoraggio

Obiettivo

Sostituire `console.log()` con un logger professionale come **Morgan** o **Winston**, che:

- registra richieste e risposte,
- mostra tempi di esecuzione,
- può salvare i log su file.

Installazione

```
npm install morgan winston
```

`server.js` (con Morgan)

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

const app = express();

// Middleware per logging delle richieste HTTP
app.use(morgan('dev')); // formato "dev" mostra metodo, rotta, tempo, stato

app.get('/', (req, res) => {
  res.send('Benvenuto nel server!');
});

app.listen(3000, () => console.log(' Server su http://localhost:3000'));
```

Esempio output console:

```
GET / 200 5.172 ms - 25
```

`logger.js` (con Winston)

```
const { createLogger, transports, format } = require('winston');

const logger = createLogger({
  level: 'info',
  format: format.combine(
    format.timestamp(),
    format.printf(info => `${info.timestamp} [${info.level}] ${info.message}`)
  ),
  transports: [
    new transports.Console(),
    new transports.File({ filename: 'server.log' }) // salva i log
  ]
});

module.exports = logger;
```

Uso:

```
const logger = require('./logger');
logger.info('Server avviato correttamente');
logger.error('Errore durante la connessione al DB');
```

4. Introduzione al Deployment

Obiettivo

Mostrare come **pubblicare online** il backend funzionante.
Così vedrete la vostra API “viva” e testabile da browser o Postman.

Opzioni gratuite

- Render
- Railway
- Vercel (supporta anche Node.js Express)

Passaggi generali

1 Inizializza un repository Git

```
git init
git add .
git commit -m "Prima versione backend"
```

2 Crea un file `.env`

```
PORT=3000
MONGO_URI=your_mongodb_atlas_url
JWT_SECRET=your_secret_key
```

3 Modifica `server.js` per leggere variabili d’ambiente

```
require('dotenv').config();
const express = require('express');
const app = express();

app.get('/', (req, res) => {
  res.send('API online ');
});

const PORT = process.env.PORT || 3000;
app.listen(PORT, () => console.log(`Server su porta ${PORT}`));
```

4 Connetti a MongoDB Atlas

1. Vai su https://cloud.mongodb.com
2. Crea un cluster gratuito
3. Copia la stringa di connessione (es. `mongodb+srv://...`)
4. Impostala in `.env`

5 Pubblica su Render o Railway

- Collega il tuo repository GitHub
- Imposta le variabili `.env` nel pannello del servizio
- Deploy automatico in pochi minuti

Esempio output:

```
Your service is live: https://backend-student.onrender.com
```

Esercizio finale

Creare:

1. una semplice API `/api/users` con Joi e Service Layer;
2. il logger Morgan attivo;
3. deploy su Render;
4. test dal browser o Postman con l'URL pubblico.