PRD: File Normalization Service

Product Requirements Document: File Normalization Service (FNS)

==========

1. Executive Summary

The FNS is a backend microservice designed to act as an ingestion middleware between a generic File Repository and a downstream AI/RAG system. Its primary responsibility is to accept various document formats (PDF, DOCX, etc.), asynchronously convert them into clean Markdown, and provide a reliable "Journaling API" for downstream consumers to fetch updates efficiently.

2. System Architecture High-Level

The system follows an Asynchronous Worker Pattern. The API accepts requests (webhooks) or scheduled triggers, which push tasks onto a queue. Workers process the heavy conversion logic, while a database maintains the state for observability and journaling.

[Diagram Placeholder: Asynchronous worker architecture]

3. Functional Requirements

3.1. Ingestion & Triggers

The service must detect new files via two methods:

* Reactive (Webhook): A REST endpoint (POST /ingest) that accepts a file path or ID.
* Proactive (Scheduled/Polling): A background job that scans the source repository to catch missed files.

3.2. File Conversion (The Core Logic)

* Input Formats: PDF, DOCX, PPTX, XLSX, CSV, HTML, TXT.
* Output Format: Standard CommonMark Markdown.
* Processing Time: Capable of handling long-running jobs (approx. 30s per file).

* Idempotency: Duplicate uploads should be detected via hash.

3.3. Reliability & Error Handling

* Retry Mechanism: Configurable MAX_RETRIES (default: 3) with backoff.

* Dead Letter Handling: After retries, status = FAILED. File moved to 'failed_conversions/' folder.

3.4. Observability (Admin API)

* States: QUEUED, PROCESSING, COMPLETED, FAILED.

* Metrics: Count of files in each state.

3.5. Journaling API (Downstream Sync)

* Delta Updates: Fetch "what changed since time T".

* Lifecycle Events: Track CREATED, UPDATED, and DELETED events.

4. API Specification (Draft)

4.1. Ingestion & Control

* POST /v1/ingest: Triggers a conversion job.

* POST /v1/trigger-scan: Manually triggers a full repo scan.

4.2. Monitoring & State

* GET /v1/admin/stats: Returns job counts.

* GET /v1/admin/jobs: List specific jobs with filters.

* GET /v1/admin/jobs/{id}: Get error logs.

4.3. Journaling

* GET /v1/journal/sync: Params: `since_timestamp`. Returns list of modified files.

5. Data Model

We require a relational database (PostgreSQL or SQLite).

[Diagram Placeholder: Entity Relationship Diagram]

Table: conversion_jobs

* id (UUID, PK)
* source_path (String, Unique)
* file_hash (String)
* status (Enum: QUEUED, PROCESSING, COMPLETED, FAILED)
* retry_count (Int)
* error_message (Text)
* created_at (Timestamp)
* updated_at (Timestamp)
* is_deleted (Bool)

6. Technical Logic Flow

6.1. The Processing Pipeline (Worker)

1. Pick Task: Worker picks up task.
2. State Update: Update DB to PROCESSING.
3. Fetch: Download file to temp storage.
4. Convert: Run conversion logic.
5. Success: Upload .md file, update DB to COMPLETED.
6. Failure: Increment retry. If MAX_RETRIES reached, move file to 'failed/' and mark FAILED.

6.2. The Journaling Logic

Query DB: SELECT * FROM conversion_jobs WHERE updated_at > requested_timestamp.

7. Configuration

---

* MAX_CONCURRENT_JOBS (e.g., 5)
* MAX_RETRIES (e.g., 3)
* STORAGE_BACKEND (e.g., S3, LOCAL)
* POLLING_INTERVAL (e.g., 300s)

8. Implementation Plan

Step 1: Core Scaffolding (FastAPI, SQLModel/SQLAlchemy, Celery).

Step 2: Converter Logic (MarkItDown/Pandoc).

Step 3: API & Worker Wiring.

Step 4: Robustness (Retries, Dead letters).

Step 5: Journaling Endpoint.