Document Manager

Architecture & Operations

Document Manager is a full-stack system for saving, tagging, summarising, and searching documents. It combines a Java 21 / Spring Boot 3 backend, a React + Vite + Tailwind frontend, and a set of services (PostgreSQL, RabbitMQ, MinIO object storage and Elasticsearch). Uploads are stored in MinIO, described in PostgreSQL, pushed through RabbitMQ for OCR and AI summarization and also indexed for search.

Technology Stack

- Backend: Spring Boot (DocumentManagerApplication), JPA/Hibernate, Flyway migrations, Validation, Spring AMQP, MinIO SDK, Elasticsearch Java client.
- Storage/Infra: PostgreSQL (documents, tags, document_tags), MinIO for binary/object storage, RabbitMQ for events and worker queues, optional Elasticsearch for text search.
- Workers: ocr-worker-py (Python, Tesseract via pdf2image), genai-worker (Python, Google Gemini for summaries). Dockerfiles are provided for both.
- Frontend: React 19, Vite, Tailwind 4, React Toastify. In production, Nginx serves the built bundle and proxies /api to the backend.

Data Model

- Document (backend/src/main/java/at/technikum/documentmanager/entity/Document.java): id (UUID), originalFilename, contentType, size, uploadedAt, storageFilename, optional summary, tags (many-to-many).
- Tag (backend/src/main/java/at/technikum/documentmanager/entity/Tag.java): id, unique name, optional color.

Backend API Surface

- Upload: POST /api/documents/upload accepts MultipartFile. DocumentServiceImpl.saveFile streams the file to MinIO, persists metadata, and publishes an upload event.
- Download: GET /api/documents/download/{id} streams from MinIO using stored storageFilename.
- Metadata update: PUT /api/documents/{id}/metadata to rename / retag MIME type.
- Replace file: PUT /api/documents/{id}/replace re-uploads the object, updates metadata.
- Delete: DELETE /api/documents/{id} removes the object from MinIO then deletes DB row.
- Summary fetch: GET /api/documents/{id}/summary returns stored AI summary if present.
- Listing: GET /api/documents returns DocumentResponse with tags.

- Tagging: POST /api/tags creates a tag; GET /api/tags lists tags; POST /api/documents/{id}/tags/{tagId} / DELETE ... attaches or removes tags.
- Search: GET /api/search?q=... calls SearchService (Elasticsearch multi-match with highlights over text, filename, summary).

Storage Strategy

- Files live in MinIO under documentmanager bucket. Naming convention is <uuid>-<sanitizedOriginalFilename> (DocumentServiceImpl.saveFile and ocr-worker-py/utils/minio_client.py share the same sanitiser).
- OCR text is stored back in MinIO as ocr/<uuid>.txt for downstream consumers or reprocessing.

Messaging and Asynchronous Pipeline

- Exchange/queues defined in MessagingConfig: topic exchange docs.exchange, upload queue docs.uploaded.q, DLX docs.dlx + DLQ docs.uploaded.dlq, plus durable queues genai-tasks and summary-results.
- Producer: UploadEventPublisher sends UploadEvent (docId, filename, type, size, uploadedAt, uploadedBy) to docs.exchange with routing key document.uploaded.
- OCR worker (ocr-worker-py/main.py):
    - Consumes docs.uploaded.q.
    - Downloads the file from MinIO, runs Tesseract OCR (perform_ocr), uploads extracted text to MinIO, and publishes two messages:
        - To genai-tasks with documentId and text for AI summarisation.
        - To indexing-tasks (queue declared in worker) with metadata + text for indexing.
- GenAI worker (genai-worker/main.py):
    - Consumes genai-tasks.
    - Uses Google Gemini (genai.Client) to generate a summary.
    - Publishes the result to summary-results.
- Consumer: SummaryListener listens to summary-results, parses SummaryMessage, and persists the summary via DocumentService.saveSummary.
- Indexing path: The search API assumes documents are indexed in Elasticsearch with fields text, filename, summary, etc. The indexing-tasks queue is emitted by the OCR worker; an indexing worker would consume it and upsert into the documents index (not yet included in this repo—add one if search should reflect OCR/summary content).

Search Service

SearchService performs a multi-match query across text, filename, and summary, returning up to 20 hits with highlight snippets. It trims or strips tags for readability and tolerates missing IDs. Configure via ELASTICSEARCH_URL and ELASTICSEARCH_INDEX (defaults to http://localhost:9200 and documents). Deploy an Elasticsearch node alongside the stack to enable this endpoint.

Frontend Flow

The React SPA lists documents, supports file upload, shows metadata and summaries, manages tags, and issues search requests. It talks to the backend through the /api routes (proxied in dev via Vite, and in production via Nginx). Tailwind provides styling; Toastify surfaces API errors.

Running the Stack

- Development: docker compose up db app rabbitmq minio minio-init starts backend dependencies and the Spring Boot app. Run npm install && npm run dev inside frontend for hot-reload UI.
- Production: docker compose up --build -d builds frontend, backend, workers, and Nginx. Ports: 8080 (backend), 5432 (Postgres), 5672/15672 (RabbitMQ + management), 9000/9090 (MinIO + console), 80 (Nginx).
- Environment: defaults are set in application.yml and .env (e.g., POSTGRES_DB, MINIO_ROOT_USER, APP_MQ_EXCHANGE, ELA STICSEARCH_URL). Set GEMINI_API_KEY for the GenAI worker.

Operational Notes

- Reliability: Upload queue is durable with DLX; failures route to docs.uploaded.dlq. The OCR and GenAI workers ack messages after processing to avoid loss.
- Security: Credentials are sourced from env vars; MinIO and RabbitMQ default to in-repo credentials - override for real deployments.
- Scaling: Backend and workers are stateless; scale out behind RabbitMQ. MinIO/PostgreSQL/Elasticsearch remain single instances unless replaced with managed services.
- Observability: Logging is standard Spring + worker stdout; add structured logging/metrics for production. Health checks are defined for Postgres and MinIO in docker-compose.yml.