NPCYF (Backend) Project

Shreya Shambhavi
Bachelor of Science in Data Science and Applications
Indian Institute of Technology Madras

Suprava Das
Project Mentor

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

IDEAS – Institute of Data Engineering, Analytics and Science Foundation
Indian Statistical Institute Kolkata

1. Abstract

The project focuses on the design, development, and implementation of a microservice-based file management system. It delivers a complete end-to-end workflow for handling file uploads, storage, and processing, all within a scalable, cloud-native architecture. It is built using `FastAPI`, uses `PostgreSQL` to store file metadata, and `MinIO` serves as S3-compatible object storage for files. Core functionalities include API endpoints for uploading CSV and Excel files, retrieving the list of stored files, and executing a data-processing pipeline. The pipeline enables users to dynamically merge any two files based on a common column, with the intermediate results temporarily cached via `fastapi-cache2`. This is followed by a dedicated endpoint that allows the cached, merged dataset to be fetched and permanently saved as a new file in `MinIO`, along with the updatation of `PostgreSQL` database. Therefore, the project serves as a practical demonstration of modern API development, data-intensive application design, and the importance of modularity in a microservice environment.

2. Introduction

In modern web applications, handling user-uploaded files is a fundamental requirement. However, storing files in a traditional database or on a server's local file system presents significant scalability and architectural challenges. In addition to this, data processing systems require efficient file storage, retrieval, and transformation capabilities. This project provides a backend solution that addresses these needs using modern technologies like `FastAPI`. It provides a system where file metadata (like filename, format, and ID) is stored in a robust `PostgreSQL` relational database, while the actual file objects are stored in a `MinIO` object storage bucket. The APIs provide a clean interface for a user to perform a complete data workflow : upload raw data files (CSV or Excel), view all stored files, store metadata in a relational database, select two files for a data-processing task (a join/merge), and save the new, transformed data as a permanent file in an object storage system. This solution is useful for structured file handling, data engineering, and analytics pipelines. The technologies involved in the project include:

- **FastAPI:** A high-performance web framework, used for building the backend services.
- **PostgreSQL:** A powerful open-source relational database, used to store and query all file metadata.
- **MinIO:** A high-performance S3-compatible object storage server, used to store the actual files (CSV, XLSX).

- **SQLAlchemy:** The Python SQL toolkit used for ORM-based database interaction.
- **Pydantic:** A powerful tool used to set up configuration and request/response validation.
- **Pandas:** The primary library used for data manipulation and processing.
- **fastapi-cache2:** The library used to implement in-memory caching.
- **Uvicorn:** The ASGI server used to run the FastAPI application.

Apart from the project, during the first two weeks of the internship, I received training on the following topics:

- Basic Statistics for Data Science
- Data Visualization
- Introduction to GitHub and Cloud Computing
- Streamlit (widget handling and database integration)
- Machine Learning (regression and classification)
- LLM Fundamentals
- Communication Skills
- Deep Learning
- Computer Vision Overview
- Basics of Natural Language Processing
- System View of Data Science

3. Objective

The primary objective of the project included the following:

1. To design and implement RESTful APIs with `FastAPI`, adhering to best practices for endpoint design and data validation using `Pydantic`.
2. To demonstrate the detached storage pattern by separating file metadata (stored in `PostgreSQL`) from file objects (stored in `MinIO`).
3. To implement a complex, multi-step data workflow:

- Ingestion (File Upload)
- Processing (File Merge)
- Storage (Save Merged File)

4. To utilize in-memory caching (`fastapi-cache2`) to manage temporary data for performance optimization.

5. To demonstrate API-driven data engineering.

4. Methodology

The application consists of the following core components:

1. **FastAPI Backend:** main application exposing all API endpoints.

2. **PostgreSQL Relational Database:** stores file metadata.

3. **MinIO S3 Object Storage:** stores actual physical files.

4. **In Memory Cache System:** holds merged/processed files temporarily.

5. **Pandas DataFrames:** used for merging and transforming datasets.

The project was developed following a structured, iterative process. The workflow is as follows:

1. **Environment Setup:** The setup begins with the creation of a dedicated virtual environment to ensure that all project dependencies remain isolated and consistent. All required dependencies were installed via `pip`, along with the updation of `requirements.txt` file. A `.env` file is then configured to store essential environment variables, including `PostgreSQL` credentials and `MinIO` connection details such as the endpoint, bucket name, and authentication keys. This centralised configuration supports secure and flexible deployment.

2. **Configuration:** A `config.py` file was created using `pydantic-settings` to load sensitive credentials (database connection strings, `MinIO` keys) from a `.env` file, keeping them out of the source code.

3. **Database Configuration:** The database configuration is defined in `database.py`, which sets up the `SQLAlchemy engine`, `session maker`, `declarative base`, and the `get_db()` dependency used to manage request-scoped database sessions. The database connection URL is constructed dynamically using environment variables, allowing seamless adaptation across different development or production environments.

4. **Data Model Definition:** The `PostgresSQL` database model is defined in `models.py`, which includes the **FileMetadata** class mapped to the `file_metadata` table. This model contains fields for file ID, file name, and file format (either CSV or XLSX). These attributes make it possible to keep track of all ingested files and maintain consistent metadata across the workflow.

5. **MinIO Integration:** The integration with `MinIO` is handled in `minio_client.py`. This module includes functionalities for uploading files to MinIO, downloading them for processing, and storing merged datasets back into the object storage system. By encapsulating these operations, the module ensures clean and consistent interaction with the `MinIO` server.

6. **CRUD Operations:** All operations related to managing file metadata are implemented in the `crud.py` module. This includes inserting new records, retrieving all stored entries, and accessing specific file metadata when required. These CRUD functions form the core interface between the API layer and the PostgreSQL database.

7. **API Endpoint Implementation:** All API-related logic is implemented in `api.py`. The `POST /file/upload` endpoint validates the uploaded file's format, stores its metadata in `PostgreSQL`, and uploads the file to `MinIO`. The `GET /files` endpoint returns all stored metadata entries. The `GET /files/merge` endpoint retrieves the relevant files from `MinIO`, loads them into pandas dataframes, performs the selected merge operation (inner, outer, left, or right) on the common column, caches the resulting merged dataset using `UUID`, and returns a preview along with the cache key. Finally, the `POST /files/save_merged` endpoint loads the cached merged data, saves the completed dataset to `MinIO`, and records its metadata in `PostgreSQL` - thereby completing the file processing and merge workflow.

GitHub Link: https://github.com/shreya-shambhavi/npcyf-backend-project

5. Analysis and Results

The Results of the project are the functional API endpoints and the successful orchestration of the data workflow. The system was validated using the `FastAPI` interactive documentation (`Swagger UI`).

API Endpoint Test Results

1. `POST /api/v1/files/upload`:

   - Action: Uploaded files, for example - `sample_file1.csv` and `sample_file1.xlsx`. Only CSV and Excel files are allowed.
   - Result: The API successfully returned a 200 OK response for each, showing the new database ID, base filename, and format.

2. `GET /api/v1/files`:
   - Action: Called the endpoint to list all files.
   - Result: The API returned a JSON array correctly showing the records for all stored files.

3. `GET /api/v1/files/merge`:
   - Action: Called the endpoint with the relational database ids of `sample_file1.csv`, `sample_file2.csv`, `common_column = id`, and `merge_type = inner`. Only files of the same format can be merged.
   - Result: The API returned a 200 OK response containing the message, a unique `cache_key (UUID)`, and a preview field with the first 5 rows of the merged data in clean JSON format.

4. `POST /api/v1/files/save_merged`:
   - Action: Called the endpoint using the `cache_key` from the previous step. The request body was `"cache_key": "your-uuid-key-here"`
   - Result: The API returned a 200 OK response with the metadata of the newly created permanent file.

6. Conclusion

This project successfully demonstrated the creation of a robust, scalable file management and processing microservice. The use of `FastAPI` provided high-performance APIs, while the architectural pattern of separating metadata (`PostgreSQL`) from file objects (`MinIO`) proved to be efficient and maintainable.
The key takeaway is that by combining these best-in-class open-source tools, it is possible to build complex, data-intensive workflows that are reliable and easy to manage. The merge-and-cache feature highlights a practical solution for handling temporary, compute-intensive results in a stateless API, allowing for a seamless user experience.

Future Work

While the project is fully functional, it provides a strong foundation for future enhancements:

1. Authentication and Security: Implement `OAuth2` authentication to secure all endpoints, ensuring only authorized users can upload or access files.

2. Large File Handling: For files larger than 1GB, the current in-memory approach with `pandas` would be inefficient. The upload and merge process could be rearchitected to use chunked/streaming operations and a background task queue (like `Celery or FastAPI’s BackgroundTasks`). Moreover, `Redis` based caching can be implemented for better performance.

3. Unit & Integration Testing: Develop a comprehensive test suite using `pytest` to ensure all helper functions and API endpoints are reliable and to prevent regressions.

4. Frontend: Provide a frontend UI for non-technical users.

7. Appendices

1. FastAPI Documentation: https://fastapi.tiangolo.com/

2. MinIO Documentation: https://docs.min.io/

3. SQLAlchemy ORM Documentation: https://docs.sqlalchemy.org/en/20/

4. fastapi-cache2 Documentation: https://pypi.org/project/fastapi-cache2/