21BCE0440
Rohan Rai

Documentation for File Sharing System Project

Project Overview

You have built a **File Sharing & Management System** backend using Go. The system supports essential features like user authentication, file uploading to S3, file metadata management with PostgreSQL, caching with Redis, and efficient concurrency handling. Additionally, optional features such as encryption, real-time WebSocket notifications, cloud hosting, and rate-limiting were also implemented. Below is the detailed documentation of the tasks completed in the project.

Task 1: User Authentication & Authorization

Objective: Implement secure user registration, login, and authorization for the file-sharing system.

Completed Work:

- **JWT-Based Authentication**: Used JSON Web Tokens (JWT) to handle user authentication. Upon successful login or registration, a token is generated, which is then used for subsequent authorized requests.
- **Middleware for Authorization**: Implemented `AuthMiddleware` to ensure only authenticated users can access protected routes like file upload, download, and deletion. The middleware extracts the JWT from the `Authorization` header and verifies its validity.

Key Files:

- `api/auth.go`: Contains `RegisterHandler`, `LoginHandler`, and `AuthMiddleware` for authentication and token management.
- `services/auth_service.go`: Handles the core authentication logic such as password hashing and token generation.

Task 2: File Upload to S3 with Metadata Storage in PostgreSQL

Objective: Implement file upload functionality to AWS S3, and store file metadata (e.g., file name, size, upload date) in PostgreSQL.

Completed Work:

- **File Upload**: Files uploaded by users are stored in an AWS S3 bucket.
- **File Metadata**: Information like file name, size, type, and upload date is stored in PostgreSQL.
- **S3 Configuration**: Configured an S3 bucket (`my-file-sharing-bucket1`) to securely upload and retrieve files.

- **Error Handling**: Handled potential errors during file upload, such as invalid file types or failed uploads.

Key Files:

- `services/s3_service.go`: Manages file uploads to AWS S3.
- `models/file.go`: Defines the database schema for storing file metadata (name, size, type, upload date, etc.).

Task 3: File Download with Authorization

Objective: Allow users to download their files, ensuring only authorized users can access their files.

Completed Work:

- **File Download with Authorization**: Implemented logic to restrict file downloads to the file's owner. This ensures that users cannot access files belonging to others.
- **Presigned URL**: Used AWS S3 presigned URLs to allow secure access to files for a limited time.

Key Files:

- `api/file.go`: Contains `DownloadFileHandler` to handle file download requests with proper authorization checks.

Task 4: File Search

Objective: Implement a search functionality for users to retrieve their files based on metadata like name, upload date, or file type.

Completed Work:

- **Search Feature**: Users can search for files based on file name, upload date, or file type using optimized queries.
- **Database Indexing**: Optimized the search functionality for large datasets by creating necessary indexes in PostgreSQL.

Key Files:

- `services/file_service.go`: Contains logic for searching files based on user queries.
- `models/file.go`: Schema includes indexing for efficient metadata searches.

Task 5: Caching Layer for File Metadata

Objective: Implement a caching mechanism to reduce database load when retrieving file metadata.

Completed Work:

- **Redis Cache**: Integrated Redis to cache file metadata. File metadata is cached upon retrieval and invalidated when the metadata is updated (e.g., file renamed).
- **Automatic Cache Expiry**: Metadata cache automatically expires after 5 minutes, ensuring cache freshness.
- **Cache Invalidation**: Cache is invalidated when metadata is updated or deleted.

Key Files:

- `services/cache_service.go`: Implements caching logic using Redis for file metadata.
- `models/file.go`: Used in combination with the Redis cache to store metadata.

Task 6: Database Interaction

Objective: Design the schema for storing file metadata, S3 locations, and user data in PostgreSQL, and ensure efficient interaction with the database.

Completed Work:

- **Schema Design**: Designed database schema for users and files, ensuring efficient storage and retrieval.
- **Database Transactions**: Handled critical operations, such as file uploads and deletions, using database transactions to ensure atomicity.
- **PostgreSQL Integration**: Used GORM to interact with PostgreSQL for storing and querying user and file data.

Key Files:

- `models/file.go`: Defines the file metadata schema.
- `models/user.go`: Defines the user schema.

Task 7: Background Job for File Deletion

Objective: Implement a background worker to periodically delete expired files from S3 and remove the corresponding metadata from the PostgreSQL database.

Completed Work:

- **Background Job with Goroutines**: Implemented a Goroutine to periodically check for expired files, delete them from S3, and remove their metadata from the database.
- **Efficient Scheduling**: Used Go's `time` package to schedule periodic cleanup jobs.

Key Files:

- `jobs/cleanup.go`: Contains the logic for periodically checking and deleting expired files.
- `services/s3_service.go`: Manages file deletion from S3.

Task 8: API Testing

Objective: Write tests for the authentication APIs (`/register` and `/login`).

Completed Work:

- **Unit Testing**: Wrote unit tests for `RegisterHandler` and `LoginHandler` to ensure correct behavior during registration and login operations.
- **Test Coverage**: Verified that tests cover scenarios such as missing fields, invalid credentials, and successful logins.

Key Files:

- `api/auth_test.go`: Contains unit tests for registration and login APIs.

Bonus Task 1: WebSocket for Real-Time File Upload Notifications

Objective: Implement a WebSocket-based system to notify users in real-time when their file uploads are completed.

Completed Work:

- **WebSocket Integration**: Implemented WebSocket connections that notify users when their file upload is complete.
- **Real-Time Notifications**: Once the file is uploaded to S3, users are notified via WebSocket.

Key Files:

- `api/websocket.go`: Manages WebSocket connections and notifications.

Bonus Task 2: File Encryption

Objective: Add encryption for files before storing them in S3, and decrypt them when accessed.

Completed Work:

- **AES-256 Encryption**: Implemented AES-256 encryption for files before uploading them to S3, ensuring file data is secure.
- **Decryption on Access**: Files are decrypted when downloaded by the user.

Key Files:

- `utils/encryption.go`: Contains the `Encrypt` and `Decrypt` functions to handle file encryption and decryption.

Bonus Task 3: Implement Rate Limiting

Objective: Add rate limiting to restrict API usage to 100 requests per user per minute.

Completed Work:

- **Rate Limiting**: Integrated rate-limiting functionality using the `tollbooth` package, restricting users to 100 requests per minute.
- **Global and User-Specific Limits**: Implemented both global rate limits and per-user limits to ensure fair usage.

Key Files:

- `api/ratelimit.go`: Implements rate limiting on API routes using `tollbooth`.

Conclusion

Throughout this project, the file-sharing system has evolved into a robust backend application with modern features like real-time notifications, encryption, and rate-limiting. The system efficiently handles user authentication, file management, and database operations, while maintaining high security and performance.