db:: filestorage table:: user File Storage API with User Authentication and Storage Limits Objective: The goal of this assignment is to build a RESTful API in Go that allows authenticated users to upload and manage files within their allocated storage space. The API should incorporate user authentication using JWT (JSON Web Tokens), file storage management with user-specific folders, and storage quota enforcement. Tasks: You are required to build a Go application that implements the following functionalities: 1. RESTful API Endpoints: a. User Registration (POST /register): i. Accepts user credentials (username and password). ii. Creates a new user in the system. iii. Should return a success or failure message with appropriate HTTP status codes. b. User Login (POST /login): i. Accepts user credentials (username and password). ii. Authenticates the user against stored credentials. iii. Upon successful authentication, generates and returns a JWT. c. File Upload (POST /upload): i. Authentication Required (JWT): Only authenticated users should be able to upload files. ii. Accepts a file upload request in multipart/form-data format. iii. Validates file size against the user's remaining storage. If the file size exceeds the remaining storage, reject the upload. iv. Creates a folder for each user based on their username (if it doesn't exist). v. Saves the uploaded file into the user's specific folder. vi. Stores file metadata (filename, original name, size, upload timestamp) in memory or a simple persistent storage (e.g., a file-based JSON database - optional). vii. Returns a success or failure message with appropriate HTTP status codes. d. Get Remaining Storage (GET /storage/remaining): i. Authentication Required (JWT): ii. Retrieves and returns the remaining storage space for the authenticated user. iii. Response should include the total storage allocated and the remaining storage. e. Retrieve Uploaded Files (GET /files): i. Authentication Required (JWT): ii. Retrieves a list of files uploaded by the authenticated user. iii. Returns a list of filenames (and optionally other metadata like size, upload date) associated with the user. iv. Consider implementing pagination if you are comfortable with it (optional bonus). 2. User Authentication with JWT: a. Implement user registration and login functionality. b. Use a library to generate and verify JWTs. c. Protect API endpoints (/upload, /storage/remaining, /files) using JWT authentication middleware. d. Store user credentials securely (consider hashing passwords using a library like bcrypt). 3. File Storage Management: a. Create a base directory for file storage. b. Within the base directory, create user-specific folders based on usernames. c. Save uploaded files in the corresponding user folders. d. Keep track of the storage used by each user. 4. Storage Limitation: a. Implement a fixed storage quota for each user. For example, set a default quota of 10MB or 50MB per user (you can configure this). b. When a user registers, allocate the storage quota to them. c. Before uploading a file, check if the user has enough remaining storage. d. Update the user's used storage after successful file uploads. 5. Get Remaining Storage: a. Implement an endpoint to retrieve the remaining storage for an authenticated user. b. Calculate remaining storage by subtracting the used storage from the total allocated storage. 6. Retrieve Uploaded Files: a. Implement an endpoint to retrieve a list of files uploaded by a user. b. Return a list of filenames (and optionally metadata) for the authenticated user. Submission Format: Github link to your private repo