Video Application: Project Architecture (Based on nilesh384/Video-App)

1. Introduction

This document outlines the architecture, features, and data flows for the Video App project. It is based on the existing codebase in the nilesh384/Video-App repository and is intended to serve as a technical reference for current and future development.

The application is built with a backend-first approach, focusing on establishing a robust API and data management system.

2. Core Features & Functionality (As Implemented)

2.1. User-Facing Features

- **User Authentication:**
    - Secure user registration with password hashing (bcrypt).
    - Login system that issues access and refresh tokens (JWT).
    - Logout functionality that clears refresh tokens.
    - Functionality to update user account details and avatars.
    - Password change and current user retrieval.
- **Channel & Profile Management:**
    - Viewing a user's public channel profile.
    - Access to watch history for logged-in users.
- **Social & Interaction:**
    - Subscription system: Users can subscribe to and unsubscribe from channels.
    - Retrieval of subscribed channels and a user's subscribers.
    - Video liking functionality.
    - Tweet-like feature for short text posts.
    - Commenting on videos.
- **Video Management:**
    - Video publishing (upload).
    - Retrieval of all videos with pagination.
    - Fetching, updating, and deleting specific videos by ID.
    - Toggling the publish status of a video.
- **Playlists:**
    - Functionality to create, update, and delete playlists.
    - Adding and removing videos from playlists.
    - Fetching user-specific and general playlists.

2.2. Backend Services

- **API Endpoints:** A comprehensive RESTful API built with Express.js to handle all application logic.
- **Database Management:** MongoDB for storing all application data, including users, videos, and social interactions.
- **File Storage & Processing:** Cloudinary for storing and managing media assets like user avatars and video thumbnails/files.
- **Middleware:** Custom middleware for authentication (verifyJWT) and file handling (Multer).

3. System Architecture

3.1. Backend Architecture

- **Language/Framework: Node.js** with the **Express.js** framework.
- **Database: MongoDB** with **Mongoose** as the Object Data Modeling (ODM) library for schema definition and data validation.
- **Authentication: JSON Web Tokens (JWT)** for stateless authentication, managed with jsonwebtoken and cookie-parser.
- **Asynchronous Handling:** Custom asyncHandler utility to wrap asynchronous route handlers and manage promises gracefully.
- **API Error Handling:** A custom API error and response structure for consistent communication with the client.
- **Media Storage: Cloudinary** for cloud-based storage of user-uploaded images and videos.
- **File Uploads: Multer** for handling multipart/form-data, used for uploading files from the client to the server before they are sent to Cloudinary.

3.2. Frontend Architecture

- *(Not yet implemented in the repository)*. The backend is set up to support a frontend framework like **React**, **Vue.js**, or **Svelte**.

4. Data & API Flows

4.1. User Registration Flow

1. A request is sent to POST /api/v1/users/register with user details (username, email, password) and an avatar file.
2. **Multer** middleware processes the file upload.
3. The avatar is uploaded to **Cloudinary**.
4. The user's password is encrypted using **bcrypt**.
5. A new user document is created and saved in the **MongoDB** users collection with the Cloudinary URL for the avatar.

6. The server responds with the created user data.

4.2. Video Upload Flow

1. An authenticated user sends a request to POST /api/v1/videos with video and thumbnail files, plus metadata (title, description).
2. The verifyJWT middleware confirms the user's identity.
3. **Multer** handles the multiple file uploads.
4. Both the video file and the thumbnail are uploaded to **Cloudinary**.
5. A new video document is created in the **MongoDB** videos collection, storing the URLs from Cloudinary and associating the video with the user.
6. The server responds with the details of the newly created video record.

4.3. Video Request Flow

1. A client requests a video's data via GET /api/v1/videos/:videoId.
2. The backend retrieves the video document from MongoDB using the provided videoId.
3. The document, containing metadata and the **Cloudinary URL** for the video file, is returned to the client.
4. The client-side video player then uses the Cloudinary URL to stream the video content directly from Cloudinary's CDN.

5. Database Schema (Mongoose Models)

User Model

- username (String, Unique)
- email (String, Unique)
- fullName (String)
- avatar (String - URL from Cloudinary)
- coverImage (String - URL from Cloudinary)
- watchHistory (Array of ObjectId refs to Video)
- password (String - Hashed)
- refreshToken (String)

Video Model

- videoFile (String - URL from Cloudinary)
- thumbnail (String - URL from Cloudinary)
- title (String)
- description (String)
- duration (Number - from Cloudinary)
- views (Number)
- isPublished (Boolean)
- owner (ObjectId ref to User)

Subscription Model

- subscriber (ObjectId ref to User)
- channel (ObjectId ref to User)

(Other models include Like, Comment, Tweet, Playlist)