Group 5 **CSCI 112 - YW**

Artifacts Documentation for VIBES (Video-Integrated Backend Entertainment System)

Brief Introduction

Our Project, VIBES (Video-Integrated Backend Entertainment System), is a streaming platform in the Entertainment/Media industry designed to provide users with personalized video recommendations based on their viewing history and preferences. The platform allows users to create accounts and log in to access their watch history and receive tailored suggestions for new content to watch. By tracking user interactions—such as titles watched, time spent viewing, and ratings—it analyzes data to provide recommendations aimed to improve the user experience.

The key features include user authentication, watch history access, title search, content rating, personalized recommendations, viewing progress tracking, user statistics, and options to filter and sort content by criteria such as popularity, genre, rating, and country.

Application Architecture

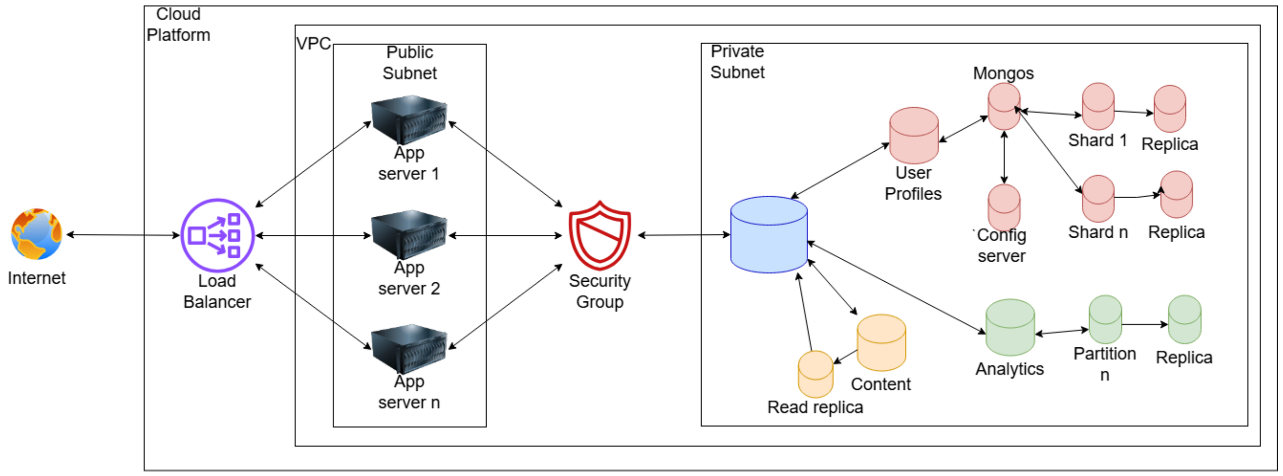

Figure 1. Ideal Application Architecture Diagram

In an ideal situation, the application architecture for VIBES would have databases clustered into user profiles, content, and analytics. The content and analytics databases would have respective read replicas while the user profiles database would employ sharding and read replicas for each shard to accommodate the expansion of the dataset. A load balancer would distribute network traffic among app servers, and a security group would control access to the resources.

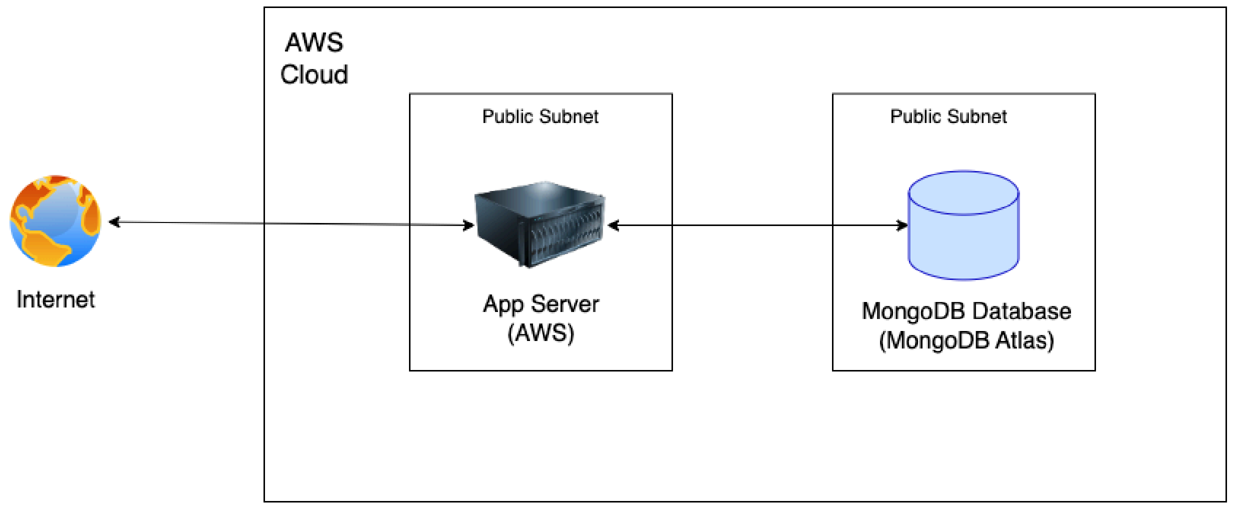

Figure 2. Actual Application Architecture Diagram

This shows our actual application architecture where the app server is an EC2 instance in a Public Subnet within the AWS VPC, allowing it to directly handle incoming user requests from the Internet. The app server communicates with an external, managed MongoDB Atlas database over the internet to store and retrieve data. The use of a Public Subnet ensures that the app server is accessible from the internet while maintaining secure connections to the MongoDB Atlas service, which is hosted outside the AWS environment. This setup reflects a cloud architecture with a public-facing application server and an external database service.

Business Process

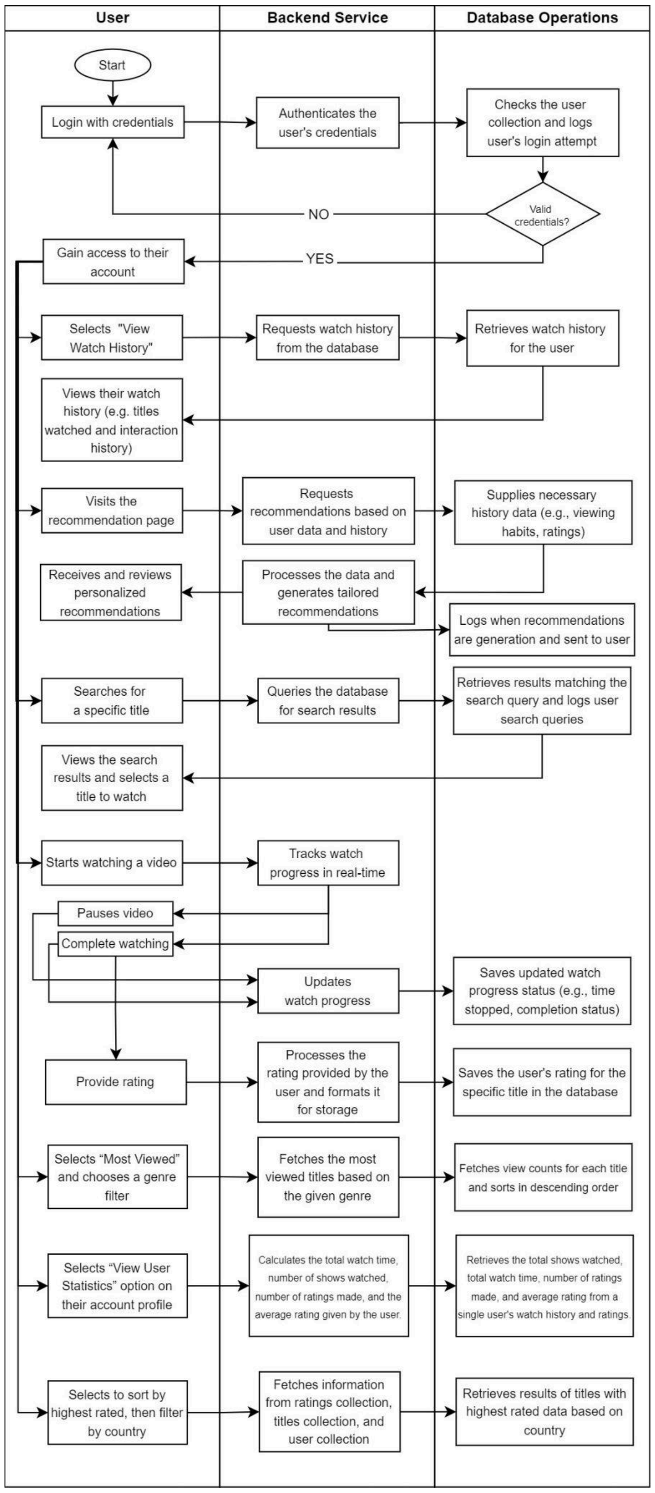
User
Backend Service
Database Operations
Start
Login with credentials
Authenticates the user's credentials
Checks the user collection and logs user's login attempt
Valid credentials?
NO
YES
Gain access to their account
Selects "View Watch History"
Requests watch history from the database
Retrieves watch history for the user
Views their watch history (e.g. titles watched and interaction history)
Visits the recommendation page
Requests recommendations based on user data and history
Supplies necessary history data (e.g., viewing habits, ratings)
Receives and reviews personalized recommendations
Processes the data and generates tailored recommendations
Logs when recommendations are generation and sent to user
Searches for a specific title
Queries the database for search results
Retrieves results matching the search query and logs user search queries
Views the search results and selects a title to watch
Starts watching a video
Tracks watch progress in real-time
Pauses video
Complete watching
Updates watch progress
Saves updated watch progress status (e.g., time stopped, completion status)
Provide rating
Processes the rating provided by the user and formats it for storage
Saves the user's rating for the specific title in the database
Selects "Most Viewed" and chooses a genre filter
Fetches the most viewed titles based on the given genre
Fetches view counts for each title and sorts in descending order
Selects "View User Statistics" option on their account profile
Calculates the total watch time, number of shows watched, number of ratings made, and the average rating given by the user.
Retrieves the total shows watched, total watch time, number of ratings made, and average rating from a single user's watch history and ratings.
Selects to sort by highest rated, then filter by country
Fetches information from ratings collection, titles collection, and user collection
Retrieves results of titles with highest rated data based on country

Figure 3. Business Process Diagram

Link to view the diagram and open with draw.io: https://tinyurl.com/CSCI112BusinessProcess

The swimlane diagram above illustrates the core business processes within the Video Streaming Platform System, distributed across three primary roles: User, Backend Service, and Database Operations. Each swimlane depicts the tasks performed by each role in sequential order.

Each swimlane distinctly shows how Users interact with the Backend Services, which then interface with Database Operations to fulfill each request. This division of roles and actions highlights the systematic flow from user requests to data storage and logging.

Schema

This section provides an overview of the data structure and organization used in our chosen NoSQL database, which is MongoDB. Unlike traditional relational databases, NoSQL databases utilize a flexible schema design that allows for dynamic and scalable data models, making it ideal for managing diverse and rapidly evolving data.

Our project uses a document-based NoSQL database, with data organized into eight collections. Each collection serves a distinct purpose within the application, holding documents that share a similar structure but can accommodate variations based on specific use cases. This schema design ensures flexibility while maintaining consistency where needed.

Below, we detail the purpose and structure of each collection, including the key fields, data types, and their respective descriptions.

1) user

user	**Type**	**Description**
_id	ObjectId	Unique identifier for the user
username	string	Username of the user
password	string	Password of the user
email	string	Email of the user
dateCreated	Date	Date when the user account was created
userCountry	string	Country of the user

2) title

title	**Type**	**Description**
_id	ObjectId	Unique identifier for the title

titleName	string	Name of the title
titleDescription	string	Description of the title
uploadDate	Date	Date when the title was uploaded
duration	int	Duration of the title in minutes
titleType	string	Type of the title (e.g., movie, series)
viewCount	int	Total number of views
ratingAverage	float	Average rating
ratingCount	int	Number of ratings
totalWatchTime	int	Total watch time in minutes
dateReleased	Date	Release date of the title
genre	array of strings	List of genres associated with the title
tags	array of strings	List of tags associated with the title

3) watchHistory

watchHistory	**Type**	**Description**
_id	ObjectId	Unique identifier for the watch history
userId	ObjectId (FK)	Foreign key referencing user._id
titleId	ObjectId (FK)	Foreign key referencing title._id
timeStopped	int	Time stopped in minutes
completionStatus	string	Indicates if the title was completely watched or not
lastUpdated	Date	Date when the title was watched last

4) eventLog

eventLog	**Type**	**Description**
_id	ObjectId	Unique identifier for the event log
userId	ObjectId (FK)	Foreign key referencing user._id

eventType	string	Describes the type of event or user interaction
details	string	Describes additional details regarding the event, if any
timestamp	Date	Timestamp of the event

5) contentSearchLog

contentSearchLog	**Type**	**Description**
_id	ObjectId	Unique identifier for the event log
userId	ObjectId (FK)	Foreign key referencing user._id
searchQuery	string	Search query text
searchedAt	Date	Timestamp of the event
resultsCount	int	Number of search results returned

6) rating

rating	**Type**	**Description**
_id	ObjectId	Unique identifier for the rating
userId	ObjectId (FK)	Foreign key referencing user._id
titleId	ObjectId (FK)	Foreign key referencing title._id
rating	float	Rating value
comment	string	User's comment on the title, if any
lastUpdated	Date	Date when the rating was given

7) loginLog

loginLog	**Type**	**Description**
_id	ObjectId	Unique identifier for the login log
userId	ObjectId (FK)	Foreign key referencing user._id
timestamp	Date	Timestamp of the login attempt

success	boolean	Status indicating if the login attempt was successful or not

8) recommendationLog

recommendationLog	Type	Description
_id	ObjectId	Unique identifier for the recommendation log
userId	ObjectId (FK)	Foreign key referencing user._id
recommendationData	array of strings	List containing ids of recommended titles
timestamp	Date	Date the recommendation was generated

Access Patterns

User Login and Account Authentication

- **Trigger**: User logs in with credentials.
- **Backend Service Action**: Validates the credentials by checking them in the database.
- **Database Access Pattern**:
    - **Read**: User Collection - Fetches user credentials to authenticate.
    - **Write**: Login Log Collection - Records the login attempt with timestamp and result (success/failure).
        - If authentication **fails**, no further access is granted. If **successful**, access to user features is enabled.

Access User's Watch History

- **Trigger**: User selects "View Watch History."
- **Backend Service Action**: Requests the user's watch history from the database.
- **Database Access Pattern**:
    - **Read**: Watch History Collection - Retrieves history records including titles watched, interaction details (e.g., time spent, viewing date).
    - **Write**: Event Log Collection - Logs access to the user's watch history for auditing purposes
- *Watch history data is crucial for generating personalized recommendations.*
- **Aggregation Pipeline to be Implemented:** Join watchHistory with the title collection.

Fetch Content Recommendations for a User

- **Trigger**: User visits the recommendations page.
- **Backend Service Action**: Requests recommendations based on user's data and history.

- **Database Access Pattern**:
    - **Read**:
        - Watch History Collection - Retrieves user's watch history, including ratings and viewing habits.
    - **Write**:
        - Event Log Collection - Logs when recommendations are generated and sent to the user.
- **Aggregation Pipeline to be Implemented**: Aggregate user's data from user, watchHistory, and rating collection to generate personalized recommendations for the user.
- *The recommendation engine uses historical data to suggest personalized content, and logs are stored to track recommendation generation.*

Search for a Specific Title

- **Trigger**: User searches for a specific title.
- **Backend Service Action**: Queries the database for matching titles.
- **Database Access Pattern**:
    - **Read**: Title (Content) Collection - Retrieves titles that match the search query (e.g., title, genre).
    - **Write**: Content Search Log Collection - Logs the search query along with user details for tracking purposes.

Pause or Stop Watching a Title

- **Trigger**: User pauses or stops watching the title.
- **Backend Service Action**: Updates the watch progress in the database.
- **Database Access Pattern**:
    - **Write:** Watch History Collection - Updates the current session record with the paused time or final status (complete/incomplete).
- *Storing this data ensures that users can resume content where they left off and provides insights into viewing habits.*

Rate Content After Viewing

- **Trigger**: User completes watching and provides a rating.
- **Backend Service Action**: Processes and stores the user's rating.
- **Database Access Pattern**:
    - **Write**: Rating Collection - Records the rating along with the user ID, title ID, and rating value, comment (if any), and the timestamp when the rating was given.
- *Ratings are used to improve content recommendations and to understand user preferences for future interactions.*

Get Most Viewed Titles of a Specific Genre

- **Trigger:** User selects "Most Viewed" and chooses a genre filter.

- **Backend Service Action:** Fetches the most viewed titles based on the given genre.
- **Database Access Pattern:**
    - **Read:** Title Collection - Fetches view counts for each title and sorts in descending order.
- **Aggregation Pipeline to be Implemented**: Unwind the genre field, match titles with the given genre, then sort the results in descending order by view counts.

<u>**Get User Statistics**</u>

- **Trigger:** User selects “View User Statistics” option in their account profile.
- **Backend Service Action:** Calculates the total watch time, number of shows watched, number of ratings made, and the average rating given by the user.
- **Database Access Pattern:**
    - **Read:** Watch History Collection - Retrieves the total number of shows watched and calculates total watch time based on the watch history of a single user.
    - **Read:** Rating Collection - Retrieves the number of ratings made and the average rating of a single user.
- **Aggregation Pipeline to be Implemented**:
    - Match records in the Watch History collection based on the user ID, then group by user ID to calculate aggregated statistics.
    - Match records in the Rating collection based on the user ID, then group by user ID to calculate aggregated statistics.

<u>**Get Highest Rated Titles by Country**</u>

- **Trigger:** User selects the option to sort by “highest rated” and filter by country.
- **Backend Service Action:** Fetches information from ratings collection, titles collection, and user collection.
- **Database Access Pattern:**
    - **Read:** Title Collection - Retrieves titles based on ratings and associated metadata.
    - **Read:** Rating Collection - Retrieves user ratings for titles.
    - **Read:** User Collection - Retrieves user country for filtering.
- **Aggregation Pipeline to be Implemented**: Match all users from the given country, join with rating collection, then sort based on total average rating.

Setup

Setting up the app server in AWS:

AWS Free Tier offers the use of EC2 instances, which are utilized to host the application that communicates with the MongoDB database. To set up the server, we used the default options available with the free tier, selecting the t2.micro instance size and Ubuntu 22.04 as the operating system. This configuration ensures cost-effective hosting while providing sufficient resources for the application during the development phase.

We chose AWS EC2 as the app server because it is one of the most popular and widely used options for cloud computing, offering scalability, reliability, and ease of management. However, for the purpose of our demo, we may use a local machine to avoid potential complications from differences in cloud and local environments, especially given the short timeframe.

Connecting to the app server:

After launching the EC2 instance, the server can be accessed using its public IP address, with the appropriate permission file for SSH access, or directly through the AWS Management Console. Once connected, pymongo is installed to facilitate communication with the MongoDB database. From this point, CRUD operations and aggregations can be performed on the database via pymongo, allowing the application to interact with the stored data.

Setting up the database in MongoDB Atlas:

MongoDB Atlas was used to set up our database due to its high availability and its ability to minimize downtime. Given that we made use of the free cluster available, its specifications are limited to those listed as follows:

- Storage: 512 MB
- RAM, vCPU: Shared
- OPS/SEC: 0 - 100

We configured it as follows:

- Cluster name: Grp5-CSCI112-FinalProj
- Username: grp5
- Password: Ra8b2hkQQ7dFiFUv
- Connection String: mongodb+srv://grp5:Ra8b2hkQQ7dFiFUv@grp5-csci112-finalproj.n88pt.mongodb.net/?retryWrites=true&w=majority&appName=Grp5-CSCI112-FinalProj

Code

We have 7 Python files for this project. We have our main.py file, which runs all our scripts. We also have the ff: createScripts.py, readScripts.py, updateScripts.py, and deleteScripts.py which contain all our queries written in functions and separated into different files according to the respective CRUD operations they implement. Lastly, we also have insertData.py which was used to populate our collections.

You may access our code through this link: https://github.com/addysalto/CSCI112-finals

main.py

- The main.py script serves as the primary entry point for testing the functionality of our video streaming platform's backend system. This script implements and handles the testing of CRUD operations, utility functions, and other key features like user login, search, recommendations, and analytics.

- **Import Statements**
  The script imports functions from the following modules:
  - **createScripts:** Functions for creating database records (e.g., users, titles, logs, etc.).
  - **readScripts:** Functions for reading and querying database records (e.g., user credentials, watch history, recommendations).
  - **updateScripts:** Functions for updating database records (e.g., watch progress, ratings).
  - **deleteScripts:** Functions for deleting database records (e.g., users, history, logs).
  - **utils:** Utility functions for generating and formatting test data.

- **Test Functions**
  The script defines several test functions, each focusing on a specific set of operations:
  - **test_create_operations()**
    - Tests the creation of users, titles, watch history, login logs, ratings, search logs, event logs, and recommendation logs.
    - Ensures that the corresponding createScripts functions are working as expected.
  - **test_read_operations(username, password, title_name)**
    - Validates the retrieval of user information, watch history, content recommendations, and title search results.
    - Prints the outputs for verification.
  - **test_update_operations(username, password, title_name)**

        - Tests the functionality of updating watch progress (e.g., marking content as "complete") and title ratings.
    - **test_delete_operations(username, password, title_name)**
        - Ensures the deletion of users, watch history, login logs, and ratings.

utils.py

- The utils.py file provides a collection of utility functions designed to simplify common operations required for the backend of our video streaming platform. This includes managing database connections, generating timestamps, and formatting data for various operations, such as creating logs, watch history entries, and rating information. The functions in this module are foundational for supporting CRUD operations and event logging throughout the system.

- **Database Connection Utilities**
    - **openConnection()**
        - Establishes a connection to the MongoDB cluster using a provided URI.
    - **closeConnection(conn)**
        - Closes an active MongoDB connection.

- **Time and Date Utilities**
    - **get_current_timestamp()**
        - Returns the current timestamp in ISO format, ensuring compatibility with MongoDB standards.
    - **random_date()**
        - Generates a random date between a specified start date (2000-01-01) and the current date (2024-12-04). This is particularly useful for creating test data with varied timestamps.

- **Data Creation Functions**
  These functions return structured dictionaries representing various entities or logs. They are used to create consistent data entries for different operations:
    a) **Login Data**
    - **create_login_data(user_id, success)**
        - Generates a login log entry containing the user's ID, the current timestamp, and the success status of the login attempt.

    b) **Watch History Data**
    - **create_watch_history_data(user_id, title_id, time_watched, completion_status="incomplete", last_updated=None)**
        - Creates a dictionary representing a user’s watch history, including: time stopped during a video, completion status (default: "incomplete"), and timestamp for the last update.

    c) **Event Log Data**

- **create_eventlg_data(user_id, event_type, details=None)**
    - Generates a log entry for a user event, including event type and optional details.

d) Search Log Data

- **create_searchlg_data(user_id, searchQuery, resultsCount)**
    - Creates a dictionary representing a user's search activity, including the search query, timestamp, and the number of results returned.

e) Rating Data

- **create_rating_data(user_id, title_id, rating, comment=None)**
    - Produces a rating entry for a specific title, containing: the user's ID and title ID, rating score and optional comments, and a timestamp for when the rating was last updated

f) Login Log Data

- **create_loginlg_data(user_id, status)**
    - Creates a dictionary to log the status of a user's login attempt.

g) Recommendation Log Data

- **create_recommendationlg_data(user_id, recommendations)**
    - Generates a log entry for recommendations served to a user, which includes: the user's ID, a list of recommended titles, and a timestamp for when the recommendation was created.

createScripts.py

- The createScripts.py file is dedicated to our Create operations. It serves as the primary interface for inserting new records into various collections in the database, including users, titles, watch history, logs, and ratings. Additionally, it includes a login() function to handle user authentication and logging of login attempts.

  Each function in this module ensures the database connection is securely opened and closed for every operation, reducing the risk of connection leaks and maintaining modularity.

- **Core Create Functions**
  These functions are responsible for adding data to the respective collections in the database. Each function does the ff: opens a connection to the database using the openConnection() utility, inserts the provided data into the appropriate collection, prints a confirmation message, including the ID of the newly inserted document, and closes the connection after the operation.
    - **create_user(user_data)**
        - Inserts a new user into the user collection.
    - **create_title(title_data)**
        - Adds a new title (e.g., movie or series) to the title collection.
    - **create_watch_history(watch_history_data)**

        - Records a user's watch history in the watchHistory collection.
    - **create_login_log(login_data)**
        - Logs a user's login attempt in the loginLog collection.
    - **create_rating(rating_data)**
        - Stores user ratings for titles in the rating collection.
    - **create_event(event_data)**
        - Logs events related to user activity in the event collection.
    - **create_search_log(search_data)**
        - Saves user search activity in the contentSearchLog collection.
    - **create_event_log(eventlg_data)**
        - Records event logs in the eventLog collection.
    - **create_recommendation_log(recommendationlg_data)**
        - Logs content recommendations for a user in the recommendationLog collection.

- **Authentication Function**
    - **login(username, password)**
        - Checks a user's credentials against the user collection.
        - Logs the login attempt (successful or failed) using the create_login_log() function.
        - Provides meaningful console feedback (e.g., "Login successful" or "Invalid password").
        - Returns the ID of the login log entry.

readScripts.py

- The readScripts.py file contains functions designed to retrieve data from the MongoDB database for ProjectVibes. These functions perform various read operations, such as fetching user information, watch history, content recommendations, search results, and statistics. Additionally, the file contains aggregation queries to process data, such as getting the most viewed videos, highest-rated shows by country, and user-specific analytics.

  Each function ensures proper connection management by opening and closing the database connection within its scope.

- **User-Related Queries**
  Functions to fetch user data based on credentials or user activity.
    - **get_user_by_credentials(username, password)**
        - Retrieves a user document matching the provided username and password.
    - **get_user_watch_history(user_id)**
        - Fetches a user's watch history and logs the access as an event.
    - **get_user_login_logs(user_id)**
        - Retrieves all login logs associated with a specific user.

- **Content-Related Queries**
  Functions that retrieve and process content-related data.
  - **get_content_recommendations(user_id)**
    - Uses a pipeline to aggregate a user's watch history and ratings to generate recommendations.
  - **search_titles(user_id, query)**
    - Searches for titles matching the query string (case-insensitive) and logs the search as an event.
  - **get_title_id_by_name(title_name)**
    - Finds a title by its name and returns its ID.

- **Aggregation-Based Queries**
  Advanced queries that process data using aggregation pipelines.
  - **get_most_viewed_videos_by_genre(genre, n)**
    - Retrieves the top n most viewed videos in a specific genre.
  - **get_user_statistics(user_id)**
    - Aggregates watch time and ratings to provide statistics for a user.
  - **get_highest_rated_shows_by_country(input_country, n)**
    - Retrieves the highest-rated shows from users in a specific country.

updateScripts.py

- The updateScripts.py file handles update operations in the ProjectVibes database. This script contains functions for updating records in the MongoDB collections related to user watch progress and title ratings. These functions allow for modifications to existing data or the creation of new records (in cases such as using the upsert option).

  All functions maintain proper connection management by opening and closing the database connection within their respective scopes.

- **Update Functions**
  - **update_watch_progress(user_id, title_id, time_stopped, completion_status)**
    - Updates a user's watch progress for a specific title.
  - **update_title_rating(user_id, title_id, rating, comment=None)**
    - Updates or creates a user's rating and optional comment for a title.

deleteScripts.py
- The deleteScripts.py file manages the deletion of records in the ProjectVibes database. It includes functions to remove users, titles, and various logs from their respective MongoDB collections. The module provides fine-grained control over database entries, ensuring proper data cleanup where necessary.

  Each function is designed with connection management in mind, ensuring the database connection is opened and closed correctly within the function's scope.

- **Delete Functions**
  - **delete_user(user_id)**
    - Deletes a user from the user collection.
  - **delete_title(title_id)**
    - Deletes a title from the title collection.
  - **delete_watch_history(user_id, title_id)**
    - Deletes a specific watch history entry.
  - **delete_login_log(user_id)**
    - Deletes all login logs associated with a user.
  - **delete_rating(user_id, title_id)**
    - Deletes a user's rating for a specific title.
  - **delete_search_log(search_log_id)**
    - Deletes a specific search log entry.

insertData.py

- The insertData.py file handles batch data insertion into the ProjectVibes database, enabling the creation of users, titles, watch histories, login logs, and ratings. It uses the helper functions from createScripts, readScripts, updateScripts, and utils modules to structure and insert data while maintaining database consistency.

  This script is primarily used for testing or initializing the database with sample data. It automates relationships between users, titles, and their associated logs.

- **Data Sources**
  - **user_data**
    - List of user dictionaries, each containing attributes like username, password, email, and user country.
  - **title_data**
    - List of title dictionaries, each containing attributes like title name, description, genre, and view count.

- **Core Function**
  - **insert_data(user_data, title_data)**
    - Inserts users, titles, and their associated records into the database.

- **Entry Point**
  - **main()**
    - Calls insert_data with predefined user_data and title_data.