## Design and Architecture Overview

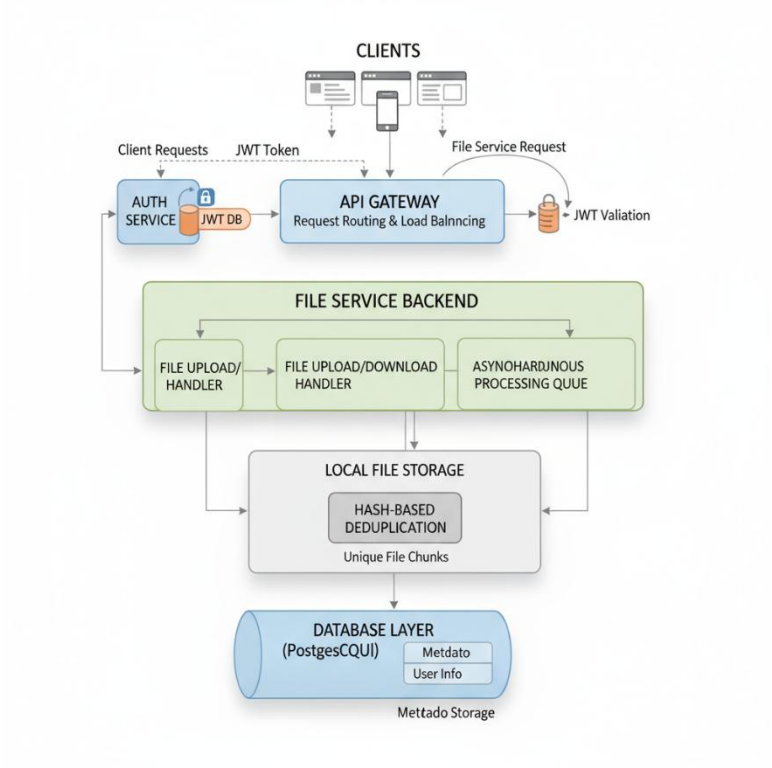
### System Overview

The File Service backend is designed as a modular, secure, and scalable microservice responsible for file management in a distributed application environment. It provides RESTful APIs to upload, list, download, delete, search, and publicly share files, all secured using JWT-based authentication.

---

### Architecture Components

1. API Layer
    - Built with Go and the Gorilla Mux router, serving HTTP endpoints.
    - Handles request parsing, authentication, and passing control to business logic.
2. Authentication
    - Encapsulated in a dedicated auth microservice issuing JWT tokens after validating user credentials.
    - File Service enforces JWT token validation as middleware, extracting user context to enforce file ownership and access policies.
3. File Storage
    - Files are stored locally on disk using SHA-256 content hashes as filenames for deduplication.
    - Metadata including filename, uploader, MIME type, size, public link, reference/download counts are stored in PostgreSQL for durability and querying.
4. Database Layer
    - PostgreSQL serves as the primary persistence layer for user and file metadata.
    - Supports trigram indexing for efficient substring searching on filenames.
    - Enforces constraints such as unique content hashes to support deduplication and integrity.




---

**Design Patterns and Principles**

- Microservices Architecture: File and Auth services are separated, enabling independent deployment and specialized scaling.
- JWT-Based Security: Stateless authentication improves scalability and secure access management.
- Deduplication using Content Hash: Prevents storage waste and manages references efficiently to multiple users.
- Asynchronous Metrics Update: Download count increments are performed asynchronously to avoid blocking I/O.
- RESTful API Design: Clear resource-based endpoints and consistent HTTP verbs improve maintainability and client consumption.
- Separation of Concerns: Authentication, file processing, and metadata management are cleanly delineated across services and layers.

---

**Scalability and Performance**

- Indexing on frequently queried fields (filename, uploader, content hash) enables low-latency search and retrieval.
- Content hash-based file naming streamlines storage lookups and prevents unnecessary duplication.
- Potential extension includes adding a distributed file system or cloud storage backend for scalability.

---

**Future Considerations**

- Implement rate limiting and quota management for users.
- Support for chunked uploads for very large files.
- Integration with CDNs or caching layers to accelerate public file access.
- Enhanced metadata search including tagging and full-text description.