Patient Document Portal — Design Document

Table of Contents

---

1. Project Summary

A simple full-stack Patient Document Portal where a single user can upload, view, download, and delete PDF medical documents. Files are stored on the backend filesystem (`uploads/`) and metadata is persisted in a SQLite database. The application runs locally for evaluation and includes a small cleanup job to remove stale files.

---

2. Tech Stack Choices (Q1–Q4)

Q1. Frontend framework: React - Why: Fast to build interactive UIs, large ecosystem, easy to scaffold with `create-react-app` and to integrate routing, forms, and fetch/XHR.

Q2. Backend framework: Node.js + Express - Why: Lightweight, minimal boilerplate for REST APIs, pairs nicely with JavaScript frontend, large middleware ecosystem (Multer for uploads), fast to iterate.

Q3. Database: SQLite (sqlite3) - Why: Simple file-based DB requiring no separate server — perfect for local exercises and single-user demo. Easy to inspect and include in repository.

Q4. If scaling to 1,000 users, consider: - Replace SQLite with PostgreSQL or managed DB (RDS). - Move file storage to object storage (AWS S3, GCS) for horizontal scaling. - Add authentication and per-user isolation (JWT + user table). - Use load-balanced backend instances behind a reverse proxy (NGINX) or managed platform. - Add background workers for expensive tasks and use caching (Redis) for hot metadata.

---

3. Architecture Overview

```
[Browser (React UI)] <--HTTPS/HTTP--> [Express API Server]
                                        |-- stores files --> uploads/ (local) OR
S3 (prod)
                                        |-- stores metadata --> SQLite (local) OR
Postgres (prod)
```

Components: - **Frontend (React):** Upload page, Documents page. Uses `fetch` / `XMLHttpRequest` to call backend. - **Backend (Express):** Endpoints for upload, list, download, delete. Uses Multer for parsing multipart `file` field and writes to `uploads/`. - **DB (SQLite):** `documents` table holding metadata: id, filename, filepath, filesize, created_at, last_accessed. - **Cleanup job:** A scheduled function in the backend that deletes files older than a TTL (configurable via env var `TTL_DAYS`).

---

4. API Specification

Base URL: `http://localhost:5000`

1) `POST /documents/upload`

- **Description:** Upload a PDF file (multipart/form-data, field name `file`).
- **Request:** `multipart/form-data` with `file` (PDF)
- **Response (201)**

```
{ "id": 12, "filename": "prescription.pdf", "filesize": 42102,
"created_at": "2025-12-09T12:34:56.000Z", "message": "Uploaded
successfully" }
```

- **Errors:** 400 (no file), 415 (invalid file type), 413 (file too large), 500 (server error)

2) `GET /documents`

- **Description:** List all uploaded documents metadata for the app (single-user assumption).
- **Response (200)**

```
[
  { "id": 12, "filename": "prescription.pdf", "filepath": "backend/
uploads/...", "filesize": 42102, "created_at": "2025-12-09T12:34:56.000Z",
"last_accessed": "2025-12-10T10:00:00.000Z" },
  ...
]
```

3) `GET /documents/:id`

- **Description:** Download stored file for the given id.
- **Behavior:** Backend updates `last_accessed` to the current timestamp and then sends the file using `res.download()` so browser saves it with `filename`.
- **Response:** Binary (Content-Disposition: attachment). 404 if not found, 410 if file missing.

4) `DELETE /documents/:id`

- **Description:** Delete file and its metadata.
- **Response (200)**

```
{ "message": "Deleted successfully", "id": 12 }
```

- **Errors:** 404 if not found, 500 for DB/file deletion issues.

---

5. Data Flow

Upload flow

1. User selects PDF in React Upload form.
2. Client validates file type and size, then sends POST `/documents/upload` as multipart form-data.
3. Express + Multer writes file to `uploads/` with a unique filename.
4. Server writes a row in `documents` table with `created_at` and `filepath`.
5. Server responds with metadata and `id`.
6. Client refreshes document list.

Download flow

1. User clicks Download link (anchor to `GET /documents/:id`).
2. Server sets `last_accessed = now` for the row, then serves file via `res.download()`.
3. Client browser triggers download with original filename.

Cleanup flow

1. Periodic job runs (configurable interval) and computes cutoff `now - TTL_DAYS`.
2. Server queries rows where `(last_accessed IS NOT NULL AND last_accessed < cutoff) OR (last_accessed IS NULL AND created_at < cutoff)`.
3. For each row: delete file from disk (if exists) and delete DB row.

---

6. Database Schema

Table: documents

Column	Type	Notes
id	INTEGER	PRIMARY KEY AUTOINCREMENT
filename	TEXT	Original filename
filepath	TEXT	Path on server (uploads/...)
filesize	INTEGER	bytes
created_at	TEXT	ISO timestamp when uploaded
last_accessed	TEXT	ISO timestamp when downloaded/viewed (nullable)

---

7. Assumptions

- Single-user application (no authentication) — matches assignment instructions.
- PDF-only uploads — validated by MIME type and filename extension.
- Max file size set to 20 MB (adjustable in Multer limits).
- Files stored locally (`uploads/`) so reviewers can inspect files.
- Cleanup is hard-delete by default. For recoverability, a soft-delete (move to `uploads/trash/`) can be implemented instead.
- Timezone: timestamps stored in ISO 8601 (UTC) to avoid ambiguity.

---

8. Operational & Deployment Notes

- Environment variables:
- `PORT` — backend port (default 5000)
- `TTL_DAYS` — days after which un-accessed files are removed (default 30)
- `CLEANUP_INTERVAL_HOURS` — how often cleanup runs (default 6)
- Development: run backend (`npm start`) and frontend (`npm start`) locally.
- Database file `db.sqlite` and `uploads/` should be in `.gitignore` (or included for demo if instructed).
- For production, use a persistent DB (Postgres) and S3 for files.

---

9. How to support 1,000 users (scaling considerations)

- **Storage & DB:** Use Postgres + S3. Store only object keys in DB.
- **Auth & Multi-tenancy:** Add users table, associate documents with user_id, implement JWT-based auth.
- **File serving:** Use pre-signed S3 URLs for downloads to reduce backend bandwidth.
- **Background jobs:** Use a worker queue (e.g., Bull with Redis) for cleanup and heavy tasks.
- **Horizontal scaling:** Deploy backend behind a load balancer, use sticky sessions or centralized session store.

- **Monitoring & backups:** Use logging (ELK/Datadog), monitor storage costs, backup DB.

---

10. Security & Privacy Considerations

- Serve over HTTPS in production.
- Limit accepted file types and set strict Multer filters.
- Scan uploads for malware (optional): use a virus-scanning service or ClamAV.
- Do not expose full file paths in public responses — prefer storing and returning safe object keys.
- Access control: authenticate users and ensure documents are only visible to their owners.
- Data retention policy: document TTL and deletion policies; if required, implement soft-delete and retention windows.

---

11. Run & Test Instructions

Backend (development):

```
cd backend
npm install
# start server
npm start
```

Frontend (development):

```
cd frontend
npm install
npm start
```

Quick manual tests: - Upload an example PDF via UI or `curl -X POST -F "file=@example_document.pdf" http://localhost:5000/documents/upload` - List: `curl http://localhost:5000/documents` - Download: open `http://localhost:5000/documents/<id>` in browser - Delete: `curl -X DELETE http://localhost:5000/documents/<id>`

Appendix: Example curl commands

Upload:

```
curl -X POST -F "file=@/path/to/example_document.pdf" http://localhost:5000/
documents/upload
```

List:

```
curl http://localhost:5000/documents
```

Download:

```
# open in browser or use curl to save as file
curl -OJ http://localhost:5000/documents/12
```

Delete:

```
curl -X DELETE http://localhost:5000/documents/12
```

End of design.md