Backend Engineering Task

Secure File Upload & Metadata Processing Microservice (Node.js)

📦 Overview

In this task, you will build a Node.js backend microservice that handles authenticated file uploads, stores associated metadata in a database, and processes those files asynchronously. This challenge simulates a core production backend component — with security, async jobs, and structured APIs.

You are not expected to build a frontend or UI. Focus on backend engineering best practices, security, and code clarity.

🎯 Objective

> Build a secure file upload service in Node.js that stores file metadata in a database, runs background processing tasks, and tracks the status of those tasks.

🧰 Required Tech Stack

Please use the following technologies:

- Node.js (>=18)
- Framework: Express.js or NestJS (NestJS recommended for structure)
- Database: PostgreSQL or SQLite (with Sequelize, Prisma, or TypeORM)
- Authentication: JWT (using libraries like `jsonwebtoken`)
- Background Jobs: BullMQ (Redis-based), or an alternative local queue
- File Handling: `multer` or `formidable`
- Environment: Local (Docker optional)

🧩 Functional Requirements

1. Authentication (JWT)

- Authenticate users using static credentials or a simple user table.
- Issue a JWT token on login (`POST /auth/login`)
- Require the token on all API requests except login and health check.

---

2. File Upload API

- `POST /upload`
  - Accepts:
    - A file (any type)
    - Optional metadata (title, description)
  - Requirements:
    - Save file to local disk or a `./uploads` folder
    - Save metadata and file path to DB
    - Add a job to background queue for processing
  - Return:
    - File ID
    - Upload status: `uploaded`

---

3. File Processing (Async Job)

- Run a background job that:
  - Reads the uploaded file
  - Simulates processing (e.g., `setTimeout`, checksum calculation, or text extraction)
  - Updates status in DB:
    - `processing → processed` or `failed`
  - Save any "extracted data" (e.g., file hash or mock result)

Use BullMQ with Redis for job queueing.

---

4. File Status API

- `GET /files/:id`
  - Auth required
  - Only return file info to the user who uploaded it
  - Include:
    - Metadata
    - Current status
    - Extracted data (if available)

🔒 Security Requirements

- Auth token required on all endpoints (except login/health)
- Only authenticated users can upload files
- Only the user who uploaded a file can access its status

- Upload size should be limited to prevent abuse

🔧 Optional Enhancements (Bonus)

- Pagination: `GET /files?page=1`
- Retry failed jobs (auto or manual)
- Upload rate limiting per user
- Dockerfile + docker-compose setup (for Node, DB, Redis)
- Swagger/OpenAPI documentation

📦 Deliverables

Please submit your work as a GitHub repo or ZIP file with the following:

Required:

- Complete Node.js project
- `README.md` with:
  - How to run locally
  - API documentation (include auth flow)
  - Your design choices
  - Known limitations or assumptions
- PostgreSQL/SQLite schema or migrations
- Code for background processor and job worker
- Example `.env` file or setup instructions

Optional:

- Postman collection or cURL scripts
- Docker setup
- Swagger/OpenAPI JSON

✅ Evaluation Criteria

Area	What We Look For
API Design	RESTful, clean, intuitive endpoints
Auth & Access Control	JWT usage, user-based access control
Async Processing	Effective background job logic with state tracking

Code Quality	Readable, modular, error-handled, structured code
DB Schema	Clear modeling of users, files, jobs, and status
Security Practices	Proper file handling, user isolation, token validation
Realism	Practical, production-like approach to a common backend scenario
Documentation	Usable setup and explanations of decisions

⏳ Time Estimate

4–8 hours depending on experience and enhancements.

You do not need to over-engineer this — clear and correct wins over complex and half-finished.

✨ Example API Flow

1. `POST /auth/login`
   → Receive JWT
2. `POST /upload` (with file + metadata)
   → Returns file ID + status `uploaded`
3. Background job picks up and processes file
   → Updates DB status to `processed`
4. `GET /files/:id`
   → Returns file info, status, and extracted result

🗃️ Basic Database Schema

1. users

Stores application users. For simplicity, passwords can be plaintext or hashed.

```
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  email VARCHAR(255) UNIQUE NOT NULL,
  password VARCHAR(255) NOT NULL,  -- Plaintext for demo or hashed if implementing auth securely
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

---

2. files

Stores uploaded file metadata and tracking information.

```
CREATE TABLE files (

  id SERIAL PRIMARY KEY,

  user_id INTEGER REFERENCES users(id) ON DELETE CASCADE,

  original_filename VARCHAR(255) NOT NULL,

  storage_path TEXT NOT NULL,

  title VARCHAR(255),

  description TEXT,

  status VARCHAR(50) CHECK (status IN ('uploaded', 'processing', 'processed',
'failed')) NOT NULL DEFAULT 'uploaded',

  extracted_data TEXT,

  uploaded_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP

);
```

3. jobs (Optional, if you want to explicitly track job metadata)

You can track job status separately or just update file records directly.

```
CREATE TABLE jobs (

  id SERIAL PRIMARY KEY,

  file_id INTEGER REFERENCES files(id) ON DELETE CASCADE,

  job_type VARCHAR(50),

  status VARCHAR(50) CHECK (status IN ('queued', 'processing', 'completed',
'failed')) NOT NULL,

  error_message TEXT,

  started_at TIMESTAMP,

  completed_at TIMESTAMP

);
```

🔑 User-to-File Relationship

- One user → many files
- A user can only see files they uploaded — this should be enforced in the API logic

---

🔄 Status Flow for Files

- `uploaded` → File is saved, job enqueued
- `processing` → Background job started
- `processed` → Job completed, result saved
- `failed` → Job encountered an error (message can go in `extracted_data` or error column)