Stack: **Node.js + Express + MongoDB (Mongoose) + JWT + Joi + Multer**
Stile modulo: **CommonJS (require/module.exports)**, facile da avviare ovunque.

TAPPA 0 — Inizializzazione progetto

0.1 Crea progetto e installa dipendenze

```
mkdir taskmanager-api && cd taskmanager-api
npm init -y

# core
npm install express mongoose cors dotenv

# auth & sicurezza
npm install bcryptjs jsonwebtoken helmet express-rate-limit

# validazione & upload
npm install joi multer

# logging
npm install morgan

# dev helper
npm install -D nodemon
```

0.2 Script, .gitignore e struttura

package.json (aggiungi script)

```
{
  "name": "taskmanager-api",
  "version": "1.0.0",
  "main": "src/server.js",
  "type": "commonjs",
  "scripts": {
    "dev": "nodemon src/server.js",
    "start": "node src/server.js"
  }
}
```

.gitignore

```
node_modules/
.env
uploads/
```

Struttura cartelle

```
src/
  server.js
  app.js
  config/
    env.js
    db.js
  models/
    user.model.js
    task.model.js
  middleware/
    auth.js
    validate.js
    errorHandler.js
  services/
    auth.service.js
    task.service.js
  controllers/
    auth.controller.js
    task.controller.js
  routes/
    auth.routes.js
    task.routes.js
  utils/
    logger.js
  uploads/          (creata automaticamente da multer)
```

.env (locale; non commit)

```
PORT=3000
MONGO_URI=mongodb+srv://<USER>:<PASS>@<CLUSTER>.mongodb.net/taskmanager
JWT_SECRET=metti-una-chiave-lunga-e-random
NODE_ENV=development
```

TAPPA 1 — Config e avvio server

1.1 Carica env e valida variabili critiche

src/config/env.js

```
// Legge variabili d'ambiente da .env in locale
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: senza questi non avviamo
['mongoUri', 'jwtSecret'].forEach((k) => {
  if (!env[k]) {
    console.error(` Variabile mancante: ${k.toUpperCase()}`);
    process.exit(1);
```

```
  }
});

module.exports = env;
```

1.2 Connessione a MongoDB (con log chiaro)

src/config/db.js

```
const mongoose = require('mongoose');
const env = require('./env');

async function connectDB() {
  try {
    await mongoose.connect(env.mongoUri);
    console.log(' MongoDB connesso');
  } catch (err) {
    console.error(' Errore connessione MongoDB:', err.message);
    process.exit(1);
  }
}

module.exports = connectDB;
```

1.3 App Express (middleware globali, rotte base)

src/app.js

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

const { notFound, errorHandler } = require('./middleware/errorHandler');
const authRoutes = require('./routes/auth.routes');
const taskRoutes = require('./routes/task.routes');

const app = express();

// Sicurezza base
app.use(helmet());

// CORS (in prod: metti origin: whitelist)
app.use(cors());

// Log HTTP in dev
app.use(morgan(process.env.NODE_ENV === 'production' ? 'combined' : 'dev'));

// Body parser JSON
app.use(express.json());

// Rate limit su /api/auth (evita abusi al login)
app.use('/api/auth', rateLimit({ windowMs: 15*60*1000, max: 100 }));

// Rotte principali
app.use('/api/auth', authRoutes);
app.use('/api/tasks', taskRoutes);

// Health check
app.get('/api/health', (req, res) => res.json({ status: 'ok' }));
```

```
// 404 e errori centralizzati
app.use(notFound);
app.use(errorHandler);

module.exports = app;
```

1.4 Entrypoint server

src/server.js

```
const env = require('./config/env');
const connectDB = require('./config/db');
const app = require('./app');

// Connette DB poi avvia
connectDB().then(() => {
  app.listen(env.port, () => {
    console.log(`🚀 Server avviato su http://localhost:${env.port}`);
  });
});
```

TAPPA 2 — Modelli Mongoose (User & Task)

2.1 User (email unica + hash password)

src/models/user.model.js

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

const userSchema = new mongoose.Schema({
  name:     { type: String, trim: true },
  email:    { type: String, required: true, unique: true, lowercase: true, trim:
true },
  passwordHash: { type: String, required: true },
  role:     { type: String, enum: ['user', 'admin'], default: 'user' }
}, { timestamps: true });

userSchema.set('toJSON', {
  transform: (doc, ret) => {
    delete ret.passwordHash; // non esporre l'hash
    return ret;
  }
});

module.exports = mongoose.model('User', userSchema);
```

2.2 Task (owner → relazione 1–N con User)

src/models/task.model.js

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

const taskSchema = new mongoose.Schema({
  title:       { type: String, required: true, trim: true, minlength: 1 },
  description: { type: String, trim: true },
  completed:   { type: Boolean, default: false },
```

```
  // Relazione: questo task appartiene a un utente
  owner:       { type: mongoose.Schema.Types.ObjectId, ref: 'User', required:
true },
  // (opzionale) file allegato
  filePath:    { type: String },    // es: /uploads/abc123.jpg
  fileName:    { type: String },
  fileType:    { type: String },
  fileSize:    { type: Number }
}, { timestamps: true });

module.exports = mongoose.model('Task', taskSchema);
```

TAPPA 3 — Middleware di autenticazione (JWT)

src/middleware/auth.js

```
const jwt = require('jsonwebtoken');
const env = require('../config/env');

// Verifica Bearer Token e inserisce req.user
module.exports = function auth(req, res, next) {
  const h = req.headers.authorization;
  if (!h || !h.startsWith('Bearer ')) {
    return res.status(401).json({ error: 'UNAUTHORIZED', message: 'Token
mancante o invalido' });
  }
  const token = h.split(' ')[1];

  try {
    const decoded = jwt.verify(token, env.jwtSecret);
    req.user = { userId: decoded.userId, role: decoded.role || 'user' };
    return next();
  } catch (err) {
    return res.status(401).json({ error: 'UNAUTHORIZED', message: 'Token non
valido o scaduto' });
  }
};
```

TAPPA 4 — Validazione con Joi (middleware generico)

src/middleware/validate.js

```
// Middleware generico: validate(schema) → controlla req.body con Joi
module.exports.validate = (schema) => (req, res, next) => {
  const { error, value } = schema.validate(req.body, { abortEarly: false,
stripUnknown: true });
  if (error) {
    return res.status(400).json({
      error: 'INVALID_INPUT',
      details: error.details.map(d => d.message)
    });
  }
  req.body = value; // usa i dati "puliti"
  next();
```

```
};
```

Schemi (Auth/Task)

```
// src/middleware/schemas.js
const Joi = require('joi');

const registerSchema = Joi.object({
  name: Joi.string().min(2).max(60).optional(),
  email: Joi.string().email().required(),
  password: Joi.string().min(6).required()
});

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

const createTaskSchema = Joi.object({
  title: Joi.string().min(1).required(),
  description: Joi.string().allow('').optional(),
  completed: Joi.boolean().optional()
});

const updateTaskSchema = Joi.object({
  title: Joi.string().min(1).optional(),
  description: Joi.string().allow('').optional(),
  completed: Joi.boolean().optional()
}).min(1); // almeno un campo

module.exports = { registerSchema, loginSchema, createTaskSchema,
updateTaskSchema };
```

TAPPA 5 — Gestione errori centralizzata

src/middleware/errorHandler.js

```
class AppError extends Error {
  constructor(status, code, message) {
    super(message);
    this.status = status;
    this.code = code;
  }
}

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

const errorHandler = (err, req, res, next) => {
  console.error('💥 Errore:', err);

  // Errori Joi (validazione)
  if (err.isJoi) {
    return res.status(400).json({ error: 'INVALID_INPUT', message: 'Dati non
validi' });
  }

  // Errori applicativi custom
  if (err instanceof AppError) {
```

```
    return res.status(err.status).json({ error: err.code, message:
err.message });
  }

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

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

TAPPA 6 — Service Layer (logica di business)

6.1 Auth Service

src/services/auth.service.js

```
const bcrypt = require('bcryptjs');
const jwt = require('jsonwebtoken');
const env = require('../config/env');
const User = require('../models/user.model');

async function register({ name, email, password }) {
  const exists = await User.findOne({ email });
  if (exists) throw new Error('EMAIL_TAKEN');

  const passwordHash = await bcrypt.hash(password, 10);
  const user = await User.create({ name, email, passwordHash });
  return user; // toJSON rimuove passwordHash
}

async function login({ email, password }) {
  const user = await User.findOne({ email });
  if (!user) throw new Error('BAD_CREDENTIALS');

  const ok = await bcrypt.compare(password, user.passwordHash);
  if (!ok) throw new Error('BAD_CREDENTIALS');

  const token = jwt.sign(
    { userId: user._id.toString(), role: user.role },
    env.jwtSecret,
    { expiresIn: '1h' }
  );

  return { token, user: { id: user._id, email: user.email, role: user.role,
name: user.name } };
}

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

6.2 Task Service

src/services/task.service.js

```
const Task = require('../models/task.model');
const { AppError } = require('../middleware/errorHandler');

async function listByOwner(ownerId) {
  return Task.find({ owner: ownerId }).sort({ createdAt: -1 });
```

```
}

async function create(ownerId, data) {
  return Task.create({ ...data, owner: ownerId });
}

async function getOne(ownerId, id) {
  const task = await Task.findOne({ _id: id, owner: ownerId });
  if (!task) throw new AppError(404, 'NOT_FOUND', 'Task non trovato');
  return task;
}

async function update(ownerId, id, updates) {
  const task = await Task.findOneAndUpdate(
    { _id: id, owner: ownerId },
    updates,
    { new: true, runValidators: true }
  );
  if (!task) throw new AppError(404, 'NOT_FOUND', 'Task non trovato');
  return task;
}

async function remove(ownerId, id) {
  const task = await Task.findOneAndDelete({ _id: id, owner: ownerId });
  if (!task) throw new AppError(404, 'NOT_FOUND', 'Task non trovato');
  return true;
}

module.exports = { listByOwner, create, getOne, update, remove };
```

TAPPA 7 — Controller (snelli: orchestrano req/res)

7.1 Auth Controller

src/controllers/auth.controller.js

```
const { register, login } = require('../services/auth.service');

exports.register = async (req, res, next) => {
  try {
    const user = await register(req.body);
    res.status(201).json(user);
  } catch (err) {
    if (err.message === 'EMAIL_TAKEN') {
      return res.status(409).json({ error: 'CONFLICT', message: 'Email già
registrata' });
    }
    next(err);
  }
};

exports.login = async (req, res, next) => {
  try {
    const data = await login(req.body);
    res.json(data);
  } catch (err) {
    if (err.message === 'BAD_CREDENTIALS') {
```

```
      return res.status(401).json({ error: 'UNAUTHORIZED', message: 'Credenziali
non valide' });
    }
    next(err);
  }
};
```

7.2 Task Controller (+ upload file)

src/controllers/task.controller.js

```
const { listByOwner, create, getOne, update, remove } =
require('../services/task.service');

exports.list = async (req, res, next) => {
  try {
    const tasks = await listByOwner(req.user.userId);
    res.json(tasks);
  } catch (err) { next(err); }
};

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

exports.detail = async (req, res, next) => {
  try {
    const task = await getOne(req.user.userId, req.params.id);
    res.json(task);
  } catch (err) { next(err); }
};

exports.patch = async (req, res, next) => {
  try {
    const task = await update(req.user.userId, req.params.id, req.body);
    res.json(task);
  } catch (err) { next(err); }
};

exports.remove = async (req, res, next) => {
  try {
    await remove(req.user.userId, req.params.id);
    res.status(204).end();
  } catch (err) { next(err); }
};

exports.attach = async (req, res, next) => {
  try {
    // req.file valorizzato da multer
    if (!req.file) {
      return res.status(400).json({ error: 'INVALID_INPUT', message: 'Nessun
file caricato' });
    }
    // Aggiorna task con metadati del file
    const updates = {
      filePath: `/uploads/${req.file.filename}`,
      fileName: req.file.originalname,
      fileType: req.file.mimetype,
      fileSize: req.file.size
```

```
    };
    const task = await update(req.user.userId, req.params.id, updates);
    res.status(201).json(task);
  } catch (err) { next(err); }
};
```

TAPPA 8 — Rotte (Auth, Task) + Multer

8.1 Config upload (multer) per immagini

Nota: esponiamo staticamente `/uploads` **solo in laboratorio**.

Aggiungi in **src/app.js**:

```
const path = require('path');
app.use('/uploads', express.static(path.join(__dirname, 'uploads')));
```

Config Multer inline nella route:

```
// src/routes/task.routes.js (snippet in alto del file)
const multer = require('multer');
const upload = multer({
  dest: 'src/uploads/',
  limits: { fileSize: 2 * 1024 * 1024 }, // 2MB
  fileFilter: (req, file, cb) => {
    if (file.mimetype.startsWith('image/')) cb(null, true);
    else cb(new Error('Tipo file non supportato (solo immagini)'));
  }
});
```

8.2 Rotte Auth

src/routes/auth.routes.js

```
const router = require('express').Router();
const { validate } = require('../middleware/validate');
const { registerSchema, loginSchema } = require('../middleware/schemas');
const authCtrl = require('../controllers/auth.controller');

router.post('/register', validate(registerSchema), authCtrl.register);
router.post('/login',    validate(loginSchema),    authCtrl.login);

module.exports = router;
```

8.3 Rotte Task (protette)

src/routes/task.routes.js

```
const router = require('express').Router();
const auth = require('../middleware/auth');
const { validate } = require('../middleware/validate');
const { createTaskSchema, updateTaskSchema } = require('../middleware/schemas');
const taskCtrl = require('../controllers/task.controller');

const multer = require('multer');
const upload = multer({
  dest: 'src/uploads/',
```

```
  limits: { fileSize: 2 * 1024 * 1024 },
  fileFilter: (req, file, cb) => file.mimetype.startsWith('image/') ? cb(null,
true) : cb(new Error('Solo immagini'))
});

// Tutte protette
router.use(auth);

router.get('/',           taskCtrl.list);
router.post('/',          validate(createTaskSchema), taskCtrl.create);
router.get('/:id',        taskCtrl.detail);
router.patch('/:id',      validate(updateTaskSchema), taskCtrl.patch);
router.delete('/:id',     taskCtrl.remove);

// Upload file su un task
router.post('/:id/upload', upload.single('file'), taskCtrl.attach);

module.exports = router;
```

TAPPA 9 — Prova in locale

9.1 Avvia

```
npm run dev
# Console:
#  MongoDB connesso
#  Server avviato su http://localhost:3000
```

9.2 Test rapidi (curl / Postman)

Health

```
curl http://localhost:3000/api/health
# {"status":"ok"}
```

Register → Login

```
curl -X POST http://localhost:3000/api/auth/register \
 -H "Content-Type: application/json" \
 -d '{"name":"Leandro","email":"leo@test.it","password":"Segreta1"}'

curl -X POST http://localhost:3000/api/auth/login \
 -H "Content-Type: application/json" \
 -d '{"email":"leo@test.it","password":"Segreta1"}'
# → prendi "token"
```

Create Task

```
curl -X POST http://localhost:3000/api/tasks \
 -H "Authorization: Bearer <TOKEN>" \
 -H "Content-Type: application/json" \
 -d '{"title":"Prima task","description":"Esempio"}'
```

List

```
curl -H "Authorization: Bearer <TOKEN>" http://localhost:3000/api/tasks
```

Upload immagine

```
curl -X POST http://localhost:3000/api/tasks/<ID>/upload \
 -H "Authorization: Bearer <TOKEN>" \
 -F "file=@/percorso/immagine.jpg"
```

TAPPA 10 — Pronti al deploy (checklist rapida)

- `npm start` funziona ✔
- `process.env.PORT` usata ✔
- `.env` **non** nel repo ✔
- `MONGO_URI` Atlas testata ✔
- `helmet`, `rate-limit`, `CORS` attivi ✔
- Health `/api/health` risponde ✔

Deploy consigliato: Render o Railway

- Imposta env nel pannello (`MONGO_URI`, `JWT_SECRET`, `NODE_ENV=production`)
- Non impostare `PORT` manualmente (lo fornisce il provider)
- Testa le rotte con Postman sull'URL pubblico

Suggerimenti didattici

- **Testate ogni step**: dopo ogni file creato, provate una rotta.
- **Commit frequenti**: "feat: aggiunge create task" → storia pulita.
- **Log e messaggi d'errore chiari** aiutano a capire cosa succede.
- **Sicurezza base** sempre attiva (helmet, rate limit, CORS mirato in prod).

VARIANTE 1 — Dashboard Front-End (HTML o React)

Obiettivo

Creare una piccola **interfaccia utente** che si connette al backend “TaskManager” già deployato online.

Step 1 — Struttura base (HTML + JS)

Crea un file `index.html` con un form di login e un’area per i task.

```
<body>
  <h1>TaskManager Frontend</h1>

  <!-- Form login -->
  <form id="loginForm">
    <input type="email" id="email" placeholder="Email" required />
    <input type="password" id="password" placeholder="Password" required />
    <button type="submit">Login</button>
  </form>

  <!-- Area per i task -->
  <div id="tasks" style="margin-top: 20px;"></div>

  <script src="main.js"></script>
</body>
```

Step 2 — Login con fetch()

Nel file `main.js` scrivi:

```
const API_URL = 'https://tuo-backend.onrender.com/api';
let token = null;

document.getElementById('loginForm').addEventListener('submit', async (e) => {
  e.preventDefault();

  const email = document.getElementById('email').value;
  const password = document.getElementById('password').value;

  const res = await fetch(`${API_URL}/auth/login`, {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ email, password }),
  });

  const data = await res.json();
  if (res.ok) {
    token = data.token;
    alert('Login effettuato!');
    caricaTask();
  } else {
    alert('Errore: ' + data.message);
```

```
  }
});
```

Step 3 — Mostrare i task

```
async function caricaTask() {
  const res = await fetch(`${API_URL}/tasks`, {
    headers: { Authorization: `Bearer ${token}` },
  });

  const tasks = await res.json();
  document.getElementById('tasks').innerHTML = tasks
    .map((t) => `<p>${t.title} — ${t.completed ? '✅' : '❌'}</p>`)
    .join('');
}
```

Variante avanzata (React)

Se vuoi farli provare con React:

- Crea un progetto con `npm create vite@latest frontend -- --template react`
- Usa `useState`, `useEffect` e `fetch` per gestire token e task
- Aggiungi `localStorage` per salvare il JWT

VARIANTE 2 — Documentazione API con Swagger

Obiettivo

Mostrare agli studenti **come si documenta un'API** in modo professionale e leggibile.

Step 1 — Installa librerie

```
npm install swagger-ui-express yamljs
```

Step 2 — Crea file `docs/openapi.yaml`

```
openapi: 3.0.0
info:
  title: TaskManager API
  version: 1.0.0
paths:
  /api/auth/login:
    post:
      summary: Effettua il login
      requestBody:
        required: true
        content:
```

```
          application/json:
            schema:
              type: object
              properties:
                email:
                  type: string
                password:
                  type: string
      responses:
        '200':
          description: Token JWT valido
```

Step 3 — Monta Swagger in `app.js`

```
const swaggerUi = require('swagger-ui-express');
const yaml = require('yamljs');
const path = require('path');

const swaggerDoc = yaml.load(path.join(__dirname, '../docs/openapi.yaml'));
app.use('/api/docs', swaggerUi.serve, swaggerUi.setup(swaggerDoc));
```

Ora apri `http://localhost:3000/api/docs`

VARIANTE 3 — Dashboard Admin (ruoli e permessi)

Obiettivo

Far capire come funziona l'autorizzazione **basata sui ruoli (admin/user)**.

Step 1 — Aggiorna modello User

```
role: { type: String, enum: ['user', 'admin'], default: 'user' }
```

Step 2 — Crea middleware `isAdmin`

```
module.exports = function isAdmin(req, res, next) {
  if (req.user.role !== 'admin') {
    return res.status(403).json({ error: 'FORBIDDEN', message: 'Accesso
negato' });
  }
  next();
};
```

Step 3 — Rotte Admin protette

```
const router = require('express').Router();
const auth = require('../middleware/auth');
const isAdmin = require('../middleware/isAdmin');
const User = require('../models/user.model');
const Task = require('../models/task.model');

router.use(auth, isAdmin);

router.get('/users', async (req, res) => res.json(await User.find()));
router.get('/tasks', async (req, res) => res.json(await Task.find()));
```

```
router.delete('/users/:id', async (req, res) => {
  await User.findByIdAndDelete(req.params.id);
  res.status(204).end();
});

module.exports = router;
```

Monta in `app.js`:

```
const adminRoutes = require('./routes/admin.routes');
app.use('/api/admin', adminRoutes);
```

VARIANTE 4 — Refresh Token e Reset Password

Obiettivo

Introdurre la gestione di sessioni e recupero password in modo realistico.

Step 1 — Refresh Token

Nel login genera **due token**:

```
const accessToken = jwt.sign({ userId: user._id }, JWT_SECRET, { expiresIn:
'15m' });
const refreshToken = jwt.sign({ userId: user._id }, JWT_SECRET, { expiresIn:
'7d' });
```

Crea una nuova rotta:

```
router.post('/refresh', async (req, res) => {
  const { refreshToken } = req.body;
  try {
    const decoded = jwt.verify(refreshToken, JWT_SECRET);
    const accessToken = jwt.sign({ userId: decoded.userId }, JWT_SECRET,
{ expiresIn: '15m' });
    res.json({ accessToken });
  } catch {
    res.status(401).json({ error: 'Token non valido o scaduto' });
  }
});
```

Step 2 — Password Reset (versione semplificata)

1. Crea campo `resetToken` in `User`
2. Crea rotta:

```
router.post('/request-reset', async (req, res) => {
  const token = Math.random().toString(36).substring(2, 10);
  await User.findOneAndUpdate({ email: req.body.email }, { resetToken: token });
  console.log(`Link di reset: /api/auth/reset-password?token=${token}`);
  res.json({ message: 'Email inviata (simulata)' });
});
```

3. Rotta per nuovo password:

```
router.post('/reset-password', async (req, res) => {
  const user = await User.findOne({ resetToken: req.body.token });
  if (!user) return res.status(400).json({ error: 'Token non valido' });
  user.passwordHash = await bcrypt.hash(req.body.password, 10);
  user.resetToken = null;
  await user.save();
  res.json({ message: 'Password aggiornata' });
});
```

VARIANTE “PRO PLUS” — Ecosistema completo

Obiettivo

Unire tutto per avere un **progetto portfolio reale full-stack**.

Struttura finale

```
backend/
  api/ (Express)
frontend/
  app/ (React)
docs/
  openapi.yaml
```

Risultato finale

- API Express con autenticazione e upload
- Dashboard utente in React o HTML
- Dashboard admin riservata
- Documentazione Swagger online
- Database MongoDB Atlas
- Tutto deployato su Render o Railway

Suggerimenti

- Partite da **una sola estensione** per volta.
- Testate ogni nuova rotta con **Postman**.
- Scrivete commit chiari:

  ```
  feat(admin): aggiunta rotta GET /users
  fix(auth): corretto bug nel refresh token
  ```

- Aggiornate il `README.md` spiegando cosa avete aggiunto.
- Collaborate in coppia: uno lavora al backend, l’altro al frontend.