Universidad Nacional de Ingeniería

Facultad de Ingeniería Eléctrica y Electrónica

Programación Orientada de Objetos (BMA15)

Documentación del Proyecto Loglytics

Integrantes:

- Yaulilahua Llancari Bruss Denis (20232613F)
- Salazar Ore Christopher Zahir (20232617A)
- Vasquez Villa Luis Enrique (20232601H)
- Paucas Rojas Jeremy Patrick (20231378C)

Profesor: Tello Chancapoma Yuri Oscar

Fecha: 30-11-24

Indice

Introducción

El proyecto **Loglytics** es una herramienta web desarrollada con **Flask** y **Python** que permite gestionar, analizar y visualizar logs de eventos de aplicaciones en tiempo real. Su propósito principal es facilitar la carga, análisis y presentación gráfica de logs de servidores y aplicaciones, proporcionando a los usuarios una interfaz amigable para monitorear y detectar problemas en sus sistemas.

Características Principales:

- **Carga de Logs:** Los usuarios pueden cargar archivos de logs a través de una interfaz web. Los logs se procesan y almacenan para su posterior análisis.
- **Análisis de Logs:** La aplicación procesa los logs para clasificarlos en diferentes tipos, como **INFO**, **WARNING**, **DEBUG** y **ERROR**, lo que permite identificar rápidamente patrones y posibles problemas.
- **Visualización de Datos:** La herramienta incluye dashboards interactivos que muestran estadísticas y gráficos sobre los logs procesados, como distribuciones de tipo de logs y detalles específicos de los errores encontrados.
- **Registro de Usuarios:** Los usuarios pueden registrarse y acceder a un servidor con un nombre de usuario y contraseña.

Objetivo del Proyecto

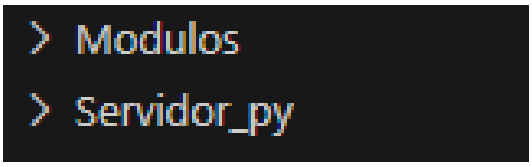
El objetivo de **Loglytics** es proporcionar a los desarrolladores y administradores de sistemas una solución sencilla y eficiente para manejar grandes volúmenes de logs, realizar análisis en tiempo real y obtener una mejor comprensión de los eventos que ocurren dentro de sus aplicaciones y servidores. Al ofrecer análisis automatizados y gráficos interactivos, la herramienta ayuda a detectar rápidamente problemas y optimizar el rendimiento de las aplicaciones.

Tecnologías Utilizadas

- **Flask:** Framework ligero de Python para el desarrollo de aplicaciones web.
- **SQLAlchemy:** ORM(mapeo relcional de objetos) para interactuar con bases de datos.
- **Matplotlib & Pandas:** Para la generación de gráficos y análisis de datos.
- **Jinja2:** Motor de plantillas utilizado para renderizar HTML dinámico.

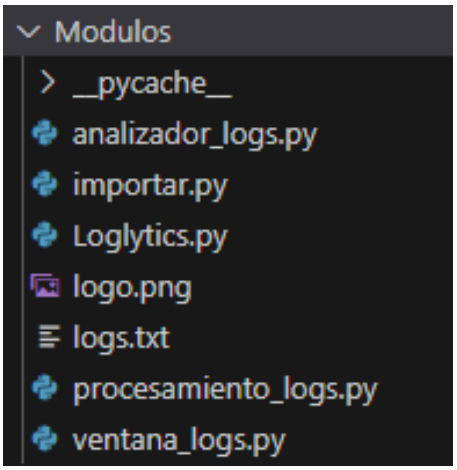
Este proyecto está diseñado para ser escalable y modular, permitiendo agregar nuevas funcionalidades y adaptarse a diferentes tipos de logs y escenarios de uso.

Para empezar, debemos mencionar que el proyecto consta de dos carpetas principales, las cuales se llaman “Modulos” y “Servidor_py”.

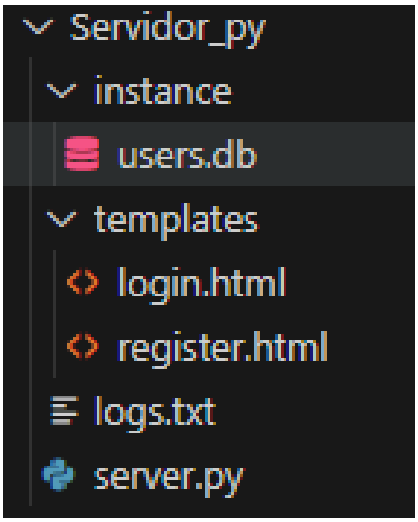


La primera carpeta contiene todos los módulos que serán necesarios para el análisis del archivo que contendrá los logs, mientras que la segunda carpeta contiene los archivos necesarios para la creación del servidor de un login, del cual se obtendrán los logs a analizar.

Contenido de la carpeta “Módulos”:



Contenido de la carpeta “Servidor_py”:



A continuación, se explicarán cada uno de los modulos usados en el proyecto.

Contenido de la carpeta “Módulos”

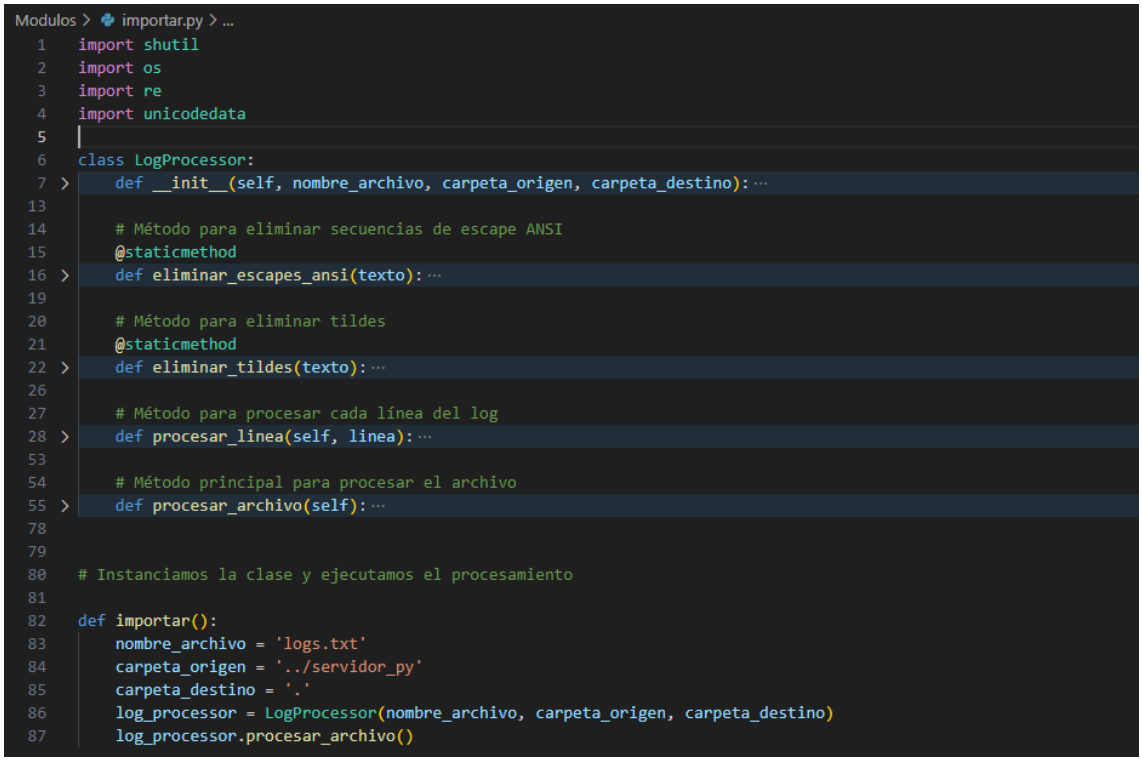
Archivo: importar.py

Este módulo se encarga de procesar archivos de logs, limpiarlos y transformarlos según ciertos patrones. El procesamiento se realiza mediante la clase LogProcessor, que gestiona la lectura, modificación y copia de archivos.

Descripción General

El módulo tiene como objetivo importar archivos de log desde un directorio de origen, procesarlos para eliminar información irrelevante (como secuencias de escape ANSI, tildes y marcas de tiempo), y luego guardarlos en un directorio de destino.

Utiliza programación orientada a objetos (POO) para organizar las funcionalidades en la clase LogProcessor.



1. Clase LogProcessor

La clase LogProcessor es responsable del procesamiento de archivos de log. Tiene varias funciones, cada una de las cuales realiza una tarea específica. A continuación, se detallan sus métodos y atributos:

Atributos

- **nombre_archivo** (str): Nombre del archivo de log que se va a procesar.
- **carpeta_origen** (str): Ruta del directorio de origen donde se encuentra el archivo de log.

- **carpeta_destino** (str): Ruta del directorio donde se almacenará el archivo procesado.
- **ruta_origen** (str): Ruta completa del archivo de log en el directorio de origen.
- **ruta_destino** (str): Ruta completa donde se guardará el archivo procesado.

Métodos

- **__init__(self, nombre_archivo, carpeta_origen, carpeta_destino)**
  Constructor que inicializa los atributos del objeto. Recibe el nombre del archivo, las rutas de origen y destino.
- **eliminar_escapes_ansi(texto)**
  Método estático que elimina secuencias de escape ANSI en el texto. Estas secuencias suelen ser usadas para formatear texto en la terminal, pero no son necesarias en los logs.

  Parámetros:

  ✓ texto (str): Texto a procesar.

  Retorna:

  ✓ Texto sin secuencias de escape ANSI.
- **eliminar_tildes(texto)**
  Método estático que elimina las tildes de un texto utilizando normalización Unicode.

  Parámetros:

  ✓ texto (str): Texto a procesar.

  Retorna:

  ✓ Texto sin tildes.
- **procesar_linea(self, linea)**
  Método que procesa cada línea del archivo de log de acuerdo con ciertos patrones:

  a) Elimina las secuencias de escape ANSI.

  b) Modifica las líneas con códigos HTTP y reemplaza ciertas partes según el código.

  c) Reemplaza patrones de fecha con un espacio vacío.

  d) Elimina tildes en los caracteres.

  Parámetros:

  ✓ linea (str): Línea del archivo de log a procesar.

Retorna:

- ✓ Línea procesada (str).

- **procesar_archivo(self)**
  Método principal que procesa el archivo completo:

a) Copia el archivo desde la carpeta de origen a la de destino.

b) Lee y procesa las líneas del archivo, omitiendo las primeras 5.

c) Guarda las líneas procesadas en el archivo de destino.

Retorna:

- ✓ Mensajes en consola sobre el estado de la operación (copia y procesamiento).

2. Función importar()

Esta es la función principal que se ejecuta cuando se importa el módulo. Instancia un objeto LogProcessor con los parámetros correspondientes y llama al método procesar_archivo().

Parámetros:

- No tiene parámetros de entrada directos, ya que los valores de nombre_archivo, carpeta_origen y carpeta_destino están definidos de manera interna.

Descripción:

La función crea una instancia de la clase LogProcessor, pasando los parámetros definidos, y luego ejecuta el procesamiento del archivo de logs.

Ejemplo de ejecución:

```
importar()
```

3. Flujo de Ejecución del Programa

1. **Instanciación de LogProcessor:** Se crea un objeto log_processor de la clase LogProcessor, pasándole el nombre del archivo de log, la carpeta de origen y la carpeta de destino.
2. **Copiado del archivo:** El archivo de log se copia desde la carpeta de origen a la carpeta de destino.
3. **Procesamiento de las líneas del archivo:** Se lee el archivo, se procesan las líneas (se eliminan secuencias de escape, se reemplazan códigos HTTP y se eliminan tildes) y finalmente se guardan las líneas procesadas en el archivo de destino.
4. **Manejo de errores:** En caso de que haya errores durante el proceso (por ejemplo, si el archivo no existe en la carpeta de origen o hay

problemas al copiar o procesar el archivo), se captura la excepción y se muestra un mensaje de error.

4. Ejemplo de Uso

```
from importar import importar

# Ejecutar la importación y procesamiento del archivo de log
importar()
```

Este ejemplo muestra cómo se puede usar la función importar() para procesar el archivo de log sin necesidad de interactuar directamente con la clase LogProcessor.

5. Dependencias Externas

El módulo utiliza las siguientes bibliotecas estándar de Python:

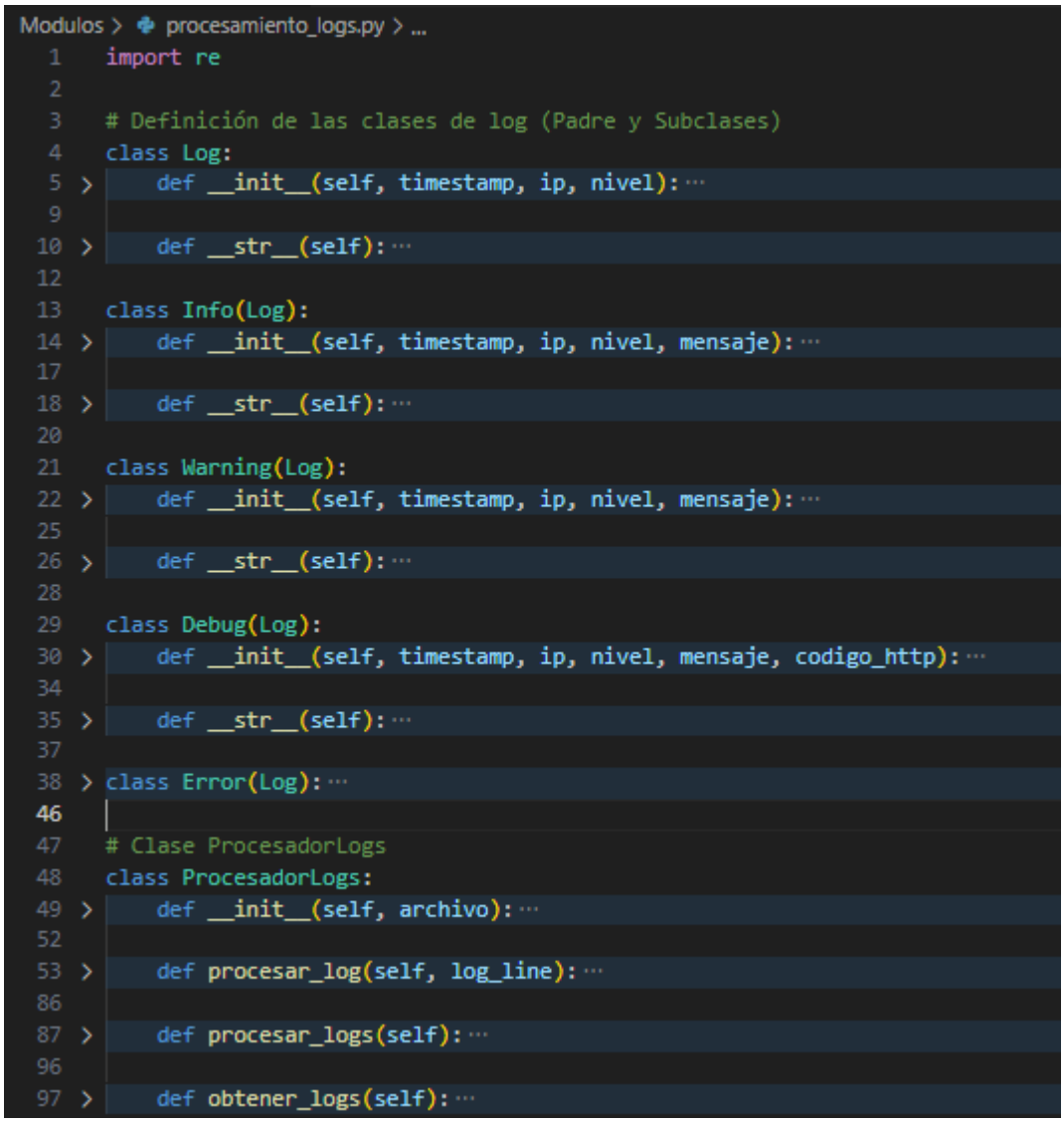
- **shutil**: Para copiar archivos.
- **os**: Para manipulación de rutas y archivos.
- **re**: Para trabajar con expresiones regulares.
- **unicodedata**: Para manipulación de caracteres Unicode.

6. Referencias y Recursos

- Documentación de la biblioteca shutil: https://docs.python.org/3/library/shutil.html
- Documentación de la biblioteca os: https://docs.python.org/3/library/os.html
- Documentación de la biblioteca re: https://docs.python.org/3/library/re.html
- Documentación de la biblioteca unicodedata: https://docs.python.org/3/library/unicodedata.html

Archivo: procesamiento_logs.py

Este módulo se encarga de procesar líneas de logs que contienen diferentes niveles de severidad (INFO, WARNING, DEBUG, ERROR) y crear instancias de clases específicas para cada tipo de log. El procesamiento se realiza mediante la clase ProcesadorLogs, que organiza y almacena los logs procesados en objetos de las clases Log, Info, Warning, Debug, y Error.



1. Clases de Log

Clase Log (Clase base)

La clase Log es la clase base de la que derivan todas las clases de logs específicas (INFO, WARNING, DEBUG, ERROR). Representa un log genérico con los atributos comunes a todos los logs.

Atributos:

- **timestamp** (str): Marca de tiempo del log, en formato YYYY-MM-DD HH:MM:SS.
- **ip** (str): Dirección IP asociada al log.
- **nivel** (str): Nivel del log (INFO, WARNING, DEBUG, ERROR).

Métodos:

- **__str__(self)**: Método que retorna una representación en cadena del log, mostrando el timestamp, la IP y el nivel.

Clases Derivadas de Log

Las siguientes clases representan los diferentes niveles de logs, cada una de ellas hereda de la clase Log y añade información adicional relevante para cada tipo de log.

- **Clase Info**
  - ✓ **Atributo adicional**: mensaje (str) - Mensaje del log.
  - ✓ **Método __str__(self)**: Retorna la cadena de texto que incluye el timestamp, la IP, el nivel de log (INFO) y el mensaje.
- **Clase Warning**
  - ✓ **Atributo adicional**: mensaje (str) - Mensaje del log.
  - ✓ **Método __str__(self)**: Retorna la cadena de texto que incluye el timestamp, la IP, el nivel de log (WARNING) y el mensaje.
- **Clase Debug**
  - ✓ **Atributos adicionales**:
    - mensaje (str) - Mensaje del log.
    - codigo_http (str) - Código HTTP asociado al log de tipo DEBUG.
  - ✓ **Método __str__(self)**: Retorna la cadena de texto que incluye el timestamp, la IP, el nivel de log (DEBUG), el mensaje y el código HTTP.
- **Clase Error**
  - ✓ **Atributos adicionales**:
    - mensaje (str) - Mensaje del log.
    - codigo_http (str) - Código HTTP asociado al log de tipo ERROR.
  - ✓ **Método __str__(self)**: Retorna la cadena de texto que incluye el timestamp, la IP, el nivel de log (ERROR), el mensaje y el código HTTP.

2. Clase ProcesadorLogs

La clase ProcesadorLogs es responsable de procesar los archivos de logs y generar las instancias correspondientes de las clases Info, Warning, Debug, y Error.

Atributos:

- **archivo** (str): Ruta del archivo de logs a procesar.
- **logs_procesados** (list): Lista donde se almacenarán los logs procesados como instancias de las clases derivadas de Log.

Métodos:

- **__init__(self, archivo)**
  Constructor que recibe el nombre del archivo de logs y lo asigna a self.archivo. También inicializa la lista self.logs_procesados para almacenar los logs procesados.
- **procesar_log(self, log_line)**
  Este método procesa una línea de log y devuelve una instancia de la clase correspondiente (INFO, WARNING, DEBUG o ERROR) dependiendo del nivel del log.
  Parámetros:
  - ✓ log_line (str): Línea de log que se desea procesar.

  Retorna:
  - ✓ Instancia de la clase correspondiente (Info, Warning, Debug, Error) o None si la línea no corresponde a un log válido.
- **procesar_logs(self)**
  Este método procesa el archivo de logs línea por línea, utilizando el método procesar_log() para obtener las instancias correspondientes. Los logs procesados se almacenan en la lista self.logs_procesados.
  Retorna:
  - ✓ Lista de logs procesados (instancias de las clases Log y sus subclases).
- **obtener_logs(self)**
  Este método devuelve la lista de logs procesados almacenada en self.logs_procesados.
  Retorna:
  - ✓ Lista de logs procesados.

3. Flujo de Ejecución del Programa

1. **Instanciación de ProcesadorLogs:**
   Se crea un objeto ProcesadorLogs pasando el nombre del archivo de logs a procesar.
2. **Procesamiento de Logs:**
   Se llama al método procesar_logs(), que lee el archivo línea por línea y crea instancias de las clases Info, Warning, Debug, o Error dependiendo del nivel de cada log.
3. **Acceso a los Logs Procesados:**
   Después de procesar el archivo, se puede acceder a los logs procesados mediante el método obtener_logs(), que retorna la lista de instancias de logs.

4. Ejemplo de Uso

```python
from procesamiento_logs import ProcesadorLogs

# Instanciar el procesador de logs con el archivo de logs a procesar
procesador = ProcesadorLogs("logs.txt")

# Procesar los logs
logs_procesados = procesador.procesar_logs()

# Acceder a los logs procesados
for log in logs_procesados:
    print(log)
```

Este ejemplo muestra cómo se puede utilizar la clase ProcesadorLogs para procesar un archivo de logs y luego imprimir cada uno de los logs procesados.

5. Dependencias Externas

El módulo utiliza la biblioteca estándar re (expresiones regulares) para extraer los valores relevantes de cada línea de log.

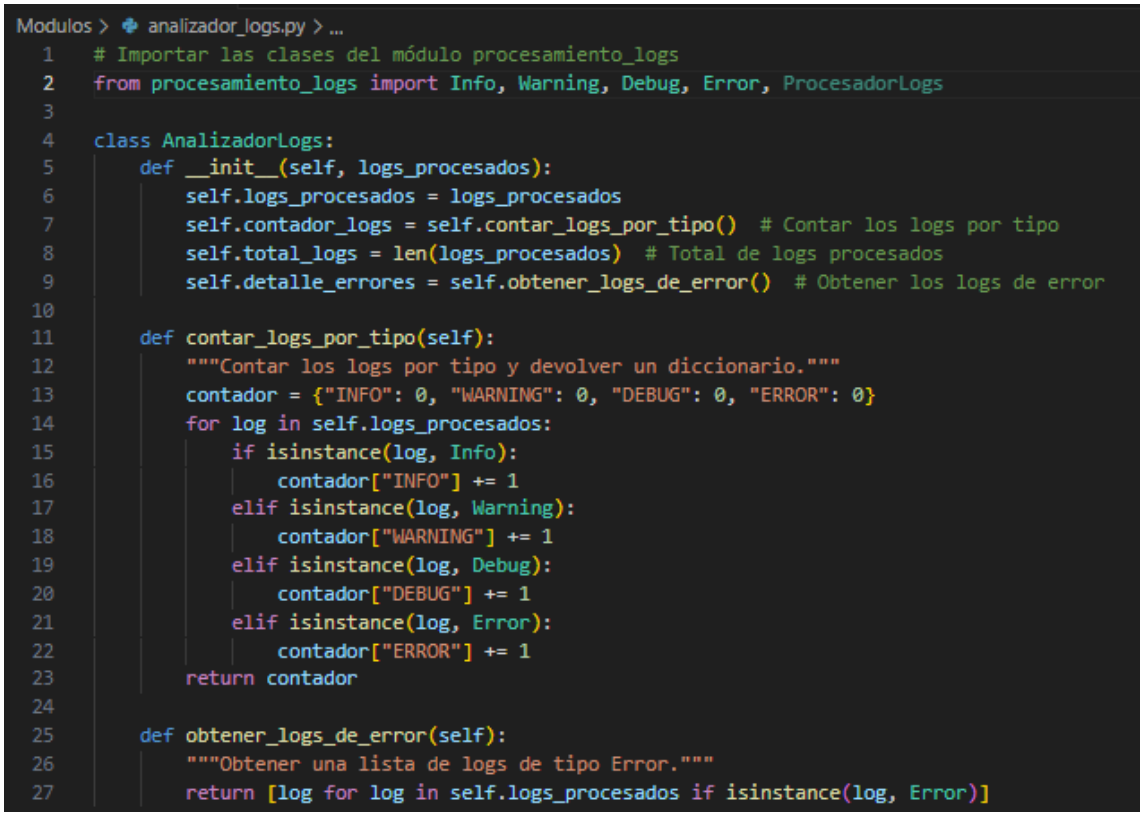
- Documentación de la biblioteca re: https://docs.python.org/3/library/re.html

6. Consideraciones

- El archivo de log debe tener un formato consistente con las expresiones regulares utilizadas en el procesamiento. Cada línea del log debe incluir un timestamp, una dirección IP, un nivel de log y, en algunos casos, un mensaje y un código HTTP.
- Si alguna línea no se ajusta al formato esperado, el método procesar_log() devuelve None, por lo que las líneas no procesadas se descartan.

Archivo: analizador_logs.py

Este módulo proporciona la clase AnalizadorLogs, que se encarga de analizar los logs procesados y realizar diversas tareas de análisis como contar los logs por tipo y extraer los logs de tipo Error. Utiliza las clases definidas en el módulo procesamiento_logs.py para trabajar con los logs procesados.



```python
# Importar las clases del módulo procesamiento_logs
from procesamiento_logs import Info, Warning, Debug, Error, ProcesadorLogs

class AnalizadorLogs:
    def __init__(self, logs_procesados):
        self.logs_procesados = logs_procesados
        self.contador_logs = self.contar_logs_por_tipo()  # Contar los logs por tipo
        self.total_logs = len(logs_procesados)  # Total de logs procesados
        self.detalle_errores = self.obtener_logs_de_error()  # Obtener los logs de error

    def contar_logs_por_tipo(self):
        """Contar los logs por tipo y devolver un diccionario."""
        contador = {"INFO": 0, "WARNING": 0, "DEBUG": 0, "ERROR": 0}
        for log in self.logs_procesados:
            if isinstance(log, Info):
                contador["INFO"] += 1
            elif isinstance(log, Warning):
                contador["WARNING"] += 1
            elif isinstance(log, Debug):
                contador["DEBUG"] += 1
            elif isinstance(log, Error):
                contador["ERROR"] += 1
        return contador

    def obtener_logs_de_error(self):
        """Obtener una lista de logs de tipo Error."""
        return [log for log in self.logs_procesados if isinstance(log, Error)]
```

1. Clase AnalizadorLogs

La clase AnalizadorLogs está diseñada para analizar los logs que han sido previamente procesados por la clase ProcesadorLogs. Proporciona métodos para contar los logs de cada tipo (INFO, WARNING, DEBUG, ERROR) y para obtener detalles específicos de los logs de error.

Atributos:

- **logs_procesados** (list): Lista de objetos Log que contienen los logs procesados. Estos objetos son instancias de las subclases Info, Warning, Debug y Error.
- **contador_logs** (dict): Diccionario que contiene el conteo de logs por tipo. Las claves son los tipos de log ("INFO", "WARNING", "DEBUG", "ERROR") y los valores son los conteos de cada tipo.
- **total_logs** (int): Número total de logs procesados.
- **detalle_errores** (list): Lista que contiene solo los logs de tipo Error (instancias de la clase Error).

Métodos:

- **__init__(self, logs_procesados)**
  Constructor que recibe una lista de logs procesados (logs_procesados) y realiza el análisis inicial:
  - ✓ Llama al método contar_logs_por_tipo() para contar los logs por tipo.
  - ✓ Asigna el total de logs procesados a total_logs.
  - ✓ Llama al método obtener_logs_de_error() para obtener todos los logs de tipo Error.
- **contar_logs_por_tipo(self)**
  Método que cuenta los logs de cada tipo (INFO, WARNING, DEBUG, ERROR). Recorre la lista logs_procesados y cuenta cuántos logs pertenecen a cada tipo utilizando la función isinstance() para verificar el tipo de cada log.
  Retorna:
  - ✓ Un diccionario con los conteos de logs por tipo, por ejemplo:

```
{"INFO": 10, "WARNING": 3, "DEBUG": 5, "ERROR": 2}
```

- **obtener_logs_de_error(self)**
  Método que obtiene todos los logs de tipo Error de la lista logs_procesados.
  Retorna:
  - ✓ Una lista de objetos Error que representan todos los logs de error procesados.

2. Flujo de Ejecución del Programa

1. **Instanciación de AnalizadorLogs:**
   Se crea un objeto AnalizadorLogs pasando una lista de logs procesados (por ejemplo, obtenida de la clase ProcesadorLogs).
2. **Análisis de los Logs:**
   - ✓ El método contar_logs_por_tipo() cuenta los logs de cada tipo (INFO, WARNING, DEBUG, ERROR) y almacena los resultados en el atributo contador_logs.
   - ✓ El atributo total_logs almacena el número total de logs procesados.
   - ✓ El método obtener_logs_de_error() crea una lista con los logs de tipo Error.
3. **Acceso a los Resultados:**

- ✓ Después de realizar el análisis, se pueden acceder a los resultados a través de los atributos contador_logs, total_logs, y detalle_errores.

3. Ejemplo de Uso

```python
from analizador_logs import AnalizadorLogs
from procesamiento_logs import ProcesadorLogs

# Instanciamos el procesador de logs y procesamos los logs de un archivo
procesador = ProcesadorLogs("logs.txt")
logs_procesados = procesador.procesar_logs()

# Instanciamos el analizador de logs con los logs procesados
analizador = AnalizadorLogs(logs_procesados)

# Obtener el contador de logs por tipo
print("Conteo de logs por tipo:", analizador.contador_logs)

# Obtener el total de logs procesados
print("Total de logs procesados:", analizador.total_logs)

# Obtener los logs de tipo Error
print("Logs de tipo Error:", analizador.detalle_errores)
```

Este ejemplo muestra cómo se puede usar el analizador para contar los logs por tipo, obtener el total de logs procesados y extraer los logs de tipo Error.

4. Dependencias Externas

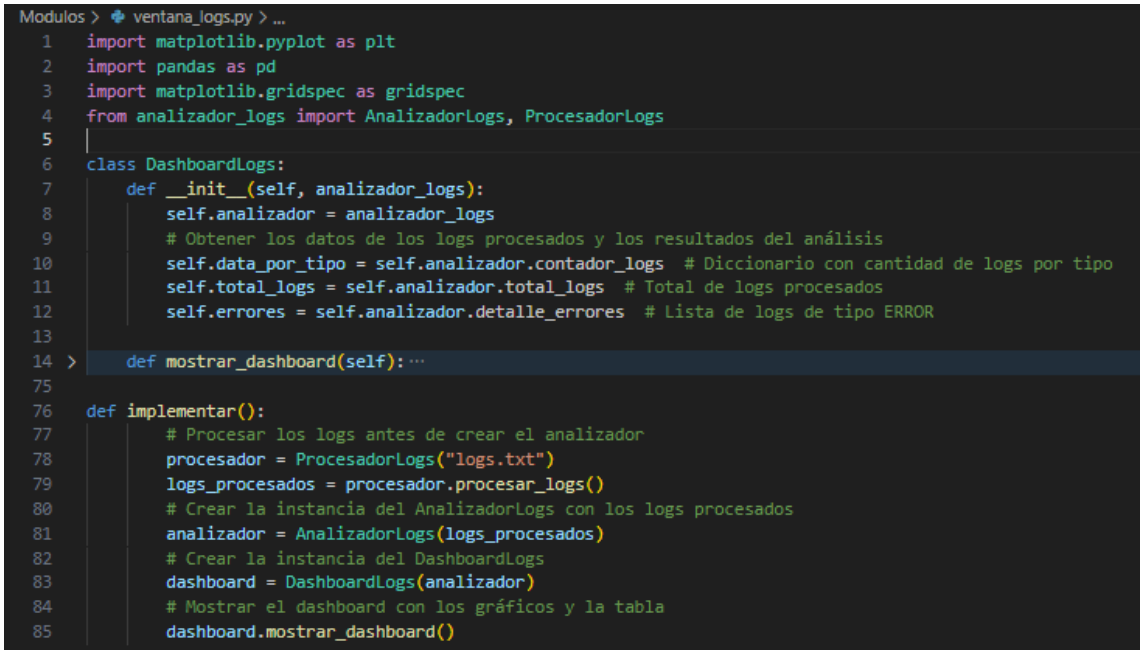
Este módulo no tiene dependencias externas adicionales, pero depende del módulo procesamiento_logs.py, que debe contener las clases Info, Warning, Debug, Error y ProcesadorLogs para funcionar correctamente.

5. Consideraciones

- **Formato Consistente de los Logs**: Los logs deben haber sido procesados previamente y deben estar representados por instancias de las clases Info, Warning, Debug y Error antes de ser analizados por AnalizadorLogs.
- **Accesibilidad de los Métodos**: Los métodos de la clase AnalizadorLogs permiten acceder a los datos analizados (conteos y logs de error), lo que facilita la obtención de estadísticas sobre los logs procesados.

Archivo: ventana_logs.py

Este módulo proporciona la clase DashboardLogs, que está diseñada para crear una interfaz de visualización (dashboard) de análisis de logs. Utiliza **matplotlib** para mostrar gráficos y una tabla con los datos extraídos de los logs procesados y analizados. La información visualizada incluye la cantidad de logs por tipo, los detalles de los logs de tipo Error, y los porcentajes de distribución de los diferentes tipos de logs.



1. Clase DashboardLogs

La clase DashboardLogs es responsable de visualizar los resultados del análisis de logs mediante gráficos y tablas. Utiliza la clase AnalizadorLogs para obtener los datos necesarios para la visualización.

Atributos:

- **analizador** (AnalizadorLogs): Instancia de la clase AnalizadorLogs que contiene los logs procesados y los resultados del análisis, como el conteo de logs por tipo y los detalles de los logs de error.
- **data_por_tipo** (dict): Diccionario que contiene la cantidad de logs por tipo (INFO, WARNING, DEBUG, ERROR).
- **total_logs** (int): Total de logs procesados.
- **errores** (list): Lista de objetos Error que representan los logs de tipo Error.

Métodos:

- **__init__(self, analizador_logs)**
  Constructor que recibe una instancia de AnalizadorLogs y obtiene los datos necesarios para la visualización, tales como:
  - ✓ data_por_tipo: El diccionario con la cantidad de logs por tipo.
  - ✓ total_logs: El total de logs procesados.
  - ✓ errores: Una lista con los logs de tipo Error.
- **mostrar_dashboard(self)**
  Método que crea y muestra un dashboard con tres secciones visuales:
  I. **Gráfico de Barras** (en la parte superior izquierda) que muestra la cantidad de logs por tipo (INFO, WARNING, DEBUG, ERROR).
  II. **Tabla de Reportes** (en la parte inferior izquierda) que lista los detalles de los logs de tipo Error (timestamp, IP, nivel, mensaje, código HTTP).
  III. **Gráfico Circular** (en la parte inferior derecha) que muestra los porcentajes de cada tipo de log con respecto al total de logs procesados.

El método utiliza **matplotlib** para crear los gráficos y **pandas** para generar la tabla de errores.

2. Flujo de Ejecución del Programa

1. **Instanciación de ProcesadorLogs:**
   Se crea un objeto ProcesadorLogs y se procesan los logs desde un archivo (por ejemplo, logs.txt).
2. **Instanciación de AnalizadorLogs:**
   Después de procesar los logs, se instancian los logs procesados en un objeto AnalizadorLogs, que realiza el análisis de los logs.
3. **Instanciación de DashboardLogs:**
   A continuación, se crea un objeto DashboardLogs, que recibe el analizador de logs y genera el dashboard visual con los resultados del análisis.
4. **Visualización del Dashboard:**
   El método mostrar_dashboard de DashboardLogs genera y muestra un gráfico de barras, una tabla de errores y un gráfico circular para visualizar el análisis de los logs.

3. Métodos de Visualización

- **Gráfico de Barras**:
  - ✓ Muestra el conteo de logs por tipo.

- ✓ Utiliza el método bar() de **matplotlib** para crear las barras.

- **Tabla de Reportes de Errores**:
  - ✓ Muestra los detalles de los logs de tipo Error.
  - ✓ Utiliza **pandas** para organizar los datos en un DataFrame y luego visualiza la tabla utilizando el método table() de **matplotlib**.
- **Gráfico Circular (Pie Chart)**:
  - ✓ Muestra los porcentajes de cada tipo de log con respecto al total de logs procesados.
  - ✓ Utiliza el método pie() de **matplotlib** para crear un gráfico circular.

4. Función implementar()

La función implementar() orquesta el flujo de trabajo, procesando los logs, analizando los resultados y mostrando el dashboard:

```
def implementar():
    # Procesar los logs antes de crear el analizador
    procesador = ProcesadorLogs("logs.txt")
    logs_procesados = procesador.procesar_logs()
    # Crear la instancia del AnalizadorLogs con los logs procesados
    analizador = AnalizadorLogs(logs_procesados)
    # Crear la instancia del DashboardLogs
    dashboard = DashboardLogs(analizador)
    # Mostrar el dashboard con los gráficos y la tabla
    dashboard.mostrar_dashboard()
```

1. **Procesar los logs**: Se procesan los logs desde un archivo (en este caso logs.txt).
2. **Analizar los logs**: Se instancian los logs procesados en un objeto AnalizadorLogs.
3. **Mostrar el Dashboard**: Se crea y muestra el dashboard utilizando la clase DashboardLogs.

5. Consideraciones

- **Dependencias Externas**: Este módulo requiere las siguientes bibliotecas externas:
  - ✓ matplotlib: Para crear gráficos.

    Documentación de la biblioteca matplotlib: https://matplotlib.org/stable/contents.html
  - ✓ pandas: Para manejar los datos en formato tabular y crear las tablas de reportes.

Documentación de la biblioteca pandas:
https://pandas.pydata.org/pandas-docs/stable/

- **Interfaz Visual**: El uso de **matplotlib** permite crear una interfaz visual interactiva que facilita la interpretación de los logs procesados.
- **Personalización de Gráficos**: Los colores de los gráficos y la tabla son personalizables. Actualmente se utilizan colores predeterminados para los gráficos y para la tabla de errores.

6. Ejemplo de Uso

```python
from ventana_logs import implementar

# Ejecutar la implementación para visualizar el dashboard
implementar()
```

Este código ejecutará el flujo completo: procesará los logs, analizará los datos y mostrará el dashboard visual con los gráficos y la tabla de reportes.

Archivo: Loglytics.py

El módulo Loglytics.py define la clase InterfazPrincipal, que es responsable de crear y gestionar la ventana principal de la aplicación "Loglytics", la cual permite a los usuarios importar archivos de logs, analizar los logs procesados y salir de la aplicación. La interfaz está construida utilizando **Tkinter** para la GUI (interfaz gráfica de usuario) y **Pillow** para cargar y mostrar el logo de la aplicación.

Modulos > Loglytics.py > ...

```python
import tkinter as tk
from tkinter import messagebox
from PIL import Image, ImageTk
import importar
import ventana_logs

class InterfazPrincipal:
    def __init__(self, root):···

    def cargar_logo(self):···

    def importar_logs(self):
        # Acción del botón Importar Logs
        try:
            # Llamamos a la función del módulo importar solo cuando el usuario presiona el botón
            importar.importar()  # Llama a la función que procesa el archivo
            messagebox.showinfo("Archivo procesado", "El archivo de logs ha sido importado y procesado correctamente.")
        except Exception as e:
            messagebox.showerror("Error", f"Hubo un error al procesar el archivo: {e}")

    def iniciar_analisis(self):
        # Acción del botón Iniciar Análisis
        try:
            # Llamamos a la función del módulo importar solo cuando el usuario presiona el botón
            messagebox.showinfo("Analisis terminado", "El archivo de logs ha sido analizado correctamente.")
            ventana_logs.implementar()  # Llama a la función que procesa el archivo
        except Exception as e:
            messagebox.showerror("Error", f"Hubo un error al analizar el archivo: {e}")

# Configuración de la ventana principal
root = tk.Tk()
app = InterfazPrincipal(root)
root.mainloop()
```

Dependencias

- **Tkinter**: Biblioteca estándar de Python para la creación de interfaces gráficas de usuario.
- **Pillow**: Biblioteca para la manipulación y carga de imágenes.

1.Clase InterfazPrincipal

Clase que gestiona la ventana principal de la aplicación, incluyendo los botones y la interacción con el usuario para importar y analizar logs.

Métodos:

- **__init__(self, root)**
  Constructor de la clase. Inicializa la ventana principal, configura el título y tamaño de la ventana, y coloca los elementos en el marco principal.

  Parámetros:

  ✓ root: Instancia de la ventana principal (Tk()), proporcionada por Tkinter.

- **cargar_logo(self)**
  Carga y ajusta la imagen del logo de la aplicación para mostrarla en la ventana. Si no se encuentra el archivo de la imagen, muestra una advertencia.

- **importar_logs(self)**
  Se ejecuta cuando el usuario presiona el botón "Importar Logs". Llama a la función importar.importar() (definida en el módulo importar.py) para procesar los logs desde el archivo. Si se completa correctamente, muestra un mensaje informativo; si ocurre un error, muestra un mensaje de error.

- **iniciar_analisis(self)**
  Se ejecuta cuando el usuario presiona el botón "Iniciar análisis". Llama a la función ventana_logs.implementar() (definida en el módulo ventana_logs.py) para generar el análisis visual de los logs procesados. Muestra un mensaje de éxito o error según corresponda.

2. Funcionalidad general

1. **Interfaz de usuario**:

   ✓ La ventana principal contiene un título ("LOGLYTICS") y tres botones:

   - **Importar Logs**: Llama a la función de importación de logs.
   - **Iniciar Análisis**: Llama a la función de análisis de logs.
   - **Salir**: Cierra la aplicación.

2. **Logo**:
    - ✓ Se carga y muestra una imagen del logo en la parte superior de la ventana. La imagen se ajusta al tamaño adecuado (100x100 píxeles) usando **Pillow**.
3. **Manejo de errores**:
    - ✓ Si ocurre algún error durante el proceso de importación o análisis de logs, se muestra un mensaje de error en un cuadro de diálogo.

3. Flujo de trabajo

1. Al iniciar la aplicación, se muestra la ventana con el logo y los botones.
2. Al presionar el botón **Importar Logs**, se procesan los archivos de logs y se muestra un mensaje informativo si se importa correctamente.
3. Al presionar el botón **Iniciar análisis**, se ejecuta el análisis de los logs y se muestran los resultados (gráficos y tabla) en una nueva ventana.
4. El botón **Salir** cierra la ventana y termina la ejecución de la aplicación.

4. Ejemplo de uso

```
import tkinter as tk
from Loglytics import InterfazPrincipal

# Crear la ventana principal
root = tk.Tk()

# Crear una instancia de la interfaz principal
app = InterfazPrincipal(root)

# Iniciar el bucle principal de Tkinter
root.mainloop()
```

Este código inicializa la ventana principal de la aplicación, mostrando la interfaz de usuario con los botones y el logo. El bucle principal de Tkinter (root.mainloop()) mantiene la ventana activa y gestionando las interacciones del usuario.

5. Archivos requeridos

- **logo.png**: Se requiere que exista un archivo de imagen llamado logo.png en el directorio donde se ejecuta el programa. Si el archivo no se encuentra, la aplicación muestra una advertencia.

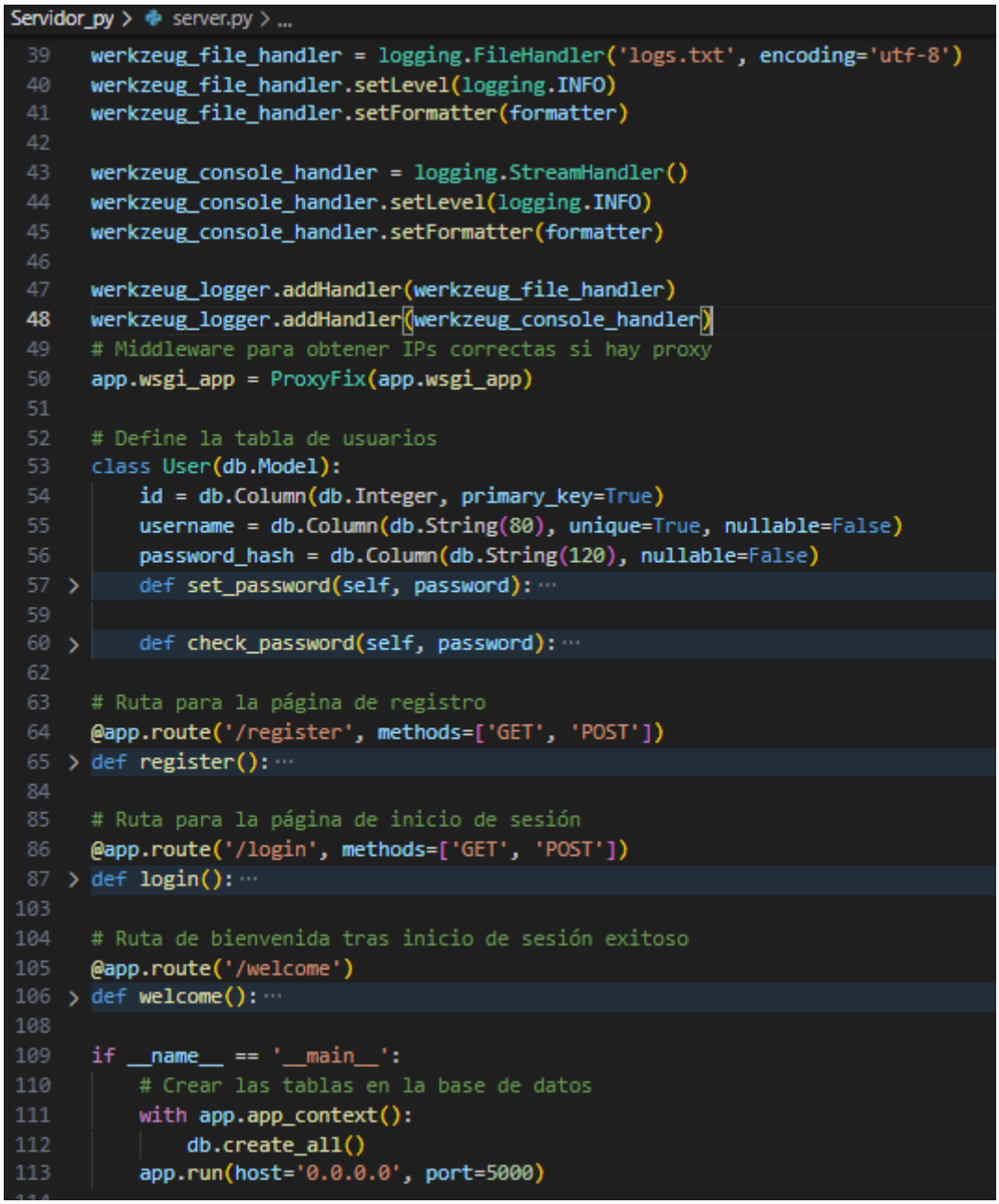
Contenido de la carpeta “Servidor_py”

Archivo: server.py

Este módulo define una aplicación web utilizando Flask que permite a los usuarios registrarse, iniciar sesión y visualizar una página de bienvenida. Además, configura un sistema de logging para capturar los eventos relevantes de la aplicación.

Servidor_py > server.py > ...

```python
import logging
from flask import Flask, request, render_template, redirect, url_for, flash
from flask_sqlalchemy import SQLAlchemy
from werkzeug.security import generate_password_hash, check_password_hash
from werkzeug.middleware.proxy_fix import ProxyFix

app = Flask(__name__)
app.config['SECRET_KEY'] = 'your_secret_key'
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///users.db'
db = SQLAlchemy(app)

# Configurar logger específico para la aplicación
app_logger = logging.getLogger("app_logger")
app_logger.setLevel(logging.INFO)

# Formato para los logs
formatter = logging.Formatter('%(asctime)s - %(message)s')

# Handler para guardar en archivo
file_handler = logging.FileHandler('logs.txt', encoding='utf-8')
file_handler.setLevel(logging.INFO)
file_handler.setFormatter(formatter)
app_logger.addHandler(file_handler)

# Handler para la consola
console_handler = logging.StreamHandler()
console_handler.setLevel(logging.INFO)
console_handler.setFormatter(formatter)
app_logger.addHandler(console_handler)

# Redirigir los logs de Werkzeug al logger de la aplicación con el mismo formato
werkzeug_logger = logging.getLogger('werkzeug')
werkzeug_logger.setLevel(logging.INFO)

# Sobrescribir el formateo de Werkzeug
for handler in werkzeug_logger.handlers[:]:
    werkzeug_logger.removeHandler(handler)
```



Dependencias

- **Flask**: Framework web ligero para Python.
- **Flask-SQLAlchemy**: Extensión de Flask que agrega soporte para bases de datos SQL.
- **Werkzeug**: Biblioteca para herramientas WSGI, como la gestión de contraseñas y middleware de proxy.
- **Logging**: Biblioteca estándar de Python para registrar mensajes y eventos.

1. Configuración de la Aplicación

- **Flask**:
  - ✓ SECRET_KEY: Clave secreta utilizada por Flask para proteger las sesiones.
  - ✓ SQLALCHEMY_DATABASE_URI: URI de la base de datos, en este caso, utiliza SQLite para almacenar los datos de usuario.
- **Logging**:
  - ✓ Se configura un logger para registrar los eventos tanto en consola como en un archivo (logs.txt).
  - ✓ Los logs de la aplicación y los logs de Werkzeug (el servidor de desarrollo de Flask) se redirigen a un archivo y a la consola.

2. Base de Datos

La base de datos está configurada utilizando **SQLAlchemy**, y se crea una tabla User para almacenar los usuarios registrados:

```python
# Define la tabla de usuarios
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True, nullable=False)
    password_hash = db.Column(db.String(120), nullable=False)

    def set_password(self, password):
        self.password_hash = generate_password_hash(password)

    def check_password(self, password):
        return check_password_hash(self.password_hash, password)
```

3. Rutas del Servidor

1. **/register (Método: GET, POST)**:
   - ✓ Permite a los usuarios registrarse proporcionando un nombre de usuario y contraseña.
   - ✓ Si el nombre de usuario ya existe en la base de datos, se muestra un mensaje de advertencia.
   - ✓ Si el registro es exitoso, se genera un log informativo.
2. **/login (Método: GET, POST)**:
   - ✓ Permite a los usuarios iniciar sesión con su nombre de usuario y contraseña.
   - ✓ Si la autenticación es exitosa, se redirige al usuario a la página de bienvenida.

- ✓ Si el inicio de sesión falla, se genera un log de advertencia.

3. **/welcome**:
   - ✓ Ruta de bienvenida que solo es accesible después de un inicio de sesión exitoso.

4. Middleware de Proxy

Se configura ProxyFix de werkzeug.middleware.proxy_fix para corregir las direcciones IP cuando la aplicación está detrás de un proxy, como en un entorno de producción.

5. Logging de la Aplicación

Se configura un sistema de logging para registrar eventos de la aplicación, tales como registros de usuarios, intentos fallidos de inicio de sesión y registros de accesos:

- Se utiliza un **FileHandler** para guardar los logs en un archivo (logs.txt).
- Se configura un **StreamHandler** para mostrar los logs en la consola.

Los logs se almacenan con un formato que incluye la hora del evento y el mensaje asociado.

6. Ejecución de la Aplicación

La aplicación se ejecuta en el host 0.0.0.0 (accesible desde cualquier IP) y en el puerto 5000. Se crea la base de datos y las tablas necesarias al arrancar la aplicación si no existen.

```
if __name__ == '__main__':
    # Crear las tablas en la base de datos
    with app.app_context():
        db.create_all()
    app.run(host='0.0.0.0', port=5000)
```

7. Estructura de Directorios

Este archivo debe estar en el mismo directorio donde se encuentren los templates HTML (login.html y register.html), ya que Flask los renderiza cuando se accede a las rutas correspondientes.

login.html - Página de Inicio de Sesión

Este archivo HTML proporciona la interfaz de usuario para la página de inicio de sesión de la aplicación web. Permite a los usuarios ingresar su nombre de usuario y contraseña para acceder a la aplicación. Si el usuario no tiene cuenta, también se proporciona un enlace para redirigir al usuario a la página de registro.

1. Estructura del archivo

Servidor_py > templates > login.html

```html
<!doctype html>
<html lang="es">
  <head>
    <title>Inicio de sesión</title>
  </head>
  <body>
    <h2>Inicio de sesión</h2>
    <form method="POST">
      <label>Nombre de usuario:</label><br>
      <input type="text" name="username" required><br>
      <label>Contraseña:</label><br>
      <input type="password" name="password" required><br><br>
      <button type="submit">Iniciar sesión</button>
    </form>
    <p><a href="{{ url_for('register') }}">¿No tienes cuenta? Regístrate aquí.</a></p>
  </body>
</html>
```

2. Explicación de los Elementos

1. **<!doctype html>**
   - Indica que el documento es un archivo HTML5.
2. **<html lang="es">**
   - Define el idioma del contenido del documento como español (es).
3. **<head>**
   - Contiene metadatos del documento. En este caso, solo se incluye un título para la página:
     - **<title>Inicio de sesión</title>**: Establece el título de la pestaña del navegador.
4. **<body>**
   - Contiene el contenido visible de la página:
     - **<h2>Inicio de sesión</h2>**: Título principal de la página de inicio de sesión.
     - **<form method="POST">**: Formulario que permite a los usuarios enviar datos al servidor mediante el método HTTP POST.
       - **Campos de entrada**:
         - **<label>Nombre de usuario:</label>**: Etiqueta para el campo de nombre de usuario.
         - **<input type="text" name="username" required>**: Campo de texto donde el usuario puede ingresar su nombre de usuario. El atributo required garantiza que el campo no se puede dejar vacío.

        - **<label>Contraseña:</label>**: Etiqueta para el campo de contraseña.
        - **<input type="password" name="password" required>**: Campo de texto donde el usuario puede ingresar su contraseña. El tipo password oculta los caracteres ingresados.
    - **<button type="submit">Iniciar sesión</button>**: Botón para enviar el formulario.
- **Enlace para redirigir a la página de registro**:
    - ** ¿No tienes cuenta? Regístrate aquí. **: Un enlace para redirigir al usuario a la página de registro si aún no tiene una cuenta.

3. Flask y Jinja2

El archivo utiliza la funcionalidad de **Jinja2** (el motor de plantillas de Flask) para generar la URL de la página de registro:

- **{{ url_for('register') }}**: Esta expresión genera la URL de la ruta registrada como register en la aplicación Flask. Esta es la URL a la que el usuario será redirigido si hace clic en el enlace para registrarse.

4. Interacción con el Backend

- El formulario de inicio de sesión envía los datos de los campos username y password al servidor a través de una solicitud HTTP POST cuando el usuario presiona el botón "Iniciar sesión".
- Si los datos coinciden con los registros en la base de datos, el servidor autentica al usuario y lo redirige a la página de bienvenida.

5. Dependencias

- **Flask**: Para el manejo de las rutas y renderizado de la plantilla.
- **Jinja2**: Para la generación dinámica de URLs y la inyección de contenido en las plantillas.

register.html - Página de Registro de Usuarios

Este archivo es la plantilla de la página de **registro** de usuarios en una aplicación web utilizando **Flask**. El formulario permite a los nuevos usuarios crear una cuenta proporcionando un nombre de usuario y una contraseña. Si el usuario ya tiene una cuenta, puede acceder a la página de inicio de sesión a través del enlace proporcionado.

1. Estructura del archivo

Servidor_py > templates > register.html

```html
<!doctype html>
<html lang="es">
  <head>
    <title>Registro</title>
  </head>
  <body>
    <h2>Registro</h2>
    <form method="POST">
      <label>Nombre de usuario:</label><br>
      <input type="text" name="username" required><br>
      <label>Contraseña:</label><br>
      <input type="password" name="password" required><br><br>
      <button type="submit">Registrarse</button>
    </form>
    <p><a href="{{ url_for('login') }}">Ya tienes una cuenta? Inicia sesión aquí.</a></p>
  </body>
</html>
```

2. Explicación de los Elementos

1. **Título de la página:**
    - <title>Registro</title>
    - El título de la página es **"Registro"**, lo que aparece en la pestaña del navegador.
2. **Formulario de Registro:**
    - <form method="POST">
    - El formulario envía los datos utilizando el método POST, lo cual es adecuado para enviar información sensible como contraseñas.
    - Los campos del formulario incluyen:
        - **Nombre de usuario** (input de tipo texto): Campo obligatorio (required) para que el usuario ingrese su nombre de usuario.
        - **Contraseña** (input de tipo password): Campo obligatorio para que el usuario ingrese su contraseña. El contenido de este campo será oculto.
    - **Botón de envío** (button de tipo submit): Permite enviar el formulario una vez que ambos campos hayan sido completados.
3. **Enlace para acceder a la página de inicio de sesión:**
    - Ya tienes una cuenta? Inicia sesión aquí.
    - El enlace redirige a los usuarios que ya tienen cuenta a la página de inicio de sesión. Utiliza la función url_for de **Flask** para generar la URL correcta de la vista login.

3. Funcionalidad:

- **Formulario POST**: Al enviar el formulario, los datos (nombre de usuario y contraseña) se envían al servidor, donde serán procesados en la vista register definida en el archivo server.py. Si el nombre de usuario no existe en la base de datos, se crea un nuevo usuario y se guarda en la base de datos.
- **Flask y Jinja2**: Usa **Flask** para renderizar el archivo HTML y la sintaxis **Jinja2** para generar dinámicamente el enlace al formulario de inicio de sesión.

4. Plantilla Jinja2:

- El archivo utiliza el sistema de plantillas **Jinja2**, el cual permite inyectar dinámicamente contenido desde el servidor.
    - {{ url_for('login') }}: Se utiliza para generar la URL correcta a la página de inicio de sesión. El nombre de la función login es pasado como argumento a url_for.

5. Flask Integration:

Este archivo es utilizado dentro de una aplicación Flask, donde el formulario se procesa en el backend (definido en el archivo server.py) y los mensajes de error o éxito se gestionan mediante las funciones de **Flask** y **Flash**.

6. Dependencias

- **Flask**: Para el manejo de las rutas y renderizado de la plantilla.
- **Jinja2**: Para la generación dinámica de URLs y la inyección de contenido en las plantillas.

Diagrama UML del proyecto