

Trabajo Práctico Integrador Programación Avanzada

Integrantes

Daniela Fernández

Micaela Rojas

Adriel Lorito

Maylén Fuenzalida

Universidad Nacional de Guillermo Brown
Tecnicatura en Programación

20/06/2025

Objetivo.....	3
Tema elegido.....	3
Motivación.....	4
Desarrollo del Proyecto.....	4
Metodología (cómo trabajamos, decisiones técnicas, uso de POO).....	5
Diseño y estructura de código(clases principales, patrones usados).....	5
Funcionamiento general del sistema.....	11
Diagrama de Clases.....	13
Resultados / dificultades (qué funcionó, qué no, cómo lo resolvimos).....	13
Conclusiones Finales.....	14

Objetivo

El objetivo de esta página es ofrecer un espacio donde se puedan encontrar y leer libros de forma gratuita. La idea es que cualquier persona, sin necesidad de registrarse ni pagar, pueda acceder fácilmente a una selección de libros.

Nos pareció útil hacer algo que facilite el acceso a la lectura, sobre todo pensando en quienes no siempre tienen la posibilidad de comprar libros o pagar una suscripción. Por eso creamos esta aplicación simple, con un catálogo de libros, un buscador, y la opción de leer o descargar directamente.

Tema elegido

El tema que elegimos fue ***Aplicaciones Web con Frameworks de Python***.

En nuestro proyecto, se usó **Flask**. Este framework se puede imaginar como si fuera una base ya armada que nos permite construir una página web sin tener que empezar todo desde cero.

Flask nos da funciones listas para usar: mostrar una página, recibir datos de un formulario, redirigir al usuario, conectar con una base de datos, etc. Nosotros sólo tenemos que ir uniendo esas piezas con nuestro código.

Flask es muy liviano y flexible, por lo tanto no tenemos que seguir una estructura complicada y se puede organizar el proyecto como resulte más cómodo. A diferencia de otros frameworks más grandes (como Django), Flask te da más libertad para hacer las cosas a tu manera.

Con Flask, por ejemplo, podés hacer (cosas que se hicieron en nuestro código):

- Una página de inicio
- Un buscador de libros (catálogo)
- Un botón para donar

Y que todo eso funcione desde un solo archivo llamado `app.py`, que es el corazón del programa.

Además, se combina fácilmente con HTML y CSS para mostrar las páginas, y también se puede conectar con bases de datos como SQLite para guardar y leer información.

Flask facilita mucho la creación de páginas web usando Python, ideal para proyectos chicos o medianos como el nuestro.

Motivación

Al momento de elegir el tema del proyecto, como grupo coincidimos en que queríamos **desafiarnos**. Nos interesaba aprender algo nuevo y, al mismo tiempo, aplicar esos conocimientos en un trabajo concreto. Por eso, optamos por trabajar con frameworks de Python, un terreno que nos llamaba la atención.

El código original lo fuimos adaptando según los requisitos que nos indicó el profesor, buscando que fuera más funcional y elaborado. Esto implicó no solo modificarlo, sino también **entender en profundidad** cómo funcionaba cada parte, lo que nos permitió afianzar conceptos y aprender otros nuevos durante el proceso.

Elegimos este proyecto porque sentíamos que, más allá del resultado final, el camino nos iba a aportar **herramientas útiles** para futuros desarrollos.

Desarrollo del Proyecto

Metodología (cómo trabajamos, decisiones técnicas, uso de POO)

Como se mencionó anteriormente, el framework que usamos fue Flask. Al usar como base una idea ya planteada, se buscó organizar el código de forma clara usando **Programación Orientada a Objetos** (POO).

Se crearon clases para representar los elementos principales del proyecto, como los libros y los usuarios. Así se separaron responsabilidades y se mantuvo el código más limpio y fácil de modificar. Por ejemplo, cada libro se define como un objeto con sus propios atributos (título, autor, portada, etc.), y todo lo que tiene que ver con su comportamiento está encapsulado dentro de su clase.

A nivel visual, usamos HTML y CSS, y todas las imágenes están dentro de la carpeta static. Además, la aplicación incluye una sección para donar (con enlace a Mercado Pago) y un login.

Nos repartimos el trabajo entre todos y nos organizamos con WhatsApp, Google Docs y GitHub para comunicarnos, escribir el informe y subir el código. Nos enfocamos en que cada parte del sistema cumpla con lo pedido y sea fácil de entender y mantener.

Diseño y estructura de código(clases principales, patrones usados)

Las clases principales del código son BaseUsuario, Usuario, UsuarioPendiente, FabricaUsuario y Libro.

A continuación se van a explicar:

Class BaseUsuario:

```

1  class BaseUsuario:
2      def __init__(self, id, nombre, email, contrasena, img):
3          self.id = id
4          self.nombre = nombre
5          self.email = email
6          self.contrasena = contrasena
7          self.img = img
8
9      def cambiar_contrasena(self, nueva_contrasena):
10         self.contrasena = nueva_contrasena
11
12     def cambiar_imagen(self, nueva_ruta):
13         self.img = nueva_ruta
14
15     def mostrar_info(self):
16         return f"{self.nombre} ({self.email})"

```

La clase BaseUsuario la hicimos para guardar y manejar los datos de cada usuario, como su nombre, correo, contraseña e imagen. Usamos POO para tener todo junto en un solo lugar y hacer que el código sea más ordenado.

Con esta clase, cada usuario puede cambiar su contraseña o su imagen, y también se puede mostrar su información.

Resuelve el problema de cómo guardar bien los datos de los usuarios y cómo darles funciones básicas.

Class Usuario:

```

TP_Final_Biblio > usuario.py > Usuario
1  import sqlite3
2  from base_usuario import BaseUsuario
3
4  class Usuario(BaseUsuario):
5      def __init__(self, id, nombre, email, contrasena, img, cargo):
6          super().__init__(id, nombre, email, contrasena, img)
7          self.cargo = cargo
8

```

La clase Usuario amplía la clase BaseUsuario, añadiendo el campo cargo, que sirve para saber qué tipo de usuario es (por ejemplo, lector o admin). Hereda todo lo demás: nombre, email, contraseña, etc., para evitar repetir código y mantener todo organizado.

Se importa **sqlite3** porque este archivo se conecta directamente a la base de datos del proyecto, llamada biblio.db, donde se guardan los usuarios. Así podemos hacer operaciones reales como guardar un nuevo usuario, buscar por email o ID, cambiar datos o eliminarlo.

Muchos métodos están marcados como **@staticmethod**, porque esas funciones no necesitan acceder directamente a los datos del objeto (self), sino que trabajan directamente con la base de datos. Es decir, no hace falta tener un usuario ya creado para poder buscar o modificar algo. Por ejemplo, para obtener todos los usuarios o eliminar uno, no necesitamos tener un objeto de Usuario antes. Esto hace el código más limpio y fácil de usar desde otras partes del programa.

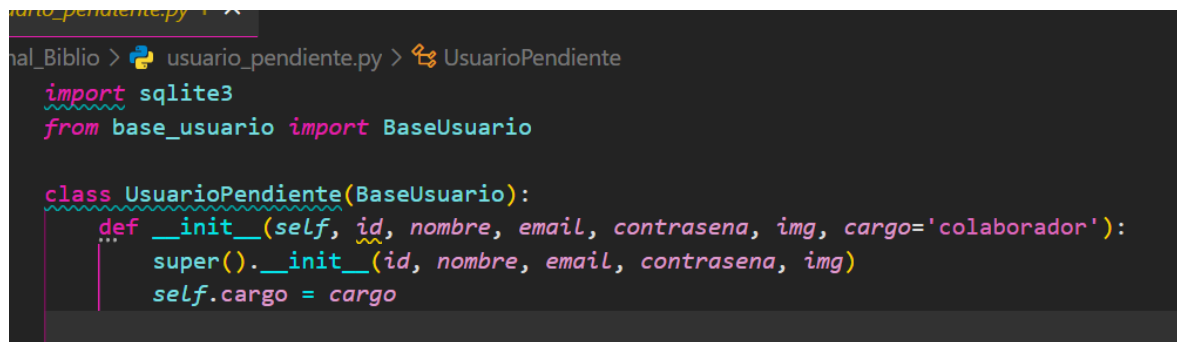
```
@staticmethod
def obtener_por_email(email):
    with sqlite3.connect("biblio.db") as conn:
        conn.row_factory = sqlite3.Row
        cursor = conn.cursor()
        cursor.execute("SELECT * FROM usuarios WHERE email=?", (email,))
        fila = cursor.fetchone()
        if fila:
            return Usuario(
                id=fila["id"],
                nombre=fila["nombre"],
                email=fila["email"],
                contraseña=fila["contrasena"],
                img=fila["img"],
                cargo=fila["cargo"]
            )
        return None
```

Por ejemplo, en esa parte del código se usa **@staticmethod** porque no necesitamos tener un usuario creado para usarla. Solo le pasamos el email y ya busca en la base de datos.

Si queremos encontrar a un usuario por su gmail, hacemos: `Usuario.obtener_por_email("ejemplo@correo.com")`

Esta clase permite gestionar de forma completa a los usuarios dentro del sistema, tanto en memoria como en la base de datos.

Class UsuarioPendiente:



```

usuario_pendiente.py
hal_Biblio > usuario_pendiente.py > UsuarioPendiente
import sqlite3
from base_usuario import BaseUsuario

class UsuarioPendiente(BaseUsuario):
    def __init__(self, id, nombre, email, contrasena, img, cargo='colaborador'):
        super().__init__(id, nombre, email, contrasena, img)
        self.cargo = cargo
  
```

La clase UsuarioPendiente es muy parecida a la clase Usuario, pero está pensada para los que todavía no fueron aprobados en el sistema. Por eso se guarda en una tabla diferente, que se llama usuarios_pendientes.

También hereda de BaseUsuario, así que no hace falta volver a escribir el código para nombre, email, contraseña o imagen. Solo se le agrega el campo cargo, que por defecto es 'colaborador', porque es el rol que suelen tener cuando se registran.

Tiene métodos simples para guardar un nuevo usuario en espera, traer todos los que están pendientes o eliminar a uno si se decide que no va a entrar. En dos de estos métodos también se marcaron como `@staticmethod` porque no hace falta tener un objeto creado para usarlos, solo llaman a la base de datos y hacen su trabajo.

Con esta clase se puede revisar quién está esperando aprobación y decidir si se lo acepta o no. En efecto se creó esta clase para tener más control sobre quién entra al sistema.

Explayando un poco más con el tema de SQLite para trabajar con la base de datos, tenemos una serie de métodos usados como por ejemplo `cursor()` para preparar y ejecutar consultas SQL. También `execute()` para mandar la consulta, y `fetchall()` que se usa después

de un SELECT para devolver los resultados. Si se hacen cambios, como insertar o borrar algo, se usa commit() para que esos cambios queden guardados.

Class FabricaUsuario:

```

from usuario import Usuario
from usuario_pendiente import UsuarioPendiente

class FabricaUsuario:
    @staticmethod
    def crear_postulante(**kwargs):
        requeridos = ["nombre", "email", "contrasena", "img"]
        for campo in requeridos:
            if campo not in kwargs:
                raise ValueError(f"Falta el parámetro obligatorio: {campo}")

        return UsuarioPendiente(
            id=None,
            nombre=kwargs["nombre"],
            email=kwargs["email"],
            contrasena=kwargs["contrasena"],
            img=kwargs["img"],
            cargo=kwargs.get("cargo", "colaborador")
        )

```

La clase FabricaUsuario se hizo para encargarse de crear usuarios nuevos, especialmente cuando alguien se registra en el sistema. Tiene dos funciones principales: una sirve para crear un UsuarioPendiente, y la otra transforma ese postulante en un Usuario cuando ya fue aprobado.

La idea de armar esta clase fue para no tener que repetir código cada vez que se registra alguien. En vez de escribir todo de nuevo, simplemente llamamos a esta clase con los datos necesarios, y ella se encarga de armar el usuario. También verifica que no falte ningún dato importante, como el nombre, correo, contraseña o imagen.

A diferencia de otras clases, esta no tiene un `__init__` porque no necesita guardar información propia ni crear objetos de sí misma. Solo funciona como una herramienta que genera objetos de otras clases, por eso todos sus métodos son `@staticmethod`.

Estamos usando programación orientada a objetos porque organizamos todo en clases, cada una con su responsabilidad. En este caso, la FabricaUsuario se encarga solo del proceso de crear usuarios.

Esta clase ayuda a mantener el orden en el sistema de registro. Es útil para este proyecto porque queremos que muchas personas puedan acceder a libros gratis, pero también necesitamos controlar quién entra, revisar su información y aceptar solo a los que cumplen con ciertos requisitos. Así, se asegura que el sistema funcione bien y que todos los usuarios estén registrados correctamente.

Class Libro:

```
import sqlite3

class Libro:
    def __init__(self, id, titulo, autor, descarga, portada, es_inicio=0):
        self.id = id
        self.titulo = titulo
        self.autor = autor
        self.descarga = descarga
        self.portada = portada
        self.es_inicio = es_inicio
```

Esta es la última clase que forma parte del sistema, y se hizo para manejar toda la información relacionada con los libros. Cada libro tiene su título, autor, link de descarga (todavía no funciona), imagen de portada y un campo extra que se usa si ese libro va a aparecer en la página principal.

Con esta clase podemos hacer varias cosas definidas en los métodos: guardar un libro nuevo, actualizarlo, eliminarlo o buscarlo por título, autor o por su ID. Algunas funciones son estáticas porque no hace falta tener un libro creado para usarlas. Por ejemplo, se pueden traer todos los libros de la base de datos o buscar uno sin necesidad de crear primero un objeto.

Con ella podemos mostrar y gestionar libros dentro del sistema. Es clave para cumplir el objetivo del proyecto: permitir que cualquier persona pueda acceder a libros gratuitos de forma ordenada y simple.

Funcionamiento general del sistema

El archivo principal del proyecto es `app.py`. Desde ahí se configuran todas las rutas que permiten que el sistema funcione desde el navegador. La primera ruta que se ejecuta es la de inicio, que se llama **home**. Esa ruta hace una consulta a la base de datos, específicamente a la tabla donde se guardan los libros destacados. Lo que hace es obtener los id de esos libros y, con ayuda de la función `obtener_por_ids` que está en la clase `Libro`, se buscan los datos completos. Una vez que se tiene esa información, se carga el HTML de inicio con las tarjetas correspondientes.

La página que se muestra en ese inicio es **index.html**. Ahí hay un menú para moverse por distintas secciones como catálogo, suscripción, donación, login y contacto. También hay un botón que lleva directo al catálogo y se muestran cinco libros seleccionados.

En la ruta de **suscripción** se abre una página que tiene un botón que redirige a la cuenta de Mercado Pago de la empresa. Ahí se puede elegir pagar una suscripción mensual de \$5000. Se descuenta de forma automática cada mes, pero también se puede cancelar cuando uno quiera.

La ruta del **catálogo** permite buscar libros. Si el usuario no escribe nada en el buscador, se muestran todos los libros disponibles. Pero si escribe algo, se filtra por título o autor, según lo que se haya cargado en el formulario. En caso de no encontrar resultados, aparece un mensaje. Para los administradores hay un catálogo especial donde, además de ver los libros, pueden modificarlos o borrarlos.

La función **donación** simplemente carga el HTML que tiene un botón para transferir el monto que desees.

En cuanto al **login**, se accede desde un formulario. Los datos se envían con el método POST y se comparan con los que hay en la base. Si coinciden, se guarda la sesión del usuario y se lo redirige a su panel. Si hay error en el mail o la contraseña, vuelve al login

con un mensaje. También se usa el método GET para que esa ruta cargue normalmente si alguien entra desde el navegador.

En el **registro**, el formulario permite crear un nuevo usuario pendiente. Se valida que los campos estén completos, que las contraseñas coincidan y que el mail no exista ya en la base. Si no se sube una imagen, se carga una por defecto. Y si se sube, se guarda en la carpeta "static/img/usuarios". Luego, con ayuda de la clase FabricaUsuario, se genera el nuevo usuario y queda esperando aprobación.

En la parte del **inicio admin**, se muestran también los libros destacados, pero además se saluda al usuario y se agregan botones para modificar o eliminar libros. Si se borra uno que estaba en la portada, se reemplaza automáticamente por otro. Esta vista sólo es accesible si hay una sesión activa, y la función busca en la tabla correspondiente los libros a mostrar.

Cuando se elige **eliminar un libro**, además de borrarlo de la base, también se borra su imagen guardada y, si era parte de los destacados, se actualiza la portada con otro libro que no esté repetido.

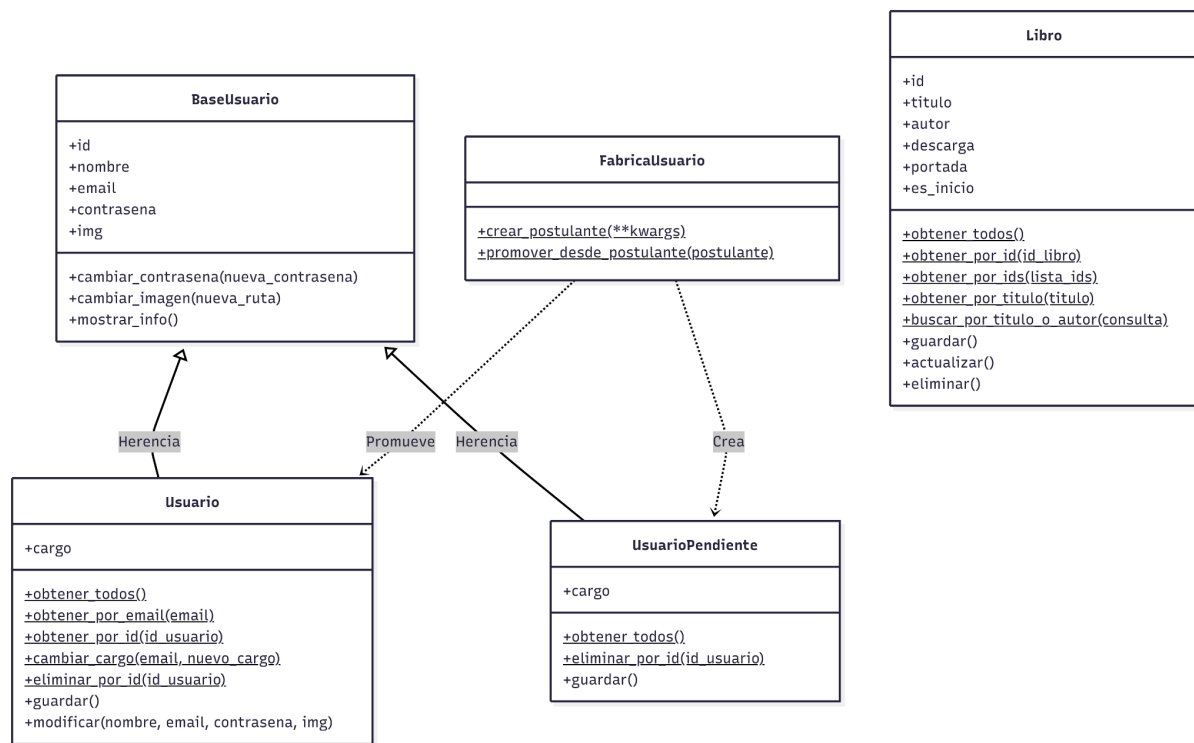
En la función de **modificar libro**, se muestran los datos actuales del libro y permite cambiar solo lo necesario. Si se sube una nueva imagen, se reemplaza por la anterior. Si no, se deja la que ya estaba. Todo se actualiza en la base y redirige al inicio admin.

En la parte de **libros inicio**, se puede cambiar qué cinco libros aparecen destacados. Si alguno de los libros escritos no existe en la base de datos, se muestra un mensaje con los errores y los libros disponibles. Si todo está bien, se actualiza la tabla con los nuevos IDs.

En el **perfil**, el usuario puede ver sus datos (nombre, mail, contraseña tapada con asteriscos, imagen y cargo) y tiene las opciones de modificar su perfil o cerrar sesión. También se muestran dos listas: postulantes y empleados. A los postulantes se los puede aceptar o rechazar, y a los empleados se los puede modificar o eliminar, siempre que tengan un cargo inferior al del usuario que está logueado.

Por último, en **modificar perfil**, se carga un formulario con los datos actuales del usuario. Se valida que las contraseñas coincidan y que el correo no esté repetido. Si se cambia la imagen y no es la predeterminada, se actualiza. Luego se guarda todo en la base y se actualizan también los datos en sesión.

Diagrama de Clases



Resultados / dificultades (qué funcionó, qué no, cómo lo resolvimos)

A lo largo del desarrollo, surgieron varios errores y decisiones que se fueron resolviendo en el momento. Uno de los temas fue cómo marcar qué libros iban a aparecer en la página de inicio. Al principio se pensó en agregar un campo extra en la tabla libros, como un booleano o un número. Sin embargo, al final se armó una tabla aparte solo para eso. No es la solución más limpia, pero el tiempo jugó en contra y se decidió seguir así. Es algo que se podría mejorar más adelante.

Otro error que surgió fue con los tipos de datos. En una parte del sistema, antes se trabajaba con diccionarios donde los id estaban guardados como cadenas (str). Pero en la base de datos, los id estaban como enteros (int), y eso causaba problemas con el inicio de sesión porque no se reconocían como iguales. Al identificar esa diferencia, se cambió para que los tipos coincidieran.

También hubo un problema cuando se pasó a usar herencia en las clases de usuario. Antes, los datos se acomodaban según el orden de los campos. Pero al hacer cambios, ese orden ya no coincidía, y por eso tomaba el email como si fuera la contraseña por ejemplo. Eso hacía que no se pudiera iniciar sesión. Se solucionó haciendo que, en vez de guiarse por la posición, el código identificara cada dato por el nombre de la columna que viene desde la base de datos, y así se asigna correctamente al atributo correspondiente.

Fueron errores chicos, pero varios, y se resolvieron revisando bien cada parte y prestando atención a lo que mostraba la consola. Se fue acomodando todo para que el sistema funcione como se esperaba.

Conclusiones Finales

(+ Reflexiones sobre la cursada)

Hacer este proyecto nos ayudó a aplicar algunos de los temas que vimos en la materia, como programación orientada a objetos y el uso de Flask. Al principio costó un poco organizarnos, pero con el tiempo fuimos acomodando las tareas y logramos que el sistema funcionara.

A lo largo del trabajo, fuimos adaptándonos a los cambios que iban surgiendo lo más rápido posible, porque el tiempo no jugó a favor. La materia nos ayudó a sentar las bases de lo que es POO y a entender la importancia de organizar el código de forma clara. Más allá de los errores o ajustes que quedaron por hacer, la experiencia nos sirvió para acercarnos a cómo se programa realmente fuera clases.

Enlace al video !!!

<https://photos.app.goo.gl/WC7ED5vBXVwpy1ny9>