Documentación de funcionalidades del back

Registro de Usuario (handle_register)

Descripción

Esta funcionalidad permite que un nuevo usuario cree una cuenta dentro de la aplicación. Se utiliza cuando un usuario se registra por primera vez desde la app móvil, completando su correo electrónico, un nombre de usuario visible y una contraseña.

Qué recibe

El registro se realiza mediante una solicitud **POST**, que puede venir en formato JSON o formulario.
En el **body** se envían tres datos obligatorios: el correo electrónico del usuario (email), el nombre de usuario que quiere mostrar (username), y la contraseña (password).
No se envían datos en **headers** (ya que no requiere autenticación previa) ni en **query parameters**.
El servidor responde con un mensaje JSON que indica el resultado del registro.

Dependencias clave

- **Flask (request, jsonify):** para recibir la solicitud HTTP y devolver la respuesta JSON.
- **SQLAlchemy (db):** para guardar el nuevo usuario en la base de datos local.
- **bcrypt:** para encriptar la contraseña antes de almacenarla.
- **AWS Cognito (Config.COGNITO_CLIENT):** servicio externo que maneja la autenticación y los tokens del usuario.
- **Función auxiliar get_secret_hash():** genera el hash requerido por Cognito para crear usuarios de forma segura.

Flujo

1. Se recibe la solicitud POST y se leen los datos del body.
2. El sistema verifica si ya existe un usuario con el mismo correo en la base de datos local. Si existe, devuelve un error indicando que el usuario ya está registrado.
3. Si no existe, se genera un **hash seguro** de la contraseña usando bcrypt, para no guardar contraseñas en texto plano.

4. Se crea el usuario en **AWS Cognito** mediante dos pasos:
    - admin_create_user: registra el usuario sin enviar email de verificación (MessageAction="SUPPRESS").
    - admin_set_user_password: establece la contraseña y la marca como permanente.
      Si Cognito devuelve el error UsernameExistsException, se responde con error indicando que el usuario ya existe.
5. Si la creación en Cognito es exitosa, se crea también el usuario en la base de datos local (Usuario) con el campo formulario_pendiente=True para indicar que aún no completó el formulario inicial de preferencias.
6. Se confirma la operación devolviendo una respuesta 200 con el mensaje “Registro exitoso”.

Inicio de Sesión (handle_login)

Descripción

Esta funcionalidad permite que un usuario ya registrado acceda a la aplicación iniciando sesión con su correo electrónico y contraseña.
Es el proceso que se usa cada vez que un usuario vuelve a abrir la app y quiere autenticarse.
Su función es validar las credenciales localmente y en Cognito, para generar los **tokens de sesión** (id_token, access_token, refresh_token) que luego permiten acceder a las funcionalidades protegidas del sistema (como ver películas recomendadas, crear grupos, o marcar favoritos).

Qué recibe

La solicitud también se realiza mediante **POST**, en formato JSON o formulario.
En el **body** se incluyen dos datos obligatorios: el correo electrónico (email) y la contraseña (password).
No utiliza **query parameters**.
Tampoco requiere **headers** especiales, ya que el usuario todavía no está autenticado.
Si las credenciales son correctas, el servidor devuelve un JSON con los tres tokens de

autenticación generados por Cognito, junto con el nombre del usuario y el estado del formulario inicial (formulario_pendiente).

Dependencias clave

- **Flask (request, jsonify):** maneja la petición y la respuesta JSON.
- **SQLAlchemy (db):** para consultar el usuario local en la base.
- **bcrypt:** para verificar la contraseña ingresada con el hash guardado en la base.
- **AWS Cognito (Config.COGNITO_CLIENT):** se usa para autenticar al usuario y generar los tokens de sesión.
- **Función auxiliar get_secret_hash():** genera el hash secreto que Cognito necesita para autenticar al usuario.

Flujo

1. Se recibe la solicitud POST y se leen los campos email y password.
2. El sistema busca en la base local un usuario con ese correo. Si no existe, devuelve un error de credenciales inválidas.
3. Si el usuario existe, se verifica la contraseña comparando el texto recibido con el hash guardado usando bcrypt.check_password_hash.
   Si la verificación falla, se devuelve un error 401 con “Error en las credenciales”.
4. Si la contraseña coincide, se arma el SECRET_HASH (requerido por Cognito) usando el email, el Client ID y el Client Secret.
5. Se solicita autenticación a **AWS Cognito** mediante initiate_auth con el flujo USER_PASSWORD_AUTH.
   - Si Cognito autoriza, devuelve los tres tokens (id_token, access_token, refresh_token), el nombre del usuario y si tiene o no pendiente el formulario inicial.
   - Si Cognito rechaza la autenticación (NotAuthorizedException), se responde con error 401.
   - Si ocurre otro error inesperado en Cognito, se responde con un error 500.
6. En caso exitoso, el backend devuelve los tokens y los datos del usuario para que la app los guarde localmente y los use en los siguientes pedidos protegidos.

Ver información del usuario (show_user_info)

Descripción

Permite obtener los datos básicos del perfil del usuario autenticado.
Se usa cuando la aplicación necesita mostrar o actualizar la información personal del usuario, como su país, nombre o las plataformas que tiene seleccionadas.

Qué recibe

Recibe una solicitud **GET** que incluye en el **header** el token JWT bajo el formato Authorization: Bearer <token>.
No utiliza parámetros en la **query** ni datos en el **body**.
El token se decodifica para extraer el correo electrónico del usuario, con el cual se obtienen sus datos en la base.

Dependencias clave

- **Flask (request, jsonify):** para manejar la petición y devolver la respuesta.
- **SQLAlchemy (joinedload):** para precargar relaciones del usuario con sus plataformas.
- **JWT:** para decodificar el token de autenticación.
- **DB (SQLAlchemy):** para consultar la base de datos local.

Flujo

1. Se obtiene el token del header Authorization.
2. Se decodifica el JWT (sin verificar firma) y se extrae el email.
3. Se busca el usuario correspondiente en la base y se cargan sus plataformas asociadas.
4. Se arma la respuesta con los datos del perfil: correo, nombre, país (id y bandera) y plataformas (id, nombre, logo).
5. Se devuelve una respuesta 200 con el JSON del perfil del usuario.

Actualizar datos del usuario (update_user_info)

Descripción

Permite que un usuario modifique su información básica (nombre, país o plataformas vinculadas).
Se utiliza cuando el usuario cambia su perfil o actualiza las plataformas que utiliza para ver películas.

Qué recibe

Recibe una solicitud **POST** con el token de autenticación en el **header** (Authorization: Bearer <token>).
El **body** puede venir en formato JSON o formulario, con uno o varios de los siguientes campos:

- nombre: nuevo nombre del usuario.
- id_pais: identificador del país.
- plataformas: lista de IDs de plataformas que reemplaza a las actuales.
  No recibe parámetros en la **query**.

Dependencias clave

- **Flask (request, jsonify):** para recibir y responder la solicitud.
- **SQLAlchemy (joinedload):** para cargar las plataformas asociadas al usuario.
- **JWT:** para obtener el correo electrónico desde el token.
- **DB (SQLAlchemy):** para guardar los cambios en la base.

Flujo

1. Se verifica que el método sea **POST**.
2. Se obtiene el email del usuario a partir del token.
3. Se busca el usuario en la base junto con sus plataformas actuales.
4. Se actualizan los campos recibidos: nombre, país y plataformas (reemplazo completo).
5. Se guardan los cambios con commit() y se devuelve una respuesta 200 confirmando la actualización.

Obtener opciones del formulario inicial (show_form)

Descripción

Devuelve los catálogos necesarios para construir el formulario inicial del usuario (onboarding).
Se usa la primera vez que el usuario entra a la app después del registro, para que seleccione país, plataformas y géneros de preferencia.

Qué recibe

Recibe una solicitud **GET** sin headers de autenticación ni datos en el body o query.
El servidor simplemente consulta las tablas Pais, Plataforma y Genero y devuelve la información para llenar el formulario.

Dependencias clave

- **Flask (request, jsonify):** para responder la solicitud.
- **SQLAlchemy (db):** para leer datos de las tablas Pais, Plataforma y Genero.

Flujo

1. Se consultan todas las filas de las tablas Pais, Plataforma y Genero.
2. Se formatean las listas con los campos relevantes:
    - Países: id, nombre y bandera.
    - Plataformas: id, nombre y logo.
    - Géneros: id y nombre.
3. Se arma una respuesta JSON con los tres catálogos.
4. Se devuelve una respuesta 200 con esos datos para mostrar en la app.

Guardar formulario inicial del usuario (save_user_form)

Descripción

Guarda las preferencias seleccionadas por el usuario al completar el formulario inicial: país, géneros favoritos, películas favoritas y plataformas que utiliza.
Se ejecuta una sola vez después del registro, para personalizar las recomendaciones y la pantalla principal.

Qué recibe

Recibe una solicitud **POST** con el token JWT en el **header** (Authorization: Bearer <token>).
El **body** puede ser JSON o formulario e incluye listas con los IDs elegidos por el usuario:

- countries: lista de países (solo se usa el primero).
- genres: IDs de géneros favoritos.
- movies: IDs de películas favoritas.
- services: IDs de plataformas seleccionadas.
  No usa parámetros en la **query**.

Dependencias clave

- **Flask (request, jsonify):** para recibir los datos y generar la respuesta.
- **SQLAlchemy (db):** para modificar las relaciones del usuario con país, géneros, películas y plataformas.
- **JWT:** para identificar al usuario mediante el token.

Flujo

1. Se leen los datos del body y se obtiene el token del header.
2. Se decodifica el JWT y se extrae el email para buscar al usuario en la base.
3. Se asignan los valores recibidos:
    - Se actualiza el país (countries[0] si existe).
    - Se reemplazan los géneros favoritos, películas favoritas y plataformas según las listas recibidas.
4. Se marca formulario_pendiente=False indicando que el usuario completó su configuración inicial.
5. Se guardan los cambios con commit() y se devuelve una respuesta 200 con el mensaje “Formulario guardado con éxito”.

Mostrar el Home personalizado al usuario (show_home_movies)

Descripción

Genera la selección de películas recomendadas para el usuario en la pantalla principal de la aplicación.

Las recomendaciones se basan en sus géneros favoritos, las plataformas que tiene configuradas y la disponibilidad de las películas en su país.
De esta forma, cada usuario ve un **home personalizado** con contenido relevante para sus preferencias.

Qué recibe

Recibe una solicitud **GET** con el token JWT en el **header** Authorization: Bearer <token>.
No usa parámetros en la **query** ni datos en el **body**, ya que toda la información se obtiene a partir del usuario identificado con el token.

Dependencias clave

- **Flask (request, jsonify):** para manejar la petición y generar la respuesta.
- **SQLAlchemy (db, joinedload, func, desc):** para consultar las películas y sus relaciones (géneros, plataformas, países).
- **JWT:** para decodificar el token y obtener el correo del usuario autenticado.
- **Modelos:** Usuario, Pelicula, Genero, PeliculaPlataformaPais.

Flujo

1. Se obtiene el token desde el header Authorization.
2. Se decodifica el JWT (sin verificación de firma) para extraer el email.
3. Se busca el usuario correspondiente en la base, cargando también sus géneros favoritos y plataformas mediante joinedload.
4. Se extraen los IDs de géneros, plataformas y país asociados al usuario.
5. Se realiza una consulta sobre las películas que cumplan con esas condiciones:
    - Coinciden con los géneros favoritos del usuario.
    - Están disponibles en alguna de las plataformas que tiene.
    - Están disponibles en el país del usuario.
6. Los resultados se ordenan por score_critica de forma descendente, se eliminan duplicados y se limita la lista a 6 películas.
7. Se devuelve una respuesta JSON con los campos básicos (id, title, poster, genres) para mostrarlas en el home de la app.

Búsqueda con paginación (request_movie_info)

Descripción

Permite buscar películas por título utilizando texto libre.
Se usa en la pantalla de búsqueda de la aplicación, cuando el usuario ingresa el nombre de una película o una palabra clave.
La búsqueda se realiza sobre la vista materializada MV_PELICULAS_COMPLETA y devuelve los resultados paginados de a 20 películas.

Qué recibe

Recibe una solicitud **GET** con dos parámetros en la **query**:

- query: texto que se busca en los títulos (obligatorio).
- page: número de página (opcional, comienza en 0).
  No requiere autenticación, por lo que no usa headers ni body.

Dependencias clave

- **Flask (request, jsonify):** para recibir los parámetros y devolver el resultado.
- **SQLAlchemy (db, func, or_):** para aplicar filtros de búsqueda con to_tsvector (búsqueda semántica) y ILIKE (coincidencia parcial).
- **Modelo:** PeliculaCompleta, que contiene los datos listos para mostrar (géneros y plataformas en formato JSON).

Flujo

1. Se valida que exista el parámetro query; si falta, se devuelve error 400.
2. Se interpreta el parámetro page como número entero, tomando 0 si no se envía o es inválido.
3. Se consulta la vista PeliculaCompleta aplicando dos estrategias de búsqueda:
   - Búsqueda por texto completo (to_tsvector y plainto_tsquery).
   - Búsqueda parcial por coincidencia (ILIKE '%query%').
4. Los resultados se ordenan por título, con un límite de 21 elementos (para detectar si hay siguiente página) y un desplazamiento calculado según la página (OFFSET page * 20).

5. Se formatea la respuesta con la información necesaria para mostrar en la app: id, título, póster, géneros, plataformas, año, duración, director, puntuación, descripción y clasificación por edad.
6. Se devuelve una respuesta 200 con la lista de películas.

Detalle de película (movie_details_screen_info)

Descripción

Devuelve toda la información de una película específica para mostrar en la pantalla de detalle.
Además, indica si la película ya fue marcada como favorita por el usuario.
Esta funcionalidad se usa cuando el usuario selecciona una película desde el home o desde una búsqueda.

Qué recibe

Recibe una solicitud **GET** con el identificador de la película en la **query** (movieId=<id>) y el token JWT en el **header** Authorization: Bearer <token>.
No incluye datos en el **body**.
El token permite identificar al usuario para verificar si la película se encuentra entre sus favoritas.

Dependencias clave

- **Flask (request, jsonify):** para manejar la petición y la respuesta.
- **SQLAlchemy (db):** para consultar la película y las relaciones del usuario.
- **JWT:** para decodificar el token y extraer el email del usuario.
- **Modelos:** Usuario, PeliculaCompleta.

Flujo

1. Se verifica que se haya recibido el parámetro movieId; si no, se responde con error 400.
2. Se obtiene el token del header y se valida que esté presente; si no, se devuelve error 401.
3. Se decodifica el JWT (sin verificación de firma) y se extrae el email del usuario.

4. Se busca el usuario en la base de datos; si no existe, se devuelve error 404.
5. Se busca la película correspondiente en la tabla PeliculaCompleta.
   Si no se encuentra, se devuelve error 404 (en el código actual se devuelve 401, pero debería ajustarse).
6. Se arma un diccionario con los datos de la película (géneros, plataformas, año, duración, director, puntuación, descripción, clasificación por edad) y se agrega el campo is_favorite, que indica si la película se encuentra en la lista de favoritas del usuario.
7. Se devuelve una respuesta 200 con el JSON completo de la película.

Crear grupo (create_group)

Descripción

Crea un nuevo grupo y agrega automáticamente como primer miembro al usuario autenticado. Se usa cuando alguien quiere iniciar un grupo para decidir películas en conjunto.

Qué recibe

Llega una solicitud **POST** con el **header** Authorization: Bearer <token> para identificar al usuario. En el **body** (JSON o form) viene el nombre del grupo bajo la clave group_name. No utiliza parámetros en **query**.

Dependencias clave

Flask (request/jsonify), Modelos (Usuario, Grupo), DB (SQLAlchemy db), helper generate_id() para crear el identificador del grupo, y JWT para leer el email desde el token.

Flujo

1. Lee group_name del body.
2. Obtiene el email decodificando el token (JWT sin verificación de firma).
3. Busca al usuario creador.
4. Genera id_grupo con generate_id().
5. Crea el Grupo con ese id y nombre, y añade al creador a la lista de miembros.

6. Hace commit y calcula el **código de unión público**: group_join_id = id_grupo * 7 + 13.
7. Devuelve **200** con { "group_join_id": <int> }.

Nota general de códigos: el **código de unión** que se comparte en pantalla es group_join_id = id_grupo * 7 + 13. Para obtener el verdadero id_grupo se calcula id_grupo = (group_join_id - 13) // 7.

Unirse a un grupo por código (add_user_to_group)

Descripción

Agrega al usuario autenticado como miembro de un grupo existente usando el **código de unión** que le compartieron. Se usa cuando un integrante quiere sumarse a un grupo ya creado.

Qué recibe

Llega una **POST** con **header** Authorization: Bearer <token>. En el **body** (JSON o form) llega group_join_id. No usa **query**.

Dependencias clave

Flask (request/jsonify), Modelos (Usuario, Grupo), DB (SQLAlchemy db), y JWT para identificar al usuario.

Flujo

1. Lee group_join_id del body.
2. Calcula id_grupo = (group_join_id - 13) // 7.
3. Obtiene el email desde el token y busca al Usuario.
4. Busca el Grupo por id_grupo.
5. Agrega al usuario a grupo.usuarios y hace commit.
6. Devuelve **200** con un mensaje de éxito.

Lista de grupos del usuario (get_user_groups)

Descripción

Devuelve todos los grupos a los que pertenece el usuario autenticado, con información básica para listar en la app. Se usa para la pantalla “Mis grupos”.

Qué recibe

Llega una **GET** con **header** Authorization: Bearer <token>. No usa **query** ni **body**.

Dependencias clave

Flask (request/jsonify), Modelos (Usuario), DB (SQLAlchemy db), y JWT para identificar al usuario.

Flujo

1. Decodifica el token (sin verificación) y obtiene el email.
2. Busca el Usuario.
3. Recorre usuario.grupos y arma una lista con { id, name, members } (cantidad de integrantes).
4. Devuelve **200** con esa lista.

Lista de usuarios de un grupo (get_group_users)

Descripción

Devuelve los integrantes de un grupo específico (email y username). Se usa para ver quiénes forman parte del grupo en la UI.

Qué recibe

Llega una **GET** que **recibe en query** el **ID real del grupo** (no el join code) como group_id=<int>. No requiere **body** y, según el código actual, tampoco exige **header** de autorización.

Dependencias clave

Flask (request/jsonify), Modelos (Grupo), DB (SQLAlchemy db).

Flujo

1. Lee group_id desde la query (ya casteado a int).
2. Busca el Grupo por id_grupo.
3. Recorre grupo.usuarios y arma la lista con { email, username }.

4. Devuelve **200** con la lista (vacía si no hay miembros).

Agregar o quitar favoritos (add_remove_favorite_movie)

Descripción

Permite al usuario autenticado **agregar o eliminar películas de su lista de favoritas**.
Se usa cuando el usuario marca una película como favorita o decide quitarla de esa lista.

Qué recibe

Recibe una solicitud **POST** con el token JWT en el **header** Authorization: Bearer <token>.
En el **body** (JSON o formulario) se envían:

- movie_id: identificador de la película (obligatorio).
- action: acción a realizar, que puede ser "add" o "remove" (por defecto "add").
  No utiliza parámetros en la **query**.

Dependencias clave

Flask (request, jsonify), Modelos (Usuario, Pelicula), DB (SQLAlchemy db), y JWT para identificar al usuario.

Flujo

1. Lee movie_id del body; si falta, devuelve error 400.
2. Busca la película en la base de datos; si no existe, devuelve error 404.
3. Decodifica el token (sin verificación de firma) y obtiene el email del usuario.
4. Busca el Usuario correspondiente; si no existe, devuelve error 404.
5. Si la acción es "remove":
    - Si la película no está en favoritas, responde 200 con mensaje informativo.
    - Si está, la quita de la lista, guarda los cambios (commit) y responde 200.
6. Si la acción es "add" (por defecto):
    - Si la película ya está en favoritas, responde 200 con mensaje informativo.
    - Si no, la agrega a la lista, guarda los cambios y responde 200.

Listar favoritos (show_favorites)

Descripción

Devuelve todas las películas que el usuario autenticado tiene marcadas como favoritas.
Se usa para la pantalla de “Mis favoritas” o para mostrar la lista personalizada del usuario.

Qué recibe

Recibe una solicitud **GET** con el token JWT en el **header** Authorization: Bearer <token>.
No usa **body** ni **query parameters**.

Dependencias clave

Flask (request, jsonify), Modelos (Usuario, Pelicula), DB (SQLAlchemy db), y JWT para autenticar al usuario.

Flujo

1. Valida que se haya enviado el token y decodifica el email.
2. Busca al Usuario en la base de datos; si no existe, devuelve error 404.
3. Obtiene la lista de películas favoritas (usuario.favoritas).
4. Devuelve una respuesta 200 con un JSON que contiene id, title y poster de cada película favorita.

Ratear una película (rate_movie)

Descripción

Permite al usuario calificar una película que ya vio, guardando o actualizando su calificación personal (rating).
Se usa para que la app pueda mostrar y almacenar las valoraciones individuales de cada usuario.

Qué recibe

Recibe una solicitud **POST** con el token JWT en el **header** Authorization: Bearer <token> y el contenido en el **body** (JSON o formulario).
El body debe incluir:

- movie_id: ID de la película a calificar (obligatorio).
- rating: valor numérico de la calificación (obligatorio).
  No utiliza **query parameters**.

Dependencias clave

Flask (request, jsonify), Modelos (Usuario, Pelicula, UsuarioVioPeli), DB (SQLAlchemy db), y JWT para identificar al usuario.

Flujo

1. Valida el token y decodifica el email (maneja posibles errores de decodificación).
2. Busca al Usuario; si no existe, devuelve error 404.
3. Obtiene movie_id y rating del body; si faltan, devuelve error 400.
4. Busca la película en la base; si no existe, devuelve error 404.
5. Consulta si ya existe una relación UsuarioVioPeli entre el usuario y la película:
   - Si existe, actualiza el campo rating.
   - Si no, crea una nueva relación con ese rating.
6. Guarda los cambios (commit) y responde 200 con mensaje de éxito.

Obtener rating del usuario de una película (get_user_rating)

Descripción

Devuelve la calificación que el usuario autenticado le dio a una película específica.
Se usa en la pantalla de detalle de una película para mostrar el rating personalizado del usuario (si existe).

Qué recibe

Recibe una solicitud **GET** con el token JWT en el **header** Authorization: Bearer <token>.
El **ID de la película** se pasa como parámetro en la **query** (movie_id=<int>).
No se envían datos en el **body**.

Dependencias clave

Flask (request, jsonify), Modelos (Usuario, UsuarioVioPeli), DB (SQLAlchemy db), y JWT para autenticar al usuario.

Flujo

1. Valida el token y extrae el email.
2. Busca el Usuario en la base de datos.
3. Obtiene movie_id desde la query.
4. Busca la relación UsuarioVioPeli (mail, id_pelicula):
    - Si existe, devuelve {"rating": <int>}.
    - Si no existe, devuelve {"rating": null}.
5. Responde con código 200 en ambos casos.

Listar películas vistas del usuario (get_seen_movies)

Descripción

Devuelve todas las películas que el usuario autenticado ha visto, junto con su rating personal (si lo tiene).
Se usa en la pantalla de “Películas vistas” o para generar estadísticas de actividad del usuario.

Qué recibe

Recibe una solicitud **GET** con el token JWT en el **header** Authorization: Bearer <token>.
No se envían datos en **body** ni **query**.

Dependencias clave

Flask (request, jsonify), Modelos (Usuario, Pelicula, UsuarioVioPeli), DB (SQLAlchemy db), y JWT para identificar al usuario.

Flujo

1. Valida el token y extrae el email.
2. Busca el Usuario en la base; si no existe, devuelve error 404.
3. Realiza un **join** entre UsuarioVioPeli y Pelicula filtrando por mail_usuario.
4. Si no hay resultados, devuelve una lista vacía [].
5. Si hay coincidencias, arma una lista con los campos id, title, rating y poster.

6. Devuelve una respuesta 200 con esa lista en formato JSON.