Containerized Video Streaming System

Author: Ahmad Emad

GitHub: Containerized-Video-Streaming-System
YouTube Video: Demonstartion
Email: ahmademad995.ae@gmail.com

Objective Overview:

The objective of this system is to develop a Flask-based web application that allows users to securely authenticate, upload, store, and stream videos. The system ensures user authentication, efficient file management, and seamless video streaming, while leveraging MySQL for database storage and Docker for containerized deployment.

Table of Contents

Introduction

This project aims to develop a Flask-based web application that enables users to securely authenticate, upload, store, and stream MP4 files. The system is designed using a microservices architecture, with each core functionality running as an independent service.

System Components & Services

- **Authentication Service** – Manages user login and credential storage in MySQL.
- **File Upload Service** – Enables users to upload MP4 files, storing metadata in the database.
- **Streaming Service** – Provides a secure interface for streaming uploaded videos.
- **Database Service** – MySQL-based service for storing user credentials and file information.
- **Redis Service** – Used for caching authentication tokens and frequently accessed data to improve performance between containers.

The system is containerized using Docker to ensure portability and easy deployment across different environments. Docker Compose is used to orchestrate multiple services, ensuring seamless communication between them. Volumes are employed to persist user data, database records, and uploaded media files, preventing data loss even when containers are restarted.

System Architecture

Authentication Service

The authentication service is built using a Dockerfile that imports a Python image, creates a working directory, and copies the required dependencies from requirements.txt. These dependencies are then automatically installed. A volume is used to copy the Flask application into the working directory, ensuring seamless code updates. Finally, a command is executed to start the application.

The Flask application integrates essential libraries to handle user authentication efficiently. These include:

1. **Flask**: A lightweight Python web framework used to create web applications.
    - **redirect**: Redirects the user to another URL.
    - **render_template**: Renders HTML templates.
    - **request**: Accesses incoming request data (e.g., form data).
    - **session**: Stores user-specific data (e.g., username) across requests.
2. **flask_mysqldb**: A Flask extension for connecting to MySQL databases, allowing you to execute SQL queries and interact with MySQL.
3. **werkzeug.security**: Provides functions for hashing and verifying passwords.

- **generate_password_hash**: Hashes a password for secure storage.
- **check_password_hash**: Verifies if a provided password matches the stored hash.

4. **flask_session**: Manages server-side sessions, storing session data in a Redis store to allow persistence across requests.
5. **redis**: A key-value store used for session management, ensuring sessions are shared across multiple Flask containers.

Structured Approach

The application renders an HTML file stored in the default directory "/templates", while CSS files are stored in the "/static" directory. The user is then faced with a form application with labels username and password. When the user submits his credentials a POST request is sent, the server processes them as follows:

1. If the user already exists, the entered password is validated against the stored hash. If validated

   The application redirects the user to the upload service
2. If the user is new, The password is hashed and stored in the database along with the username and a unique user ID is generated
3. Upon successful authentication, the user is redirected to the upload service.

Upload Service

The upload service is built using a Dockerfile that pulls a Python image, creates a working directory and a subdirectory '/videos' to store the mp4 files in, then maps it with the volume 'videos', then installs dependencies from requirements.txt. Another volume is used to copy the Flask application into the working directory, ensuring seamless code updates. The Flask application is then executed via a command in the container.

The Flask application integrates essential libraries to handle file uploads efficiently. These include:

1. **Flask**: A lightweight Python web framework for building web applications.
   - **redirect**: Redirects the user to another URL after an action.
   - **render_template**: Renders and serves HTML templates to the client.
   - **request**: Accesses data sent in HTTP requests (e.g., form data, file uploads).
   - **session**: Stores user-specific data (e.g., user authentication status) across multiple requests.
2. **flask_mysqldb**: A Flask extension to connect and interact with MySQL databases, enabling you to execute SQL queries and retrieve results.
3. **flask_session**: Manages server-side sessions, allowing session data to be stored in a Redis database, which helps persist user sessions across multiple requests and servers.

4. **redis**: A fast, in-memory key-value store used for session storage and sharing session data across multiple Flask containers.
5. **os**: Provides utility functions for interacting with the operating system, such as working with file paths and file operations (e.g., saving files to a directory).

Structured Approach

The upload service renders an HTML file with a form application prompting the user to upload an mp4 file or the option to proceed to the streaming page if the user wants to stream a previously uploaded video, after selecting the designated file to upload the upload button activates. On submission the form sends a POST request to the server and is processed as follows:

1. The file is retrieved using the request.files method.
2. An SQL query is executed to retrieve the user ID using his username that was stored in the session.
3. Another SQL command is executed to insert the filename into the database referencing the user ID as the foreign key.
4. The filename is concatenated with the shared 'volume' directory to save the video in the new path.
5. Finally the application redirects the user to the streaming page giving the user the option to stream his uploads

Stream Service

The stream service is built using a Dockerfile that pulls a Python image, creates a working directory, a subdirectory '/videos' that maps the volume 'videos' to stream the uploaded mp4 files from, then installs dependencies from requirements.txt. Another volume is used to copy the Flask application into the working directory, ensuring seamless code updates. The Flask application is then executed via a command in the container.

The Flask application integrates essential libraries to handle file uploads efficiently. These include:

1. **Flask**: A lightweight Python web framework for building web applications.
    - **Response**: Used to generate a custom HTTP response, such as streaming video content.
    - **render_template**: Renders HTML templates to be served to the user.
    - **session**: Manages session data, typically storing user information across multiple requests.
    - **request**: Provides access to the incoming request data (e.g., query parameters for selecting videos).
2. **flask_mysqldb**: A Flask extension for interacting with MySQL databases, allowing to query data and retrieve information about the videos associated with the authenticated user.

3. **flask_session**: Manages session data server-side, using Redis as the storage backend, ensuring persistent sessions across multiple Flask containers.
4. **redis**: A fast, in-memory key-value store used to store session data, enabling sessions to be shared and persisted across different instances of the Flask application.
5. **os**: Provides utility functions for interacting with the operating system, such as working with file paths and handling file-related operations (e.g., locating video files for streaming).

Structured Approach

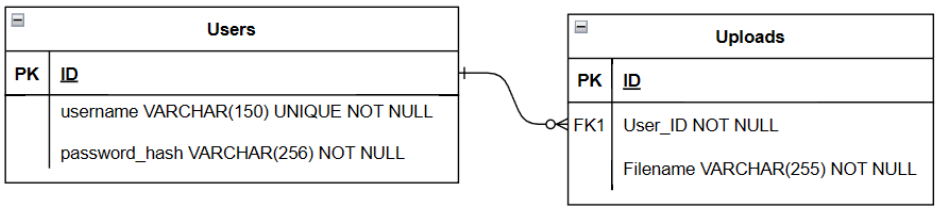
The stream service renders an HTML file providing the user with a list of the videos he/she uploaded to choose from. Once the user choose the video he/she wants to stream, the play video button activates for the video to start. This approach is processed as follows:

1. An SQL query is executed to retrieve the user ID using the username that was stored in the session.
2. Another SQL query is executed to retrieve the filenames' referenced by the user ID
3. An HTML file is rendered providing the user with all the videos he/she uploaded to select from
4. Once a video is selected play video button activates and calls the stream_video function passing the video name as a parameter
5. Stream_video function retrives the file path from the shared volume then generates the file as a video

Database Service (MySQL)

The database service is built using MySQL:8.0 image and initialized using a SQL script inside the host directory 'db-data/init.sql' creating the tables and applying constraints. The image initialization directory is mapped with the host initialization directory allowing MySQL to automatically run our script. Another mapping happens between the volume 'db-data' and the directory '/var/lib/mysql' within the container. This mapping allows to keep tables data stored and won't be lost when the container is stopped or removed.

ERD



Caching Service (Redis)

Redis is used in our Flask-based microservices architecture as a session store. Since our application consists of multiple containers (e.g., authentication, upload, and streaming services), using Redis ensures session data is shared across all services, allowing users to stay authenticated across different containers.

In our setup, Redis runs as a separate container and is accessed by other services. This is achieved using Docker Compose. other services connect to Redis by referring to its service name (redis) as the host:

```
services:
  auth_service:
    depends_on:
      - redis
```

Configuration

Each Flask service (authentication, upload, and streaming) connects to Redis using the following configuration:

```
app.config['SESSION_TYPE'] = 'redis'
app.config['SESSION_PERMANENT'] = True
app.config['SESSION_USE_SIGNER'] = True
app.config['SESSION_KEY_PREFIX'] = 'flask_session:'
app.config['SESSION_REDIS'] = redis.StrictRedis(host='redis', port=6379, db=0)
app.config['SECRET_KEY'] = 'secret'
```

- **SESSION_TYPE**: Specifies Redis as the session storage.
- **SESSION_PERMANENT:** When True, the session will persist even after the browser is closed
- **SESSION_REDIS**: Connects to the Redis container.
- **SESSION_USE_SIGNER**: Ensures session data integrity.
- **SESSION_KEY_PREFIX**: Adds a prefix to Redis session keys to avoid conflicts.

Conclusion

Redis deployment in our application ensures:

- **Persistent session management** across multiple containers.
- **Fast, in-memory storage** for session data.
- **Scalability and reliability**, preventing session loss even if a container restarts.

Volumes

The following named volumes are used for persistent storage across service restarts:

- **videos**: Stores uploaded video files, shared between stream and upload services.
- **db-data**: Stores MySQL database files for data persistence.
- **redis_data**: Stores Redis data with append-only persistence enabled.

Ports Mapping

Service	Internal Port	External Port
MySQL (db)	3306	3306
Redis (redis)	6379	6379
Stream Service	5002	5002
Upload Service	5001	5001
Auth Service	5000	5000

Conclusion

This Flask-based media management system provides a robust, scalable, and secure platform for user authentication, MP4 file uploads, and video streaming. By utilizing a containerized microservices architecture, the system ensures modularity and ease of maintenance. Each service—authentication, upload, streaming, database, and caching—operates independently, allowing for better resource management and fault isolation.

Key advantages of this architecture include:

- **Scalability:** Each service can be scaled independently based on demand. For example, if user authentication requests increase, additional instances of the authentication service can be deployed without affecting the upload or streaming services.
- **Improved Performance:** The integration of Redis for session management and metadata caching significantly reduces database queries, improving response times and overall system efficiency.
- **Data Persistence and Reliability:** The use of Docker volumes ensures that database records and uploaded files remain available even when containers restart, maintaining data integrity across service instances.
- **Secure Access Control:** The system implements secure authentication and session management using MySQL and Redis, preventing unauthorized access to stored media files. Additionally, sensitive credentials are managed via environment variables, reducing the risk of security vulnerabilities.
- **Seamless Deployment and Management:** By using Docker Compose, all services are orchestrated efficiently, making it easy to start, stop, and update the system with minimal downtime. This allows developers to maintain and enhance individual services without disrupting the entire application.

Moving forward, this system can be expanded with additional functionalities, such as user roles and permissions, video transcoding for adaptive streaming, and advanced analytics to track user engagement. Additionally, integrating cloud-based storage solutions and content delivery networks (CDNs) could further enhance performance and scalability.

By leveraging Flask, MySQL, Redis, and Docker, this project demonstrates how modern web applications can efficiently handle media content while maintaining high performance and security. It serves as a foundation for building scalable media management solutions in both personal and enterprise-level applications.