UNIVERSIDAD DE PANAMA

CENTRO REGIONAL UNIVERSITARIO DE COCLÉ

FACULTAD DE INFORMÁTICA, ELECTRÓNICA Y COMUNICACIONES

LICENCIATURA EN INGENIERÍA EN INFORMÁTICA

INFORMÁTICA TEÓRICA

PROYECTO SEMESTRAL

DESARROLLO DE UN GESTOR DE PELÍCULAS Y SERIES

PROFESOR

LUIS DOMÍNGUEZ

MANUAL DE USUARIO

Autores

- Moises Ramos
- Nidia Rojas
- Cristhian Alonso
- Julio Gonzalez

Índice

DESARROLLO DE UN GESTOR DE PELÍCULAS Y SERIES

Introducción

Bienvenido al manual de usuario del Catálogo de Películas y Series, una aplicación desarrollada con Flask que permite gestionar de manera eficiente y sencilla una colección de películas y series. Este proyecto tiene como objetivo proporcionar una herramienta intuitiva para registrar, consultar, actualizar y eliminar información relevante sobre películas y series.

Objetivo del Proyecto

El objetivo de este proyecto es proporcionar una herramienta eficiente y fácil de usar para la gestión de películas y series, permitiendo a los usuarios registrar, consultar, actualizar y eliminar información de manera intuitiva.

Tecnologías Utilizadas

Este proyecto utiliza diversas tecnologías y herramientas, incluyendo:

- Flask para el desarrollo del backend.
- SQLite como base de datos.
- Docker para la contenerización y despliegue.
- HTML y CSS para el frontend.

Estructura del Manual

Este manual está dividido en varias secciones que cubren desde la instalación y configuración del sistema hasta la interacción con la interfaz de usuario y la API.

Beneficios para el Usuario

Con este sistema, los usuarios podrán gestionar su colección de películas y series de manera eficiente, con una interfaz amigable y funcionalidades avanzadas como búsqueda y filtrado.

I. Visión General del Flujo de la Aplicación

El flujo de datos en la aplicación comienza cuando el usuario interactúa con los formularios HTML en el frontend. Estos datos son enviados al backend de Flask, donde se procesan y se almacenan en la base de datos SQLite. Las respuestas del servidor son luego enviadas de vuelta al frontend para ser mostradas al usuario.

1. Roles y responsabilidades

- ✓ Frontend (HTML): Los formularios HTML permiten a los usuarios interactuar con la aplicación. Cada formulario está diseñado para capturar datos relacionados con las películas y realizar acciones sobre el catálogo.
- ✓ Backend (Flask): El backend de Flask maneja la lógica de la aplicación. Recibe las solicitudes del frontend (formularios HTML), procesa los datos y actualiza la base de datos según las acciones solicitadas (crear, leer, actualizar, eliminar).
- ✓ Base de Datos (SQLite)**:** Almacena la información de las películas y series.

2. Interacción entre componentes

Cuando un usuario envía un formulario para agregar una nueva película, el frontend envía una solicitud POST al endpoint correspondiente en el backend. Flask recibe esta solicitud, valida los datos y los guarda en la base de datos. Luego, el backend envía una respuesta al frontend confirmando la operación.

II. Interacción con los Formularios HTML en Flask

Los formularios HTML son una interfaz gráfica que permite a los usuarios interactuar con la aplicación. En el caso de tu aplicación Flask con CRUD, los formularios HTML tienen un propósito claro:

1. Estructura del código:

Cada bloque de película está envuelto en una estructura que centra todo el contenido de cada sección. Dentro de este bloque hay un enlace <a> que contiene una imagen <img> que, al hacer clic en ella, lleva al usuario a la página web correspondiente de la película.

2. Enlaces

Los enlaces se definen mediante la etiqueta <a>, que permite redirigir a los usuarios a una URL cuando hacen clic sobre ella. Dentro de esta etiqueta <a>, puedes colocar varios tipos de contenido, como imágenes o texto. En este caso, el enlace envuelve una imagen (usando la etiqueta <img>), lo que

significa que cuando un usuario hace clic sobre la imagen, será redirigido a la URL que está especificada en el atributo href del enlace.

3. **Imágenes de las películas**

Está definida con la etiqueta <img>, que es la encargada de mostrar la imagen en la página web. El atributo src dentro de la etiqueta <img> especifica la ubicación o la dirección del archivo de la imagen, como por ejemplo "No_hables_con_extraños.jpg", lo que indica que la imagen está almacenada en esa ruta específica. Esto permite que el navegador cargue y muestre la imagen asociada a cada película.

Además, se utiliza el atributo height="200", que establece la altura de la imagen a 200 píxeles. Esto ajusta el tamaño de la imagen sin distorsionarla, ya que mantiene la proporción original entre el ancho y la altura.

4. **Agrupación por categorías**

Se utiliza la etiqueta <font> para mostrar el título de la categoría, como "Comedia", con un estilo específico. A través de los atributos de la etiqueta, como size, color o face, se pueden definir características visuales, como el tamaño, color y tipo de letra del texto, para hacerlo más llamativo y diferenciarlo de otros elementos en la página.

III. Detalles del Backend en Flask

El núcleo del backend en Flask es nuestro programa **main.py,** ya que es donde tenemos configurado las rutas fundamentales de la aplicación, manejo de errores y la lógica principal

1. **Comandos y manejadores**

```
@app.cli.command()
def test():
    tests = unittest.TestLoader().discover('tests')
    unittest.TextTestRunner().run(tests)
```

Aquí en el programa main, contamos con el comando CLI(test) el cual nos va a permitir ejecutar pruebas automatizadas.

```
@app.errorhandler(404)
def not_found(error):
        return render_template('404.html', error=error)

@app.errorhandler(500)
def server_error(error):
        return render_template('500.html', error=error)
```

Además, los manejadores de errores como el 404 y el 500, que lo que van a hacer es renderizar paginas HTML personalizadas para errores.

2. Endpoints

a) /

En este punto realizamos una solicitud GET, obtenemos la IP del usuario y lo redirigimos a /hello. Es importante establecer un contexto inicial para el usuario.

b) /hello

Realizamos solicitudes GET, que van a recuperar las tareas del usuario que es autenticado (**get_todos**) y luego se renderiza la plantilla **hello.html** con la tareas y formularios.

En las solicitudes POST, se valida el formulario de nueva tarea es decir **TodoForm**. Se agrega una tarea a Firestone (**put_todo**) y redirige. Su importancia es el punto central para gestionar tareas.

c) /todos/delete/<todo_id>

Su acción es de eliminar una tarea especifica usando **delate_todo**, lo cual es muy importante porque gestiona directamente la eliminación.

d) /todos/update/<todo_id>

Realiza solicitudes GET Y POST, el **get** busca y muestra una tarea específica para edición y el **post** actualiza la tarea con **update_todo**. También es muy importante porque nos permite a los usuarios modificar tareas existentes.

e) /api/data

Realiza solicitudes **POST** y procesa datos JSON enviados desde una API externa. Lo cual es muy importante porque expone la funcionalidad del Backend a otros sistemas.

Ahora continuamos con la importancia del programa **firestore_service.py** que contienen las funciones CRUD que interactúan directamente con Firestore.

1. **Clase Todo**

Define propiedades como descripción, estado, email, compañía, etc. También incluye método como: **to_dict** que convierte objetos a diccionarios para guardar en Firestore. Está el método **from_firestore** que es para instanciar objetos desde documentos Firestore. Y su importancia es que nos facilita el manejo estructurado de las tareas.

2. **Funciones CRUD**
    a. **get_users**: Nos va a recuperar todos los usuarios desde la colección **users**.
    b. **get_todos**: Va a obtener todas las tareas asociadas a un usuario.
    c. **put_todo**: Agrega una nueva tarea a Firestore.
    d. **update_todo**: Actualiza los campos específicos de una tarea.
    e. **delete_todo**: va a eliminar una tarea especifica.

Programa views

Este programa lo que va a hacer es manejar las rutas para el inicio de sesión. Registro, logout y tareas.

1. **Rutas /login**

Posee métodos HTTP como el GET Y POST

GET: va a renderizar la página de inicio de sesión con un formulario (login_form)

POST: En POST, valida credenciales y autentica usuarios existentes. Si las credenciales no coinciden o el usuario no existe muestra un mensaje con flash y redirige a la página de inicio. Su importancia es que nos proporciona acceso seguro a usuarios registrados mediante validación.

2. **/signup**

También cuenta con métodos GET y POST, en get se va a renderizar el formulario de registro (signup_form) y en post se va a validar el formulario y se registra un nuevo usuario si no existe. Esto nos proporciona un registro seguro.

3. **Logout**

Utiliza el método GET para cerrar la sesión del usuario actual como **logout_user** y muestra un mensaje de flash y va a redirigir a la pagina de inicio de sesión. Esto nos va a garantizar que los usuarios puedan cerrar sesión de forma segura.

4. **Tasks**

Aquí contaremos con el método GET que va a verificar que el usuario este autenticado, obtendrá tarea del usuario actual desde Firestore con **get_todos**. Si el encabezado de la solicitud incluye Accept: application/json, devuelve las

tareas en formato JSON, si no de lo contrario va a renderizar la plantilla **task.html** con las tareas. Esto es de gran importancia porque nos va a ofrecer acceso tanto para el frontend como para clientes API que consumen datos.

Interacciones Generales

1. **Conexión Frontend-Backend**

El frontend va a envía datos a través de formularios o solicitudes API y el backend procesa las solicitudes y responde con datos (JSON/HTML) o acciones (redirecciones).

2. **Gestión de Seguridad**:

Flask-Login garantiza que solo usuarios autenticados puedan acceder a rutas protegidas, así podremos tenerlas contraseñas seguras gracias a generate_password_hash.

3. **Persistencia**:

Firestore va a almacena datos de usuarios y tareas, como funciones como las CRUD van a gestionar la persistencia de forma modular.

Flujo de Datos

1. **Inicio de Sesión**

El Usuario envía credenciales a /login y el backend va a validar con get_user y autenticar con login_user.

2. **Registro**

Cuando el usuario envía datos a /signup el backend verificara si existe y crea al usuario con user_put.

3. **Gestión de Tareas**

/hello nos va a mostrar, agrega, edita o elimina tareas según la acción solicitada.

4. **Interacción API**

api/data recibirá los datos en JSON y realiza acciones específicas.

Importancia Global del Sistema

1. **Modularidad**

Se realizará la separación de vistas (views), lógica de negocio (main.py) y persistencia (firestore_service.py).

2. **Seguridad**

Usamos Flask-Login para la autenticación y las contraseñas hasheadas para proteger datos sensibles.

3. **Escalabilidad**:

Firestore va a permitir gestionar grandes volúmenes de datos de manera eficiente.

4. **Integración API**

Puntos de entrada API (/api/data) amplían las posibilidades de integración.

Facilidad de uso para el usuario

- ✓ Estructura Clara y Organizada: El manual está dividido en secciones bien definidas, lo que facilita la navegación y la comprensión del contenido.
- ✓ Las instrucciones, especialmente en las secciones que tratan sobre la interacción con los formularios HTML, son claras y fáciles de seguir.
- ✓ La descripción de la estructura del frontend (formularios HTML, enlaces, imágenes) permite a los usuarios comprender la disposición visual de la aplicación.