Spa Motos y Carros V1

Fecha de publicación: 29 de noviembre de 2024

Repositorio

URL Github

Autores

El proyecto se llevó a cabo por:

Maicol Esteven Zapata Serna - Lider del proyecto
Jaime Luis Tovar Olivero - Desarrollador Full Stack
Juan David Ruiz Murillo - Desarrollador Back End
Andres Ospina Ruiz - Desarollador Front End
Daniel Serna Alvarez -
Juan Jose Montoya Rua - Documentador de Software
Cristian Andres Zapata Cartagena - Tester QA

Descripción general

El proyecto se basa en desarrollar una aplicación para la gestión de un SPA Automotriz especializado en motos y carros. La aplicación está diseñada especialmente para ofrecer un servicio integral y de excelente al personal del SPA, mejorando procesos y teniendo una grata experiencia del usuario

Propósito

El propósito del proyecto es automatizar las operaciones de un SPA de motos y carros. Se busca brindar una solución eficiente que permita gestionar citas, registrar clientes y administrar la caja de servicios, entre otras funciones. Esta automatización mejorará la eficiencia operativa y la experiencia del cliente.

Objetivos

- **Facilitar la gestión de clientes:** El administrador del sistema tendrá la potestad ingresar clientes, modificarlos y realizar un seguimiento detallado de los clientes que más visitan el SPA
- **Generar reportes detallados:** Proveer información sobre servicios realizados, generando así un impacto significativo para mejorar el rendimiento del negocio
- **Potenciar la experiencia del cliente**: Con funcionalidades como recordatorios, notificaciones y encuestas de satisfacción.
- **Mejorar la gestión del SPA**: Organizar las tareas del personal, asignar servicios y optimizar el uso de recursos.

Tecnologías utilizadas

- **Frontend**: Para interfaz de usuario intuitiva y amigable, mediante el framework `Streamlit`, este genera una interacción eficiente con otras librerías, lo que permite una conexión sencilla con el backend
- **Backend**: Se desarrolla una gestión - pass
- **Base de Datos**: El almacenamiento seguro de datos relacionados con clientes, vehículos y servicios, se trabajará en la librería `Pandas` con creación de DataFrames y escritura de archivos CSV
- **Seguridad**: Se llevó a cabo una restricción de autenticación, creando un diseño de inicio de sesión, evitando así el acceso deliberado al sistema si no se tienen las credenciales de acceso

La aplicación está desarrollada 100 % en Python, y la mayoría de las operaciones se realizan mediante solicitudes de Streamlit

Uso y manejo del GitHub

Se generó ramificación del proyecto de acuerdo a los videos (cursos) compartidos por el profesor.
Se generó la rama MAIN, de esta se desprende la rama DEV, en la cual se generó la mayoría del código y la rama TEST, en la cual la persona de QA probó el código implemeantado y su funcionamiento

Requisitos

- Requisitos funcionales.
  1. Registrar y configurar diferentes servicios ofrecidos (lavado, encerado, mantenimiento etc.)
  2. Registrar nuevos clientes con la información básica (nombre, fecha de nacimiento, contacto etc), y que estos estén asociados a sus vehículos
  3. Consultar el historial de servicios realizado para cada cliente
  4. Llevar un registro de facturación y ventas del SPA
  5. Actualizar información del cliente si es necesario
  6. Interfaz para la navegación entre diferentes secciones del sistema como Inicio, Servicios, Clientes, Servicios
- Requisitos no funcionales:
  1. Rendimiento del sistema
  2. Usabilidad
  3. Mantenibilidad
  4. Compatibilidad

Instalacion y configuracion

Pasos de instalación:

Los requisitos básicos para la instalación del programa son Pandas, Python en su última versión 1.13.0, Streamlit, la instalación se ejecuta bajo el archivo requeriments, el cual almacena dos datos
`Streamlit` , `Pandas`

```
streamlit
pandas
```

Se ejecuta dentro de la terminal o CMD el siguiete comando:

`pip install -r requirements.txt`

Luego de esto para ejecutar la aplicación:

```
python -m streamlit run main.py
```

Funcionamiento general del programa

▼ back_util_functions.py

- Módulo de fecha y hora
  La función obtener_fecha_hora devuelve la fecha y la hora actuales formateadas:

  Fecha: "YYYY-MM-DD".
  Hora: "HH:MM:SS AM/PM".

- Clase Gestion_Clientes
  Esta clase permite gestionar información de clientes y registrar sus datos en un archivo CSV (clientes.csv).

  Principales funciones:
  cargar_dataframe: Carga los datos de clientes desde el archivo CSV o crea uno vacío si no existe.

  existe_cliente: Verifica si un cliente ya está registrado mediante su cédula.
  registrar_cliente: Agrega un nuevo cliente asignando automáticamente un ID único.

  editar_cliente: Modifica los datos de un cliente existente.

  listado_clientes: Genera un diccionario de cédulas y nombres para facilitar búsquedas rápidas.

- Clase Gestion_Vehiculos
  Administra la información de los vehículos registrados en un archivo CSV (vehicles.csv).

  Principales funciones:
  cargar_dataframe: Carga los datos de vehículos desde el CSV o crea uno vacío si no existe.

  registrar_vehiculo: Añade un nuevo vehículo al archivo CSV, asociándolo con un propietario.

  editar_vehiculo: Permite modificar información de un vehículo ya registrado.

  diccionario_tipos_vehiculos: Retorna un diccionario con

tipos de vehículos y sus categorías desde un archivo CSV separado (vehicles_types.csv).

- Clase Gestion_Servicios
  Maneja los servicios disponibles y sus precios en un archivo CSV (price_services.csv).

  Principales funciones:
  registrar_servicio: Registra un nuevo servicio con su precio, tipo de vehículo, y detalles.

  diccionario_precios_categoria: Genera un diccionario jerárquico de servicios organizados por tipo de vehículo y categoría.

  dataframe_temp_services: Procesa servicios seleccionados por un cliente, asociándolos con precios y generando un DataFrame temporal.

  cargar_servicio_vehiculo: Registra servicios utilizados por un vehículo y genera una factura.
- Clase Gestion_Usuarios
  Administra a los usuarios de la aplicación en un archivo CSV (users.csv).

  Principales funciones:

  registrar_usuario: Crea un nuevo usuario asignándole un rol y un estado activo.
  cargar_dataframe: Carga datos desde el CSV o crea un archivo vacío si no existe.
- Organización General
  Este sistema utiliza Pandas para leer y escribir datos en archivos CSV.
  Manipular DataFrames para procesar y relacionar información

▼ clients.py

De esta clase sale un registro (dB) en formato CSV de los CLIENTES

- **Importaciones:**
  import streamlit as st
  from datetime import datetime
  from dateutil.relativedelta import relativedelta
  from time import sleep
  from navigation import make_sidebar

from pages.back_util_functions import Gestion_Clientes, Gestion_Vehiculos
from pages.front_util_functions import validate_email, validate_cedula, validate_celular
Se importan las bibliotecas principales:

Streamlit (st): Para crear la interfaz web.

datetime y **dateutil.relativedelta**: Para el manejo de fechas.

sleep: Para agregar pausas breves en la ejecución.

- **Inicialización de la aplicación:**

  make_sidebar()
  st.session_state.df_state_clientes = False
  st.session_state.df_state_vehiculos = False
  st.session_state.validaciones_data = True

  Se inicializan variables globales utilizando st.session_state:

  df_state_clientes y df_state_vehiculos: Indican si se han cargado los datos de clientes y vehículos.

  validaciones_data: Bandera para verificar si las validaciones de datos han sido exitosas.

- **Manejar el texto ingresado:**

  if "input_text" not in st.session_state:
  st.session_state.input_text = ""
  def update_text():
  st.session_state.input_text =
  st.session_state.input_text.upper()

  Se define una variable de estado (`input_text`) que almacena texto ingresado por el usuario.

  `update_text()` : Convierte automáticamente el texto ingresado a mayúsculas.

- **Agregar nuevo cliente:**

  @st.dialog("Agregar nuevo cliente")

def btn_agregar():
...

Esta función implementa la lógica para agregar un nuevo cliente:

Se solicita la información del usuario:

`cedula`: Verificada con `validate_cedula`.

`nombre`, `telefono`, `fecha_nacimiento`, `email`: Validaciones adicionales para teléfono y correo.

Verificaciones: Revisa si la cédula es válida y si ya existe en el sistema

Guardar clientes: Si las validaciones son exitosas, se guarda el cliente utilizando `gestion_clientes`

- **Editar cliente:**

  @st.dialog("Editar cliente")
  def btn_editar(dict_values):
  ...

  Este proceso permite editar los datos de un cliente existente

  Mostrar datos actuales:
  Se cargan los datos del cliente seleccionado en campos de entrada.
  Los datos actualizados se envían a `Gestion_Clientes` para guardarse

- **Agregar vehículo a un cliente:**

  @st.dialog("Agregar vehículo a cliente")
  def btn_agregar_vehiculo(dict_values):
  ...
  Gestiona la creación de un vehículo asociado a un cliente:

  Entradas requeridas: Placa, propietario, categoría,

marca, modelo y cilindraje, además de esto validaciones específicas según el tipo de vehículo

Guardar vehículo: Se registra el vehículo con Gestion_Vehiculos

- **Visualizar clientes:**

st.header('Administración de Clientes')
left, middle, right = st.columns(3)
if not st.session_state.df_state_clientes:
df_clientes = Gestion_Clientes()
st.session_state.df_clientes =
df_clientes.cargar_dataframe()
st.session_state.df_state_clientes = True

Tabla de clientes: Se utiliza st.dataframe para mostrar los clientes en una tabla interactiva.

A parte de esto se cuenta con los siguientes botones de acción

`Agregar`: Abre el formulario para agregar un cliente.

`Editar`: Abre el formulario para editar el cliente seleccionado.

`Agregar Vehículo`: Abre el formulario para asociar un vehículo al cliente

- **Visualización de vehículos del cliente seleccionado:**

st.write('*Seleccione un cliente para ver sus vehículos*')
if dict_clientes_values is not None:
st.subheader(f"Vehículos de
{dict_clientes_values['nombre']}")

if not st.session_state.df_state_vehiculos:
    df_vehiculos = Gestion_Vehiculos()
    st.session_state.df_vehiculos =
df_vehiculos.dataframe_front(str(dict_clientes_values['id'
    st.session_state.df_state_vehiculos = True

Filtrar vehículos: Si un cliente está seleccionado, se muestran los vehículos asociados.

Tabla interactiva: Los vehículos se presentan en una tabla similar a la de los clientes.

▼ config.py

De esta clase sale un registro (dB) en formato CSV de los CLIENTES

- **Importaciones:**

  ```
  from navigation import make_sidebar
  import streamlit as st
  from time import sleep
  import pandas as pd
  import os
  from PIL import Image
  from pages.back_util_functions import Gestion_Usuarios
  import json
  ```

  Streamlit (st): Para la construcción de la interfaz web.

  pandas: Para manejar datos tabulares como los usuarios.

  Pillow (Image): Para trabajar con imágenes.

  json: Para manejar datos almacenados en formato JSON.

  navigation.make_sidebar: Crea una barra lateral para navegación (importada de un módulo externo).

  time.sleep: Pausa breve para procesos de retroalimentación al usuario.

  Gestion_Usuarios: Clase que maneja la lógica relacionada con usuarios (importada).

- **Barra lateral y inicialización**

  ```
  make_sidebar()
  st.session_state.df_state = False
  ```

  `make_sidebar`: Inserta una barra lateral (no está detallada en este código).

  `st.session_state.df_state`: Variable global que indica si los

datos de usuarios ya han sido cargados.

- **Agregar nuevo usuario:**

  if "input_text" not in st.session_state:
  st.session_state.input_text = ""
  def update_text():
  st.session_state.input_text =
  st.session_state.input_text.upper()

  Se define una variable de estado (`input_text`) que almacena texto ingresado por el usuario.

  `update_text()` : Convierte automáticamente el texto ingresado a mayúsculas.

- **Agregar nuevo cliente:**

  @st.dialog("Agregar nuevo cliente")
  def btn_agregar():
  ...

  Esta función implementa la lógica para agregar un nuevo cliente:

  Se solicita la información del usuario:

  `cedula` : Verificada con `validate_cedula` .

  `nombre` , `telefono` , `fecha_nacimiento` , `email` : Validaciones adicionales para teléfono y correo.

  Verificaciones: Revisa si la cédula es válida y si ya existe en el sistema

  Guardar clientes: Si las validaciones son exitosas, se guarda el cliente utilizando `gestion_clientes`

- **Editar cliente:**

  @st.dialog("Editar cliente")
  def btn_editar(dict_values):

...

Este proceso permite editar los datos de un cliente existente

Mostrar datos actuales:
Se cargan los datos del cliente seleccionado en campos de entrada.
Los datos actualizados se envían a `Gestion_Clientes` para guardarse

- **Agregar vehículo a un cliente:**

  @st.dialog("Agregar vehículo a cliente")
  def btn_agregar_vehiculo(dict_values):
  ...
  Gestiona la creación de un vehículo asociado a un cliente:

  Entradas requeridas: Placa, propietario, categoría, marca, modelo y cilindraje, además de esto validaciones específicas según el tipo de vehículo

  Guardar vehículo: Se registra el vehículo con Gestion_Vehiculos

- **Visualizar clientes:**

  st.header('Administración de Clientes')
  left, middle, right = st.columns(3)
  if not st.session_state.df_state_clientes:
  df_clientes = Gestion_Clientes()
  st.session_state.df_clientes =
  df_clientes.cargar_dataframe()
  st.session_state.df_state_clientes = True

  Tabla de clientes: Se utiliza st.dataframe para mostrar los clientes en una tabla interactiva.

  A parte de esto se cuenta con los siguientes botones de acción

  `Agregar`: Abre el formulario para agregar un cliente.

  `Editar`: Abre el formulario para editar el cliente seleccionado.

`Agregar Vehículo`: Abre el formulario para asociar un vehículo al cliente

- **Visualización de vehículos del cliente seleccionado:**

st.write('*Seleccione un cliente para ver sus vehículos*')
if dict_clientes_values is not None:
st.subheader(f"Vehículos de
{dict_clientes_values['nombre']}")

if not st.session_state.df_state_vehiculos:
    df_vehiculos = Gestion_Vehiculos()
    st.session_state.df_vehiculos =
df_vehiculos.dataframe_front(str(dict_clientes_values['id'
    st.session_state.df_state_vehiculos = True

Filtrar vehículos: Si un cliente está seleccionado, se muestran los vehículos asociados.

Tabla interactiva: Los vehículos se presentan en una tabla similar a la de los clientes.

▼ front_util_functions.py

- **Importaciones:**

import pandas as pd
import re

pandas (pd): Para manejar datos tabulares almacenados en un archivo CSV.
re (módulo de expresiones regulares): Para validar patrones en cadenas, como correos electrónicos y números.

- **Validación de usuario y contraseña**

def validate_user(user, password):
df = pd.read_csv('pages/data/users.csv')

user_exist = df[(df['usuario'] == user) &
(df['contrasena'] == password)]

Entrada:

user: Nombre de usuario ingresado.
password: Contraseña ingresada.
Acción:
Carga los datos de usuarios desde un archivo CSV (users.csv).
Busca coincidencias en las columnas usuario y contraseña usando filtros de pandas.

```
if not user_exist.empty:
    if user_exist['esta_activo'].iloc[0] == True:
        return True, user_exist['rol'].iloc[0], 'Inicio de
Sesión Exitoso'
    else:
        return False, None, f"Acceso Denegado. El usuario
\\'{user}\\' se encuentra inactivo, comuníquese con el
administrador para mas detalles."
else:
    return False, None, f'Acceso Denegado. Usuario o
contraseña incorrectos'
```

Salida:
Si el usuario existe y está activo:
Devuelve True, el rol del usuario (admin o usuario), y un mensaje de éxito.
Si el usuario existe pero está inactivo:
Devuelve False, None (sin rol) y un mensaje indicando que el usuario está inactivo.
Si el usuario o contraseña no coinciden:
Devuelve False, None y un mensaje de error.

- **Validación de correos electrónicos:**

```
def validate_email(email):
pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-
Z]{2,}(?:\.[a-zA-Z]{2,})*$'

if re.match(pattern, email):
    return True
else:
    return False
```

Entrada: Un correo electrónico (email).
Acción:
Usa una expresión regular para validar el formato del correo:
Inicia con letras, números o caracteres especiales

(._%+-).
Sigue con un @, luego un dominio, y termina con una extensión válida (como .com, .org).
Permite subdominios adicionales (example.co.uk).
re.match(pattern, email) verifica si el correo coincide con el patrón.
Salida:
Devuelve True si el correo es válido.
Devuelve False si no cumple el patrón.

- **Validación de cédulas:**

```
def validate_cedula(cedula):
if not re.fullmatch(r'^\d{1,10}$', cedula):
return False
return True
```

Entrada: Una cédula (cedula).
Acción:
Usa una expresión regular para verificar que:
Consiste únicamente en dígitos (^\d+$).
Tiene entre 1 y 10 dígitos ({1,10}).
re.fullmatch asegura que toda la cadena cumpla con este formato.
Salida:
Devuelve True si la cédula es válida.
Devuelve False si no lo es.

- **Validación de números de celular:**

```
def validate_celular(numero):
if len(numero) == 10 and numero.isdigit():
if int(numero[0]) == 3 or int(numero[0]) == 6:
return True
return False
```

Entrada: Un número de celular (numero).
Acción:
Comprueba que:
Tiene exactamente 10 dígitos (len(numero) == 10).
Está compuesto solo por números (numero.isdigit()).
Comienza con 3 o 6 (int(numero[0])).
Salida:
Devuelve True si cumple todos los criterios.

Devuelve False en caso contrario.

- **Agregar vehículo a un cliente:**

  @st.dialog("Agregar vehículo a cliente")
  def btn_agregar_vehiculo(dict_values):
  ...
  Gestiona la creación de un vehículo asociado a un cliente:

  Entradas requeridas: Placa, propietario, categoría, marca, modelo y cilindraje, además de esto validaciones específicas según el tipo de vehículo

  Guardar vehículo: Se registra el vehículo con Gestion_Vehiculos

- **Visualizar clientes:**

  st.header('Administración de Clientes')
  left, middle, right = st.columns(3)
  if not st.session_state.df_state_clientes:
  df_clientes = Gestion_Clientes()
  st.session_state.df_clientes =
  df_clientes.cargar_dataframe()
  st.session_state.df_state_clientes = True

  Tabla de clientes: Se utiliza st.dataframe para mostrar los clientes en una tabla interactiva.

  A parte de esto se cuenta con los siguientes botones de acción

  `Agregar`: Abre el formulario para agregar un cliente.

  `Editar`: Abre el formulario para editar el cliente seleccionado.

  `Agregar Vehículo`: Abre el formulario para asociar un vehículo al cliente

- **Visualización de vehículos del cliente seleccionado:**

  st.write('*Seleccione un cliente para ver sus vehículos*')
  if dict_clientes_values is not None:

st.subheader(f"Vehículos de
{dict_clientes_values['nombre']}")

if not st.session_state.df_state_vehiculos:
    df_vehiculos = Gestion_Vehiculos()
    st.session_state.df_vehiculos =
df_vehiculos.dataframe_front(str(dict_clientes_values['id'
    st.session_state.df_state_vehiculos = True

Filtrar vehículos: Si un cliente está seleccionado, se muestran los vehículos asociados.

Tabla interactiva: Los vehículos se presentan en una tabla similar a la de los clientes.

▼ principal.py

- **Importaciones:**

  from navigation import make_sidebar
  import streamlit as st
  import pandas as pd
  from time import sleep
  from pages.back_util_functions import
  Gestion_Servicios, Gestion_Vehiculos
  import logging
  from streamlit.logger import get_logger

  navigation y make_sidebar: Función personalizada para añadir un menú lateral de navegación.
  Streamlit (st): Framework utilizado para crear interfaces interactivas.
  pandas (pd): Biblioteca para manipular datos tabulares.
  Gestion_Servicios y Gestion_Vehiculos: Clases externas encargadas de gestionar servicios de lavado y datos de vehículos.
  logging: Configura el nivel de registro para evitar mensajes de error innecesarios.

- **Variables iniciales**

  st.session_state.df_temp_services = False
  if 'reset_services' not in st.session_state:
  st.session_state.reset_services = False

session_state: Se utiliza para almacenar datos temporales durante la sesión (como servicios seleccionados).
df_temp_services: Bandera para rastrear si ya se ha generado un DataFrame temporal de servicios.
reset_services: Bandera para reiniciar las selecciones de servicios cuando sea necesario.

- **Función btn_iniciar_servicio**

  Propósito:
  Registrar la transacción de servicio en el sistema.
  Parámetros:
  dataframe_in: Información tabular de los servicios seleccionados.
  dict_in: Información adicional del vehículo y propietario.
  Proceso:
  Usa la clase Gestion_Servicios para cargar los datos del servicio en el sistema (guardado o procesamiento interno).

- **Carga de datos de vehículos (motos y carros):**

  placas = Gestion_Vehiculos()
  placas = placas.listado_placas_clientes()

  Se instancia la clase `Gestion_Vehiculos` para cargar un **diccionario de placas** con la siguiente estructura:

  placas = {
  "ABC123": ["cedula", "nombre", "tipo_vehiculo", "categoria"],
  ...
  }

- **Seleccion de vehículo y datos asociados:**

  vehiculo = left.selectbox('Vehículo (Placa)', options=placas.keys(), key="selectbox_2")
  tipo_vehiculo_input = middle.text_input('Tipo de Vehículo', value=placas[vehiculo][2], disabled=True)
  cedula = middle.text_input('Cédula Propietario',

```
value=placas[vehiculo][0], disabled=True)
categoria_input = right.text_input('Categoría',
value=placas[vehiculo][3], disabled=True)
nombre = right.text_input('Nombre Propietario',
value=placas[vehiculo][1], disabled=True)
```

Selección de placa: Despliega una lista de vehículos disponibles (placas) mediante un selectbox.
Información adicional:
Al seleccionar una placa, se muestran automáticamente los datos asociados al vehículo:
Tipo de vehículo (ej. "Sedán").
Cédula del propietario.
Categoría del vehículo (ej. "Familiar").
Nombre del propietario.
Estos campos están deshabilitados (disabled=True) para evitar edición.

- **Seleccionar los servicios:**

```
servicios_precios = Gestion_Servicios()
servicios_precios =
servicios_precios.diccionario_precios_categoria()
```

diccionario_precios_categoria devuelve un diccionario con los servicios disponibles por categoría y tipo de vehículo. Ejemplo:

```
servicios_precios = {
"Sedán": {
"Familiar": {"Lavado Completo": 20000, "Lavado
Motor": 10000},
"General": {"Encerado": 15000, "Desinfección": 12000}
},
...
}
```

- **Opciones de servicios principales y adicionales:**

```
servicios = st.multiselect(
'Seleccione los servicios',
servicios_precios[placas[vehiculo][2]][placas[vehiculo]
[3]].keys(),
default=st.session_state.get('selected_services', []),
```

```
key='selected_services')
servicios_adicionales = st.multiselect(
'Seleccione los servicios adicionales',
servicios_precios[placas[vehiculo][2]]['General'].keys(),
default=st.session_state.get('selected_aditionals_service
[]),
key='selected_aditionals_services')
```

Servicios principales: Basados en el tipo y categoría del vehículo.
Servicios adicionales: Opciones genéricas aplicables a cualquier vehículo.
Se utiliza multiselect para permitir seleccionar múltiples servicios.

- **Se crea diccionario temporal:**

```
dict_temp_services =  {
'placa' : vehiculo,
'tipo_vehiculo': placas[vehiculo][2],
'categoria': placas[vehiculo][3],
'cedula' : placas[vehiculo][0],
'servicio': servicios+servicios_adicionales
}
```

Almacena la información seleccionada en un diccionario temporal (`dict_temp_services`), incluyendo la placa, tipo de vehículo, categoría, cédula y servicios seleccionados.

- **Botón de inicio:**

```
if len(dict_temp_services['servicio']) > 0:
st.subheader('Resumen')
if not st.session_state.df_temp_services:
servicios_temp = Gestion_Servicios()
df_servicios_temp =
servicios_temp.dataframe_temp_services(dict_temp_serv
st.dataframe(df_servicios_temp[['servicio',
'precio_formateado']])

if "btn_init_service" not in st.session_state:
    if st.button('Iniciar Servicio', type='primary'):
        btn_iniciar_servicio(df_servicios_temp,
```

dict_temp_services)
st.toast('Se ha iniciado un servicio exitosamente')
sleep(1)
st.session_state.reset_services = True
st.rerun()
else:
st.session_state.df_temp_services = False

Generación del resumen:
Si hay servicios seleccionados, se crea un DataFrame temporal (dataframe_temp_services) para visualizar un resumen de los servicios y precios.
Usa st.dataframe para mostrar los datos en un formato tabular interactivo.
Iniciar servicio:
Si se presiona el botón "Iniciar Servicio":
Llama a btn_iniciar_servicio para registrar la transacción.
Muestra un mensaje de confirmación (st.toast).
Reinicia el estado de selección (reset_services) y recarga la página (st.rerun).

▼ services.py

- **Importaciones y configuración inicial:** Se importan módulos para la interfaz y las clases `Gestion_Servicios` y `Gestion_Vehiculos`, que gestionan datos relacionados con servicios y vehículos.
  Se inicializan variables en `st.session_state` para mantener el estado durante la sesión, como el estado del DataFrame de servicios (`df_state_servicios_precios`).
- **Agregar un nuevo servicio** `btn_agregar`:

  Permite ingresar los datos de un nuevo servicio: nombre, precio, categoría, tipo de vehículo, y detalles.

  Usa `Gestion_Vehiculos` para obtener las categorías y tipos de vehículos.

  Guarda el servicio en el sistema utilizando `Gestion_Servicios.registrar_servicio`.

  Incluye validaciones básicas y muestra mensajes de éxito o error.
- **Editar un servicio** `btn_editar`:

  Permite editar los datos de un servicio seleccionado.

Utiliza un formulario con datos precargados del servicio (como nombre, teléfono y email).

Guarda los cambios usando `Gestion_Clientes.editar_cliente`.**Encabezado y botones**:Muestra el título "Administración de Servicios".

- **Servicio principal:**

  Muestra el título "Administración de Servicios".
  Incluye un botón para agregar servicios (Agregar) que activa el diálogo correspondiente.
  Carga inicial del DataFrame:
  Si no se ha cargado previamente, se obtiene un DataFrame con los servicios existentes usando Gestion_Servicios.cargar_dataframe y se almacena en session_state.
  Tabla interactiva:
  Muestra los servicios en una tabla con columnas configuradas (ej. ID, servicio, tipo de vehículo, precio, etc.).
  Permite seleccionar filas individuales para editar o realizar acciones futuras.

▼ vehicles.py

Configuración inicial

`make_sidebar()`: Crea una barra lateral personalizada para la navegación (función importada).

Inicializa el estado de sesión con:

`df_state`: Determina si el DataFrame de vehículos ya está cargado.

`input_text`: Una variable para texto interactivo (ej. en mayúsculas).

El diálogo permite modificar los datos de un vehículo seleccionado en una tabla. Funciona de la siguiente manera:

Datos iniciales mostrados:

- La placa aparece como un campo deshabilitado, pero puede habilitarse para edición si se marca la casilla *Editar placa*.
- Se utiliza el método `Gestion_Clientes.listado_clientes()` para generar una lista de propietarios y asignar un propietario al vehículo.

Categorías y tipos de vehículos:

- Usa `Gestion_Vehiculos.diccionario_tipos_vehiculos()` para mostrar listas desplegables de tipos y categorías.
- Si el tipo es "Moto", ajusta dinámicamente los valores permitidos de cilindraje usando un diccionario.

Guardar cambios:

- Al guardar, llama a `Gestion_Vehiculos.editar_vehiculo` para actualizar los datos.
- Si se edita la placa, se pasa un argumento adicional (`placa_nueva`) al método.
- Muestra un mensaje de éxito y recarga la página con `st.rerun()`.

Visualización de vehículos

- **Carga del DataFrame**:

  Usa `Gestion_Vehiculos.dataframe_front_gestion()` para cargar los datos de vehículos y los almacena en `st.session_state.df`.

  El estado de carga está controlado por `st.session_state.df_state` para evitar recargas innecesarias.

- **Tabla interactiva**:

  Muestra una tabla de vehículos con columnas configuradas (ej. placa, tipo, propietario, cilindraje, etc.).

  Permite la selección de una fila mediante `selection_mode=['single-row']`.

Botón para editar vehículos

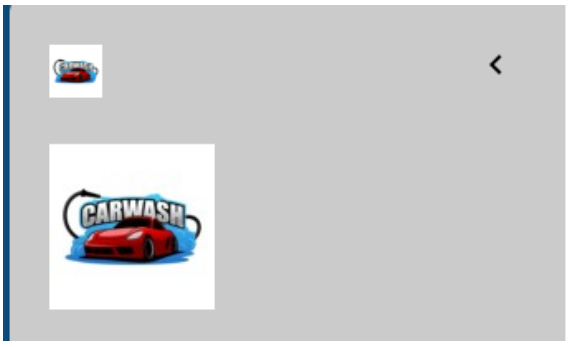
- Si se selecciona una fila en la tabla:
  - Extrae los datos del vehículo seleccionado y los pasa al diálogo de edición (`btn_editar_vehiculo`).
- Si no se selecciona ningún vehículo:
  - Muestra un mensaje de alerta indicando que primero debe seleccionarse un registro.

▼ main.py

Carga de recursos
nombre_empresa.json:
Se utiliza para obtener el nombre de la empresa desde un archivo JSON.
Ejemplo del contenido del archivo:
json

{
"nombre": "Mi Empresa"
}

logo.png:
Si existe, se carga y muestra en la interfaz, en la columna izquierda.



- **Diseño de la interfaz**

La pantalla está dividida en dos columnas (left y right):

Columna izquierda:
Muestra el logo de la empresa si el archivo logo.png existe.
Columna derecha:
Muestra el nombre de la empresa como título.
Proporciona campos para que el usuario ingrese su nombre de usuario y contraseña.

- Validación de credenciales:
  Llama a la función validate_user(username, password) que retorna una tupla con los siguientes valores:
  validacion[0]: True si las credenciales son válidas; de lo contrario, False.
  validacion[1]: Rol del usuario (si la validación es exitosa).
  validacion[2]: Mensaje de éxito o error.

- Si las credenciales son válidas:
  Almacena el estado de inicio de sesión en st.session_state:
  logged_in: Se establece como True.
  role: Almacena el rol del usuario autenticado.
  Muestra un mensaje de éxito (st.toast).
  Redirige al usuario a la página principal (pages/principal.py) usando st.switch_page.

- Si las credenciales no son válidas:
  Muestra un mensaje de error en pantalla (st.error).

Comportamiento dinámico

- Persistencia del estado:
  Utiliza st.session_state para recordar que el usuario ha iniciado sesión y cuál es su rol. Esto facilita controlar el acceso a otras páginas sin necesidad de volver a iniciar sesión.

- Redirección automática:
  La función st.switch_page redirige al usuario a la página principal si la validación es exitosa.

▼ navigation.py

Se utiliza para dar contexto a la página, en un conjunto de funciones se crea la barra personalizada en `Streamlit`

Importaciones

- `streamlit` **(st)**: Proporciona herramientas para construir la interfaz de usuario.
- `time.sleep`: Se utiliza para pausar brevemente el flujo (por ejemplo, al cerrar sesión).
- `get_script_run_ctx`: Extrae el contexto de ejecución actual del script (como información de la página).
- `get_pages`: Obtiene las páginas registradas en la aplicación de Streamlit.
- `os` **y** `json`: Permiten manejar archivos y cargar datos en formato JSON (como el nombre de la empresa).

Función `get_current_page_name`

Propósito:

Devuelve el nombre de la página actual que el usuario está viendo.

Proceso:

Obtiene el contexto de ejecución actual con `get_script_run_ctx()`.

Recupera todas las páginas de la aplicación con `get_pages("")`.

Usa el `page_script_hash` (hash único para cada página) del contexto para identificar el nombre de la página actual.

Función `make_sidebar`

Propósito:

Crea la barra lateral personalizada dependiendo del estado de sesión del usuario y su rol.

Proceso:

Carga de recursos:

- Obtiene el nombre de la empresa desde un archivo JSON (`nombre_empresa.json`).
- Carga el logo desde `logo.png`.

Construcción de la barra lateral:

Muestra el logo y el nombre de la empresa.

Dependiendo del estado de sesión:

Si el usuario está autenticado (`logged_in` en `st.session_state`):

Genera enlaces a páginas específicas según el rol del usuario (`admin` o `usuario`).

Muestra un botón **"Cerrar Sesión"**, que al presionarlo llama a la función `logout()`.

Si el usuario **no está autenticado** y la página actual no es `"main"`, redirige automáticamente a la página principal (`main.py`) usando `st.switch_page`.

Función `logout`

Propósito:

Cerrar la sesión del usuario y redirigirlo a la página principal.

Proceso:

1. Marca al usuario como no autenticado (`st.session_state.logged_in = False`).
2. Muestra un mensaje de cierre de sesión.
3. Pausa brevemente con `sleep(0.5)` para un efecto visual.
4. Redirige a la página principal (`main.py`) usando `st.switch_page`.

Límites de solicitud

Para garantizar una experiencia de desarrollo coherente para todos los usuarios, para el proyecto no se tienen limitantes, ya que todo se ejecuta desde la web, desde un localhost

Conclusión

El desarrollo del sistema de gestión para un SPA fue una experiencia enriquecedora, tanto desde el punto de vista técnico como humano. Durante el proceso, se buscó resolver las necesidades operativas de una manera eficiente, reduciendo la carga de trabajo manual y permitiendo que tanto los empleados como los clientes se beneficiaran de una experiencia más ágil y profesional.

El proyecto no solo mejoró la organización interna del negocio, sino que también abrió las puertas a una gestión más precisa y transparente. Al centralizar toda la información en un solo sistema, los administradores ahora tienen acceso a datos en tiempo real, lo que facilita la toma de decisiones y el análisis del desempeño del SPA.

Trabajar en este proyecto fue un recordatorio de lo importante que es comprender las dinámicas específicas de un negocio antes de desarrollar soluciones tecnológicas. No se trató solo de escribir código, sino de escuchar, analizar y reflejar los procesos operativos del SPA en cada funcionalidad del sistema.

Por supuesto, este es solo el inicio. Hay muchas oportunidades para seguir mejorando, sin embargo, lo más valioso de este proyecto es que demuestra cómo la tecnología puede transformar un negocio, facilitando el trabajo y generando valor tanto para quienes lo operan como para quienes lo utilizan.

En definitiva, esta experiencia fue un paso adelante no solo para el SPA, sino también para nosotros como desarrolladores, aprendiendo de cada desafío y creciendo con cada solución.