



PROYECTO SGE

2ª EVALUACIÓN

CFGS Desarrollo de Aplicaciones Multiplataforma
Informática y Comunicaciones

< Gestor de Películas >

Año: 2025

Fecha de presentación: 11 de febrero de 2025

Nombre y Apellidos: Ángel Chicote Veganzones

Email: angel.chiveg@educa.jcyl.es

ÍNDICE:

Introducción	4
Estado del arte	4
Definición de Arquitectura de Microservicios.....	4
Definición de API.....	4
Estructura de una API.....	4
Formas de Crear una API en python	5
Flask.....	5
FastApi.....	5
Descripción general del proyecto.....	6
Objetivos	6
Entorno de trabajo	6
Documentación técnica:	7
Análisis del sistema	7
Funcionalidades de la api.....	7
Funcionalidades de la Aplicación En Flutter	8
Diseño de la base de datos	8
Implementación	9
Estructura del Proyecto.....	10
Principales Librerías Utilizadas.....	18
Pruebas.....	18
Pruebas unitarias en la api.....	18
Pruebas de integración en el cliente	19
Despliegue de la aplicación.....	20
Manual de usuario	21
Inicio de sesión y registro.....	21
Pantallas Principales.....	21
Navegación inferior:	21
Navegación Lateral	21
Pantalla de Películas.....	22
Formulario Crear películas	23
Detalles Película	25
Pantalla de Listas.....	26
Detalles de una lista	28
Pantalla de Perfil	31
Pantalla de Configuración	32
Manual de instalación:.....	33

Conclusiones y posibles ampliaciones	33
Conclusiones	33
Dificultades.....	33
Posibles ampliaciones	34
Bibliografía	34

[Repositorio de la API](#)
[Repositorio del cliente Flutter](#)

INTRODUCCIÓN

El proyecto consiste en el desarrollo de una aplicación que permite gestionar películas y organizarlas en listas. La funcionalidad principal se centra en la realización de operaciones CRUD (Crear, Leer, Actualizar y Eliminar) tanto para las películas como para las listas. El sistema está compuesto por una API desarrollada en Python utilizando el framework FastAPI y un cliente multiplataforma desarrollado en Flutter. La API utiliza Pydantic para la validación y definición de modelos de datos, SQLAlchemy para la interacción con una base de datos SQLite y pytest para la realización de pruebas unitarias.

La aplicación cuenta con un sistema de autenticación basado en JWT (JSON Web Tokens), lo que garantiza que todas las operaciones protegidas (como la creación o modificación de películas y listas) sean accesibles únicamente para usuarios autenticados. El cliente en Flutter, desarrollado en IntelliJ, utiliza la API cuando se ejecuta en la plataforma web; en las demás plataformas (Android, iOS, etc.), la aplicación se conecta a una base de datos SQLite local para mejorar la experiencia de usuario y garantizar la disponibilidad offline.

Esta memoria detalla la concepción, el diseño, la implementación, las pruebas y el despliegue de la aplicación, así como la integración entre el backend y el frontend, ofreciendo una visión completa del estado actual del desarrollo y proponiendo posibles mejoras y ampliaciones futuras.

ESTADO DEL ARTE

DEFINICIÓN DE ARQUITECTURA DE MICROSERVICIOS

La **arquitectura de microservicios** es un estilo de diseño de software en el que las aplicaciones se estructuran como una colección de pequeños servicios independientes, cada uno ejecutando un proceso único y comunicándose mediante protocolos ligeros (usualmente HTTP/REST o mensajes). Esta aproximación contrasta con la arquitectura monolítica tradicional, ya que permite un mayor desacoplamiento, escalabilidad y flexibilidad en el desarrollo, despliegue y mantenimiento de cada componente.

En el contexto de nuestro proyecto, aunque la aplicación se puede considerar de tamaño moderado, se ha seguido un diseño modular que facilita la separación de responsabilidades (por ejemplo, la autenticación, la gestión de películas y la gestión de listas) y permite la escalabilidad futura si se decidiera migrar hacia un enfoque de microservicios en un entorno de producción.

DEFINICIÓN DE API

Una **API (Application Programming Interface)** es un conjunto de reglas y definiciones que permiten la comunicación entre diferentes componentes de software. En el ámbito del desarrollo web, una API expone endpoints que pueden ser consumidos por clientes (como aplicaciones móviles o web) para realizar operaciones específicas (por ejemplo, obtener datos o enviar información).

ESTRUCTURA DE UNA API

Protocolo y Métodos Utilizados

La API desarrollada se basa en el protocolo **HTTP** y utiliza métodos estándar como:

- **GET:** Para la obtención de recursos (por ejemplo, obtener una lista de películas).
- **POST:** Para la creación de nuevos recursos (por ejemplo, registrar un usuario o agregar una película).
- **PUT/PATCH:** Para la actualización de recursos existentes.
- **DELETE:** Para la eliminación de recursos.

Cada endpoint sigue una estructura clara y semántica, facilitando la comprensión y el consumo de la API por parte de los clientes.

Partes de las URL en una API

Una URL de una API generalmente se compone de:

- **Dominio/Base URL:** Ejemplo: `https://api.miaplicacion.com/`
- **Versión de la API:** Ejemplo: `/v1/`
- **Recursos:** Ejemplo: `/peliculas/`, `/listas/`
- **Parámetros y/o identificadores:** Ejemplo: `/peliculas/{id}`, donde `{id}` es el identificador único de la película.

Esta estructura facilita la identificación de la versión y la gestión de rutas en el servidor, permitiendo escalabilidad y facilidad de mantenimiento.

FORMAS DE CREAR UNA API EN PYTHON

En el ecosistema Python existen múltiples frameworks para el desarrollo de APIs. Los dos más destacados son **Flask** y **FastAPI**.

FLASK

Flask es un microframework ligero que permite el desarrollo rápido de aplicaciones web y APIs. Su principal ventaja reside en su simplicidad y la amplia comunidad que lo respalda. Sin embargo, en aplicaciones que requieren un rendimiento elevado y una validación estricta de datos, Flask puede requerir la integración de librerías adicionales, lo que en ocasiones aumenta la complejidad del código.

FASTAPI

FastAPI es un framework moderno y de alto rendimiento para la creación de APIs con Python. Se basa en **Starlette** para el manejo asíncrono y en **Pydantic** para la validación de datos, lo que permite la definición de modelos de datos de forma declarativa y robusta. Entre sus ventajas destacan:

- **Velocidad de desarrollo:** Gracias a la generación automática de documentación (Swagger y Redoc).
- **Alto rendimiento:** Comparable al de Node.js y Go.
- **Facilidad de uso:** Tipado estático y validación de datos integrados.

Para este proyecto se ha optado por **FastAPI** debido a su alto rendimiento, facilidad de integración con librerías modernas y la generación automática de documentación, lo que facilita el mantenimiento y la escalabilidad de la API.

DESCRIPCIÓN GENERAL DEL PROYECTO

OBJETIVOS

El objetivo principal del proyecto es desarrollar una aplicación robusta y escalable para la gestión de películas y listas de reproducción, que cumpla con los siguientes requerimientos:

- **Gestión de películas:** Permitir la creación, lectura, actualización y eliminación (CRUD) de registros de películas.
- **Organización en listas:** Facilitar la agrupación de películas en listas personalizadas, también con operaciones CRUD.
- **Autenticación y seguridad:** Implementar un sistema de registro e inicio de sesión que utilice JWT para proteger los endpoints sensibles de la API.
- **Interacción multiplataforma:** Desarrollar un cliente en Flutter que se conecte a la API cuando se ejecute en web, mientras que en dispositivos móviles la aplicación se apoya en una base de datos local (SQLite) para garantizar el funcionamiento offline.

Estos objetivos pretenden ofrecer una solución completa que no solo facilite la gestión de contenidos, sino que también asegure una experiencia de usuario segura y eficiente, adaptándose a distintos entornos de ejecución.

ENTORNO DE TRABAJO

Para la realización del proyecto se han utilizado diversas herramientas y tecnologías:

- **Lenguaje de Programación:** Python
 - Se ha utilizado Python por su sencillez, versatilidad y amplia disponibilidad de librerías modernas.
- **Framework Backend:** FastAPI
 - Permite el desarrollo rápido de APIs de alto rendimiento, con validación automática de datos y documentación integrada.
- **Validación de Datos:** Pydantic
 - Utilizado para definir y validar los modelos de datos, garantizando la integridad y coherencia de la información.
- **ORM (Object Relational Mapping):** SQLAlchemy
 - Facilita la interacción con la base de datos SQLite mediante un mapeo objeto-relacional, simplificando las operaciones CRUD.

- **Base de Datos:** SQLite
 - Base de datos ligera y de fácil configuración, ideal para el desarrollo y pruebas de aplicaciones de tamaño moderado.
- **Pruebas:** pytest
 - Se han implementado pruebas unitarias y de integración para asegurar la calidad y robustez del código.
- **IDE para Backend:** PyCharm
 - Entorno de desarrollo integrado que facilita la codificación, depuración y gestión del proyecto.
- **Cliente Multiplataforma:** Flutter
 - Framework para el desarrollo de aplicaciones móviles y web. Se ha utilizado IntelliJ para el desarrollo del cliente Flutter.
- **Control de Versiones:** Git
 - Se ha utilizado Git para el control de versiones y la gestión colaborativa del código.
- **Autenticación:** JWT
 - Implementación de autenticación basada en tokens JSON Web Tokens, que garantiza la seguridad de las operaciones sensibles.

En conjunto, estas herramientas permiten un desarrollo ágil, modular y escalable, asegurando tanto la calidad del software como la facilidad de mantenimiento a largo plazo.

DOCUMENTACIÓN TÉCNICA:

ANÁLISIS DEL SISTEMA

El sistema se compone de dos grandes módulos: la **API** y el **Cliente**. A continuación, se detallan las funcionalidades principales de cada uno:

FUNCIONALIDADES DE LA API

Gestión de Usuarios:

- Registro de nuevos usuarios.
- Inicio de sesión mediante autenticación JWT.
- Recuperación y gestión de datos de usuario.

Operaciones CRUD para Películas:

- Creación de registros de películas con atributos como título, director, descripción, año, etc.
- Obtención de la lista de películas disponibles.

- Actualización de información de películas existentes.
- Eliminación de registros de películas.

Operaciones CRUD para Listas:

- Creación de listas personalizadas.
- Asociación de películas a listas.
- Visualización y edición de listas.
- Eliminación de listas o películas que contiene.

Manejo de Errores y Excepciones:

- Respuestas estandarizadas para errores (por ejemplo, autenticación fallida, recurso no encontrado, etc.).

FUNCIONALIDADES DE LA APLICACIÓN EN FLUTTER

Interfaz de Usuario Amigable:

- Diseño responsive para web y aplicaciones móviles.
- Navegación intuitiva entre secciones (Películas, listas, perfil, detalles de películas...).

Integración con la API:

- Consumo de endpoints para la gestión de películas y listas cuando se ejecuta en web.

Seguridad y Autenticación:

- Pantallas de registro e inicio de sesión.
- Gestión del token JWT para el consumo seguro de la API.

Pruebas de Integración:

- Verificación del correcto funcionamiento de la comunicación entre la aplicación y la API.

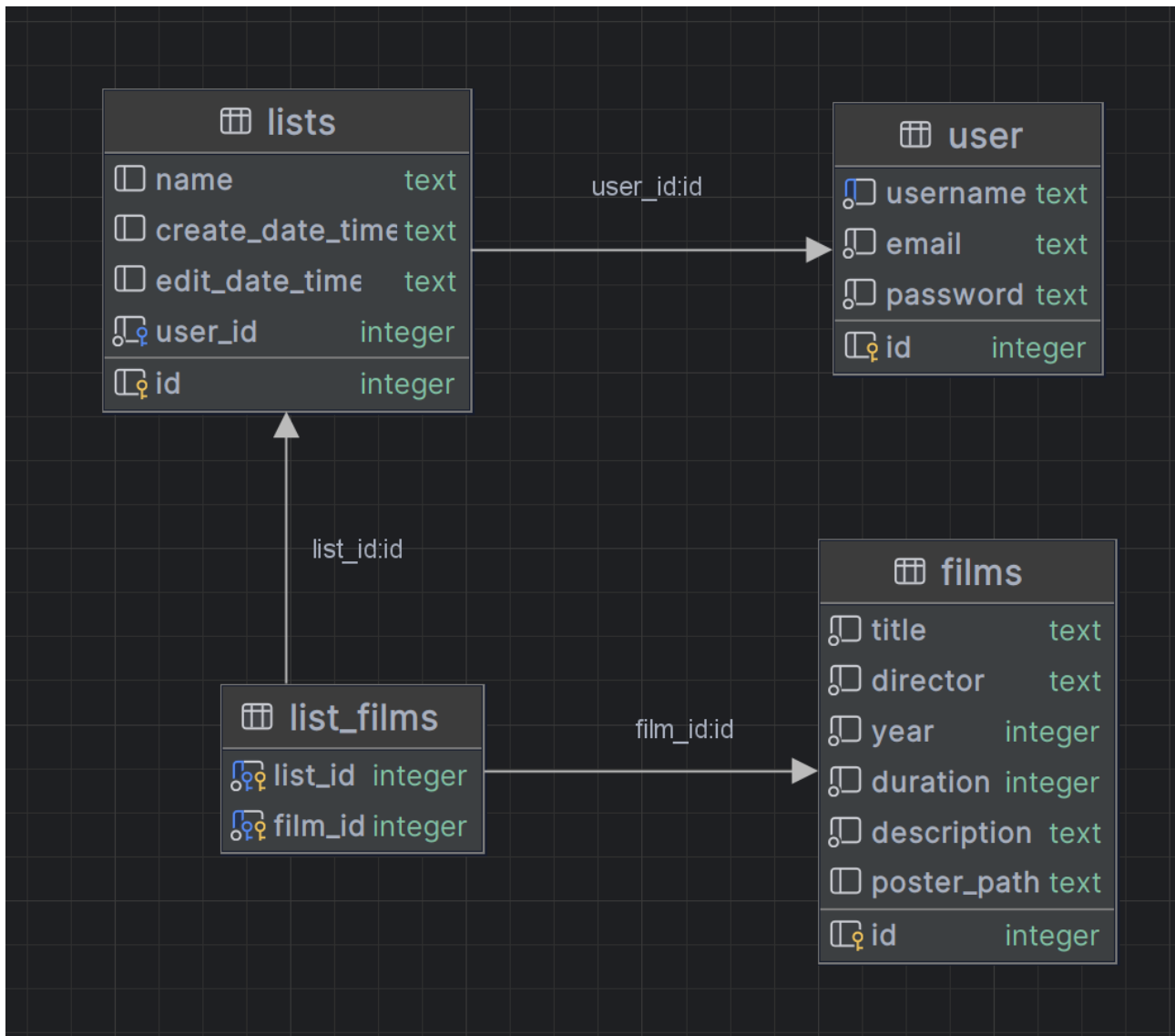
DISEÑO DE LA BASE DE DATOS

El diseño de la base de datos se ha centrado en la simplicidad y eficiencia, utilizando SQLite. Se han definido las siguientes tablas principales:

Tablas Principales

- **Usuarios:**
 - **Campos:** id (PK), nombre de usuario, email, contraseña (hash).
 - **Relaciones:** Cada usuario puede tener asociadas múltiples listas.
- **Películas:**
 - **Campos:** id (PK), título, director, descripción, año, duración, etc.
 - **Relaciones:** Las películas pueden estar asociadas a una o varias listas.

- **Listas:**
 - **Campos:** id (PK), nombre de la lista, fecha de creación y edición, id_usuario (FK).
 - **Relaciones:** Cada lista pertenece a un usuario y puede contener múltiples películas.
- **Películas-Listas (Tabla Intermedia):**
 - **Campos:** id_película (FK), id_lista (FK).
 - **Función:** Permite la relación de muchos a muchos entre películas y listas



IMPLEMENTACIÓN

La implementación del sistema se ha estructurado en distintos módulos que facilitan el mantenimiento y la escalabilidad. A continuación, se describe la organización general del código:

ESTRUCTURA DEL PROYECTO

API

Raíz del proyecto:

- **.env**

Archivo que contiene variables de entorno, como la clave secreta para JWT y otras configuraciones sensibles.

```
SECRET_KEY=clave_secreta
ALGORITHM=HS256
ACCESS_TOKEN_EXPIRE_MINUTES=30
```

- **auth.py**

Agrupar las funciones y utilidades para la autenticación, como la generación y validación de tokens JWT.

```
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="/token/")

pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

load_dotenv()
SECRET_KEY = os.getenv("SECRET_KEY", "clave_por_defecto")
ALGORITHM = os.getenv("ALGORITHM", "HS256")
ACCESS_TOKEN_EXPIRE_MINUTES = int(os.getenv("ACCESS_TOKEN_EXPIRE_MINUTES", 30))

def create_access_token(data: dict, expires_delta: timedelta | None = None):
    """Genera un token JWT con los datos proporcionados."""
    to_encode = data.copy()
    expire = datetime.now(UTC) + (expires_delta or timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES))
    to_encode.update({"exp": expire})
    return encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)

def verify_token(token: str):
    """Verifica la validez del token y devuelve su contenido."""
    try:
        payload = decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        return payload
    except ExpiredSignatureError:
        # El token ha expirado
        return None
    except PyJWTError:
        # Otro error relacionado con el token
        return None

def hash_password(plain_password: str) -> str:
    return pwd_context.hash(plain_password)

# Test
if __name__ == '__main__':
    # Crear un token
    data_test = {"sub": "user123"}
    token_test = create_access_token(data_test)
    print("Token:", token_test)
```

```
# Verificar un token
decoded_data = verify_token(token_test)
print("Decoded data:", decoded_data)
```

- **database.py**

Configura la conexión a la base de datos SQLite y expone el método o función para obtener dicha conexión.

```
DATABASE_URL = "sqlite:///./films.db"

engine = create_engine(DATABASE_URL, connect_args={"check_same_thread": False})
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()

def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

- **films.db**

Archivo de la base de datos SQLite que almacena la información de la aplicación.

- **main.py**

Es el punto de entrada de la API. Aquí se inicializa FastAPI, se incluyen las rutas definidas en los routers y se configura el servidor.

```
app = FastAPI(title="API Gestión de películas", description="API para organizar en listas tus películas")

app.include_router(users_routes.router)
app.include_router(film_routes.router)
app.include_router(list_routes.router)
app.include_router(token_routes.router)

# Crear las tablas en la base de datos cogiendo las definiciones de models.
Base.metadata.create_all(bind=engine)

# Para que funcione con el cliente de flutter web:
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"], # Cambiar esto a ["http://localhost:3000"] si solo es para desarrollo ¿?
    allow_credentials=True,
    allow_methods=["*"], # Permite todos los métodos (GET, POST, PUT, DELETE, OPTIONS)
    allow_headers=["*"], # Permite todos los encabezados
)

if __name__ == '__main__':
    uvicorn.run('main:app', port=8000, reload=True)

# Docs
# http://127.0.0.1:8000/docs
# http://127.0.0.1:8000/redoc
```

- **requirements.txt**

Lista de dependencias del proyecto (por ejemplo, fastapi, uvicorn, sqlalchemy, pydantic, pytest, etc.) que se instalarán mediante pip.

```
fastapi~=0.115.8
uvicorn~=0.34.0
pydantic~=2.10.4
SQLAlchemy~=2.0.38
passlib~=1.7.4
pydantic[email]
starlette~=0.45.3
python-dotenv~=1.0.1
pyjwt
passlib[bcrypt]
jose~=1.0.0
pytest
pytest-asyncio
```

Módulo app

- **models**

Contiene los modelos de datos definidos con SQLAlchemy. Cada archivo puede representar una entidad (por ejemplo, películas, usuarios, listas).

```
class UserModel(Base):
    """
    Definición del modelo de usuario que va a representar una tabla en la base
    de datos.
    """
    __tablename__ = "users"

    id = Column(Integer, primary_key=True, index=True)
    username = Column(String, unique=True, nullable=False)
    email = Column(String, nullable=False)
    password = Column(String, nullable=False)

    # Relaciones
    # El primer argumento significa el modelo con el que se representa la rela-
    # ción.
    # El segundo argumento representa el atributo de la otra clase que indica la
    # relación
    # El tercer argumento indica que si se elimina un usuario se eliminan todas
    # las listas asociadas.
    lists = relationship("ListModel", back_populates="user", cascade="all, de-
    lete-orphan")
```

```
class FilmModel(Base):
    """
    Definición del modelo de película que va a representar una tabla en la base
    de datos.
    """
    __tablename__ = "films"

    id = Column(Integer, primary_key=True, index=True) # por defecto es autoin-
    crement
    title = Column(String, nullable=False)
    director = Column(String, nullable=False)
    year = Column(Integer, nullable=False)
    duration = Column(Integer, nullable=False)
```

```

description = Column(String, nullable=False)
poster_path = Column(String, nullable=False)

# Relaciones
lists = relationship("ListModel", secondary=list_films_table, back_populates="films")

```

```

class ListModel(Base):
    """
    Definición del modelo de una lista que va a representar una tabla en la base de datos.
    """
    __tablename__ = "lists"

    id = Column(Integer, primary_key=True, index=True) # por defecto es autoincrement
    name = Column(String, nullable=False)
    create_date_time = Column(String, nullable=False)
    edit_date_time = Column(String, nullable=False)

    # Relaciones
    # Representa la columna de la base de datos para la clave foránea.
    # En la ForeignKey hay que indicar la columna de la base de datos con la que se relaciona, importante poner
    # el mismo nombre de tabla que el indicado en el modelo. En este caso users en minúscula y plural.
    user_id = Column(Integer, ForeignKey("users.id"), nullable=False)

    # Representa un objeto de la clase indicada, en este caso UserModel, que construye la relación.
    # back_populates indica que la relación es bidireccional y, por lo tanto, debe de haber un atributo lists.
    user = relationship("UserModel", back_populates="lists") # Relación con el modelo de usuario
    films = relationship("FilmModel", secondary=list_films_table, back_populates="lists")

list_films_table = Table(
    "list_films", Base.metadata,
    Column("list_id", Integer, ForeignKey("lists.id"), primary_key=True),
    Column("film_id", Integer, ForeignKey("films.id"), primary_key=True)
)

```

- **routers**

Define los endpoints de la API agrupados por funcionalidad. Cada archivo puede manejar un conjunto de rutas (por ejemplo, rutas para películas, rutas para usuarios).

No voy a mostrar todo el código porque es mucho, solo mostraré algún ejemplo.

```

router = APIRouter(prefix="/films", tags=["Películas"])

@router.get("/countAll", response_model=int)
def count_films(db: Session = Depends(get_db), token: str = Depends(oauth2_scheme)):
    if not verify_token(token):
        raise HTTPException(status_code=401, detail="Invalid token")

```

```
try:
    return film_service.count(db)
except SQLAlchemyError as e:
    raise HTTPException(status_code=500, detail=f"Error al calcular el
número de películas: {str(e)}")

# Importante! en Depends(get_db) es solo get_db no get_db().
@router.get("/", response_model=List[FilmResponse])
def get_films(db: Session = Depends(get_db), token: str = De-
pends(oauth2_scheme)):
    if not verify_token(token):
        raise HTTPException(status_code=401, detail="Invalid token")

    try:
        return film_service.get_films(db)
    except SQLAlchemyError as e:
        db.rollback()
        raise HTTPException(status_code=500, detail=f"Error al obtener las
películas: {str(e)}")

@router.get("/{film_id}", response_model=FilmResponse | None)
def find_by_id(film_id: int, db: Session = Depends(get_db), token: str = De-
pends(oauth2_scheme)):
    if not verify_token(token):
        raise HTTPException(status_code=401, detail="Invalid token")

    try:
        return film_service.find_by_id(film_id, db)
    except SQLAlchemyError as e:
        db.rollback()
        raise HTTPException(status_code=500, detail=f"Error al obtener las
películas: {str(e)}")

@router.post("/create", response_model=int)
def create_film(film: FilmCreate, db: Session = Depends(get_db), token: str =
Depends(oauth2_scheme)):
    if not verify_token(token):
        raise HTTPException(status_code=401, detail="Invalid token")

    try:
        return film_service.create_film(film, db)
    except SQLAlchemyError as e:
        db.rollback()
        raise HTTPException(status_code=500, detail=f"Error al crear la pelí-
cula: {str(e)}")

@router.post("/update", response_model=bool)
def update_film(film: FilmUpdate, db: Session = Depends(get_db), token: str =
Depends(oauth2_scheme)):
    if not verify_token(token):
        raise HTTPException(status_code=401, detail="Invalid token")

    try:
        return film_service.update_film(film, db) is not None
    except SQLAlchemyError as e:
        db.rollback()
        raise HTTPException(status_code=500, detail=f"Error al actualizar la pe-
lícula: {str(e)}")
```

```

@router.delete("/{film_id}", response_model=bool)
def delete_film(film_id: int, db: Session = Depends(get_db), token: str = Depends(oauth2_scheme)):
    if not verify_token(token):
        raise HTTPException(status_code=401, detail="Invalid token")

    try:
        return film_service.delete_film(film_id, db)
    except SQLAlchemyError as e:
        db.rollback()
        raise HTTPException(status_code=500, detail=f"Error al actualizar la película: {str(e)}")

@router.get("/of_list/{list_id}", response_model=List[FilmResponse])
def find_all_by_list_id(list_id: int, db: Session = Depends(get_db), token: str = Depends(oauth2_scheme)):
    if not verify_token(token):
        raise HTTPException(status_code=401, detail="Invalid token")

    try:
        return film_service.find_all_by_list_id(list_id, db)
    except SQLAlchemyError as e:
        db.rollback()
        raise HTTPException(status_code=500, detail=f"Error al obtener las películas de la lista: {str(e)}")

```

Al igual que con el ejemplo anterior hay otros tres ficheros routes:

1. list_routes: con los endpoints de las listas.
 2. User_routes: con los endpoints de los usuarios.
 3. token_routes: con un endpoint necesario para autenticarse en Swagger y poder realizar pruebas.
- **schemas**
 Contiene los esquemas de datos definidos con Pydantic, que se usan para la validación y serialización de la información que ingresa o sale de la API.

```

class FilmCreate(BaseModel):
    """
    Representa los datos que se envían en la petición para crear una nueva película.
    """
    title: str
    director: str
    year: int
    duration: int
    description: str
    poster_path: str

class FilmUpdate(BaseModel):
    """
    Representa los datos que se envían en la petición para editar una película.
    """
    id: int
    title: str
    director: str

```

```

    year: int
    duration: int
    description: str
    poster_path: str

class FilmResponse(BaseModel):
    """
    Representa los datos enviados como respuesta de una petición a la api
    """
    id: int
    title: str
    director: str
    year: int
    duration: int
    description: str
    poster_path: str

    # Es necesario para trabajar con pydantic y SQLAlchemy
    # Esto indica a pydantic que al instanciar la clase se van a usar atributos
    # de la clase modelo en lugar
    # de un diccionario
    class Config:
        from_attributes = True

```

Al igual que con el ejemplo anterior hay otros tres ficheros schemas:

1. list_schemas: con las definiciones de clases para las listas.
 2. User_routes: con las definiciones de clases para los usuarios.
 3. token_routes: con la definición necesaria para representar los datos recibidos por **OAuth2PasswordBearer**.
- **services**
Implementa la lógica de negocio y las operaciones CRUD, utilizando SQLAlchemy para interactuar con la base de datos.

```

def get_films(db: Session):
    return db.query(FilmModel).all()

def find_by_id(film_id: int, db: Session):
    return db.query(FilmModel).filter(FilmModel.id == film_id).first()

def create_film(film: FilmCreate, db: Session):
    db_film = FilmModel(**film.model_dump())
    db.add(db_film)
    db.commit()
    db.refresh(db_film)
    return db_film.id

def update_film(film: FilmUpdate, db: Session):
    db_film = find_by_id(film.id, db)
    if db_film:
        # Editar película

```



```
        db_film.title = film.title
        db_film.director = film.director
        db_film.year = film.year
        db_film.duration = film.duration
        db_film.description = film.description
        db_film.poster_path = film.poster_path

        # Guardar cambios
        db.commit()
        db.refresh(db_film)

    return db_film

def delete_film(film_id: int, db: Session):
    db_user = find_by_id(film_id, db)
    if db_user:
        # Eliminar película
        db.delete(db_user)
        db.commit()
        return True

    return False

def count(db: Session):
    return db.query(FilmModel).count()

def find_all_by_list_id(list_id: int, db: Session):
    # Obtener lista:
    db_list = db.query(ListModel).filter(ListModel.id == list_id).first()
    print(db_list)
    # Si existe, devolver sus películas.
    if db_list:
        return db_list.films
    return None
```

Al igual que con el ejemplo anterior hay otros dos ficheros services:

1. list_service: con las funciones CRUD de las listas.
2. user_service: con las funciones CRUD de los usuarios.

Tests

- **test/film_service_test.py**
Archivo que contiene las pruebas (unitarias o de integración) para validar el funcionamiento del servicio de películas.

CLIENTE EN FLUTTER

El cliente de flutter es demasiado complejo como para explicarlo todo aquí, ya que es el proyecto de la asignatura de Diseño de Interfaces.

PRINCIPALES LIBRERÍAS UTILIZADAS

- **FastAPI:** Para el desarrollo del API REST.
- **Pydantic:** Para la validación y manejo de datos.
- **SQLAlchemy:** Para la interacción con la base de datos relacional.
- **pytest:** Para la implementación y ejecución de pruebas.
- **uvicorn:** Servidor ASGI para ejecutar la aplicación en desarrollo y producción

PRUEBAS

La realización de pruebas es fundamental para garantizar la calidad y fiabilidad del sistema. Se han diseñado pruebas unitarias y de integración:

PRUEBAS UNITARIAS EN LA API

```
# Configurar base de datos de prueba en memoria
SQLALCHEMY_DATABASE_URL = "sqlite:///memory:"
engine = create_engine(SQLALCHEMY_DATABASE_URL, connect_args={"check_same_thread": False})
TestingSessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

@pytest.fixture(scope="function")
def db_session():
    """Crea una sesión de prueba con una base de datos en memoria"""
    Base.metadata.create_all(bind=engine)
    db = TestingSessionLocal()
    try:
        yield db
    finally:
        db.close()
        # Borrar la base de datos después de los tests.
        Base.metadata.drop_all(bind=engine)

def test_create_film(db_session):
    """Prueba la creación de una película"""
    new_film = FilmCreate(
        title="Interstellar",
        director="Christopher Nolan",
        year=2014,
        duration=169,
        description="Ciencia ficción sobre viajes espaciales.",
        poster_path="/path/to/poster.jpg"
    )

    film_id = film_service.create_film(new_film, db_session)
    assert film_id is not None

    film = db_session.query(FilmModel).filter(FilmModel.id == film_id).first()
    assert film is not None
    assert film.title == "Interstellar"

def test_get_films(db_session):
```

```

    """Prueba obtener películas desde la base de datos"""
    film1 = FilmModel(title="Inception", director="Christopher Nolan",
year=2010, duration=148,
                        description="Sueños dentro de sueños.",
poster_path="/path/inception.jpg")
    film2 = FilmModel(title="The Matrix", director="Wachowski", year=1999, dura-
tion=136,
                        description="Simulación y realidad.",
poster_path="/path/matrix.jpg")

    db_session.add(film1)
    db_session.add(film2)
    db_session.commit()

    films = film_service.get_films(db_session)
    assert len(films) == 2
    assert films[0].title == "Inception"
    assert films[1].title == "The Matrix"

if __name__ == "__main__":
    pytest.main(["-v", "--tb=short"])

```

PRUEBAS DE INTEGRACIÓN EN EL CLIENTE

Se encuentra en el directorio “test” de la raíz del proyecto de flutter.

```

Future<void> main() async {
  late FilmApiService filmService;
  final film = Film(
    id: null,
    title: 'Inception',
    director: 'Christopher Nolan',
    year: 2010,
    duration: 148,
    description: 'A mind-bending thriller',
    posterPath: 'https://placeholder.co/900x1600/png',
  );

  // NO se si va a ser necesario qu el token no sea nulo.
  final user = await UserApiService().login('angel', '12345678');
  final token = user?.token;

  setUp(() async {
    filmService = FilmApiService();
  });

  test('Debe insertar y recuperar una película', () async {
    film.id = await filmService.insert(token, film);
    expect(film.id, isNotNull);

    final Film? retrievedFilm = await filmService.findById(token, film.id!);
    expect(retrievedFilm, isNotNull);
    expect(retrievedFilm?.title, 'Inception');
  });

  test('Debe actualizar una película existente', () async {
    final updatedFilm = Film(
      id: film.id,
      title: "${film.title} (Actualizado)",
      director: film.director,
      year: film.year,
    );

```

```
        duration: film.duration,
        description: film.description,
        posterPath: film.posterPath,
    );

    final bool updated = await filmService.update(token, updatedFilm);
    expect(updated, isTrue);

    final Film? retrievedFilm = await filmService.findById(token, updated-
Film.id!);
    expect(retrievedFilm?.title, 'Inception (Actualizado)');
  });

test('Debe eliminar una película', () async {

    final bool deleted = await filmService.delete(token, film.id!);
    expect(deleted, isTrue);

    final Film? retrievedFilm = await filmService.findById(token, film.id!);
    expect(retrievedFilm, isNull);
  });
}
```

DESPLIEGUE DE LA APLICACIÓN

El despliegue del sistema se ha realizado inicialmente en un entorno local, facilitando pruebas y desarrollo iterativo. Entre los aspectos destacados del despliegue se encuentran:

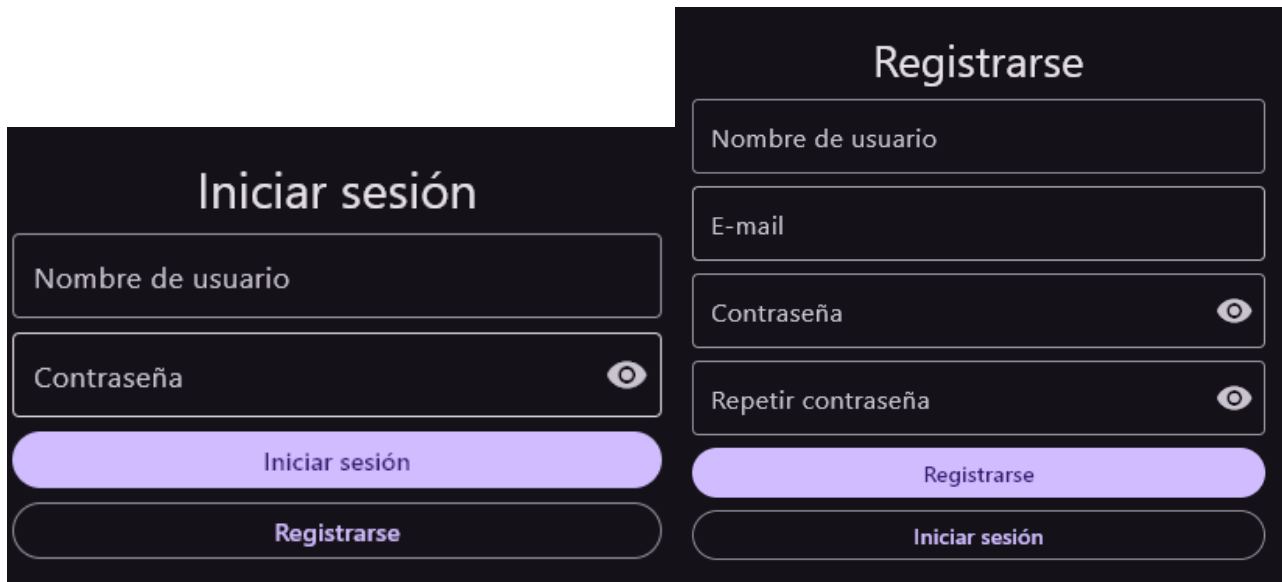
La API se ejecuta utilizando uvicorn, lo que permite levantar un servidor de desarrollo.

Se ha configurado la conexión a la base de datos SQLite, adecuada para entornos de prueba.

MANUAL DE USUARIO

INICIO DE SESIÓN Y REGISTRO

Si no tienes una cuenta puedes crear una nueva.



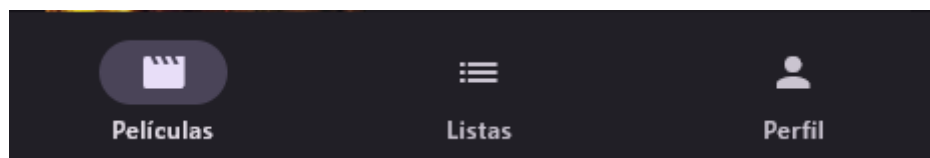
The image displays two side-by-side login and registration forms on a dark background. The left form, titled 'Iniciar sesión', features input fields for 'Nombre de usuario' and 'Contraseña' (with a toggle icon), and buttons for 'Iniciar sesión' and 'Registrarse'. The right form, titled 'Registrarse', features input fields for 'Nombre de usuario', 'E-mail', 'Contraseña' (with a toggle icon), and 'Repetir contraseña' (with a toggle icon), and buttons for 'Registrarse' and 'Iniciar sesión'.

Una vez iniciada la sesión se quedará guardada hasta que decidas manualmente cerrarla.

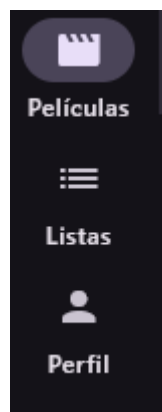
PANTALLAS PRINCIPALES

En función del dispositivo aparecerá un tipo de navegación u otro:

NAVEGACIÓN INFERIOR:




NAVEGACIÓN LATERAL




PANTALLA DE PELÍCULAS

Aparece una lista o una tabla en función del tamaño de la pantalla:




Películas




Taxi Driver

Martin Scorsese

1976 | 1 h 51 m

Para sobrellevar el insomnio crónico que sufre desde su regreso de Vietnam, Travis Bickle (Robert De Niro) trabaja como taxista nocturno en Nueva York. Es un hombre insociable que apenas tiene contacto con los demás, se pasa los días en el cine y vive prendado de Betsy (Cybill Shepherd), una atractiva rubia que trabaja como voluntaria en una campaña política. Pero lo que realmente obsesiona a Travis es comprobar cómo la violencia, la sordidez y la desolación dominan la ciudad. Y un día decide pasar a la acción.




Interstellar

Christopher Nolan

2014 | 2 h 55 m

Al ver que la vida en la Tierra está llegando a su fin, un grupo de exploradores dirigidos por el piloto Cooper (McConaughey) y la científica Amelia (Hathaway) emprende una misión que puede ser la más importante de la historia de la humanidad: viajar más allá de nuestra galaxia para descubrir algún planeta en otra que pueda garantizar el futuro de la raza humana.



Se7en

David Fincher

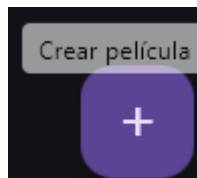
1995 | 2 h 7 m

Al ver que la vida en la Tierra está llegando a su fin, un grupo de exploradores dirigidos por el piloto Cooper (McConaughey) y la científica Amelia (Hathaway) emprende una misión que puede ser la más importante de la historia de la humanidad: viajar más allá de nuestra galaxia para descubrir algún planeta en otra que pueda garantizar el futuro de la raza humana.

+

FORMULARIO CREAR PELÍCULAS

El botón flotante de abajo a la derecha permite acceder al formulario para crear una nueva película:



← Crear película

Título

Director

Año del estreno

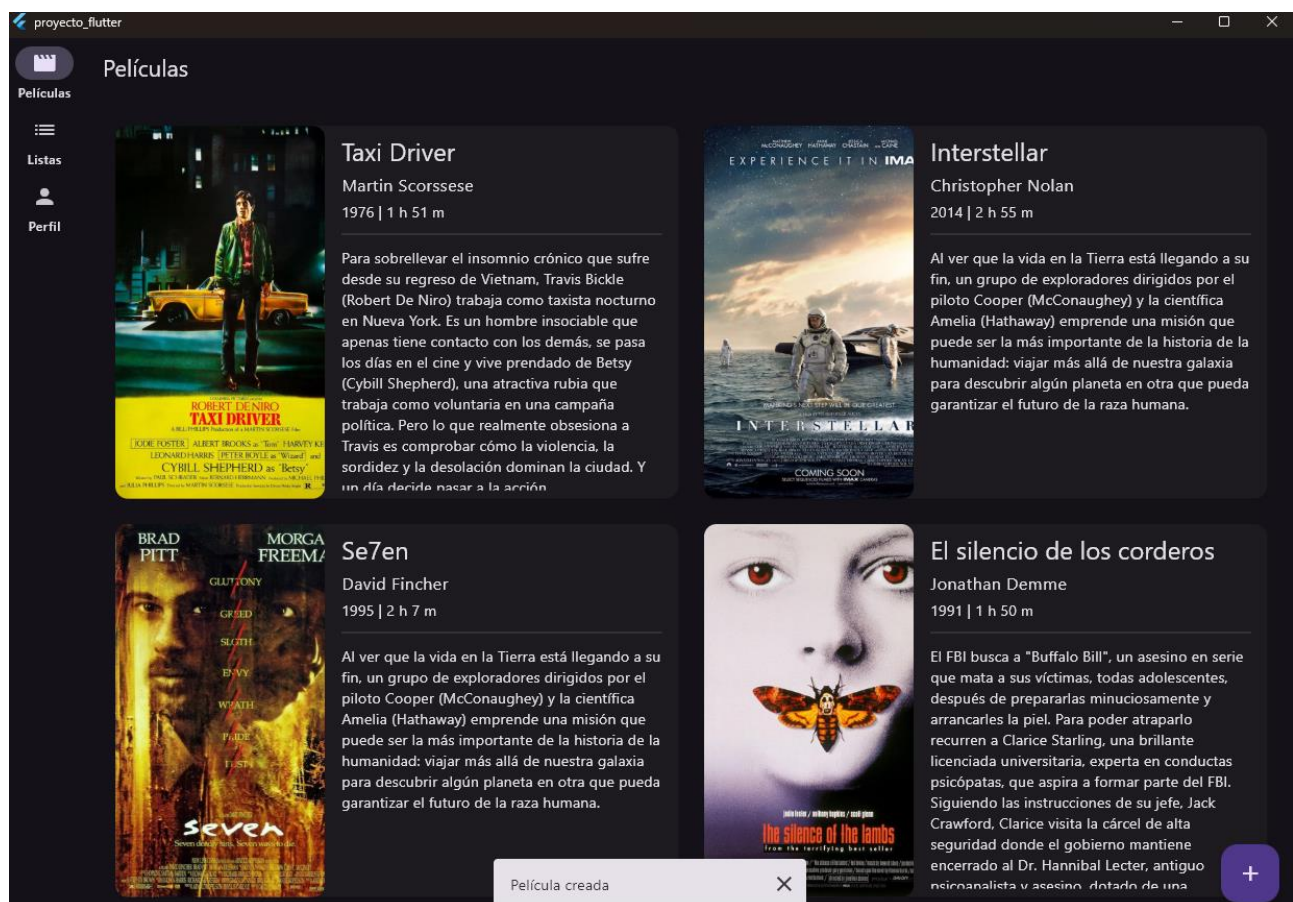
Haga click para introducir la duración

Descripción

Galería

✓

Una vez rellenado el formulario ya podemos guardar el formulario pulsando al mismo botón de antes. Solo que ahora tiene una apariencia distinta ya que sirve para confirmar la creación de la película.



DETALLES PELÍCULA



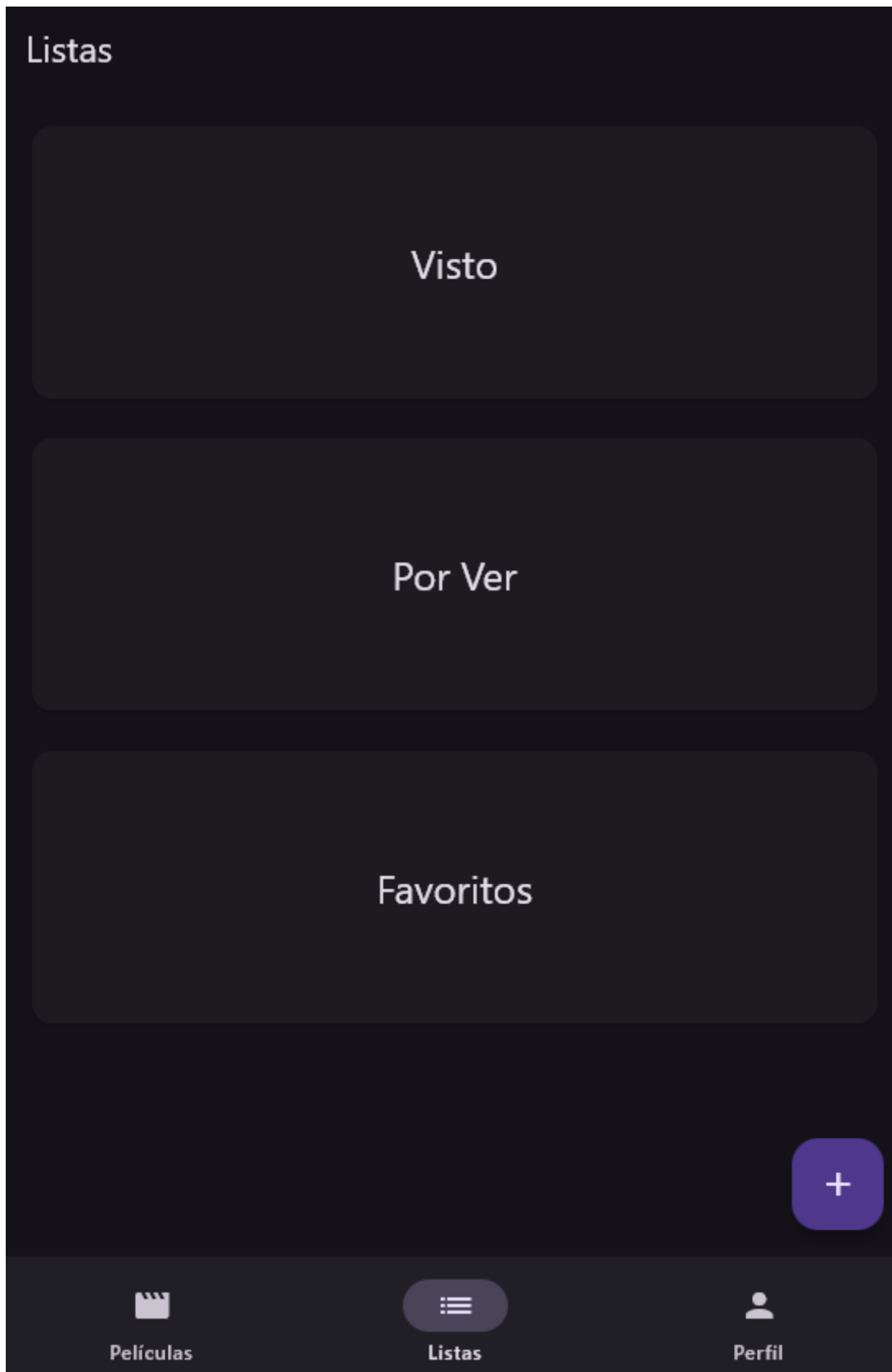
En esta pantalla se muestra toda la información de la película.

Arriba encontraremos dos botones:



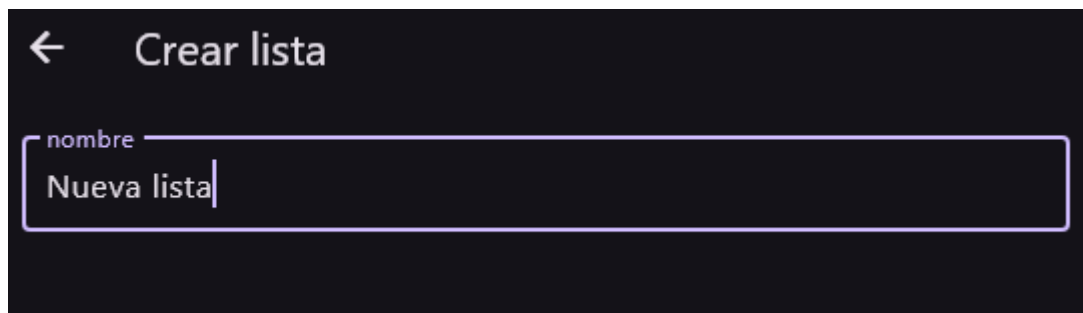
Para editar y eliminar la película.

PANTALLA DE LISTAS



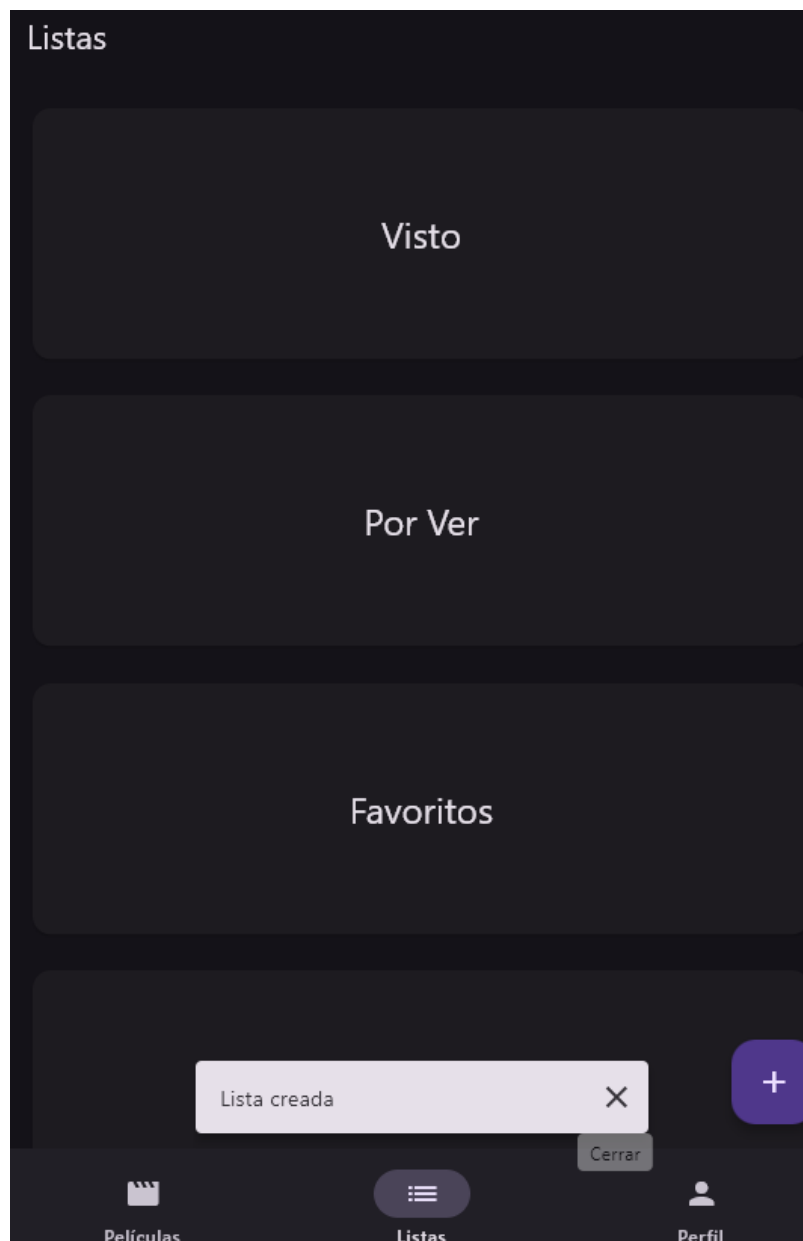
En esta sección se muestran nuestras listas.

Al igual que en la pantalla películas hay un botón abajo a la derecha que lleva al formulario para crear una nueva lista:



← Crear lista

nombre Nueva lista



DETALLES DE UNA LISTA



Si entramos en una lista podemos ver las películas que contiene.

En la parte superior encontramos al igual que antes dos botones, uno para editar la lista y otro para eliminarla.

Mas abajo esta la tarjeta de una película. También cuenta con un botón para eliminarla de la lista.

Abajo a la derecha está el botón para añadir una película a la lista.

Se7en

David Fincher

1995 | 2 h 7 m

Al ver que la vida en la Tierra está llegando a su fin, un grupo de exploradores dirigidos por el piloto Cooper (McConaughey) y la científica Amelia (Hathaway) emprende una misión que puede ser la más importante de la historia de la humanidad: viajar más allá de nuestra galaxia para descubrir algún planeta en otra que pueda garantizar el futuro de la raza humana.

El silencio de los corderos

Jonathan Demme

1991 | 1 h 50 m

El FBI busca a "Buffalo Bill", un asesino en serie que mata a sus víctimas, todas adolescentes, después de prepararlas minuciosamente y arrancarles la piel. Para poder atraparlo recurren a Clarice Starling, una brillante licenciada universitaria, experta en conductas psicópatas, que aspira a formar parte del FBI. Siguiendo las instrucciones de su jefe, Jack Crawford, Clarice visita la cárcel

Películas

Listas

Perfil

 Visto  



Taxi Driver

Martin Scorsese

1976 | 1 h 51 m

Para sobrellevar el insomnio crónico que sufre desde su regreso de Vietnam, Travis Bickle (Robert De Niro) trabaja como taxista nocturno en Nueva York. Es un hombre insociable que apenas tiene contacto con los demás, se pasa los días en el cine y vive prendado de Betsy (Cybill Shepherd), una atractiva rubia que trabaja como voluntaria en una campaña política. Pero lo que realmente obsesiona a Travis es comprobar



El silencio de los corderos

Jonathan Demme

1991 | 1 h 50 m

El FBI busca a "Buffalo Bill", un asesino en serie que mata a sus víctimas, todas adolescentes, después de prepararlas minuciosamente y arrancarles la piel. Para poder atraparlo recurren a Clarice Starling, una brillante licenciada

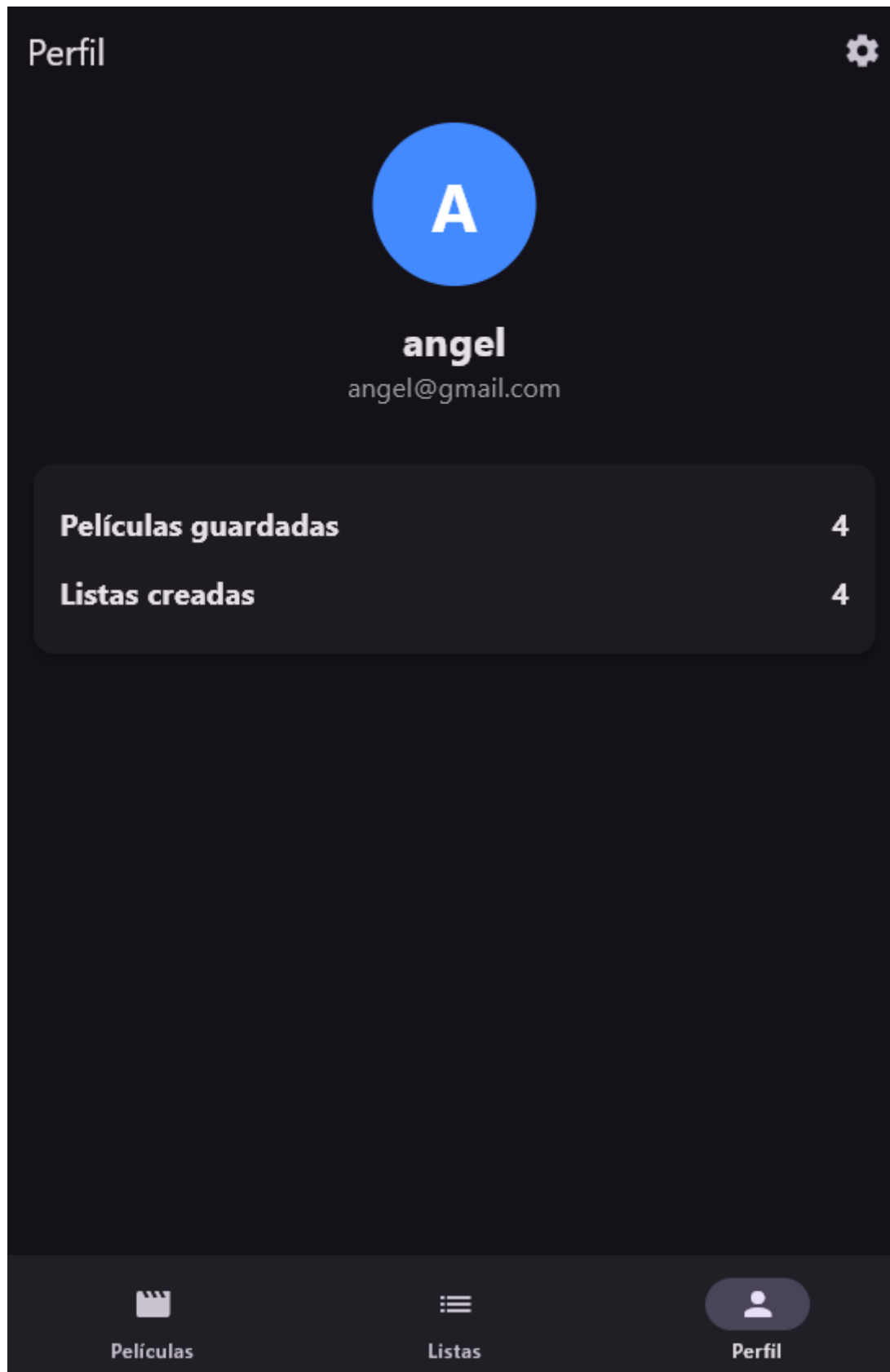
Película añadida a la lista

Añadir película

su jefe. Jack Crawford. Clarice visita la cárcel

 Películas  Listas  Perfil

PANTALLA DE PERFIL



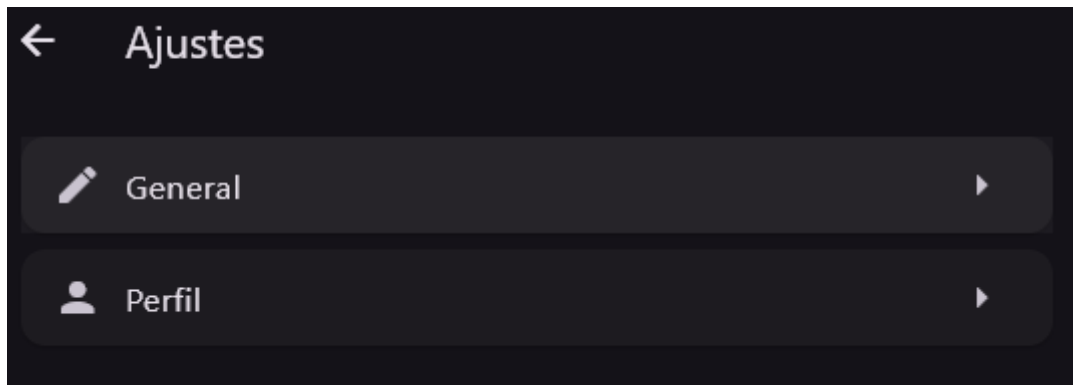
Muestra la información del usuario:

- Nombre.
- Correo electrónico.
- Número de películas guardadas.

- Listas creadas.

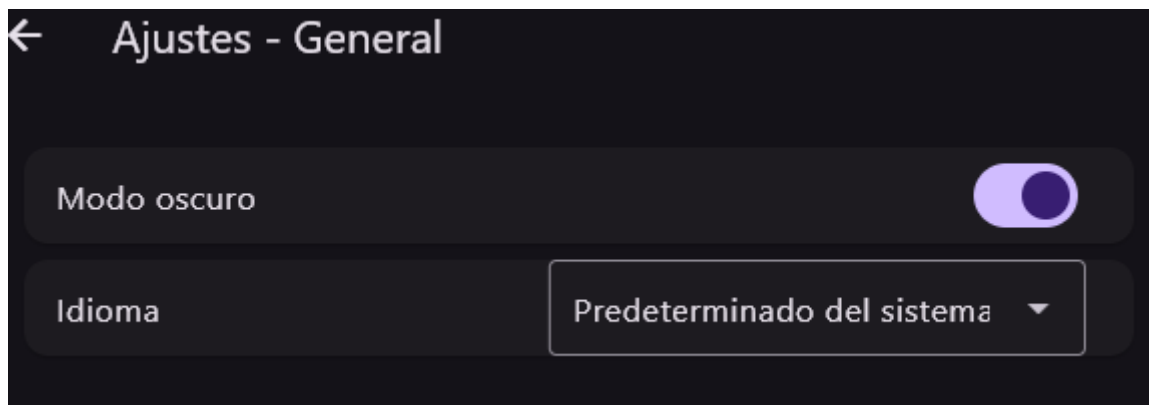
En la parte superior se encuentra el botón para ir a la configuración.

PANTALLA DE CONFIGURACIÓN



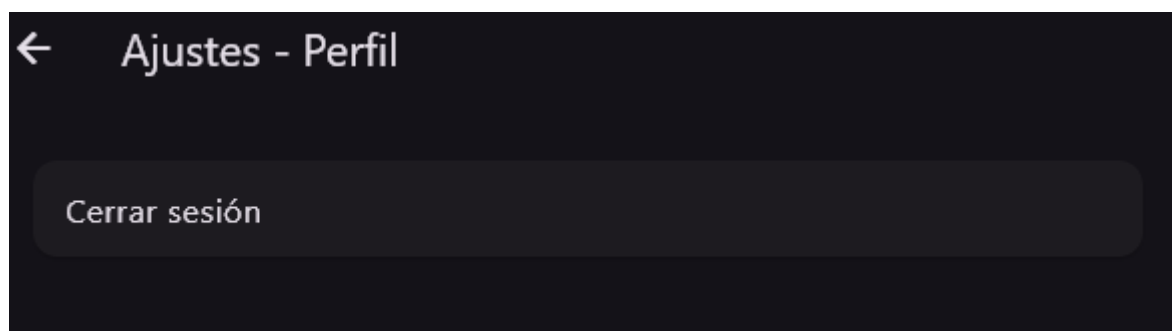
La configuración está dividida en dos submenús.

CONFIGURACIÓN GENERAL:



Permite cambiar el tema y el idioma de la aplicación.

PERFIL:



Contiene un botón para cerrar sesión.

MANUAL DE INSTALACIÓN:

Requisitos Previos:

- Instalación de Python y de las librerías necesarias.
- Configuración de Git y clonación del repositorio.

Pasos para Desplegar la API:

- Instalación de dependencias (por ejemplo, mediante `pip install -r requirements.txt`).
- Configuración del archivo `config.py` con los parámetros de la base de datos y la clave secreta para JWT.
- Ejecución del servidor con `uvicorn main:app --reload`, o desde `main.py`.

Pasos para Desplegar la Aplicación Flutter:

- [Instalación de Flutter](#) y configuración de IntelliJ / Android Studio / VSC.
- Verificar con el comando `flutter doctor` si todo está instalado correctamente.
- Como solo necesitamos la parte de web es necesario tener instalado Chrome o Edge.
- [Clonación del repositorio](#) (**importante, rama fastApi**).
- Ejecución de la aplicación en modo web (**importante, solo de esta forma usa al API**).

CONCLUSIONES Y POSIBLES AMPLIACIONES

CONCLUSIONES

El desarrollo del proyecto ha permitido consolidar conocimientos en el desarrollo de APIs REST con FastAPI, la integración de sistemas mediante JWT y el uso de Flutter para el desarrollo multiplataforma. Entre las conclusiones destacadas se encuentran:

Desarrollo Modular y Escalable:

- La utilización de FastAPI junto con Pydantic y SQLAlchemy ha permitido un desarrollo rápido y modular, facilitando la integración de nuevas funcionalidades.

Integración Multiplataforma:

- La separación del cliente web (basado en la API) y la versión móvil (con SQLite local) garantiza flexibilidad y optimiza la experiencia del usuario según el entorno de ejecución.

Calidad del Código y Pruebas:

- La implementación de pruebas unitarias e integradas con pytest ha contribuido a la robustez del sistema, permitiendo detectar y corregir errores de forma temprana.

Seguridad:

- La implementación de JWT para la autenticación asegura que solo usuarios autorizados pueden acceder a operaciones sensibles, garantizando la integridad del sistema.

DIFICULTADES

Durante el desarrollo se han identificado algunas dificultades, entre las que destacan:

Gestión de Sesiones y Tokens:

- La correcta gestión y renovación de tokens JWT para mantener sesiones seguras en distintos entornos.

Adaptación Cliente - Servidor:

- Transformar la aplicación ya existente para utilizar la api, en lugar de la base de datos local.

POSIBLES AMPLIACIONES

El proyecto ofrece diversas oportunidades de mejora y ampliación, entre las que se pueden considerar:

Migración a Microservicios:

- Separar la gestión de autenticación, películas y listas en servicios independientes para mejorar la escalabilidad y mantenimiento.

Ampliación del Cliente Móvil:

- Incluir funcionalidades offline más avanzadas, como la sincronización automática de cambios cuando se restablezca la conexión a Internet.

Integración con Plataformas Externas:

- Permitir la importación de datos desde APIs de terceros (por ejemplo, IMDB o TMDb) para enriquecer la información de las películas.

BIBLIOGRAFÍA

A continuación, se presenta una lista de referencias y recursos consultados durante el desarrollo del proyecto:

1. Documentación de FastAPI:

[FastAPI Official Documentation](#)

2. Documentación de Pydantic:

[Pydantic Official Documentation](#)

3. Documentación de SQLAlchemy:

[SQLAlchemy Official Documentation](#)

4. Documentación de pytest:

[pytest Official Documentation](#)

5. Documentación de Flutter:

[Flutter Official Documentation](#)