FastAPI File Management with PostgreSQL, Minio, Caching

Atishay Jain,

Bachelors of Technology (ECE), Batch of 2026, DTU

Project Guide / Mentor Name : Suparva Das

Period of Internship: 25th August 2025 – 31st October 2025

Report submitted to: IDEAS – Institute of Data Engineering, Analytics and Science Foundation, ISI Kolkata

FastAPI File Management Service

Abstract

This report details the implementation of a scalable file management and data processing service built on a modern, distributed architecture. The solution utilizes FastAPI for high-performance API handling,
PostgreSQL for relational metadata persistence, and MinIO (S3-compatible) for highly available binary
object storage. The service successfully implements endpoints for file ingestion, metadata retrieval, temporary data merging via Pandas and In-Memory Caching, and permanent storage of merged datasets. The resulting system demonstrates a robust, production-ready approach to separating and managing different types of data assets, ensuring data integrity, scalability, and processing efficiency.

1. Introduction

In modern data pipelines, the ability to rapidly ingest, process, and persist both structured metadata and large binary files is critical for business operations and analytics. This project addresses this requirement by creating a flexible microservice that integrates high-speed asynchronous processing with reliable persistence mechanisms. The service design acknowledges the fundamental difference between relational metadata (small, structured, indexed) and raw file binaries (large, unstructured, high-volume). The core goal was to implement four specific, interdependent tasks that collectively manage the lifecycle of an uploaded data file, from initial ingestion and processing to final merged output and cataloging within the metadata store.

2. Objective

The primary objective was to architect and implement a production-ready file management service that leverages asynchronous Python capabilities to meet the following functional requirements:

1. File Upload (POST /files/upload): Accept and validate large CSV/XLSX files, distributing the file binary to the object store and its critical metadata to the relational database.
2. File Retrieval (GET /files): Provide a single, reliable index of all persistently stored file metadata for client consumption and reference.
3. Temporary Merging (GET /files/merge): Fetch two stored files, perform a complex, user defined data merge (join) operation based on a common column, and temporarily cache the resulting high-volume dataset for high-speed access.
4. Permanent Save (POST /files/save_merged): Retrieve the processed, cached dataset and commit it to permanent object storage, simultaneously updating the metadata catalog and performing necessary cache cleanup.

3. Methodology

1. Architecture and Core Technologies
   The project employs a clear separation of concerns, utilizing specialized, high-performance technologies for each layer of the application. This microservice approach enhances both deployability and resource utilization

Component	Technology	Rationale and Role
API Service Layer	FastAPI (Python)	Chosen for its high performance (due to Starlette and Uvicorn) and its automatic data validation based on Python type hints and Pydantic models. It acts as the central coordinator.
Object Storage	MinIO (S3 Compatible)	Provides industry-standard, scalable, and high-throughput storage for large binary files, decoupling the API service from the physical file data.
Metadata Persistence	PostgreSQL	A reliable, robust relational database ideal for ensuring the integrity and quick retrieval of structured metadata (File ID, Filename, Format) via efficient indexing.
Temporary Caching	FastAPI Cache (In Memory/NullPool)	Provides fast, transient storage using Python memory, critical for holding the large merged dataset momentarily, ensuring the Task 4 endpoint can access the result without redundant merging.
Data Manipulation	Pandas	A foundational Python library used for reading data stream bytes into DataFrames, performing the vectorized merge operations (pd.merge), and serializing the results back into JSON.

3.2. Data Flow Diagram

The architectural model is centralized around the FastAPI service, which coordinates all actions between the two persistent stores and the transient cache layer. This design pattern ensures system resilience: if the database fails, the file binaries remain safe in MinIO, and vice versa.
Figure 1: Conceptual Data Flow Diagram illustrating the interactions between the FastAPI layer, the MinIO Object Store, the PostgreSQL Metadata Store, and the Caching Layer.

4. Analysis and Results

The implemented system successfully processes data through four distinct, verifiable stages, demonstrating full integration between all components and strict adherence to the functional contract of each API endpoint.

Task 1: POST /files/upload — Success Analysis
The endpoint successfully executes a dual-write operation, which is critical for resilience.

Action	Technical Detail	Verification in Code & Demo
File Upload	Reads file stream using UploadFile asynchronously	Confirms file handling is non-blocking.
Object Storage (MinIO)	minio_service.upload_file() streams bytes to the bucket.	File object visible in the MinIO Console with correct size/name.
Metadata Logging (PostgreSQL)	SQLAlchemy creates and commits a new FileMetadata record.	New row visible immediately in the PostgreSQL files table via PGAdmin.
Integration Proof	This atomic task confirms the functional connection between FastAPI, MinIO, and PostgreSQL.	

Task 2: GET /files — Metadata Integrity
The retrieval endpoint demonstrated the integrity of the PostgreSQL database as the single source of truth for file metadata, confirming that file metadata is indexed and searchable.

- Result: A seamless 200 OK status returns a list of all FileMetadata records, confirming the stability and proper indexing of the SQLAlchemy ORM mapping.

Task 3: GET /files/merge — Data Processing and Caching
This task involved the most computational complexity, validating the use of Pandas and the high speed In-Memory Cache.

- Process Details: Raw binaries fetched from MinIO are quickly buffered and converted into Pandas DataFrames. The merge (how='inner') is performed efficiently, and the large result set is serialized to JSON string format to ensure optimal cache storage.
- Result: The merged dataset is successfully serialized and stored in the In-Memory Cache, identified by a unique cache_key (UUID). This key is returned along with a data preview, validating the data manipulation and caching contract. The use of NullPool prevents connection pooling conflicts during high-demand I/O operations.

Task 4: POST /files/save_merged — Permanent Persistence and Cleanup
This final commitment endpoint proved the system's ability to manage the transition of data from transient memory to permanent storage, which is a key requirement for reliable data pipelines.

- Persistence Flow: The system validates the cache_key , deserializes the large JSON string back into a Pandas DataFrame, and then executes a dual-write: the raw file to MinIO and the new record to PostgreSQL.
- Cache Cleanup: Crucially, the last step involves calling cache_backend.delete(cache_key) , fulfilling the requirement to immediately remove the high-volume data from the In-Memory Cache, ensuring system resources are efficiently managed.

5. Conclusion

The developed FastAPI file management service successfully achieved all stated objectives, validating the utility of a microservice approach for data processing tasks. The technical solution is characterized by its modularity, utilizing the complementary strengths of each technology: the speed of FastAPI, the scalability and resilience of MinIO, the reliability of PostgreSQL for transactional integrity, and the efficiency of Pandas for in-memory data transformation. The four implemented tasks create a robust, end-to-end data lifecycle solution that is suitable for reliable production deployment.

6. Appendices

6.1. PostgreSQL Database Schema

```
-- Table used for file metadata persistence
CREATE TABLE files (
    id SERIAL PRIMARY KEY,
    filename VARCHAR(255) UNIQUE NOT NULL,
    format VARCHAR(10) NOT NULL
);
CREATE UNIQUE INDEX ix_files_filename ON files (filename);
```

6.2. Key Code Snippet: Merge and Cache Logic

The following snippet demonstrates the core logic for fetching files, merging, and caching the result:

```
# Snippet from Task 3: merge_files_temporarily

# 1. Fetch Metadata and Data (from PG and MinIO)
meta1 = db.query(FileMetadata).filter(FileMetadata.id ==
file_id_1).first()
# ... fetch meta2 ...
data1 = minio_service.fetch_file(meta1.filename)
data2 = minio_service.fetch_file(meta2.filename)

# 2. Read into DataFrames and Merge
df1 = get_file_dataframe(data1, meta1.format)
df2 = get_file_dataframe(data2, meta2.format)
merged_df = pd.merge(df1, df2, on=common_column, how='inner')

# 3. Store Merged Dataset Temporarily in In-Memory Cache
merged_json = merged_df.to_json(orient="records")
cache_key = str(uuid.uuid4())
cache_backend = FastAPICache.get_backend()
await cache_backend.set(cache_key, merged_json, expire=3600) #
Store for 1 hour

# 4. Return Key and Preview
return {"cache_key": cache_key, "preview":
merged_df.head().to_dict('records')}
```