**Storage Microservice Design Document**

**Introduction**

**This document outlines the design of a storage microservice intended to act as a centralized hub for storing and managing files of various extensions. The service will provide functionalities for file upload, retrieval, deletion, and metadata management. It will be designed to be robust, scalable, and secure, making it suitable for integration with multiple other microservices.**

**1. Functional and Non-functional Requirements**

**Functional Requirements:**

* **File Uploading**: The microservice must allow users to upload files, saving them to a specified directory and storing their metadata in a database.
* **File Retrieval**: Users must be able to retrieve files using a unique file identifier.
* **File Deletion**: Users must be able to delete files by their unique identifier.
* **Metadata Retrieval**: Users must be able to retrieve file metadata, including the file name, content type, size, and storage path.

**Non-functional Requirements:**

* **Performance**: The service should handle multiple file uploads and downloads concurrently without significant delays.
* **Scalability**: The service should be able to scale horizontally to handle increased load.
* **Security**: The service should ensure that only authorized users can upload, retrieve, or delete files.
* **Reliability**: The service should ensure data integrity and handle failures gracefully, providing meaningful error messages.
* **Maintainability**: The code should be modular and easy to maintain, allowing for future enhancements.

**2. High-Level and Low-Level Design**

**High-Level Design:**

* **Microservice Architecture**: The file service operates as a standalone microservice within a cloud-native architecture.
* **Components**:
  + **API Layer**: Exposes endpoints for file upload, retrieval, deletion, and metadata access.
  + **Database**: Stores file metadata (e.g., SQL Server, PostgreSQL).
  + **File Storage**: Saves files on the local file system or a cloud storage solution (e.g., AWS S3, Azure Blob Storage).

**Low-Level Design:**

* **Endpoints**:
  + POST /files - Uploads a file.
  + GET /files/{fileId} - Retrieves a file.
  + DELETE /files/{fileId} - Deletes a file.
  + GET /files/metadata/{fileId} - Retrieves file metadata.
* **Data Model**:
  + **FileMetadata**: Represents file properties (FileName, ContentType, Size, StoragePath).
* **Storage Implementation**:
  + Uses the local file system with a structured directory based on date for organization.

**3. Type of Storage**

**Storage Type**:

* **Local File System**: Currently, the service saves files to a local directory specified in the configuration. This is suitable for small-scale applications or development environments.

**Why**:

* **Simplicity**: Easy to set up and manage for initial development and testing.
* **Performance**: Fast access to files without network latency.

**How Files will be Saved and Retrieved**:

* **Saving**: Files are saved in a structured directory based on the current date (year/month/day) to avoid clutter and manage them easily. Each file is given a unique ID (ULID) when uploaded.
* **Retrieval**: Files are retrieved using the unique identifier stored in the metadata, ensuring that the correct file is accessed.

**4. Communication with Other Microservices**

**Communication Method**:

* **REST API**: Other microservices will communicate with the file service using HTTP requests to the exposed API endpoints.

**Data Exchange**:

* **JSON**: Data will be exchanged in JSON format for both requests and responses, making it easy to work with in various programming environments.

**Example**:

* A user service that needs to store user documents can send a POST request to the file service’s /files endpoint with the file data.
* Similarly, a reporting service can fetch files by sending a GET request to /files/{fileId} to retrieve necessary documents.

**Summary**

This design ensures a structured approach to file management within a microservices architecture while addressing both functional and non-functional requirements effectively. The choice of local file storage is suitable for initial setups, with the potential for future migration to cloud-based solutions as the application scales.