Luxury Wheels App

From: Tiago Luís Abreu Silva

Education: Professional Training in Python Programming

Training institution: Tokio School

Python: version 3.12.0

Final project of the course

Context:

Luxury Wheels is a car rental company that aims to improve its management and sales. In other words, it will be necessary to develop a website (Proposal A), where clients should be able to register and rent a vehicle, and an application (Proposal B) to enhance fleet management, allowing the company to manage all its vehicles.

Chosen Proposal

Proposal B: Assume responsibility for developing an application through which Luxury Wheels can manage its vehicle fleet, and execute the following functionalities:

1. Insert vehicles, clients, reservations, and select the payment methods that clients can use.

2. Manage (register, modify, list, remove) vehicles, clients, reservations, and select the payment methods that clients can use.

3. Export information in Excel or CSV format for vehicles, clients, reservations, and payment methods.

4. Initial dashboard with information such as:
    - Rented vehicles and the remaining days of the reservation;
    - Last registered clients;
    - Quantity of available vehicles, by type and category;
    - Monthly reservations and the total financial amount;
    - Vehicles with upcoming inspection expiry date (15 days);
    - Vehicles with upcoming legalization expiry date (15 days).

Technical Criteria

1. All the necessary information for the project must be stored in SQL databases (module 5, exercise 4):
    - Vehicle;
    - Clients;
    - Reservations;
    - Payment Methods.

2. All information about the vehicles must be in the database, such as:
    - Brand;
    - Model;
    - Category;
    - Vehicle type;
    - Capacity (number of people);
    - Image address of the vehicle;
    - Daily rate;
    - Date of last inspection;
    - Date of next inspection;
    - Date of last legalization;
    - Date of next legalization;
    - Etc.

3. The system should alert the fleet manager that vehicles will require inspection 5 days before the date of the next inspection.

4. The fleet manager should have the option to indicate that the vehicle is under maintenance, and it will be unavailable for rental.

Imports

```python
import tkinter as tk  # Provides a Python interface to the Tk GUI toolkit.
from tkinter import *  # Imports all classes, functions, and constants from tkinter module.
from tkinter import ttk, font  # Additional widgets and font handling utilities for Tkinter.
from tkinter import filedialog  # Dialogs for file and directory selection.
from flask import Flask  # Web framework for creating web applications in Python.
from flask_sqlalchemy import SQLAlchemy  # SQLAlchemy integration for Flask web applications.
from sqlalchemy import create_engine, extract, desc  # SQL toolkit and Object-Relational Mapper (ORM) for Python.
from sqlalchemy.exc import OperationalError  # Exceptions related to database operations in SQLAlchemy.
from flask_bcrypt import Bcrypt  # Password hashing utilities for Flask web applications.
from PIL import ImageTk, Image  # Python Imaging Library for image manipulation.
import os  # Provides functions to interact with the operating system.
import pandas as pd  # Data manipulation and analysis library.
import tkinter.font as tkFont  # Additional font utilities for Tkinter.
from datetime import datetime, timedelta, timezone  # Date and time utilities.
import re  # Regular expression operations.
import numpy as np  # Numerical computing library for arrays, matrices, and mathematical functions.
from tkcalendar import *  # Calendar widget for Tkinter.
```

Classes

Reservation:

```python
class Reservation(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    pick_up_date = db.Column(db.String(50), nullable=False)
    drop_off_date = db.Column(db.String(50), nullable=False)
    vehicle_id = db.Column(db.String(50), nullable=False)
    rental_duration = db.Column(db.Integer, nullable=False)
    cost_per_day = db.Column(db.Integer, nullable=False)
    total_cost = db.Column(db.Integer, nullable=False)
    full_name = db.Column(db.String(100), nullable=False)
    phone_number = db.Column(db.String(20), nullable=False)
    email = db.Column(db.String(100), nullable=False)
    id_passport_number = db.Column(db.String(20), nullable=False)
    date_of_birth = db.Column(db.String(50), nullable=False)
    nationality = db.Column(db.String(50), nullable=False)
    emergency_contact_name = db.Column(db.String(100), nullable=False)
    emergency_contact_number = db.Column(db.String(20), nullable=False)
    billing_address = db.Column(db.String(200), nullable=False)
    payment_method = db.Column(db.String(50), nullable=False)
    photos = db.Column(db.String(300), nullable=False)
    reservation_state = db.Column(db.String(15), default="In progress")
    date_confirm_completed_renting = db.Column(db.String(15), default="Still in progress")
    code_confirm_completed_renting = db.Column(db.String(20), default="Employee Code")
    date_cancel_reservation = db.Column(db.String(15), default="Still in progress")
    code_cancel_reservation = db.Column(db.String(20), default="Employee Code")
    last_update = db.Column(db.String(15), default="No previous update")
    code_last_update = db.Column(db.String(20), default="Employee Code")
    insertion_date = db.Column(db.Date, default=Lambda: datetime.now(timezone.utc).date())
    code_insertion = db.Column(db.String(20), default="Employee Code")

    def __init__(self, pick_up_date, drop_off_date, vehicle_id, rental_duration, cost_per_day, total_cost, full_name, phone_number,
                 email, id_passport_number, date_of_birth, nationality, emergency_contact_name, emergency_contact_number,
                  billing_address, payment_method, photos,  **kwargs):
        self.pick_up_date = pick_up_date
        self.drop_off_date = drop_off_date
        self.vehicle_id = vehicle_id
        self.rental_duration = rental_duration
        self.cost_per_day = cost_per_day
        self.total_cost = total_cost
        self.full_name = full_name
        self.phone_number = phone_number
        self.email = email
        self.id_passport_number = id_passport_number
        self.date_of_birth = date_of_birth
        self.nationality = nationality
        self.emergency_contact_name = emergency_contact_name
        self.emergency_contact_number = emergency_contact_number
        self.billing_address = billing_address
        self.payment_method = payment_method
        self.photos = photos
        super().__init__(**kwargs)

    def __repr__(self):
        return f"Reservation(id={self.id}, Full Name={self.full_name}, Vehicle={self.vehicle_id})"
```

Payment:

```python
class Payment(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    total_pay = db.Column(db.Integer, nullable=False)
    payment_method = db.Column(db.String(50), nullable=False)
    f_name = db.Column(db.String(100), nullable=False)
    p_number = db.Column(db.String(50),  nullable=False)
    id_number = db.Column(db.String(50), nullable=False)
    bill_address = db.Column(db.String(50), nullable=False)
    vehicle_id = db.Column(db.String(20), nullable=False)
    photos = db.Column(db.String(300), nullable=False)
    last_update = db.Column(db.String(15), default="No previous update")
    code_last_update = db.Column(db.String(20), default="Employee Code")
    insertion_date = db.Column(db.Date, default=lambda: datetime.now(timezone.utc).date())
    code_insertion = db.Column(db.String(20), default="Employee Code")

    def __init__(self, total_pay, payment_method, f_name, p_number, id_number, bill_address, vehicle_id, photos, **kwargs):
        self.total_pay = total_pay
        self.payment_method = payment_method
        self.f_name = f_name
        self.p_number = p_number
        self.id_number = id_number
        self.bill_address = bill_address
        self.vehicle_id = vehicle_id
        self.photos = photos
        super().__init__(**kwargs)

    def __repr__(self):
        return f"Payment(id={self.id}, Full Name={self.f_name}, Vehicle={self.vehicle_id})"
```

Client:

```python
class Client(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    f_name = db.Column(db.String(100), nullable=False)
    dob = db.Column(db.String(15), nullable=False)
    p_n_indicative = db.Column(db.String(15), nullable=False)
    p_number = db.Column(db.String(50), unique=True, nullable=False)
    email = db.Column(db.String(50), unique=True, nullable=False)
    nationality = db.Column(db.String(50), nullable=False)
    address = db.Column(db.String(50), nullable=False)
    id_type = db.Column(db.String(20), nullable=False)
    id_number = db.Column(db.String(50), unique=True, nullable=False)
    credit_number = db.Column(db.String(50), unique=True, nullable=False)
    bill_address = db.Column(db.String(50), nullable=False)
    p_car = db.Column(db.String(50), nullable=False)
    p_moto = db.Column(db.String(50), nullable=False)
    em_name = db.Column(db.String(100), nullable=False)
    m_license = db.Column(db.String(50), nullable=False)
    p_em_indicative = db.Column(db.String(15), nullable=False)
    em_number = db.Column(db.String(50), unique=True, nullable=False)
    photos = db.Column(db.String(300), nullable=False)
    renting = db.Column(db.String(20), default="No")
    code_renting = db.Column(db.String(20), default="Employee Code")
    last_update = db.Column(db.String(15), default="No previous update")
    code_last_update = db.Column(db.String(20), default="Employee Code")
    insertion_date = db.Column(db.Date, default=lambda: datetime.now(timezone.utc).date())
    code_insertion = db.Column(db.String(20), default="Employee Code")

    def __init__(self, f_name, dob, p_n_indicative, p_number, email, address, nationality, id_type, id_number,
                 credit_number, bill_address, p_car, p_moto, m_license, em_name, p_em_indicative, em_number, photos, **kwargs):
        self.f_name = f_name
        self.dob = dob
        self.p_n_indicative = p_n_indicative
        self.p_number = p_number
        self.email = email
        self.address = address
        self.nationality = nationality
        self.id_type = id_type
        self.id_number = id_number
        self.credit_number = credit_number
        self.bill_address = bill_address
        self.p_car = p_car
        self.p_moto = p_moto
        self.m_license = m_license
        self.em_name = em_name
        self.p_em_indicative = p_em_indicative
        self.em_number = em_number
        self.photos = photos
        super().__init__(**kwargs)

    def __repr__(self):
        return f"Client(id={self.id}, Full Name={self.f_name}, Email={self.email})"
```

Vehicle:

```python
class Vehicle(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    vehicle_type = db.Column(db.String(50), nullable=False)
    category = db.Column(db.String(50), nullable=False)
    segment = db.Column(db.String(50), nullable=False)
    brand = db.Column(db.String(50), nullable=False)
    model = db.Column(db.String(50), nullable=False)
    year = db.Column(db.String(15), nullable=False)
    license_plate = db.Column(db.String(20), unique=True, nullable=False)
    seats = db.Column(db.Integer, nullable=False)
    wheels = db.Column(db.Integer, nullable=False)
    color = db.Column(db.String(20), nullable=False)
    fuel = db.Column(db.String(20), nullable=False)
    doors = db.Column(db.Integer, nullable=False, default=0)
    gearbox = db.Column(db.String(20), nullable=False)
    photos = db.Column(db.String(300), nullable=False)
    cc = db.Column(db.String(20), default="Not Relevant")
    availability = db.Column(db.String(20), default="Available")
    rented = db.Column(db.String(20), default="No")
    code_rented = db.Column(db.String(20), default="Employee Code")
    for_inspection = db.Column(db.String(20), default="No")
    code_inspection = db.Column(db.String(20), default="Employee Code")
    for_legalization = db.Column(db.String(20), default="No")
    code_legalization = db.Column(db.String(20), default="Employee Code")
    last_update = db.Column(db.String(15), default="No previous update")
    code_last_update = db.Column(db.String(20), default="Employee Code")
    next_inspection = db.Column(db.String(15), nullable=False)
    next_legalization = db.Column(db.String(15), nullable=False)
    insertion_date = db.Column(db.Date, default=lambda: datetime.now(timezone.utc).date())
    code_insertion = db.Column(db.String(20), default="Employee Code")

    def __init__(self, vehicle_type, category, segment, brand, model, year, license_plate, seats, wheels, doors,
                 color, fuel, gearbox, photos, **kwargs):
        self.vehicle_type = vehicle_type
        self.category = category
        self.segment = segment
        self.brand = brand
        self.model = model
        self.year = year
        self.license_plate = license_plate
        self.seats = seats
        self.wheels = wheels
        self.doors = doors
        self.color = color
        self.fuel = fuel
        self.gearbox = gearbox
        self.photos = photos
        super().__init__(**kwargs)

    def __repr__(self):
        return f"Vehicle(id={self.id}, brand={self.brand}, model={self.model})"
```

Employee:

```python
class Employee(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    full_name = db.Column(db.String(50), nullable=False)
    username = db.Column(db.String(30), nullable=False, unique=True)
    password = db.Column(db.String(60), nullable=False)
    employee_code = db.Column(db.String(30), unique=True, nullable=False)
    employee_type = db.Column(db.String(30), nullable=False)

    def __init__(self, full_name, username, password, employee_code, employee_type,  **kwargs):
        self.full_name = full_name
        self.username = username
        self.password = password
        self.employee_code = employee_code
        self.employee_type = employee_type
        super().__init__(**kwargs)

    def __repr__(self):
        return f"Employee(id={self.id}, Full Name={self.full_name}, Employee Type={self.employee_type})"
```

Lw:

```
236 class Lw:
237     WINDOW_WIDTH = 1000
238     WINDOW_HEIGHT = 620
239     LOGO_PATH = 'resources/lwheels.png'
240
241
242     def __init__(self, root, flask_app):
243         self.root = root
244         self.flask_app = flask_app
245
246         self.app_context = flask_app.app_context()
247         self.app_context.push()
248
249         self.root.overrideredirect(True)
250         self.logo_image = PhotoImage(file=self.LOGO_PATH)
251         self.canvas = tk.Canvas(self.root, width=self.WINDOW_WIDTH, height=self.WINDOW_HEIGHT)
252         self.canvas.pack()
253         self.canvas.create_image(self.WINDOW_WIDTH // 2, self.WINDOW_HEIGHT // 2, image=self.logo_image,
254                                  anchor=tk.CENTER)
255
256         self.root.configure(bg='black')
257
258         screen_width = self.root.winfo_screenwidth()
259         screen_height = self.root.winfo_screenheight()
260         center_x_main = (screen_width - self.WINDOW_WIDTH) // 2
261         center_y_main = (screen_height - self.WINDOW_HEIGHT) // 2
262
263         self.root.geometry(f"{self.WINDOW_WIDTH}x{self.WINDOW_HEIGHT}+{center_x_main}+{center_y_main}")
264
265         self.root.after(3000, self.show_main_window)
266
267     def bind_hover_effects(self, button): ...
270
271     @staticmethod
272     def on_button_enter(event): ...
274
275     @staticmethod
276     def on_button_leave(event): ...
278
279     def green_bind_hover_effects(self, button): ...
282
283     @staticmethod
284     def green_on_button_enter(event): ...
286
287     @staticmethod
288     def green_on_button_leave(event): ...
290
291     def red_bind_hover_effects(self, button): ...
294
295     @staticmethod
296     def red_on_button_enter(event): ...
298
299     @staticmethod
300     def red_on_button_leave(event): ...
302
303     def orange_bind_hover_effects(self, button): ...
306
307     @staticmethod
308     def orange_on_button_enter(event): ...
310
311     @staticmethod
312     def orange_on_button_leave(event):
313         event.widget.config(bg="#C56C00")
314
315     def load_data(self, new_window): ...
333
334     def load_image(self, new_window): ...
348
349     def create_section_window(self, section): ...
```

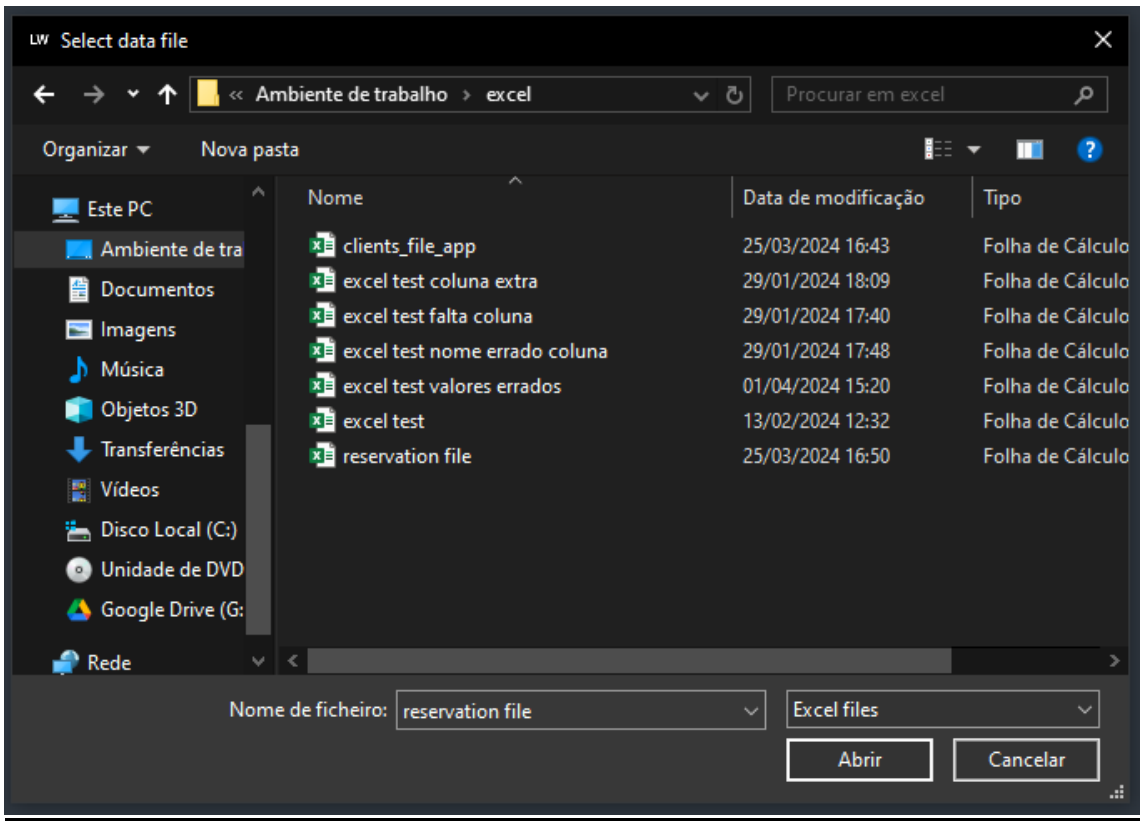
```
388     def photo_viewer(self, window, photos, view_mode=None, result_callback=None, updated_photos=None): ...
523
524     def change_row_color(self, tree, row_index, color): ...
529
530     def toggle_combo_text(self, result, combobox): ...
535
536     def toggle_entry_colors(self, result, entry): ...
541
542     def toggle_entry_colors_ifnan(self, result, entry): ...
547
548     def toggle_button_colors(self, result, button): ...
553
554     def pop_warning(self, window, variable, warning, choice_callback=None, photos_callback=None): ...
2022
2023    def validate_data(self, type_of_data, num, alpha, defined, empty): ...
2093
2094    def verify_photo_path(self, possible_photo_paths): ...
2120
2121    def check_if_repeated(self, valid, column): ...
2144
2145    def datepicker(self, window, entry, date_type, button=None, pick_date=None, costs=None): ...
2197
2198    def check_employee_code(self, code, must_be_manager=False): ...
2216
2217    def calculate_date(self, date, add_one=False): ...
2273
2274    def new_employee(self): ...
2453
2454    def insert_vehicle_section(self, new_window): ...
3968
3969    def insert_clients_section(self, new_window): ...
5387
5388    def make_reservations_section(self, new_window): ...
7091
7092    def manage_vehicles_section(self, new_window): ...
8431
8432    def manage_clients_section(self, new_window): ...
9470
9471    def manage_reservations_section(self, new_window): ...
10719
10720   def payments_section(self, new_window): ...
10952
10953   def employees_section(self, new_window): ...
11112
11113   def show_authenticated_frame(self, authenticated_username, success_message=None): ...
11632
11633   def show_non_authenticated_frame(self, error_message=None, success_message=None): ...
11686
11687   def show_main_window(self, authenticated=False, authenticated_username=None, success_message=None, error_message=None): ...
11707
```

Applying effects when hovering the mouse cursor over different buttons:

```python
    def bind_hover_effects(self, button):
        button.bind("<Enter>", self.on_button_enter)
        button.bind("<Leave>", self.on_button_leave)

    @staticmethod
    def on_button_enter(event):
        event.widget.config(bg="#205a3d")

    @staticmethod
    def on_button_leave(event):
        event.widget.config(bg="#000000")

    def green_bind_hover_effects(self, button):
        button.bind("<Enter>", self.green_on_button_enter)
        button.bind("<Leave>", self.green_on_button_leave)

    @staticmethod
    def green_on_button_enter(event):
        event.widget.config(bg="#205a3d")

    @staticmethod
    def green_on_button_leave(event):
        event.widget.config(bg="#004d00")

    def red_bind_hover_effects(self, button):
        button.bind("<Enter>", self.red_on_button_enter)
        button.bind("<Leave>", self.red_on_button_leave)

    @staticmethod
    def red_on_button_enter(event):
        event.widget.config(bg="#ff6666")

    @staticmethod
    def red_on_button_leave(event):
        event.widget.config(bg="#800000")

    def orange_bind_hover_effects(self, button):
        button.bind("<Enter>", self.orange_on_button_enter)
        button.bind("<Leave>", self.orange_on_button_leave)

    @staticmethod
    def orange_on_button_enter(event):
        event.widget.config(bg="#FF8C00")

    @staticmethod
    def orange_on_button_leave(event):
        event.widget.config(bg="#C56C00")
```

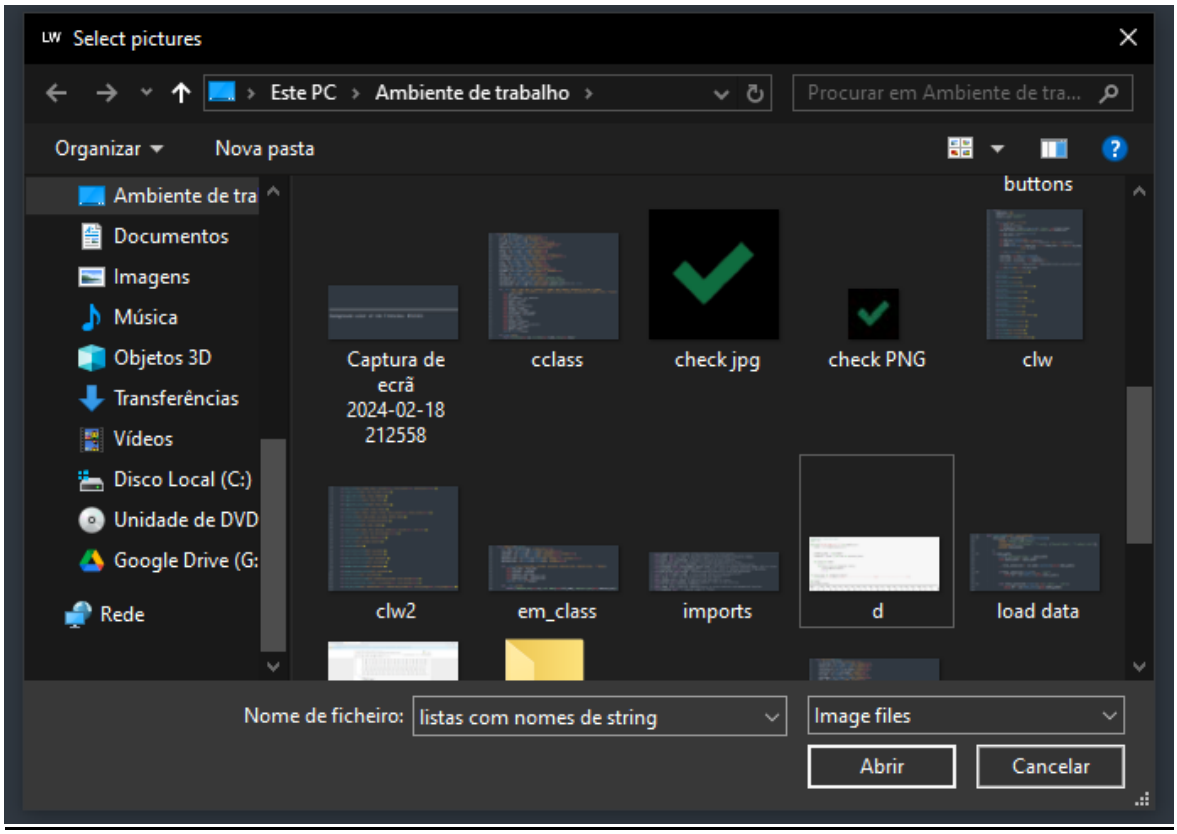
Loading data from a file selected by the user (used in the sections of Insert Vehicles/Clients/Reservations):

```python
    def load_data(self, new_window):
        data_path = filedialog.askopenfilename(
            title="Select data file",
            filetypes=[("CSV files", "*.csv"), ("Excel files", "*.xlsx;*.xls")],
            parent=new_window
        )
        if data_path:
            print("Selected file:", data_path)
            self.data_path = data_path

            _, file_extension = os.path.splitext(self.data_path)

            if file_extension.lower() == '.csv':
                self.df = pd.read_csv(self.data_path)


            elif file_extension.lower() in ('.xlsx', '.xls'):
                self.df = pd.read_excel(self.data_path)
```



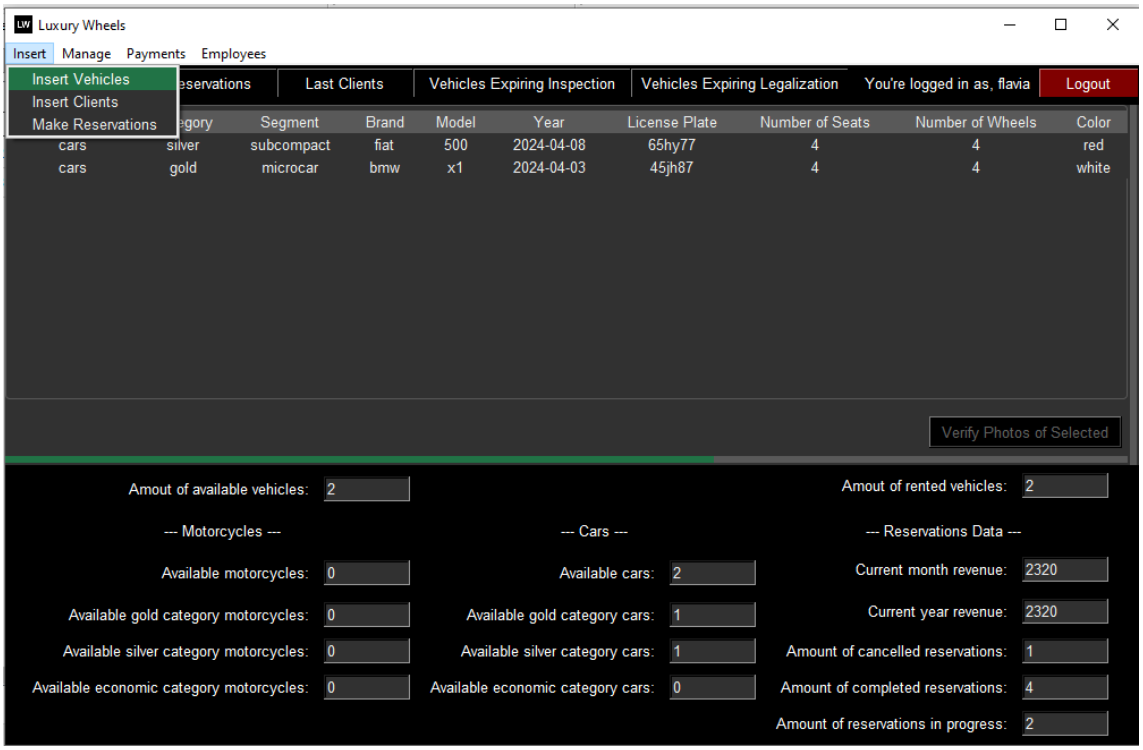
Loading images selected by the user (used in the sections of Insert/Manage Vehicles/Clients/Reservations):

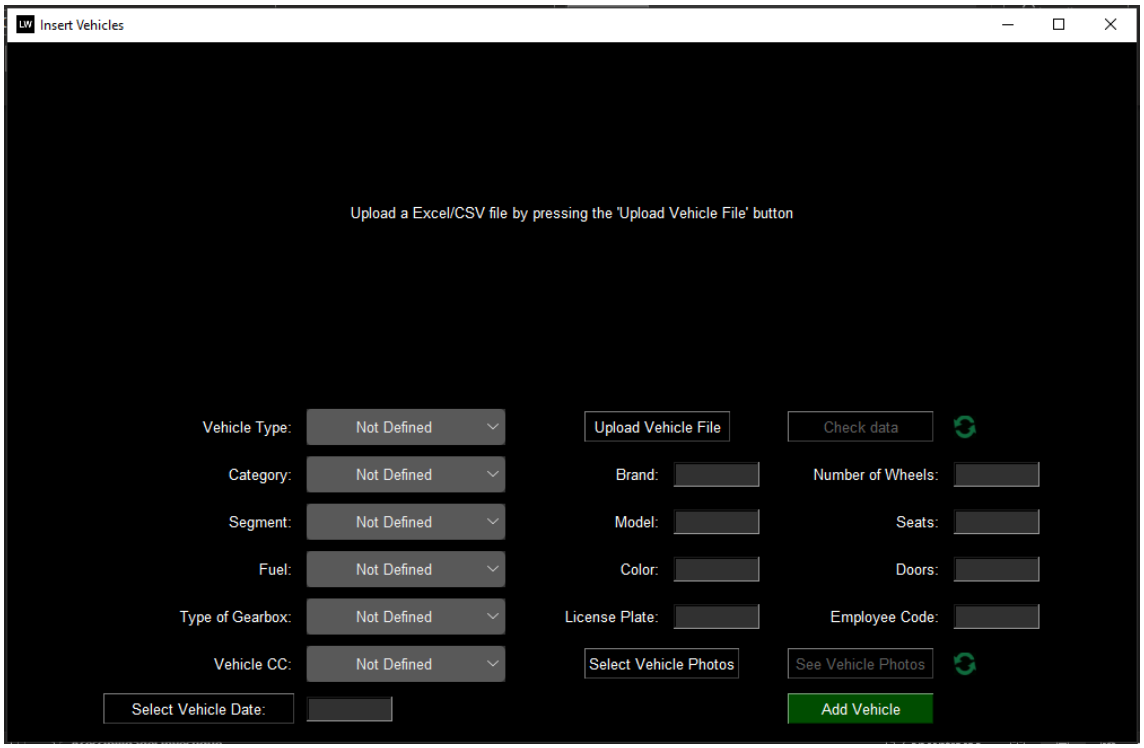
```python
    def load_image(self, new_window):
        file_paths = filedialog.askopenfilenames(
            title="Select pictures",
            filetypes=[("Image files", "*.png;*.jpg;*.jpeg")],
            parent=new_window
        )
        if file_paths:
            print("Selected files:", file_paths)

            text = file_paths
            cleaned_text = [s.replace("'", "").strip() for s in text]

            formatted_text = ", ".join(cleaned_text)
            self.photo_paths = formatted_text
```

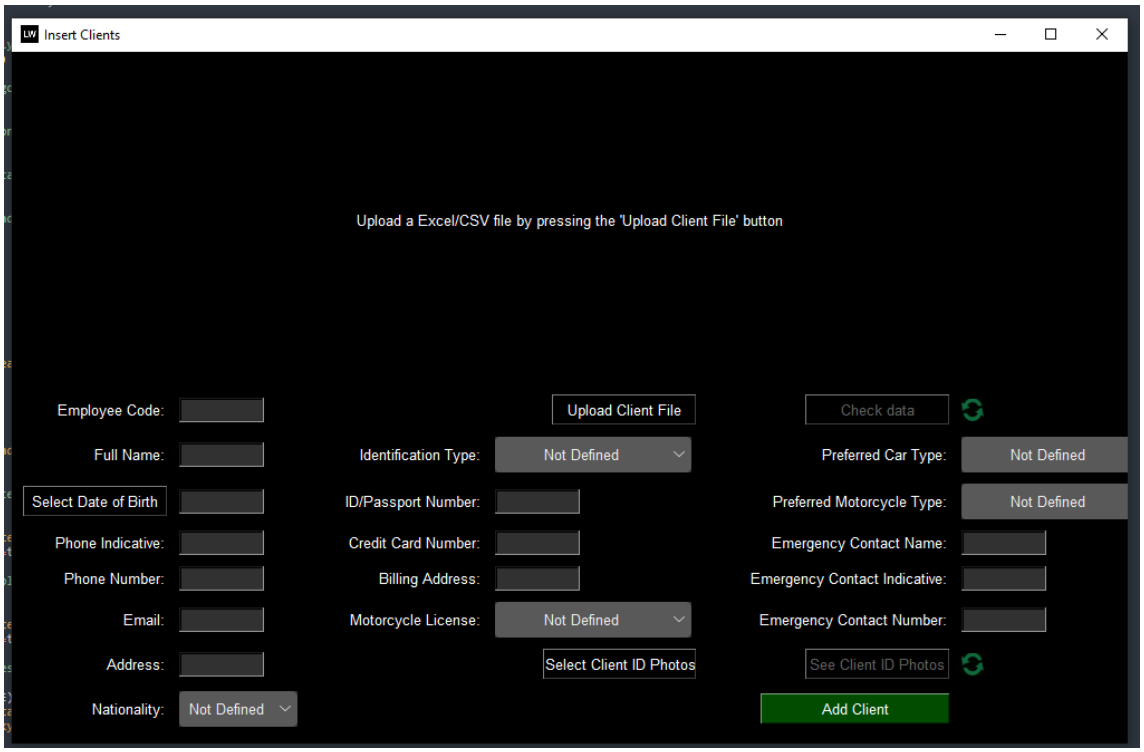


Responsible for creating a new window (or "top-level window"), depending on the section the user wants to use (by selecting with the cursor). The window will display different information based on the selected section, utilizing the same window for all sections.

```python
    def create_section_window(self, section):
        new_window = tk.Toplevel(self.root)
        new_window.title(section)
        new_window.iconphoto(True, PhotoImage(file='resources/lw.png'))

        new_window.geometry(f"{self.WINDOW_WIDTH}x{self.WINDOW_HEIGHT}")

        screen_width = self.root.winfo_screenwidth()
        screen_height = self.root.winfo_screenheight()
        center_x_new_window = (screen_width - self.WINDOW_WIDTH) // 2
        center_y_new_window = (screen_height - self.WINDOW_HEIGHT) // 2

        new_window.geometry(f"+{center_x_new_window}+{center_y_new_window}")

        new_window.resizable(True, True)

        new_window.configure(bg='black')

        self.root.attributes('-disabled', True)


        if section == "Insert Vehicles":
            self.insert_vehicle_section(new_window)
        elif section == "Insert Clients":
            self.insert_clients_section(new_window)
        elif section == "Make Reservations":
            self.make_reservations_section(new_window)
        elif section == "Manage Vehicles":
            self.manage_vehicles_section(new_window)
        elif section == "Manage Clients":
            self.manage_clients_section(new_window)
        elif section == "Manage Reservations":
            self.manage_reservations_section(new_window)
        elif section == "Payment Records":
            self.payments_section(new_window)
        elif section == "Employees Information":
            self.employees_section(new_window)

        def close_new_window():
            self.root.attributes('-disabled', False)
            new_window.destroy()

        new_window.protocol("WM_DELETE_WINDOW", close_new_window)
```




Insert Vehicles
Upload a Excel/CSV file by pressing the 'Upload Vehicle File' button
Vehicle Type:
Not Defined
Upload Vehicle File
Check data
Category:
Not Defined
Brand:
Number of Wheels:
Segment:
Not Defined
Model:
Seats:
Fuel:
Not Defined
Color:
Doors:
Type of Gearbox:
Not Defined
License Plate:
Employee Code:
Vehicle CC:
Not Defined
Select Vehicle Photos
See Vehicle Photos
Select Vehicle Date:
Add Vehicle


Insert Clients
Upload a Excel/CSV file by pressing the 'Upload Client File' button
Employee Code:
Upload Client File
Check data
Full Name:
Identification Type:
Not Defined
Preferred Car Type:
Not Defined
Select Date of Birth
ID/Passport Number:
Preferred Motorcycle Type:
Not Defined
Phone Indicative:
Credit Card Number:
Emergency Contact Name:
Phone Number:
Billing Address:
Emergency Contact Indicative:
Email:
Motorcycle License:
Not Defined
Emergency Contact Number:
Address:
Select Client ID Photos
See Client ID Photos
Nationality:
Not Defined
Add Client

Window for viewing and manipulating photos (used in the sections of Insert/Manage Vehicles/Clients/Reservations):

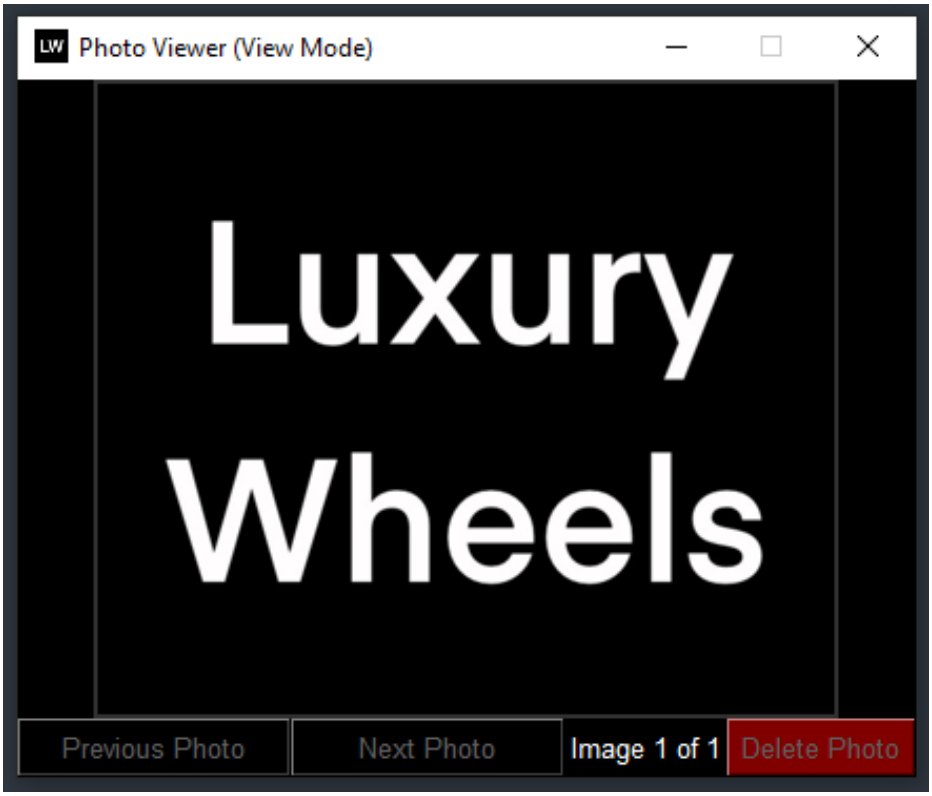
```python
    def photo_viewer(self, window, photos, view_mode=None, result_callback=None, updated_photos=None):
        if isinstance(photos, str):
                paths = photos
                print(paths)
                photos = paths.split(', ')

        changed_photos = photos
        def load_image(file_path):
            try:
                original_image = Image.open(file_path)
                width, height = original_image.size
                aspect_ratio = width / height
                target_width = 350
                target_height = int(target_width / aspect_ratio)
                resized_image = original_image.resize((target_width, target_height))
                return ImageTk.PhotoImage(resized_image)
            except Exception as e:
                print(f"Error loading image: {e}")
                return None

        def update_photo(image_number):
            nonlocal photo_label, status, next_photo_button, previous_photo_button, delete_photo_button

            photo_label.grid_forget()
            delete_number = image_number - 1

            self.my_img = load_image(photos[image_number - 1])
            if self.my_img:
                photo_label = Label(photo_view_window, image=self.my_img)
                photo_label.grid(row=0, column=0, columnspan=4)
            next_photo_button.config(command=lambda: update_photo(image_number + 1))
            previous_photo_button.config(command=lambda: update_photo(image_number - 1))

            if view_mode != "Edit Mode":
                delete_photo_button.config(state=DISABLED)

            delete_photo_button.config(command=lambda: delete_photo(image_number - 1))

            status.config(text="Image {} of {}".format(image_number, len(photos)))

            if len(photos) == 1 or image_number == 1:
                previous_photo_button.config(state=tk.DISABLED)
            elif image_number > 1:
                previous_photo_button.config(state=tk.NORMAL)

            if image_number == len(photos):

                next_photo_button.config(state=tk.DISABLED)
            else:
                next_photo_button.config(state=tk.NORMAL)

        def delete_photo(delete_number):
            nonlocal photos

            try:
                del photos[delete_number]
                if not photos:
                    on_close(result_callback)
                else:
                    update_photo(1)
            except Exception as e:
                print(f"Error deleting photo: {e}")
```

```python
        def on_close(result_callback):
            print(original_photos)
            print(changed_photos)
            if len(original_photos) != len(', '.join(changed_photos)):
                def handle_choice(option):
                    if option == "confirm":
                        if len(changed_photos) == 0:
                            updated_photos = "nan"
                            result_callback("confirm", updated_photos)
                            photo_view_window.destroy()
                            print("Confirm changes")
                        else:
                            updated_photos = ', '.join(changed_photos)
                            result_callback("confirm", updated_photos)
                            photo_view_window.destroy()
                            print("Confirm changes")
                    elif option == "cancel":
                        updated_photos = original_photos
                        result_callback("cancel", updated_photos)
                        photo_view_window.destroy()
                        print("Cancel changes")

                changes_warning = "Apply changes to photos?"
                self.pop_warning(photo_view_window, changes_warning, "photochanges", lambda option: handle_choice(option))
                print("There were some changes on photos")

            else:
                updated_photos = original_photos
                result_callback("cancel", updated_photos)
                photo_view_window.destroy()


        photo_view_window = tk.Toplevel(window)
        if view_mode != None:
            photo_view_window.title(f"Photo Viewer ({view_mode})")
        else:
            photo_view_window.title("Photo Viewer")
        photo_view_window.iconphoto(True, PhotoImage(file='resources/lw.png'))
        photo_view_window.configure(bg='black')
        photo_view_window.resizable(False, False)
        photo_view_window.grab_set()

        photo_label = Label(photo_view_window)
        status = Label(photo_view_window, text="Image 1 of {}".format(len(photos)), fg="white", bg="black")

        previous_photo_button = tk.Button(photo_view_window, text="Previous Photo", width=15, borderwidth=2,
                                          fg="white", bg="black", state=tk.DISABLED)
        next_photo_button = tk.Button(photo_view_window, text="Next Photo", width=15, borderwidth=2,
                                      fg="white", bg="black")
        delete_photo_button = tk.Button(photo_view_window, text="Delete Photo", width=10, borderwidth=2,
                                        fg="white", bg="#800000")


        previous_photo_button.grid(row=1, column=0, sticky="we")
        self.bind_hover_effects(previous_photo_button)

        next_photo_button.grid(row=1, column=1, sticky="ew")
        self.bind_hover_effects(next_photo_button)

        status.grid(row=1, column=2, sticky="ew")

        delete_photo_button.grid(row=1, column=3, sticky="ew")
        self.red_bind_hover_effects(delete_photo_button)

        update_photo(1)

        original_photos = ', '.join(photos)
```

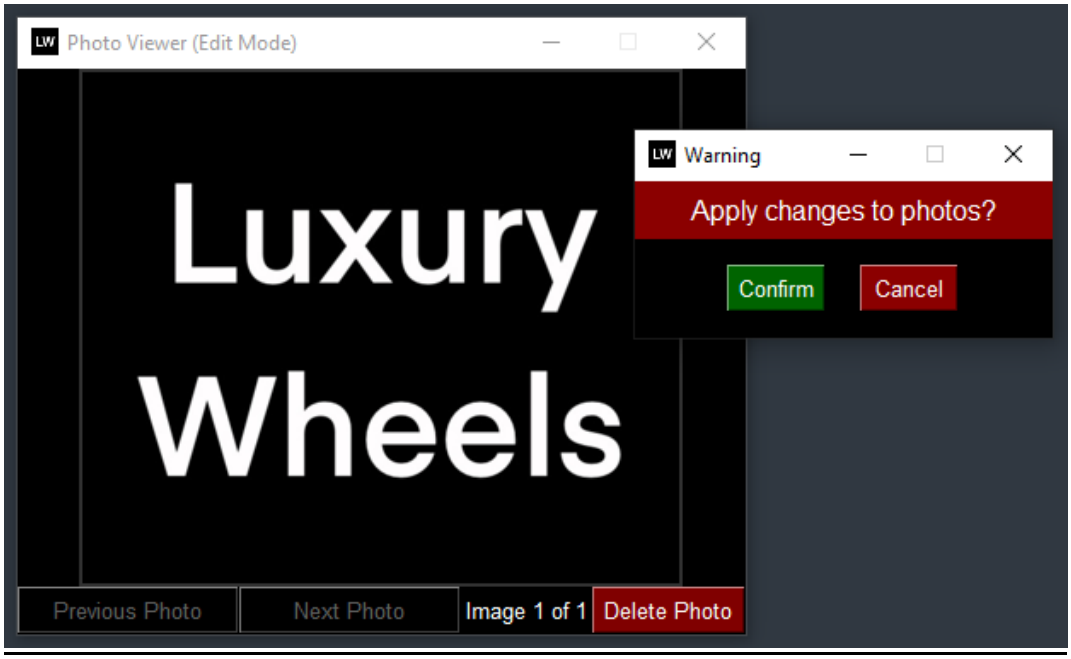
```python
        if view_mode == "Edit Mode":
            photo_view_window.protocol("WM_DELETE_WINDOW", lambda: on_close(result_callback))

            photo_view_window.wait_window(photo_view_window)
```

It has two viewing modes:

View Mode
(Does not allow deleting photos)

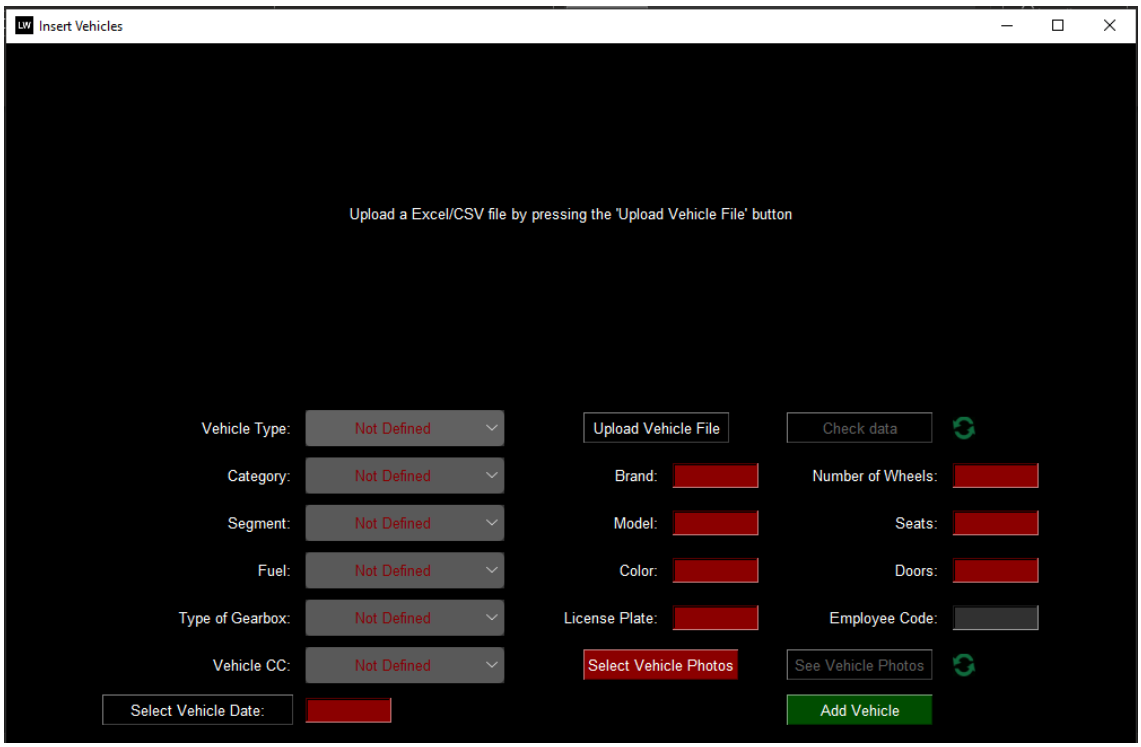


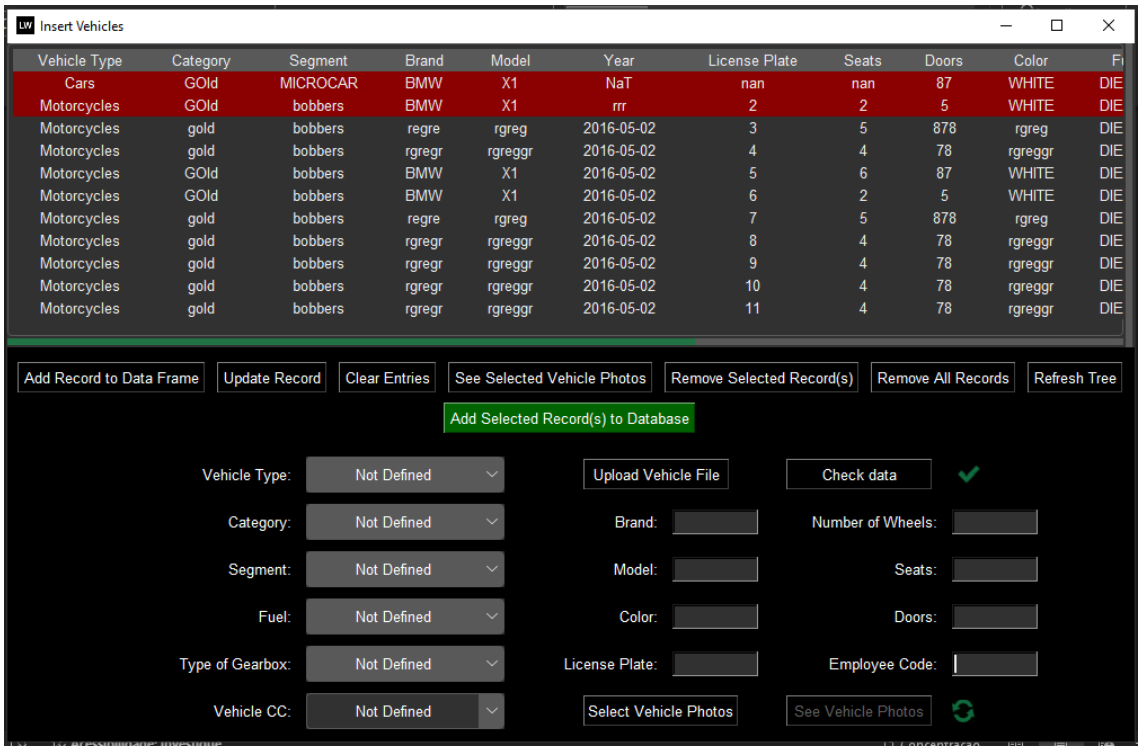
Edit Mode
(Allows deleting photos and requires confirmation if there are changes)



Applying color change effects to lines/text/text inputs/buttons if an error occurs:

```python
    def change_row_color(self, tree, row_index, color):
        item_id = tree.get_children()[row_index]
        tag_name = f"row_{row_index}_tag"
        tree.item(item_id, tags=(tag_name,))
        tree.tag_configure(tag_name, background=color)

    def toggle_combo_text(self, result, combobox):
        if result == 0:
            combobox["foreground"] = "darkred"
        else:
            combobox["foreground"] = "white"

    def toggle_entry_colors(self, result, entry):
        if result == 0:
            entry.configure(bg="darkred")
        else:
            entry.configure(bg="#313131")

    def toggle_entry_colors_ifnan(self, result, entry):
        if result == 0:
            entry.configure(bg="darkred")
        else:
            entry.configure(bg="#313131")

    def toggle_button_colors(self, result, button):
        if result == 0:
            button.configure(bg='darkred')
        else:
            button.configure(bg="black")
```





<u>Responsible for creating a new window (or "top-level window"), a window that serves as a warning or notification, either to alert about an error or to confirm an action:</u>

```
552     def pop_warning(self, window, variable, warning, choice_callback=None, photos_callback=None):
553         warning_pop = tk.Toplevel(window)
554         warning_pop.title("Warning")
555         warning_pop.iconphoto(True, tk.PhotoImage(file='resources/lw.png'))
556         warning_pop.resizable(0,0)
557         warning_pop.configure(bg="black")
558         warning_pop.grab_set()
559
560
561         def choice(option):
562             warning_pop.destroy()
563             if choice_callback:
564                 choice_callback(option)
565
566
567
568         if isinstance(variable, list): ...
1500
1501        elif isinstance(variable, tuple): ...
1534
1535        elif isinstance(variable, str): ...
1749
1750        elif isinstance(variable, dict): ...
1812
1813        elif isinstance(variable, pd.core.frame.DataFrame): ...
```

Analyzes the type of variable and then checks which specific warning is required (within that type) by searching for the text that accompanies the variable, for example:

List

```python
if isinstance(variable, list):

    if warning == "invalidformat":
        invalid_format = f"There were {len(variable)} file(s) with Invalid Format or Extension"
        label_invalid_format = tk.Label(warning_pop, text=invalid_format, font=("Helvetica", 12),
                                        fg="white", bg="darkred")
        label_invalid_format.pack(pady=5)


        for index, invalid in enumerate(variable, start=1):
            invalid_label = tk.Label(warning_pop, text=f"{invalid}\n_____", font=("Helvetica", 10),
                                     fg="white", bg="black")
            invalid_label.pack()

    elif warning == "missingheading":
        missing_heading = f"There were {len(variable)} column heading missing"
        label_missing_heading = tk.Label(warning_pop, text=missing_heading, font=("Helvetica", 12),
                                         fg="white", bg="darkred")
        label_missing_heading.pack(pady=5)


        for index, missing in enumerate(variable, start=1):
            missing_label = tk.Label(warning_pop, text=f"{missing}", font=("Helvetica", 10),
                                     fg="white", bg="black")
            missing_label.pack()
```

Tuple

```python
elif isinstance(variable, tuple):
    if warning == "missingheadunmatched":
        missing_heading = f"There were {len(variable[0])} column heading missing"
        label_missing_heading = tk.Label(warning_pop, text=missing_heading, font=("Helvetica", 12),
                                         fg="white", bg="darkred")
        label_missing_heading.pack(ipadx=10)


        for index, missing in enumerate(variable[0], start=1):
            missing_head_label = tk.Label(warning_pop, text=missing, font=("Helvetica", 10),
                                          fg="white", bg="black")
            missing_head_label.pack()

        unmatched_heading = f"There were {len(variable[1])} column heading unmatched"
        label_unmatched_heading = tk.Label(warning_pop, text=unmatched_heading, font=("Helvetica", 12),
            fg="white", bg="darkred")
        label_unmatched_heading.pack(pady=(20,5), padx=10)

        for index, unmatched in enumerate(variable[1], start=1):
            unmatched_head_label = tk.Label(warning_pop, text=unmatched, font=("Helvetica", 10),
                fg="white", bg="black")
            unmatched_head_label.pack()

    elif warning == "not_unique_license_plate":
        license_plate_warning = f"There were {len(variable[0])} vehicles with repeated license plate"
        label_license_plate_warning = tk.Label(warning_pop, text=license_plate_warning, font=("Helvetica", 12),
                                               fg="white", bg="darkred")
        label_license_plate_warning.pack(ipadx=10)

        for repeated in variable[1]:
            repeated_license_label = tk.Label(warning_pop, text=repeated, font=("Helvetica", 10),
                fg="white", bg="black")
            repeated_license_label.pack(pady=5)
```

Text

```python
elif isinstance(variable, str):
    if warning == "nanselectedphoto":
        nan_selected_photo_warning = f"'Vehicle Photos' of the selected item is empty"
        label_nan_selected_photo_warning = tk.Label(warning_pop, text=nan_selected_photo_warning, font=("Helvetica", 12),
                                                    fg="white", bg="darkred")
        label_nan_selected_photo_warning.pack(pady=5, padx=10)

        empty_selected_photos_label = tk.Label(warning_pop, text=f"Index: {variable}", font=("Helvetica", 10),
                                               fg="white", bg="black")
        empty_selected_photos_label.pack()

    elif warning == "noselectedtoupdate":
        no_selected_update_warning = f"Error while trying to update record"
        label_no_selected_update_warning = tk.Label(warning_pop, text=no_selected_update_warning, font=("Helvetica", 12),
                                                    fg="white", bg="darkred")
        label_no_selected_update_warning.pack(pady=5, padx=10)

        no_selected_update_label = tk.Label(warning_pop, text=variable, font=("Helvetica", 10),
                                            fg="white", bg="black")
        no_selected_update_label.pack()
```

Dictionary

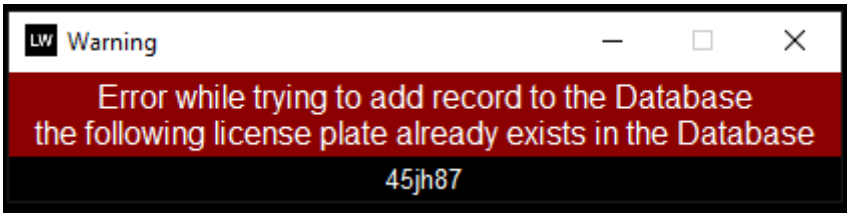
```
1750        elif isinstance(variable, dict):
1751            if warning == "validateaddrecord":
1752                validate_add_record_warning = f"Please confirm to add record"
1753                label_validate_add_record_warning = tk.Label(warning_pop, text=f"{validate_add_record_warning}", font=("Helvetica", 12),
1754                                                            fg="white", bg="darkgreen")
1755                label_validate_add_record_warning.pack(pady=5, padx=20)
1756
1757                if 'Pick-up Date' in variable:
1758                    if 'Total Cost' in variable:
1759                        limit = 16
1760                    else:
1761                        limit = 5
1762                elif 'Model' in variable:
1763                    limit = 14
1764                else:
1765                    limit = 17
1766
1767                for index, (key, value) in enumerate(variable.items()):
1768                    if index < limit:
1769                        validate_add_record_label = tk.Label(warning_pop, text=f"{key} : {value[0]}", font=("Helvetica", 10),
1770                                                             fg="white", bg="black")
1771                        validate_add_record_label.pack(pady=3)
1772                    else:
1773                        break
1774
1775                photos_validate_label_button_frame = tk.Frame(warning_pop)
1776                photos_validate_label_button_frame.pack()
1777                photos_validate_label_button_frame.configure(bg="black")
1778
1779                photos_to_validate_label = tk.Label(photos_validate_label_button_frame, font=("Helvetica", 10),
1780                                                    fg="white", bg="black")
1781                photos_to_validate_label.grid(row=0, column=0)
1782
1783                if next(iter(variable)) == 'Vehicle Type':
1784                    paths = variable['Vehicle Photos'].strip("()")
1785                    photos_to_validate_label.config(text="Vehicle Photos :")
1786                elif next(iter(variable)) == 'Full Name':
1787                    paths = variable['Client ID Photos'].strip("()")
1788                    photos_to_validate_label.config(text="Client ID Photos :")
1789                elif next(iter(variable)) == 'Pick-up Date':
1790                    paths = variable['Receipt Photos'].strip("()")
1791                    photos_to_validate_label.config(text="Receipt Photos :")
1792
1793                split_paths = paths.split(', ')
1794
1795                photos_to_validate_button = Button(photos_validate_label_button_frame, text="Verify Photos", fg="white",
1796                                                   bg="black", width=10, command=lambda: self.photo_viewer(warning_pop, split_paths, "View Mode"))
1797                photos_to_validate_button.grid(row=0, column=1)
1798
1799                confirm_cancel_frame = tk.Frame(warning_pop)
1800                confirm_cancel_frame.pack(pady=15)
1801                confirm_cancel_frame.configure(bg="black")
1802
1803                confirm_record_validation_button = Button(confirm_cancel_frame, text="Confirm", fg="white",
1804                                                          bg="darkgreen", width=10, command=lambda: choice("confirm"))
1805                confirm_record_validation_button.grid(row=0, column=0, padx=10)
1806                self.green_bind_hover_effects(confirm_record_validation_button)
1807
1808                cancel_record_validation_button = Button(confirm_cancel_frame, text="Cancel", fg="white",
1809                                                         bg="darkred", width=10, command=lambda: choice("cancel"))
1810                cancel_record_validation_button.grid(row=0, column=1, padx=10)
1811                self.red_bind_hover_effects(cancel_record_validation_button)
```

Dataframe

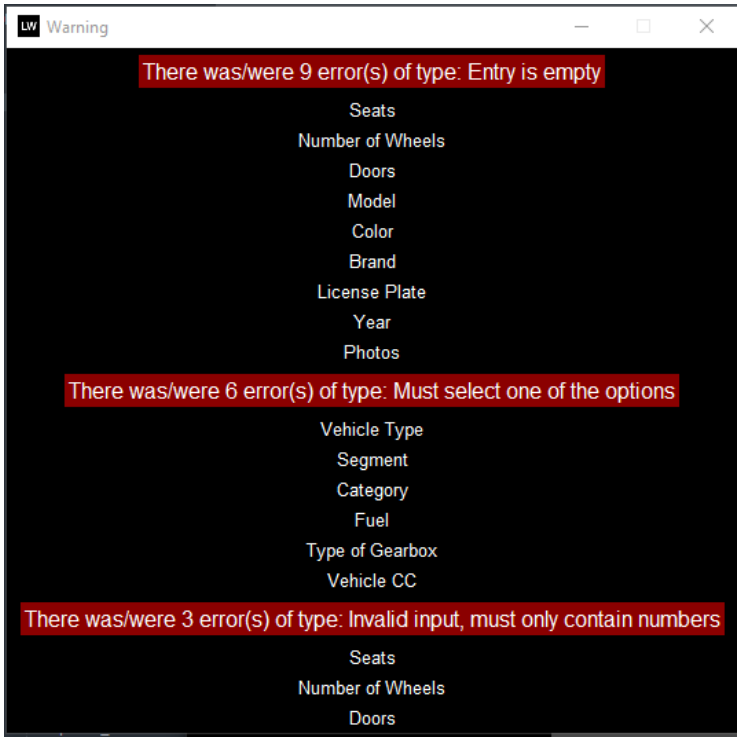
```python
        elif isinstance(variable, pd.core.frame.DataFrame):
            print("Is instance of Dataframe")

            if warning == "dfdatabvalidadd": ...

            elif warning == "export":

                export_warning = f"Please select the desired Export Format"
                label_export_warning = tk.Label(warning_pop, text=export_warning, font=("Helvetica", 12),
                                                fg="white", bg="darkgreen")
                label_export_warning.pack(pady=10)

                format_frame = tk.Frame(warning_pop)
                format_frame.pack(pady=(5,15))
                format_frame.configure(bg="black")

                column_list = variable.columns.tolist()

                current_date = str(datetime.now(timezone.utc))[:19].replace(":", "-")

                def export_excel(file_name):
                    file_name += "-Excel"
                    variable.to_excel(f"{file_name}.xlsx", index=False)
                    warning_pop.destroy()

                def export_csv(file_name):
                    file_name += "-CSV"
                    variable.to_csv(f"{file_name}.csv", index=False)
                    warning_pop.destroy()


                if "pick_up_date" in column_list:
                    single_item = f"Reservation-ID {variable.at[0, 'id']} - {current_date}"
                    multiple_items = f"Reservations {current_date}"
                elif "dob" in column_list:
                    single_item = f"Client-ID-Passport {variable.at[0, 'id_number']} - {current_date}"
                    multiple_items = f"Clients {current_date}"
                elif "vehicle_type" in column_list:
                    single_item = f"Vehicle-License Plate {variable.at[0, 'license_plate']} - {current_date}"
                    multiple_items = f"Vehicles {current_date}"
                else:
                    single_item = f"Payment-ID {variable.at[0, 'id']} - {current_date}"
                    multiple_items = f"Payments {current_date}"

                if len(variable) == 1:
                    file_name = single_item
                    print(file_name)
                else:
                    file_name = multiple_items
                    print(file_name)


                csv_button = Button(format_frame, text="CSV Format", fg="white",
                                    bg="black", width=10, command=lambda: export_csv(file_name))
                csv_button.grid(row=0, column=0, padx=10)
                self.bind_hover_effects(csv_button)

                excel_button = Button(format_frame, text="Excel Format", fg="white",
                                    bg="black", width=10, command=lambda: export_excel(file_name))
                excel_button.grid(row=0, column=1, padx=10)
                self.bind_hover_effects(excel_button)
```

Examples of warnings/confirmations:

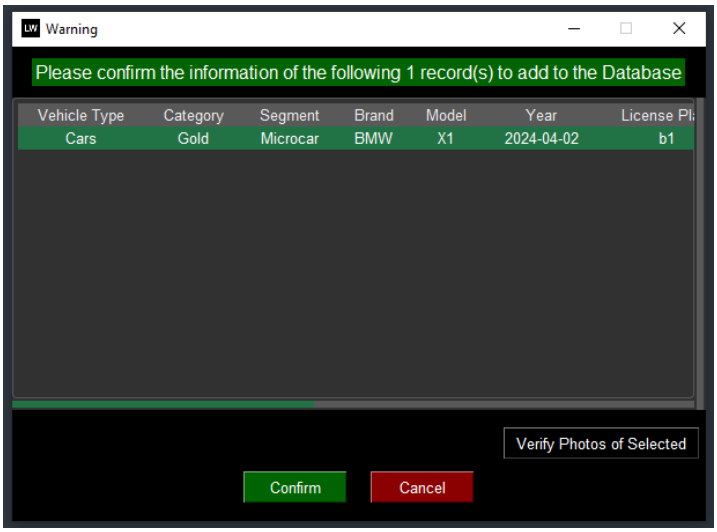
Warning when trying to insert a vehicle with a license plate that already exists in the database:



Warning when trying to insert a vehicle without filling in all the necessary information:



Confirmation request to insert the vehicle into the database:



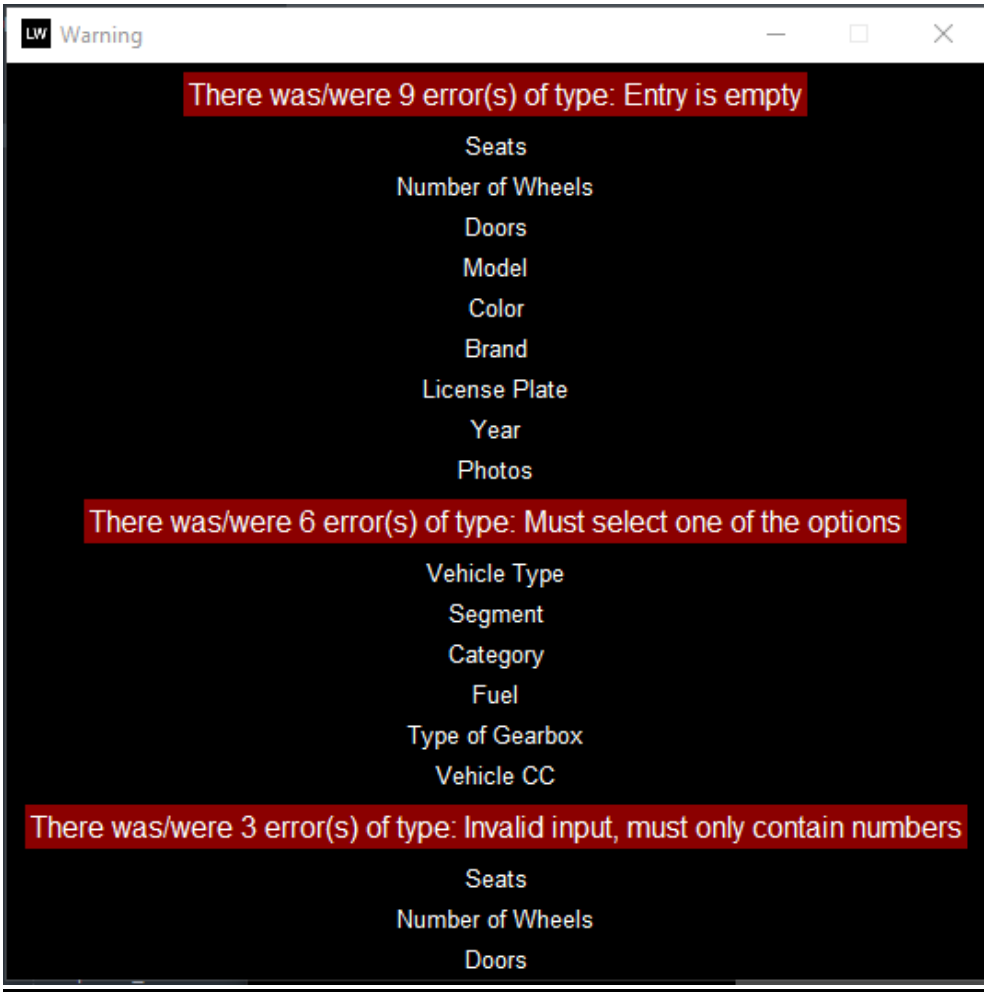
Responsible for validating the information that the user wishes to insert into the database:

Example:

- Input for names only allows letters;
- Input for the number of vehicle doors only allows numbers;
- Check that none of the inputs are left unfilled/undefined.

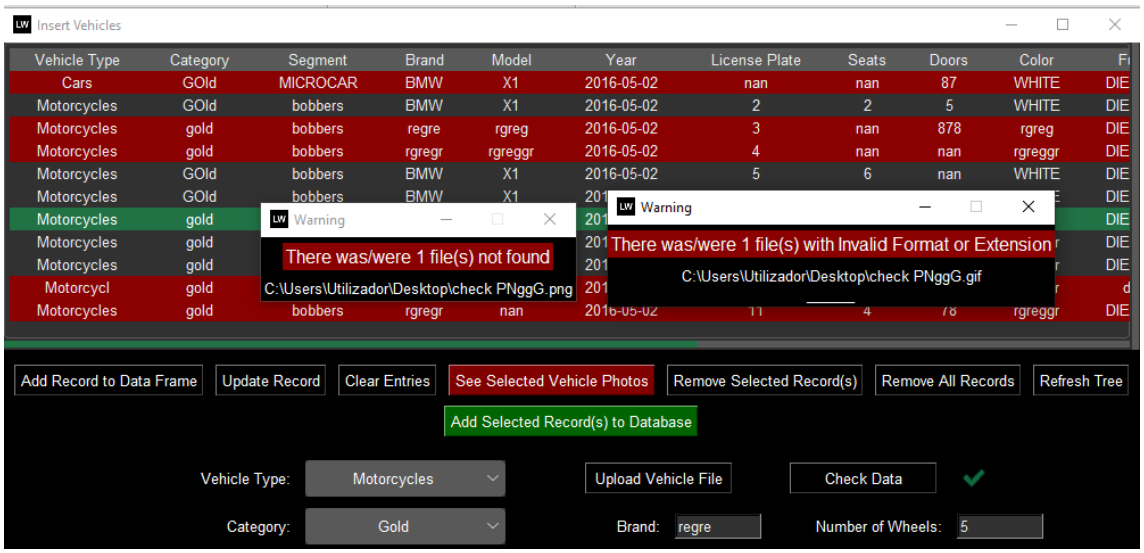
```python
    def validate_data(self, type_of_data, num, alpha, defined, empty):
        global not_num, not_alpha, not_defined, is_empty, errors_found

        if type_of_data == "entries":
            not_num = ["Invalid input, must only contain numbers"]
            for column_num, value_num in num.items():
                first_value = value_num[0]
                try:
                    int(first_value)
                except ValueError:
                    not_num.append(column_num)

            not_alpha = ["Invalid input, must only contain letters"]
            for column_word, value_word in alpha.items():
                first_value = value_word[0]
                clean_first_value = re.sub(r'[^\w\s]', '', first_value).replace(' ','')
                if all(char.isalpha() for char in clean_first_value):
                    pass
                else:
                    not_alpha.append(column_word)

            not_defined = ["Must select one of the options"]
            for column_defined, value_defined in defined.items():
                first_value = value_defined[0]
                if first_value == "Not Defined":
                    not_defined.append(column_defined)
                else:
                    pass

            is_empty = ["Entry is empty"]
            for column_all, value_all in empty.items():
                first_value = value_all[0]
                if len(first_value) == 0 or str(first_value).lower() == "empty" or str(first_value).lower() == "0":
                    is_empty.append(column_all)
                else:
                    pass

        elif type_of_data == "data_add_database":
            not_num = ["Invalid input, must only contain numbers"]
            for column_num, value_num in num.items():
                try:
                    int(value_num)
                except ValueError:
                    not_num.append(column_num)

            not_alpha = ["Invalid input, must only contain letters"]
            for column_word, value_word in alpha.items():
                clean_first_value = re.sub(r'[^\w\s]', '', value_word).replace(' ','')
                if all(char.isalpha() for char in clean_first_value):
                    pass
                else:
                    not_alpha.append(column_word)

            not_defined = ["Must select one of the options"]
            for column_defined, value_defined in defined.items():
                first_value = value_defined[0]
                second_value = value_defined[1]
                if first_value.lower() not in [val.lower() for val in second_value] or first_value.lower() == 'not defined':
                    not_defined.append(column_defined)
                else:
                    pass

            is_empty = ["Entry is empty"]
            for column_all, value_all in empty.items():
                if len(value_all) == 0 or value_all.lower() == "nan" or value_all.lower() == "0":
                    is_empty.append(column_all)
                else:
                    pass

        errors_found = is_empty, not_defined, not_alpha, not_num
```

With the validation result and using methods for color change, the user receives information about the errors:



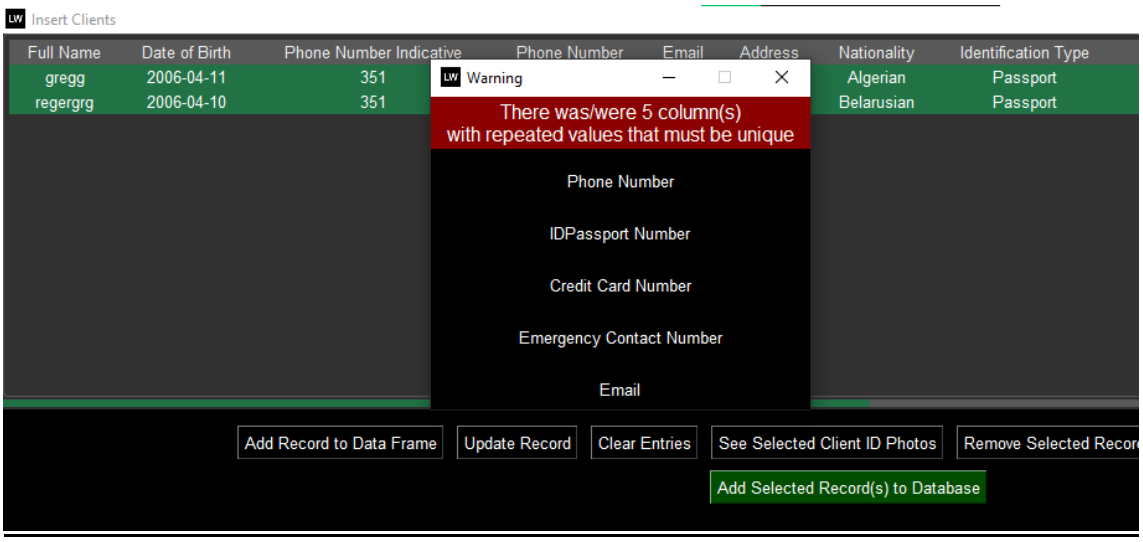
Responsible for validating the photo addresses found in the CSV/Excel file selected by the user in the sections Insert Vehicles/Clients/Reservations:

```python
def verify_photo_path(self, possible_photo_paths):
    global invalid_photo_paths, valid_photo_paths, valid_photo_type, invalid_photo_type

    paths_list = possible_photo_paths.split(',')

    paths_list = [path.strip() for path in paths_list]

    allowed_extensions = ['.png', '.jpg', '.jpeg']

    invalid_photo_type = []
    valid_photo_type = []
    for path in paths_list:
        if any(path.lower().endswith(ext) for ext in allowed_extensions):
            valid_photo_type.append(path)
        else:
            invalid_photo_type.append(path)

    invalid_photo_paths = []
    valid_photo_paths = []
    for path in valid_photo_type:
        try:
            print(f"Image open: {Image.open(path)}")
            Image.open(path)
            valid_photo_paths.append(path)
        except FileNotFoundError:
            invalid_photo_paths.append(path)
```



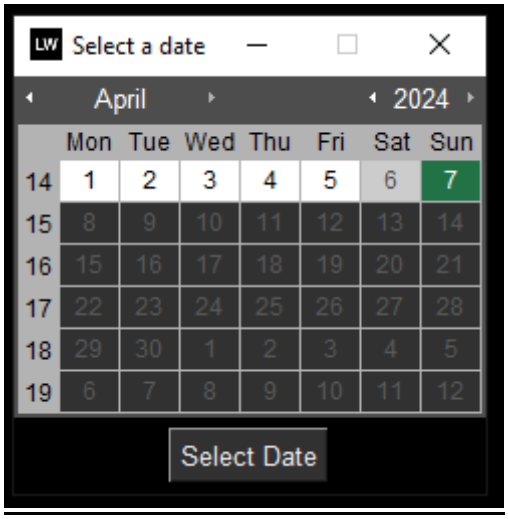
Responsible for checking if there are repeated values that should be unique when inserting multiple elements into the database:

```python
    def check_if_repeated(self, valid, column):
        check_repeated = [re.sub(r'[^\w\s]', '', str(self.df.at[record, column]).lower()) for record in valid]

        repeated_value = []
        single_value = []
        for record in check_repeated:
            if record in single_value:
                repeated_value.append(record)
            else:
                single_value.append(record)

        not_unique = []
        for record in valid:
            if re.sub(r'[^\w\s]', '', str(self.df.at[record, column]).lower()) in repeated_value:
                not_unique.append(record)

        column_repeated = []
        if len(not_unique) > 0:
            column_repeated.append(column)

        warning_repeated = not_unique, repeated_value, column

        return valid, warning_repeated
```



Date selection window:

```python
    def datepicker(self, window, entry, date_type, button=None, pick_date=None, costs=None):
        picker_calendar = tk.Toplevel(window)
        picker_calendar.title("Select a date")
        picker_calendar.iconphoto(True, tk.PhotoImage(file='resources/lw.png'))
        picker_calendar.resizable(0,0)
        picker_calendar.configure(bg="black")
        picker_calendar.grab_set()

        def select_date():
            selected_date = cal.get_date()
            entry.config(state=tk.NORMAL)
            entry.delete(0, tk.END)
            entry.insert(0, selected_date)
            entry.config(state="readonly")
            if button is not None:
                if pick_date is not None:
                    button.config(state=DISABLED)
                    costs()
                else:
                    button.config(state=NORMAL)

            picker_calendar.destroy()


        current_date = datetime.now().date()
        five_days_later = current_date + timedelta(days=5)

        cal = Calendar(picker_calendar, selectmode='day', date_pattern='yyyy-mm-dd', date=current_date)
        cal.pack()

        if date_type == "reservation":
            if hasattr(self, 'df'):
                if pick_date is not None:
                    date = datetime.strptime(pick_date, '%Y-%m-%d').date()
                    cal.config(mindate=date)
                else:
                    pass
            elif not hasattr(self, 'df'):
                if pick_date is not None:
                    date = datetime.strptime(pick_date, '%Y-%m-%d').date()
                    cal.config(mindate=date)
                else:
                    cal.config(mindate=current_date, maxdate=five_days_later)
        elif date_type == "vehicle_date":
            cal.config(maxdate=current_date)
        elif date_type == "dob":
            must_be_adult = datetime.now() - timedelta(days=18*365)
            current_date = must_be_adult.date()
            cal.config(maxdate=current_date)

        get_date_button = tk.Button(picker_calendar, text="Select Date", command=select_date)
        get_date_button.pack(pady=5)
```



Sets certain limits on possible date selections depending on the purpose of the selected date:

- If it's to select the client's date of birth, it only presents dates where the client is at least 18 years old (taking into account the current date);

- To select the vehicle's date, it's not possible to select a date later than the current date;

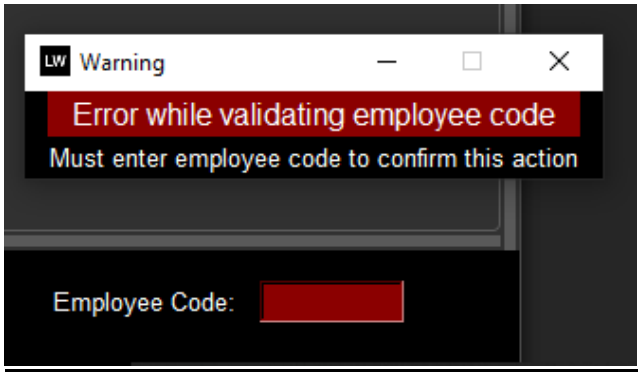
- To select the start date of a reservation, it cannot be earlier than the current date and cannot be later than the current date + 5 days;

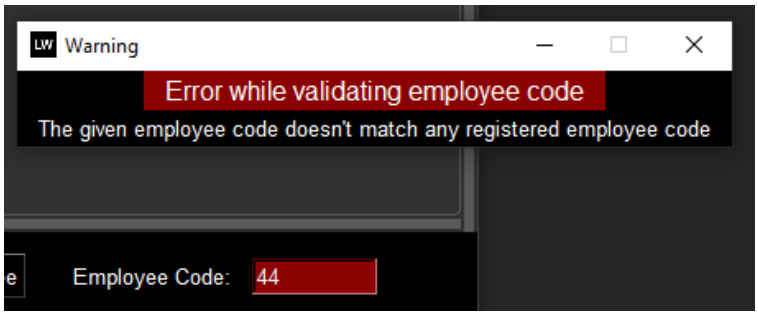
- If we are selecting the start date of a reservation after uploading a CSV/Excel file, it's possible to select a date earlier than the current date because we might be inserting a reservation that was made before the current date.

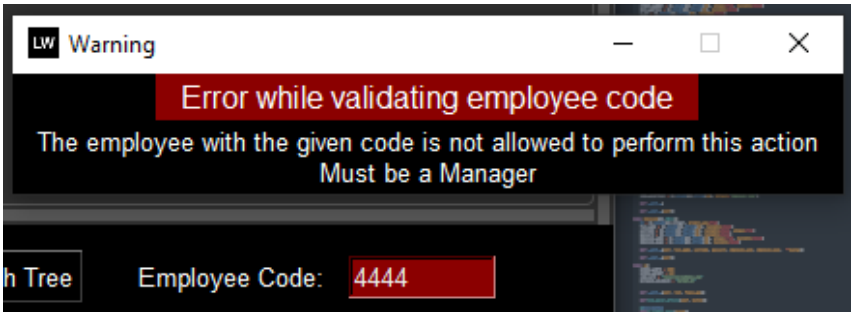
It's only possible to select the end date of a reservation after selecting the start date, and the end date of the reservation will be equal to or later than the start date.

Responsible for verifying if the employee code is compatible with any existing employee, and for certain actions, it also verifies if it corresponds to a manager:

```python
    def check_employee_code(self, code, must_be_manager=False):

        if code == "":
            code_result = "Must enter employee code to confirm this action"
        else:
            employee = None
            employee = Employee.query.filter(Employee.employee_code.ilike(code)).first()
            if employee is not None:
                if must_be_manager is True:
                    if employee.employee_type == "Manager":
                        code_result = "valid"
                    else:
                        code_result = "The employee with the given code is not allowed to perform this action\nMust be a Manager"
                else: code_result = "valid"
            else:
                code_result = "The given employee code doesn't match any registered employee code"

        return code_result
```





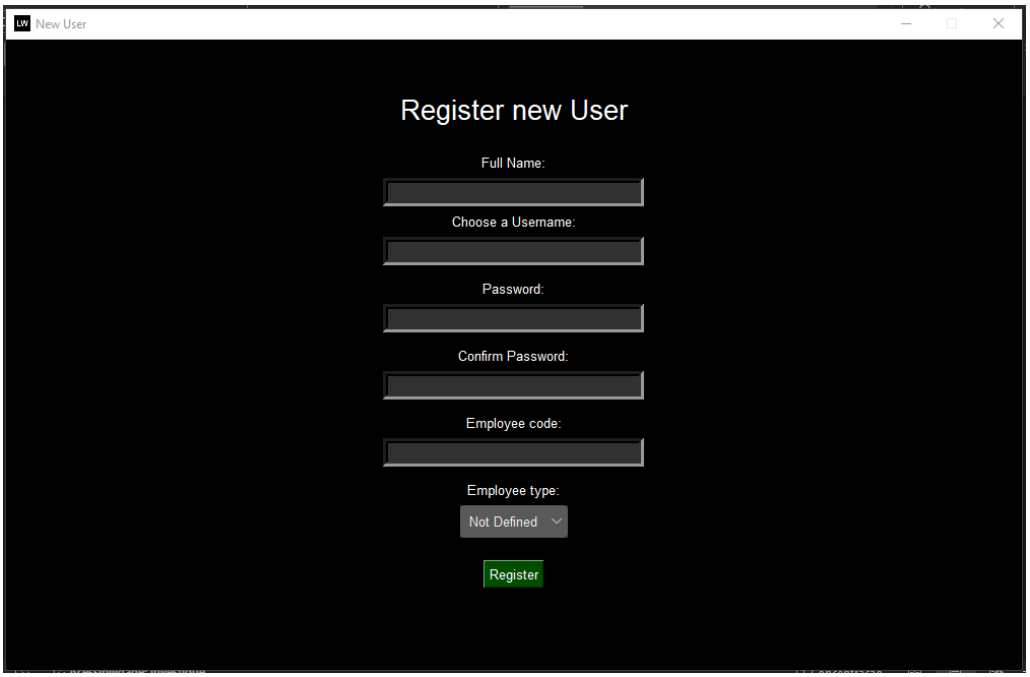


Responsible for calculating the date of the next legalization or inspection of the vehicle, used when inserting the vehicle into the database and subsequently if any changes are made regarding these dates in the manage vehicles section:

```python
    def calculate_date(self, date, add_one=False):
        date_object = datetime.strptime(date, "%Y-%m-%d")
        given_day = date_object.day
        given_month = date_object.month
        given_year = date_object.year

        current_day = datetime.now().day
        current_month = datetime.now().month
        current_year = datetime.now().year

        if add_one is False:
            if given_month == current_month:
                if given_day >= current_day:
                    new_date = datetime(current_year, given_month, given_day)
                elif given_day < current_day:
                    if given_month == 2:
                        if given_day == 29:
                            new_date = datetime(current_year + 1, given_month, given_day-1)
                            print("The next date is scheduled for:", new_date.strftime("%Y-%m-%d"))
                        else:
                            new_date = datetime(current_year + 1, given_month, given_day)
                            print("The next date is scheduled for:", new_date.strftime("%Y-%m-%d"))
                    else:
                        new_date = datetime(current_year + 1, given_month, given_day)
                        print("The next date is scheduled for:", new_date.strftime("%Y-%m-%d"))
            elif given_month > current_month:
                new_date = datetime(current_year, given_month, given_day)
                print("The next date is scheduled for:", new_date.strftime("%Y-%m-%d"))
            elif given_month < current_month:
                if given_month == 2:
                    if given_day == 29:
                        new_date = datetime(current_year + 1, given_month, given_day-1)
                        print("The next date is scheduled for:", new_date.strftime("%Y-%m-%d"))
                    else:
                        new_date = datetime(current_year + 1, given_month, given_day)
                        print("The next date is scheduled for:", new_date.strftime("%Y-%m-%d"))
                else:
                    new_date = datetime(current_year + 1, given_month, given_day)
                    print("The next date is scheduled for:", new_date.strftime("%Y-%m-%d"))

        elif add_one is True:
            if given_month == 2:
                if given_day == 29:
                    new_date = datetime(current_year + 1, given_month, given_day-1)
                    print("The next date is scheduled for:", new_date.strftime("%Y-%m-%d"))
                else:
                    # Next inspection is in the next year at the given month and day
                    new_date = datetime(current_year + 1, given_month, given_day)
                    print("The next date is scheduled for:", new_date.strftime("%Y-%m-%d"))
            else:
                new_date = datetime(current_year + 1, given_month, given_day)
                print("The next date is scheduled for:", new_date.strftime("%Y-%m-%d"))

        new_date = str(new_date.date())

        return new_date
```

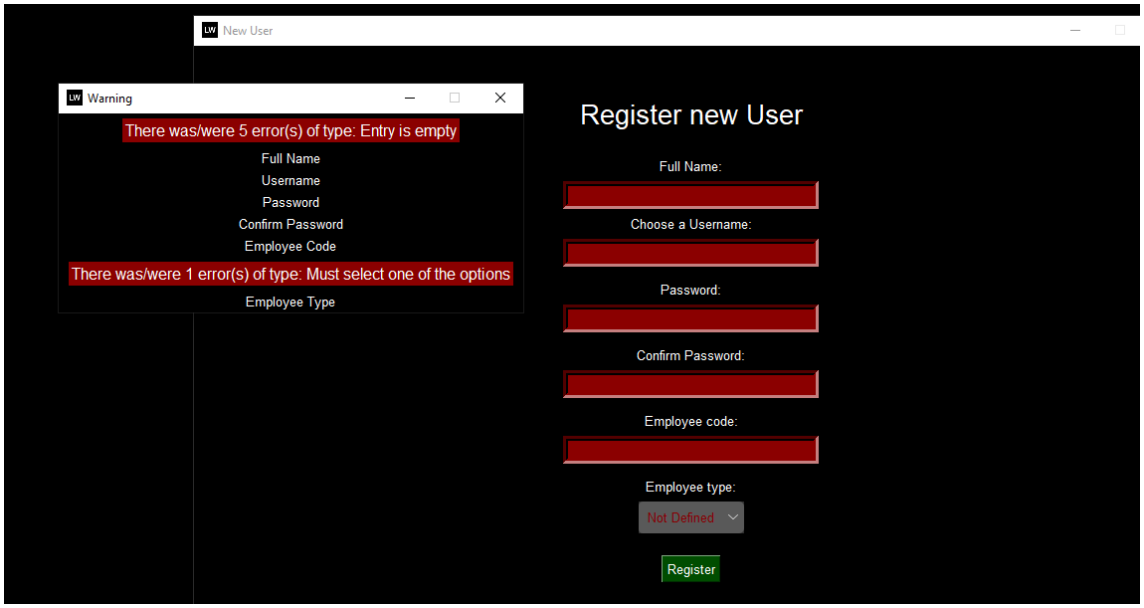
- Checks if the month of the provided date is greater or less than the current month;
- If it's less, automatically sets the next date to be one year from now;
- If it's greater, assumes that the next date will be in the current year;
- If it's equal, performs the same analysis on the days as was done with the months;
- If the provided day and month is February 29th and it needs to add a year, it will change the day to the 28th because there is no February 29th in the following year;
- Vehicle's date to calculate the next inspection;
- Date of insertion into the database to calculate the next legalization.

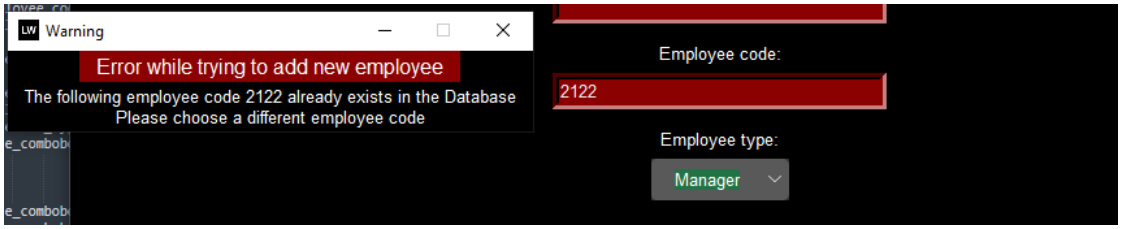
Creation of new employees:

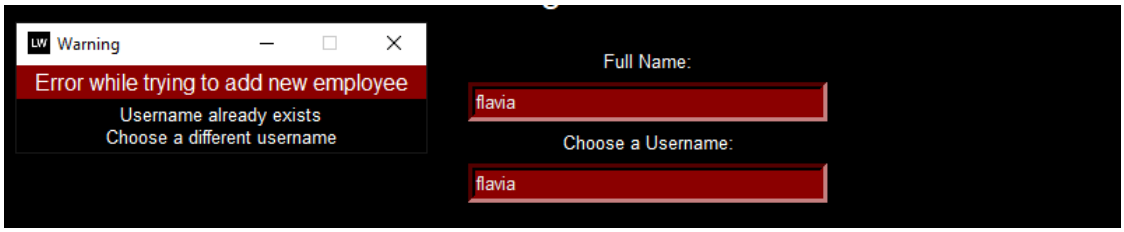


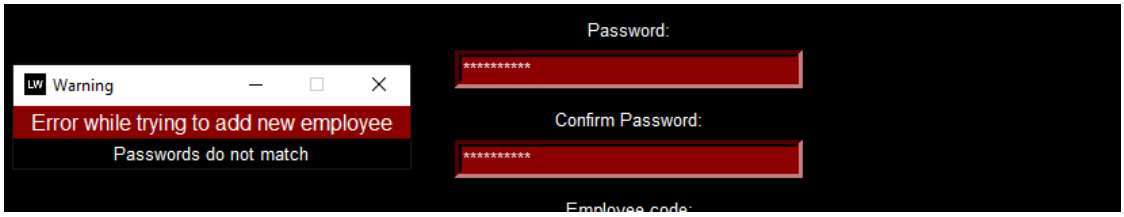

```python
        def confirm_register():
            try:
                must_be_number = {}

                must_not_have_number = {
                        'Full Name': (new_fullname_entry.get(), new_fullname_entry)
                    }

                must_be_defined = {
                        'Employee Type': (selected_type.get(), type_combobox)
                }

                must_not_be_empty = {

                    'Full Name': (new_fullname_entry.get(), new_fullname_entry),
                    'Username': (new_username_entry.get(), new_username_entry),
                    'Password': (new_password_entry.get(), new_password_entry),
                    'Confirm Password': (confirm_password_entry.get(), confirm_password_entry),
                    'Employee Code': (employee_code_entry.get(), employee_code_entry)
                    }

                self.validate_data("entries", must_be_number, must_not_have_number, must_be_defined, must_not_be_empty)

                if any(len(error_list) > 1 for error_list in errors_found):
                    result_of_validation = "Error Found"
                    print("Error Found")
                else:
                    result_of_validation = "No Error Found"
                    print("No Error Found")

                if result_of_validation == "No Error Found":
                    try:
                        with self.flask_app.app_context():
                            db.create_all()
                            code =  re.sub(r'[^\w\s]', '', str(employee_code_entry.get()).lower())
                            existing_code = Employee.query.filter(Employee.employee_code.ilike(code)).first()
                            if existing_code is not None:
                                self.toggle_entry_colors(0, employee_code_entry)
                                warning = f"The following employee code {code} already exists in the Database\nPlease choose a different employee code"
                                self.pop_warning(new_employee_window, warning, "employeecodeexists")
                            else:
                                self.toggle_entry_colors(1, employee_code_entry)
                                if new_password_entry.get() == confirm_password_entry.get():
                                    self.toggle_entry_colors(1, new_password_entry)
                                    self.toggle_entry_colors(1, confirm_password_entry)
                                    if len(new_password_entry.get()) < 10:
                                        warning="Password is too short\nPlease use at least 10 characters"
                                        self.pop_warning(new_employee_window, warning, "erroraddemploye")
                                        return
                                    existing_user = Employee.query.filter(Employee.username.ilike(new_username_entry.get())).first()
                                    if existing_user:
                                        warning="Username already exists\nChoose a different username"
                                        self.pop_warning(new_employee_window, warning, "erroraddemploye")
                                        return

                                    password_hash = bcrypt.generate_password_hash(new_password_entry.get()).decode('utf-8')

                                    new_employee = Employee(full_name=new_fullname_entry.get(),
                                        username=new_username_entry.get(),
                                        password=password_hash,
                                        employee_code=employee_code_entry.get(),
                                        employee_type=selected_type.get())

                                    db.session.add(new_employee)

                                    db.session.commit()
                                    new_employee_window.destroy()
```

It contains different verifications that ensure the data entered by the user is of the type expected by the program. It also ensures that certain data (e.g., username) is not duplicated. In case of any incorrect or duplicate input (when it should be unique), the program informs the user of the errors in question.



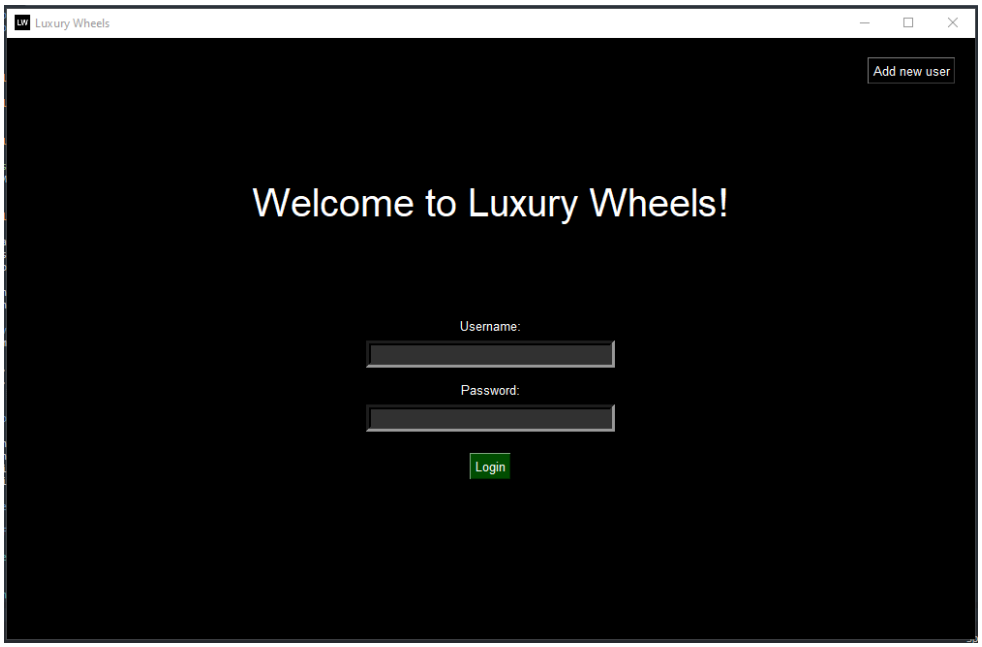






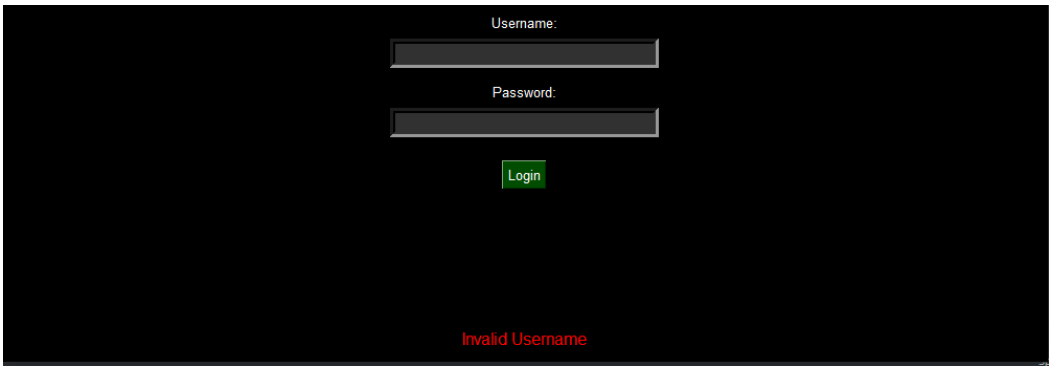
When starting the program, the name of the company is displayed:

This presentation lasts for 3 seconds. After this time, the login window is displayed:

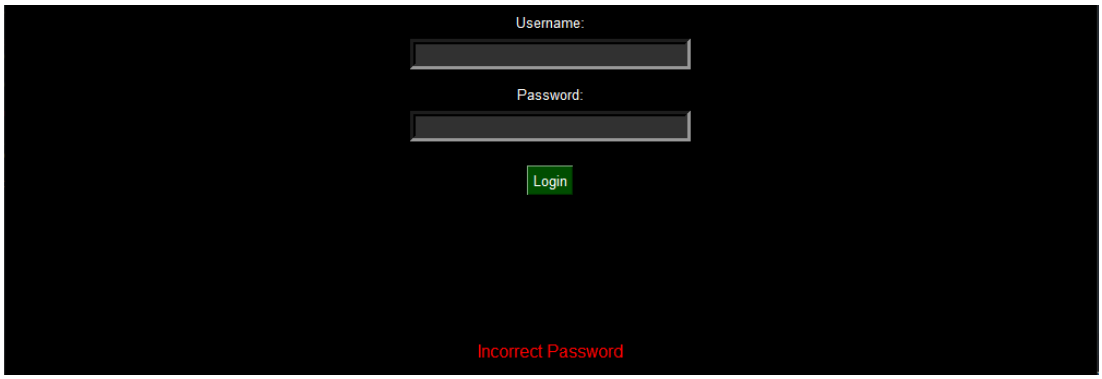


To perform the login, the user must enter their username and password. The program verifies if there are any errors in both pieces of data.

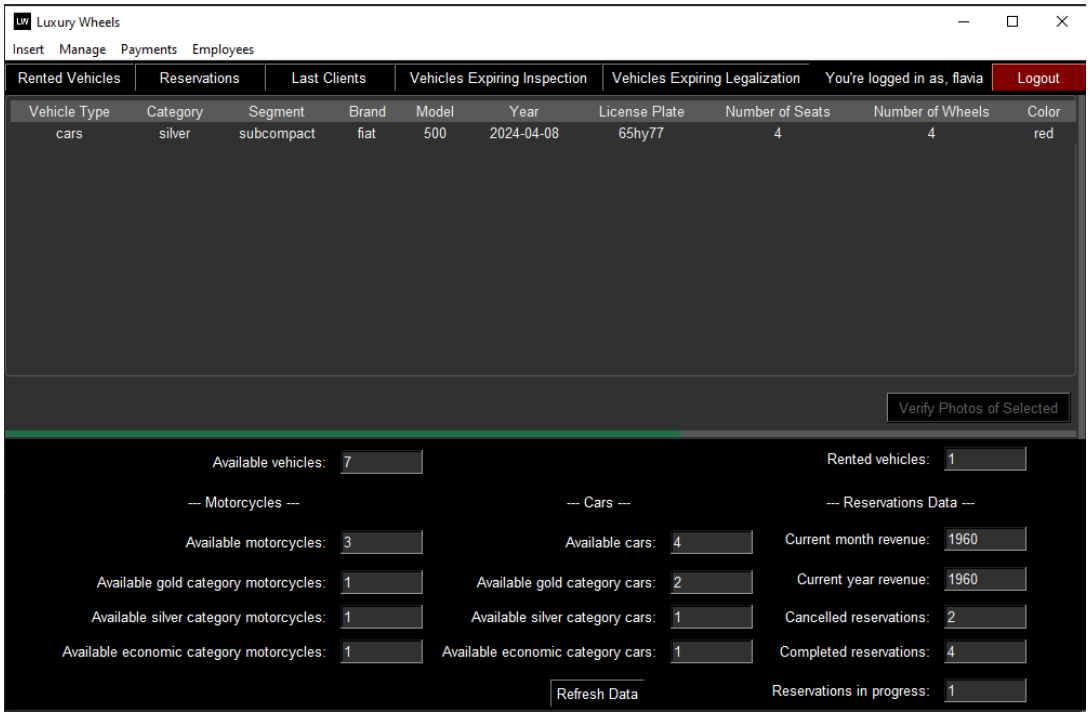
If an incorrect username is entered:



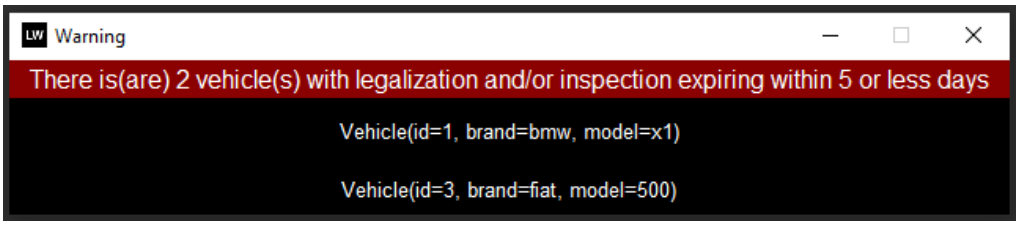
If an incorrect password is entered:



When the entered data is correct, the main window is displayed:

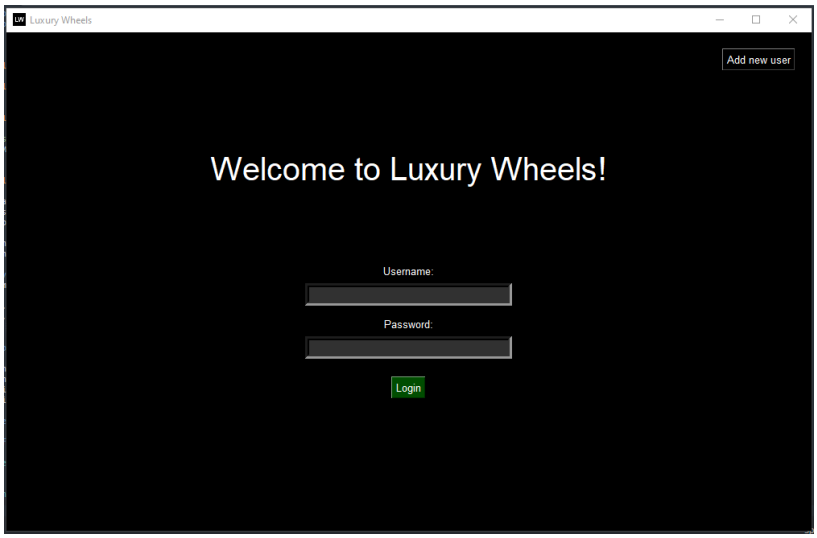


If the user is a manager, and if any vehicle has its legalization/inspection expiring, the manager is alerted to this situation:



The window used for both situations is the same; what changes is the content of the window, depending on whether the user has been successfully authenticated or not.

If the user has not logged in, the visible content will be:



Code responsible for the window above:

```python
    def show_non_authenticated_frame(self, error_message=None, success_message=None):
        button_text = "Add new user"
        button_command = self.new_employee
        add_user_button = tk.Button(self.root, text=button_text, command=button_command, fg="white", bg="black")
        add_user_button.pack(side=tk.TOP, anchor=tk.NE, pady=(20, 0), padx=(0, 20))

        label_text = "Welcome to Luxury Wheels!"
        label_font = ("Helvetica", 30)
        label1 = tk.Label(self.root, text=label_text, font=label_font, fg="white", bg="black")
        label1.pack(expand=True, pady=(30, 20))

        username_label = tk.Label(self.root, text="Username:", fg="white", bg="black")
        username_label.pack(side=tk.TOP, padx=10, pady=2)
        username_entry = Entry(self.root, width=35, borderwidth=5)
        username_entry.pack(side=tk.TOP, pady=2)

        password_label = tk.Label(self.root, text="Password:", fg="white", bg="black")
        password_label.pack(side=tk.TOP, padx=10, pady=(10, 2))
        password_entry = Entry(self.root, show="*", width=35, borderwidth=5)
        password_entry.pack(side=tk.TOP, pady=2)

        def login():
            try:
                existing_user = Employee.query.filter(Employee.username.ilike(username_entry.get())).first()

                if existing_user:
                    user_password = existing_user.password
                    if bcrypt.check_password_hash(user_password, password_entry.get()):
                        authenticated_username = existing_user.username
                        self.show_main_window(authenticated=True, authenticated_username=authenticated_username,
                                              success_message="User logged in successfully!")
                    else:
                        self.show_main_window(authenticated=False, error_message="Incorrect Password")
                else:
                    self.show_main_window(authenticated=False, error_message="Invalid Username")
            except Exception as e:
                print(e)


        login_button = tk.Button(self.root, text="Login", command=login, fg="white", bg="#004d00")
        login_button.pack(side=tk.TOP, pady=(20, 130))
        self.green_bind_hover_effects(login_button)
        self.bind_hover_effects(add_user_button)

        error_label = tk.Label(self.root, text=error_message or "", foreground="red", background="black",
                               font=("Helvetica", 12))
        error_label.pack(side=tk.TOP, pady=(0, 10))

        if success_message:
            success_label = tk.Label(self.root, text=success_message, font=("Helvetica", 12),
                                     fg="green", bg="black")
            success_label.pack(side=tk.TOP, pady=(0, 10))
            self.root.after(3000, lambda: success_label.destroy())
```

Every time we click on the login button, the program verifies the username and password:

```python
    def login():
        try:
            existing_user = Employee.query.filter(Employee.username.ilike(username_entry.get())).first()

            if existing_user:
                user_password = existing_user.password
                if bcrypt.check_password_hash(user_password, password_entry.get()):
                    authenticated_username = existing_user.username
                    self.show_main_window(authenticated=True, authenticated_username=authenticated_username,
                                          success_message="User logged in successfully!")
                else:
                    self.show_main_window(authenticated=False, error_message="Incorrect Password")
            else:
                self.show_main_window(authenticated=False, error_message="Invalid Username")
        except Exception as e:
            print(e)
```

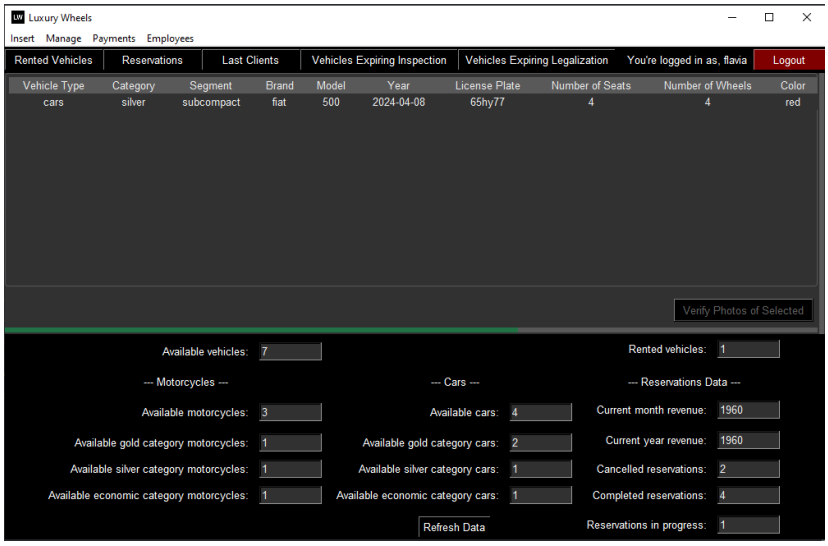
If the data is valid, the program receives information that the user has been authenticated:

```python
self.show_main_window(authenticated=True, authenticated_username=authenticated_username,
                      success_message="User logged in successfully!")
```

The 'show_main_window' method takes the authentication confirmation as a parameter, and the initial dashboard will be displayed:

```python
def show_main_window(self, authenticated=False, authenticated_username=None, success_message=None, error_message=None):
    for widget in self.root.winfo_children():
        widget.destroy()

    with self.flask_app.app_context():
        db.create_all()

    self.canvas.destroy()
    self.root.withdraw()
    self.root.overrideredirect(False)
    self.root.title("Luxury Wheels")
    self.root.resizable(1, 1)
    self.root.iconphoto(True, PhotoImage(file='resources/lw.png'))
    self.root.geometry(f"{self.WINDOW_WIDTH}x{self.WINDOW_HEIGHT}")
    self.root.deiconify()

    if authenticated:
        self.show_authenticated_frame(authenticated_username, success_message)
    else:
        self.show_non_authenticated_frame(error_message, success_message)
```

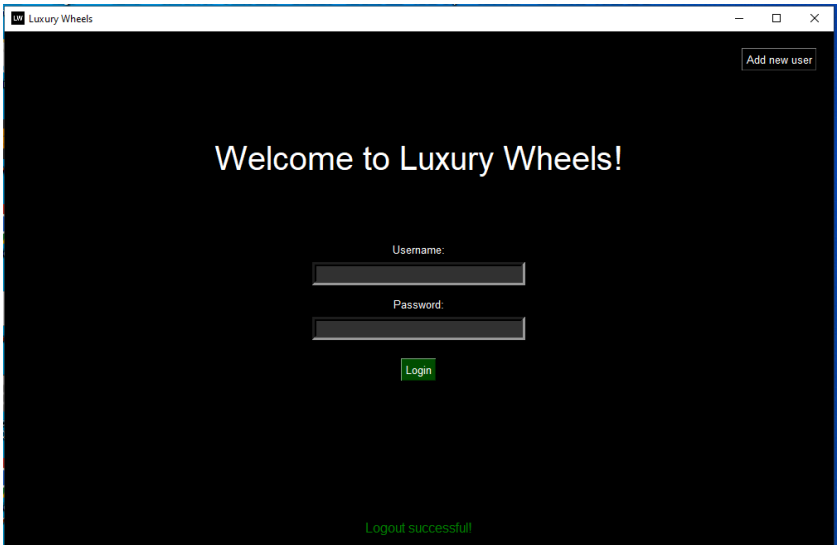
Window after authentication:



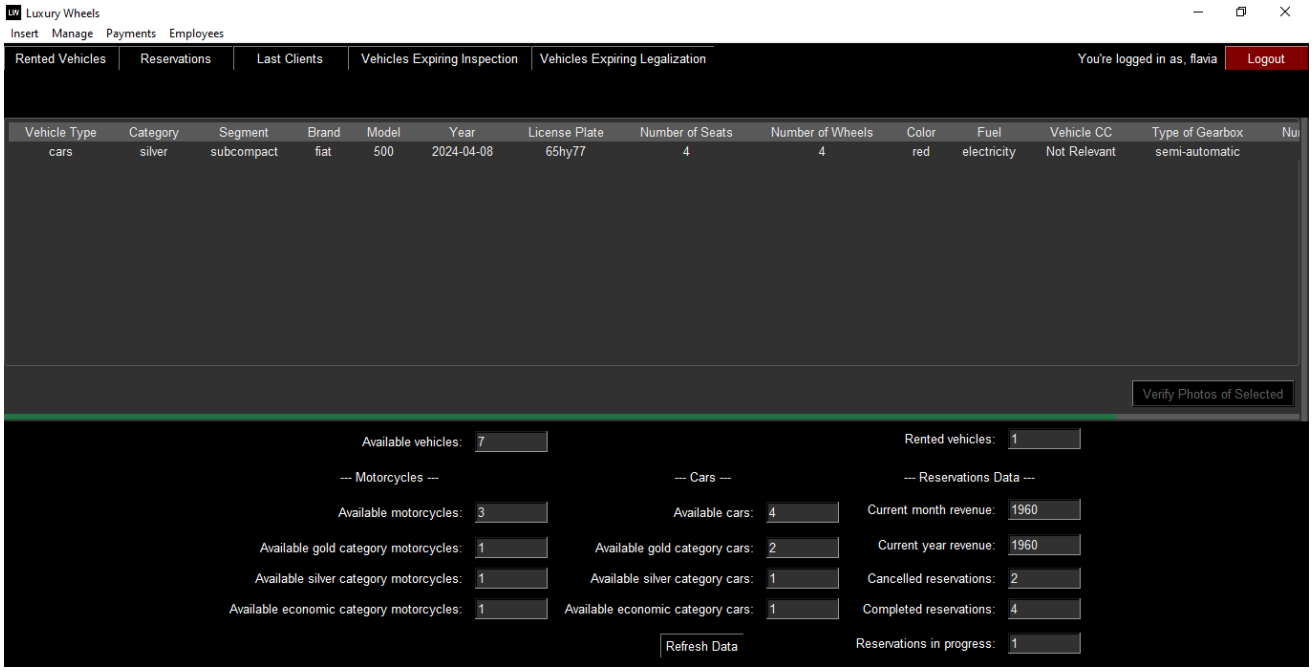
When we click on the logout button, the program receives information that the user is no longer authenticated and presents the content that allows the user to login again:

```python
def logout():
    print("Cleanup tasks completed. Logging out...")
    success_message = "Logout successful!"
    self.show_main_window(authenticated=False, success_message=success_message)
```

Window after logout:



Initial Dashboard



After login, an initial dashboard is presented containing the following information:

- Available vehicles – number of available vehicles;
- Rented vehicles – number of rented vehicles;
- Available motorcycles – number of available motorcycles;
- Available gold category motorcycles – number of available gold category motorcycles;
- Available silver category motorcycles – number of available silver category motorcycles;
- Available economic category motorcycles – number of available economic category motorcycles;
- Available cars – number of available cars;
- Available gold category cars – number of available gold category cars;
- Available silver category cars – number of available silver category cars;
- Available economic category cars– number of available economic category cars;
- Current month revenue – revenue of the current month, for these calculations, only completed reservations are taken into account;
- Current year revenue – revenue of the current year, for these calculations, only completed reservations are taken into account;
- Cancelled reservations – number of cancelled reservations;
- Completed reservations – number of completed reservations;
- Reservations in progress – number of reservations in progress.

This information is automatically obtained every time the login is performed. While using the program, for example, if we cancel a reservation, we can update this information using the “Refresh Data” button.

Piece of code responsible for retrieving information from the database, performing calculations (using that information) when necessary (e.g., calculating revenues), and inserting the data into the respective text boxes:

```python
def refresh_data():
    vehicles_available = Vehicle.query.filter_by(availability="Available").all()
    reservations_in_progress = Reservation.query.filter_by(reservation_state="In progress").all()

    current_month = datetime.now().month
    current_year = datetime.now().year

    current_month_reservations = Reservation.query.filter(
        extract('month', Reservation.insertion_date) == current_month,
        extract('year', Reservation.insertion_date) == current_year,
        Reservation.reservation_state != 'Cancelled'
    ).all()

    total_revenue_current_month = sum(reservation.total_cost for reservation in current_month_reservations)

    current_year_reservations = Reservation.query.filter(
        extract('year', Reservation.insertion_date) == current_year,
        Reservation.reservation_state != 'Cancelled'
    ).all()

    total_revenue_current_year = sum(reservation.total_cost for reservation in current_year_reservations)
    cancelled_reservations = Reservation.query.filter(Reservation.reservation_state == 'Cancelled').all()
    completed_reservations = Reservation.query.filter(Reservation.reservation_state == 'Completed').all()
    in_progress_reservations = Reservation.query.filter(Reservation.reservation_state == 'In progress').all()
    motorcycles_available = Vehicle.query.filter_by(availability="Available", vehicle_type="motorcycles").all()
    economic_motorcycles_available = Vehicle.query.filter_by(availability="Available", vehicle_type="motorcycles", category="economic").all()
    silver_motorcycles_available = Vehicle.query.filter_by(availability="Available", vehicle_type="motorcycles", category="silver").all()
    gold_motorcycles_available = Vehicle.query.filter_by(availability="Available", vehicle_type="motorcycles", category="gold").all()
    cars_available = Vehicle.query.filter_by(availability="Available", vehicle_type="cars").all()
    economic_cars_available = Vehicle.query.filter_by(availability="Available", vehicle_type="cars", category="economic").all()
    silver_cars_available = Vehicle.query.filter_by(availability="Available", vehicle_type="cars", category="silver").all()
    gold_cars_available = Vehicle.query.filter_by(availability="Available", vehicle_type="cars", category="gold").all()

    read_only = [amount_available_entry, amount_rented_entry, month_revenue_entry, year_revenue_entry, amount_reservations_cancelled_entry,
    amount_reservations_completed_entry, amount_reservations_in_progress_entry, available_cars_entry, available_gold_cars_entry, available_silver_cars_entry,
    available_economic_cars_entry, available_motorcycles_entry, available_gold_motorcycles_entry, available_silver_motorcycles_entry, available_economic_motorcycles_entry]

    for entry in read_only:
        entry.config(state=tk.NORMAL)
        entry.delete(0, END)

    amount_available_entry.insert(0, len(vehicles_available))
    amount_rented_entry.insert(0, len(in_progress_reservations))
    month_revenue_entry.insert(0, total_revenue_current_month)
    year_revenue_entry.insert(0, total_revenue_current_year)
    amount_reservations_cancelled_entry.insert(0, len(cancelled_reservations))
    amount_reservations_completed_entry.insert(0, len(completed_reservations))
    amount_reservations_in_progress_entry.insert(0, len(in_progress_reservations))
    available_motorcycles_entry.insert(0, len(motorcycles_available))
    available_economic_motorcycles_entry.insert(0, len(economic_motorcycles_available))
    available_silver_motorcycles_entry.insert(0, len(silver_motorcycles_available))
    available_gold_motorcycles_entry.insert(0, len(gold_motorcycles_available))
    available_cars_entry.insert(0, len(cars_available))
    available_economic_cars_entry.insert(0, len(economic_cars_available))
    available_silver_cars_entry.insert(0, len(silver_cars_available))
    available_gold_cars_entry.insert(0, len(gold_cars_available))

    for entry in read_only:
        entry.config(state="readonly")
```

In addition to the information described above, the initial dashboard also contains the following buttons:

Rented Vehicles	Reservations	Last Clients	Vehicles Expiring Inspection	Vehicles Expiring Legalization

- Rented Vehicles;
- Reservations;
- Last Clients;
- Vehicles Expiring Inspection;
- Vehicles Expiring Legalization.

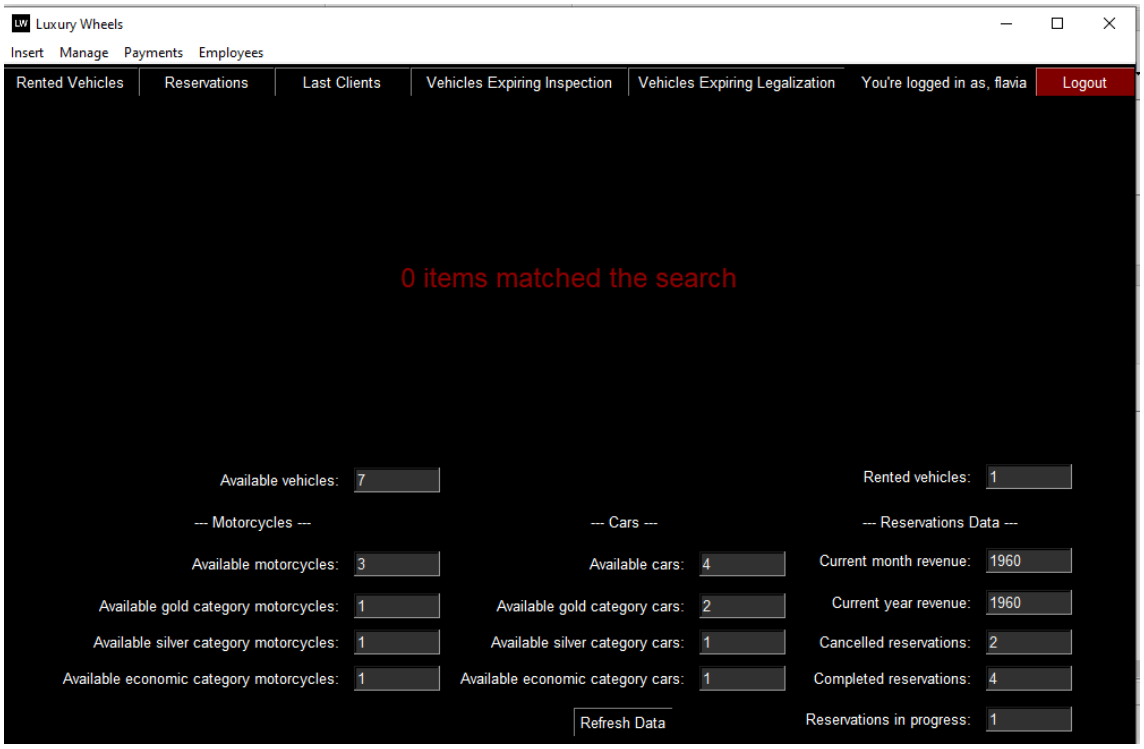
When clicking on any of these buttons, a table with the related information is displayed.

For example, clicking on the "Reservations" button displays a table with all reservations that are in the database:

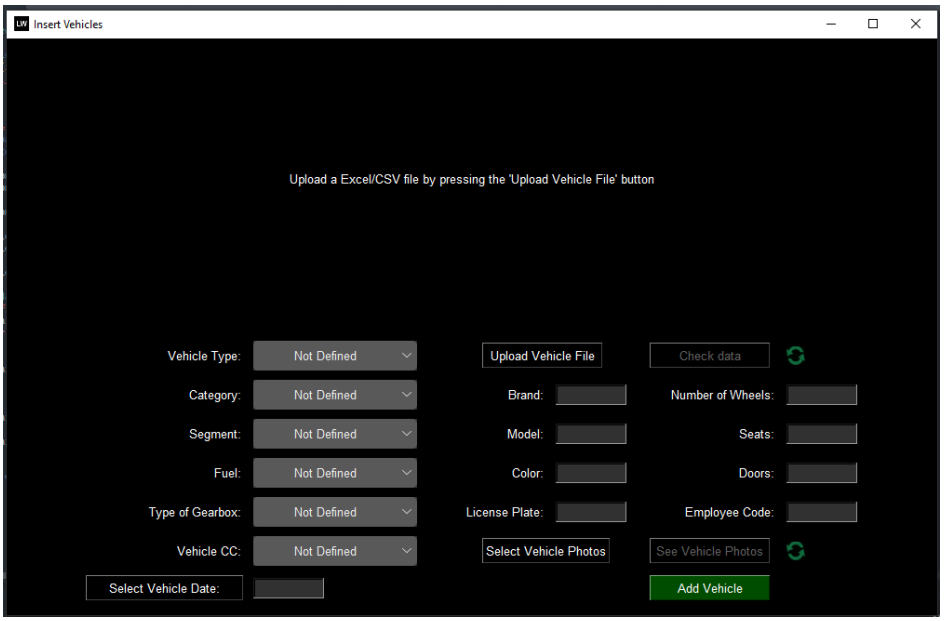
Pick-up Date	Drop-off Date	Vehicle License Plate	Rental Duration	Cost Per Day	Total Cost	Full Name	Phone Number	Email	ID/Passport Number	Date of Birth	Na
2024-04-16	2024-04-16	12qw43	1	80	80	flavia	91111111	flagmailcom	145862154	2006-04-03	Por
2024-04-15	2024-04-15	12qw43	1	80	80	flavia	91111111	flagmailcom	145862154	2006-04-03	Por
2024-04-15	2024-04-17	45jh87	3	120	360	tiago	9111111	tlgmailcom	34653222	2006-04-12	Por
2024-04-14	2024-04-14	45jh87	1	120	120	flavia	91111111	flagmailcom	145862154	2006-04-03	Por
2024-04-15	2024-04-26	45jh87	12	120	1440	tiago	9111111	tlgmailcom	34653222	2006-04-12	Por
2024-03-09	2024-03-19	12qw43	11	80	880	tiago	9111111	tlgmailcom	34653222	2006-04-12	Por
2024-04-16	2024-04-25	65hy77	10	80	800	tiago	9111111	tlgmailcom	34653222	2006-04-12	Por

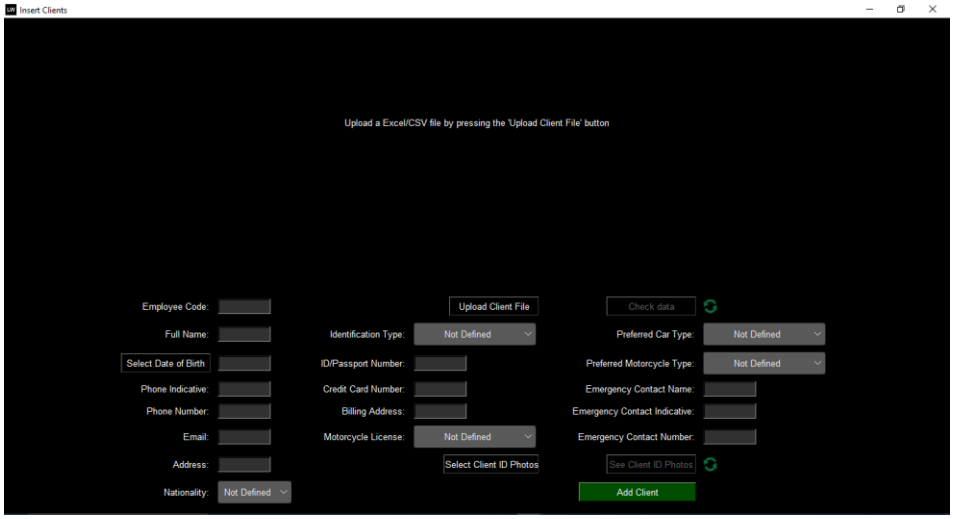
Verify Photos of Selected

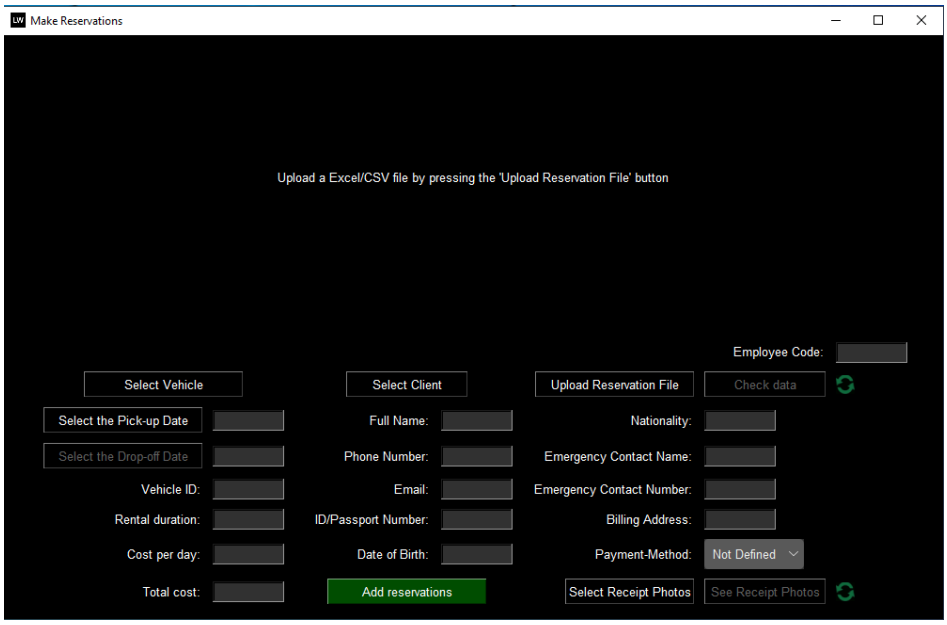
If, for example, you click the button to view vehicles with expiring legalization and/or inspection, and at that moment there are none to which this applies, the program will not have data to create the table. In this case, the program displays the following warning:



Sections to insert Vehicles/Clients/Reservations/Payments:







All sections have the following functions in common:

```
2449        def check_data(): ...
3445
3446        def check_if_df(): ...
3454
3455        def check_if_photos(): ...
```

def check_data():

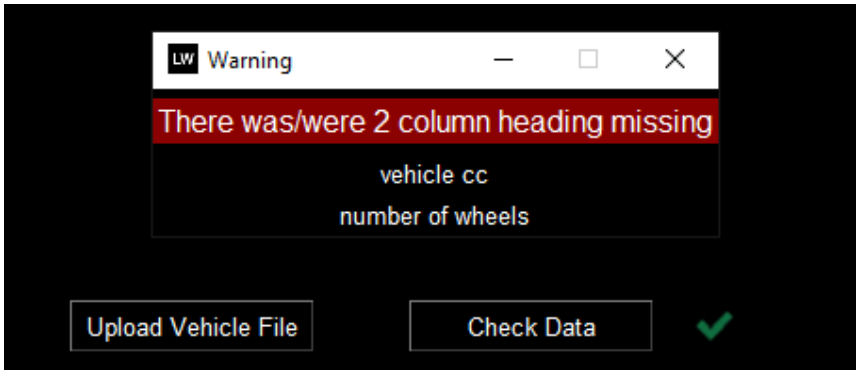
Responsible for verifying if the CSV/Excel file is compatible with what is expected by that section, as well as the information it contains.

It checks if the file is compatible with the expected format as follows:

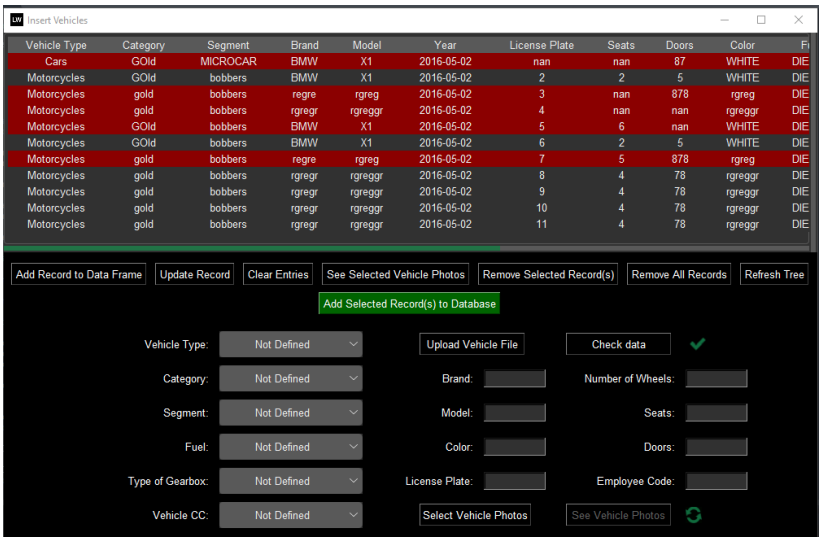
- Checks if the file contains the necessary columns for all the information that the program needs;
- Checks if there are extra columns or if any column is missing.

In the event of any of the above errors occurring, the program informs the user about the error in question.

<u>Error when trying to open a vehicles file where two of the columns are missing:</u>



If no errors occur, the program loads the file and displays its data:



Within the function `check_data()`, we have the following functions:

```
def verify_data():
```

- Analyzes the data in each row and column, if an error occurs (e.g., empty cells or wrong input type), the row containing the error is highlighted in red, as shown in the table image on the previous page.

```
def refresh_tree():
```

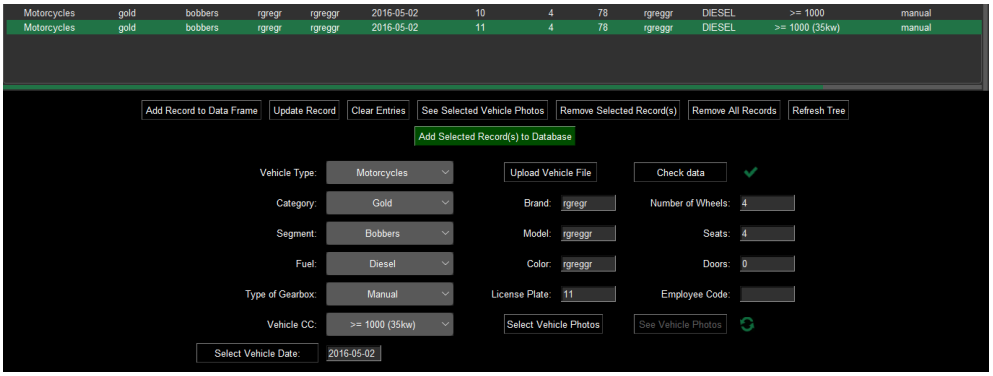
- Updates the information we see, necessary after making any changes to the DataFrame (e.g., deleting an element);
- With the data from the inserted file, we create the DataFrame that we subsequently use to input information into the table (Treeview widget); for this reason, it is necessary to update the table after making any changes to the DataFrame.

```
def clear_entries():
```

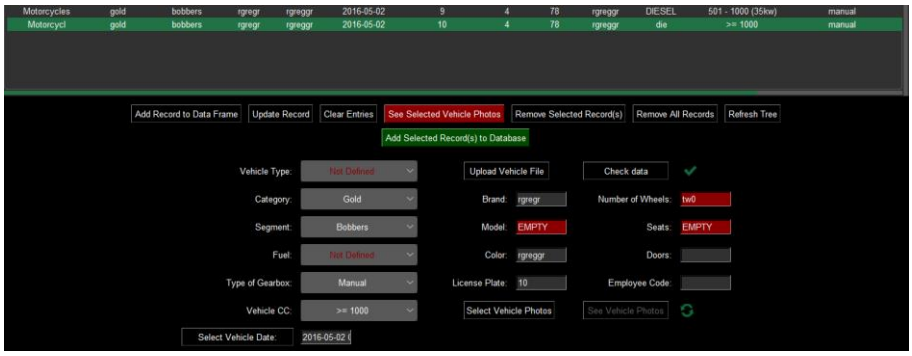
- Clears the text entries/selection options;
- Restores the original color of the text/text entries/buttons (in case any errors occurred previously);
- Triggered by clicking the button or also used automatically throughout the program (e.g., after adding a vehicle to the database).

```
def select_record(e):
```

- Automatically fills the text entries/options with the respective information when selecting one of the elements in the table.



- If the row of the selected element is red, as a way to easily identify the error, the entries/options that contain the error are also highlighted in red.



```
def update_record():
```

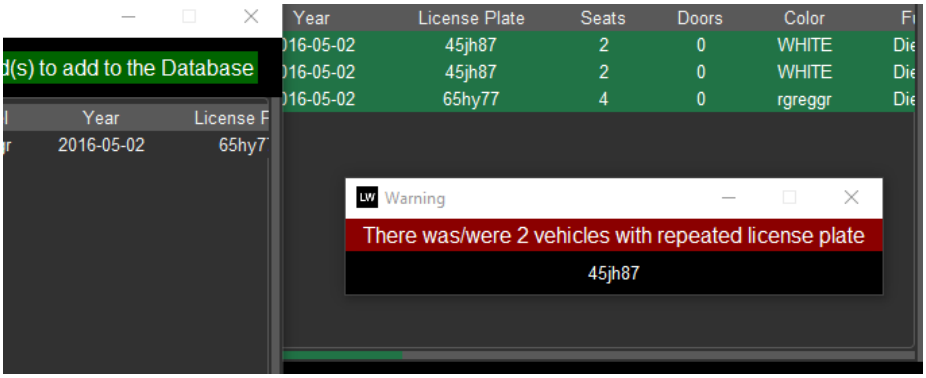
- Allows updating the data of the selected element in the table, so the user can update or correct any incorrect data without having to make that change in the file.

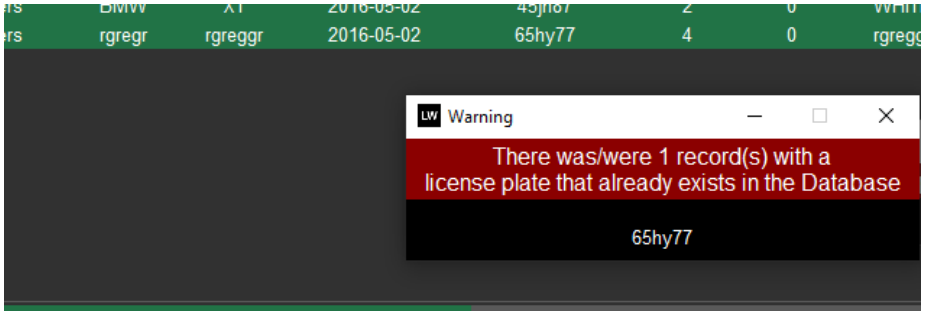
```
def remove_selected():
```

- Used to delete one or more elements from the table.

```
def add_selected_to_database():
```

- Used to add one or more elements from the table to the database;
- In addition to the normal data verification seen earlier, it also checks if unique data is not repeated in the selected elements or if it already exists in the database.





- In the Clients and Reservations insert section, other data must be unique, so in each section, the verification parameters are specific.

```
def add_record():
```

- Allows adding new elements to the table.

```
def remove_all():
```

- Removes all elements from the table.

```
def selected_vehicle_photos():
```

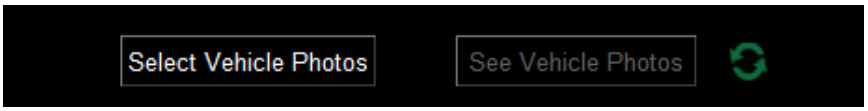
- Opens the photo viewing window of the selected element.

def check_if_df():

It checks if a file has been uploaded when the update button is clicked, if yes, the button to view the table becomes active:

def check_if_photos():

It checks if a photo has been uploaded when the update button is clicked; if yes, the button to view photos becomes active:



In addition to these functions common to all sections, each section also has a function that allows inserting Vehicles/Clients/Reservations into their respective tables in the database, without needing to use a data file.

Insert Vehicles

```
3469        def add_vehicle_db():
```

Insert Clients

```
4914        def add_client_db():
```

Insert Reservations

```
6431        def add_reservations_db():
```

In the section where we can make/insert Reservations, we have some functions that are not present in the sections for inserting Vehicles and Clients:

- costs_of_rent;
- client_info.

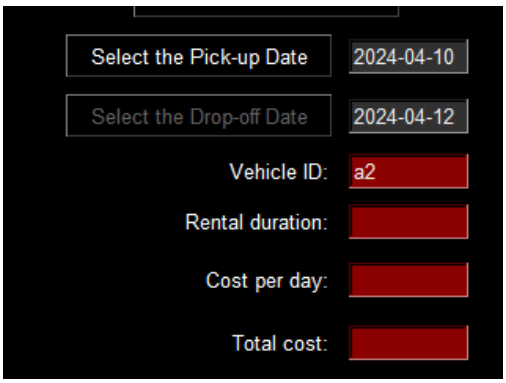
```
def costs_of_rent(*args): ...
```

Automatically calculates the duration of the reservation (in days). With the result of this calculation and after obtaining the category of the chosen vehicle (Gold/Silver/Economic), it presents the daily cost and the total cost of the reservation for that vehicle.

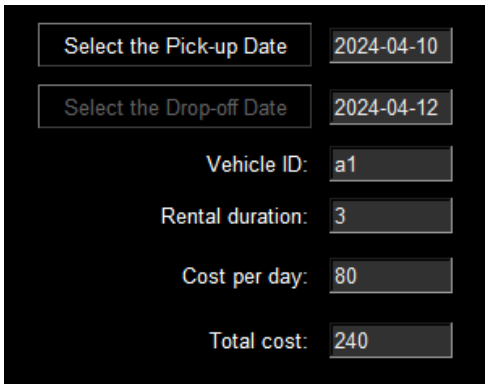
All these automatic calculations are only performed after selecting:

- Start date of reservation;
- End date of reservation;
- Enter a valid license plate (which is in the database). The license plate is what the program will use to identify the vehicle and obtain the category to use in the calculations. If the license plate is incorrect, the text entries turn red.
- The user can enter the license plate by typing it into the text entry or by using a button that allows viewing the vehicles in the database and selecting from there (this functionality will be presented later).

License plate not found



License plate found

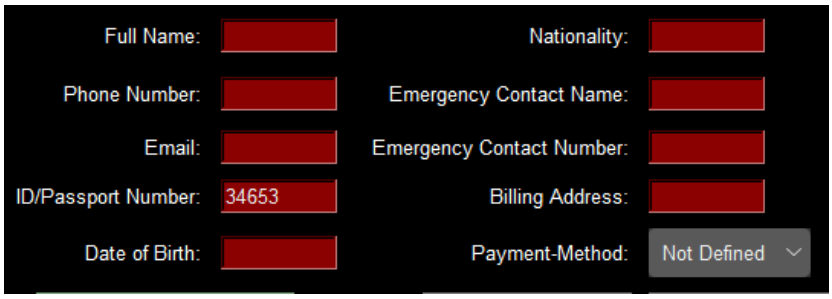



```
def client_info(*args): ...
```

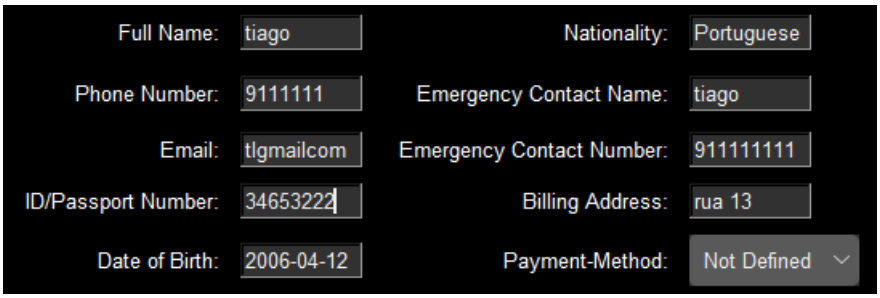
Similar to the previous function, this one also retrieves information from the database, automatically, in this case, the data of the client to whom the identification number entered in the text entry belongs. If the entered identification number does not exist in the database, the text entries turn red.

The user can enter the identification number by typing it into the text entry or by using a button that allows viewing the clients in the database and selecting from there (this functionality will be presented later).

Client not found

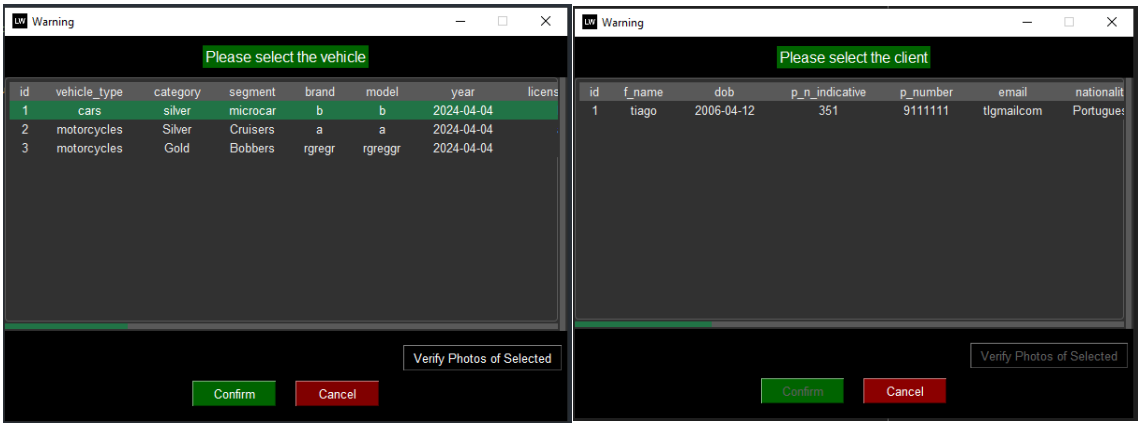


Client found



<u>Buttons to select the Vehicle/Client directly from the Database</u>

These buttons allow you to view and select, using the notification window, the vehicles/clients that are in the database, making it easier for the user to select the correct vehicle/client.



In the case of the vehicle selection window, when selecting the vehicle, it checks whether the vehicle is available or unavailable. The button to confirm the selection only becomes active if the selected vehicle is available.

Piece of code that checks the availability of the selected vehicle:

```
if str(df.at[x, 'availability']) == 'Available':
    confirm_record_selection_button.config(state=NORMAL)
else:
    confirm_record_selection_button.config(state=DISABLED)
```

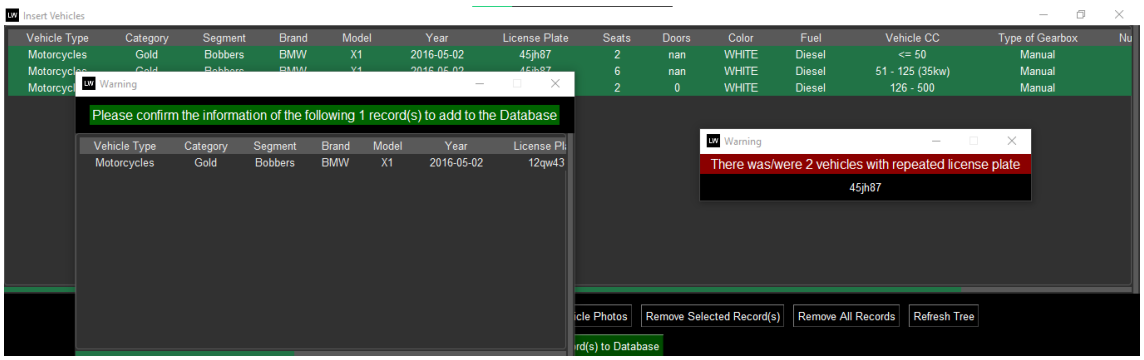
Specific verifications and data changes when inserting Vehicles/Clients/Reservations

Insert Vehicles:

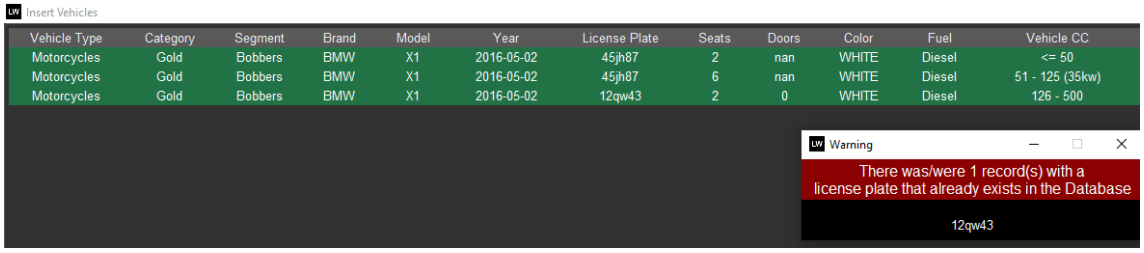
Verifications

- If we are inserting more than one vehicle at a time, the program checks if there are duplicated license plates in the selected items. After this verification, it also ensures that none of the license plates in the list exists in the database. If any duplicated or existing license plates are found, the respective item is not inserted. After inserting the remaining items, the user is alerted to the existing license plate;
- If we are inserting only one item, it checks if the license plate entered by the user does not exists in the database. If it exists, the item is not added, and the user is informed that the license plate already exists in the database.

Warning of vehicles in the list with duplicated license plates:



Warning of existing license plate in the Database:



Changes in the Database after inserting a Vehicle

availability	rented	code_rented	for_inspection	code_inspection	for_legalization	code_legalization	last_update	code_last_update	next_inspection	next_legalization	insertion_date	code_insertion
Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter
Available	No	Employee Code	No	Employee Code	No	Employee Code	No previous update	Employee Code	2025-04-04	2025-04-07	2024-04-07	2122

After inserting a vehicle into the Database, the following columns are automatically filled:

- availability – column that will identify the availability of the vehicle throughout the use of the program;
- rented – column to identify if the vehicle is currently rented;
- code_rented – code of the employee/user who confirmed the rental of the vehicle;
- for_inspection – column to identify if the vehicle was sent for inspection;
- code_inspection – code of the employee/user who sent the vehicle for inspection;

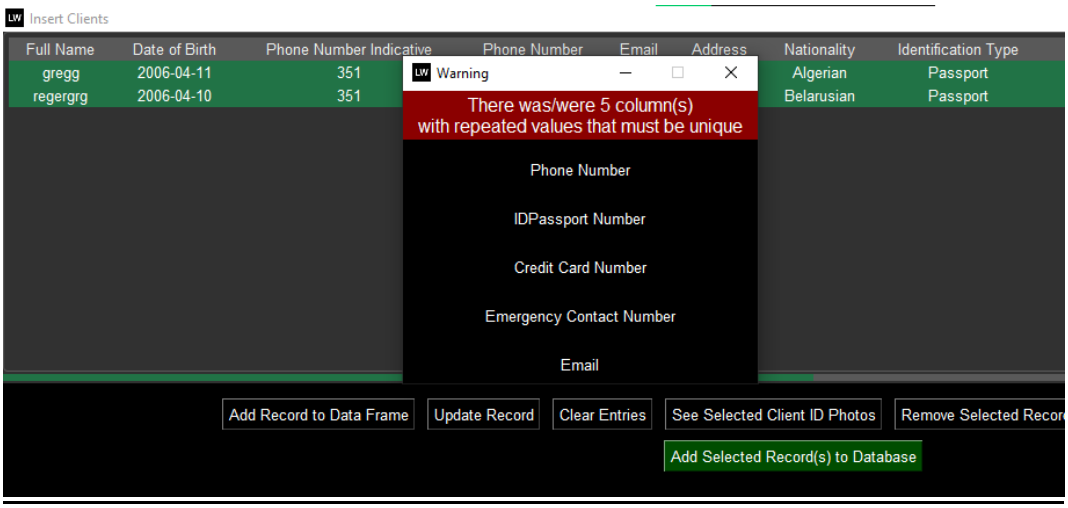
- for_legalization – column to identify if the vehicle was sent for legalization;
- code_legalization – code of the employee/user who sent the vehicle for legalization;
- last_update – date of the last update of data for this vehicle;
- code_last_update – code of the employee/user who made the last update;
- next_inspection – date of the next inspection (date obtained using the "calculate_date" function, previously seen);
- next_legalization – date of the next legalization (date obtained using the "calculate_date" function, previously seen);
- insertion_date – date on which the vehicle was inserted into the Database;
- code_insertion – code of the employee/user who inserted the vehicle into the Database.

Insert Clients:

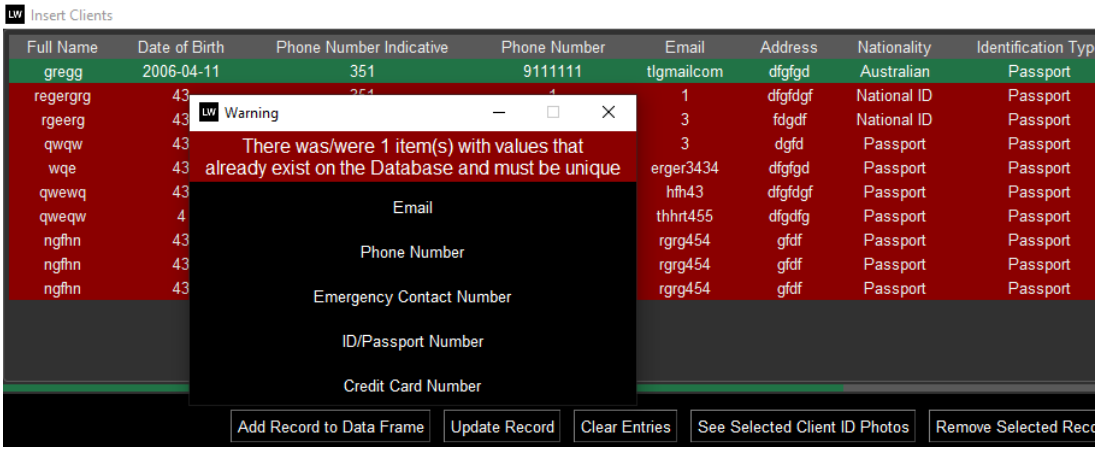
Verifications

- If we are inserting more than one client at once, the program checks if the following data is duplicated among the selected items:
  - Contact number;
  - Email;
  - Identification number;
  - Credit card number;
  - Emergency contact number.
- After this check, it also verifies that none of this data already exists in the database. If any duplicated or existing data is found, the respective item is not inserted. After inserting the remaining items, the user is alerted to the existing data;
- If we are inserting only one item, it checks that the data (which should be unique) entered by the user does not already exists in the Database. If it exists, the item is not added, and the user is informed about the existing data in the Database.

Warning of clients in the list with duplicated data:



Warning of existing data in the Database:



Changes in the Database after inserting a Client

renting	code_renting	last_update	code_last_update	insertion_date	code_insertion
Filter	Filter	Filter	Filter	Filter	Filter
No	Employee Code	No previous update	Employee Code	2024-04-07	2122

After inserting a client into the Database, the following columns are automatically filled:

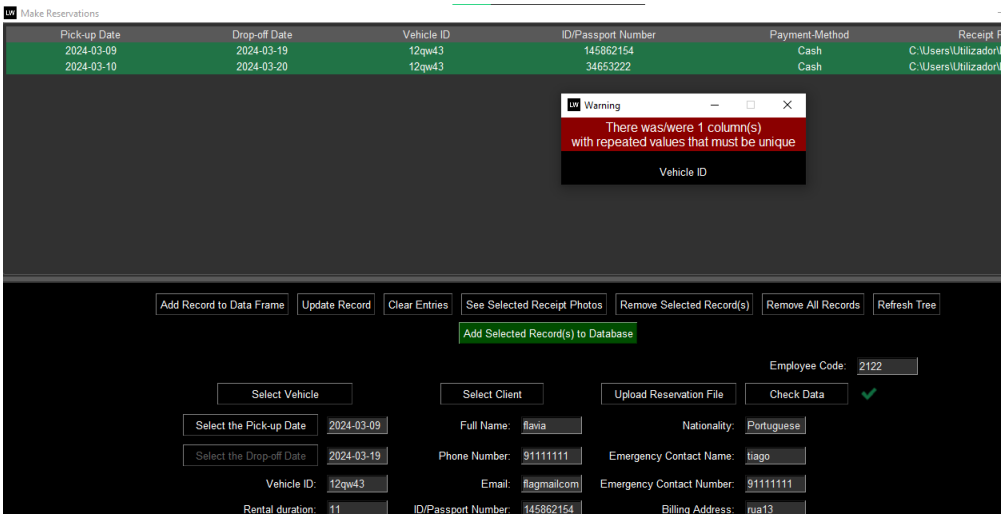
- renting – column to identify if the client currently has any active reservation;
- code_renting – code of the employee/user who confirmed the reservation;
- last_update – date of the last update of this client's data;
- code_last_update – code of the employee/user who performed the last update;
- insertion_date – date on which the client was inserted into the Database;
- code_insertion – code of the employee/user who inserted the client into the Database.

<u>Insert Reservations:</u>

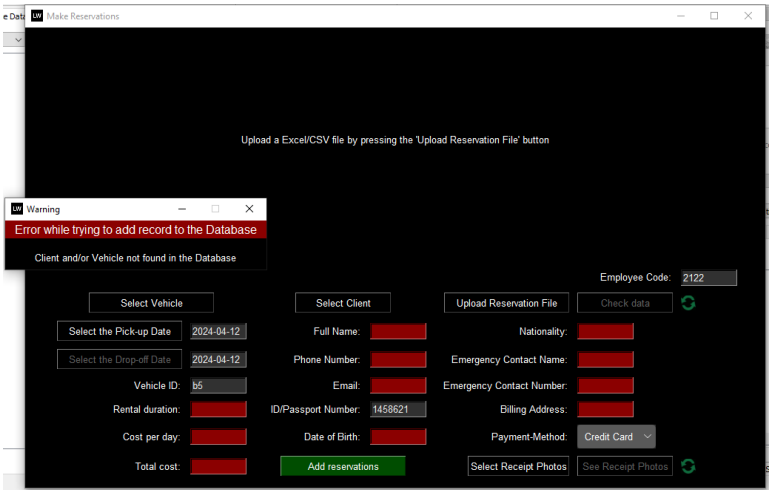
Verifications

- If we are entering more than one reservation at once, the program checks if there are duplicated license plates among the selected items;
- It verifies if the vehicle ID (license plate) and the client ID (identification number) are in the Database, the vehicle and client must have already been inserted into the Database before the reservation;
- It checks if the desired vehicle for that reservation is available; it may be unavailable due to already being rented, needing legalization and/or inspection;
- It verifies if the reservation end date does not exceed the date of the next legalization and/or inspection;
- In the case of the desired vehicle being a motorcycle, it checks if the client in question has a driver's license that enables them to drive vehicles with the engine displacement of the desired vehicle.

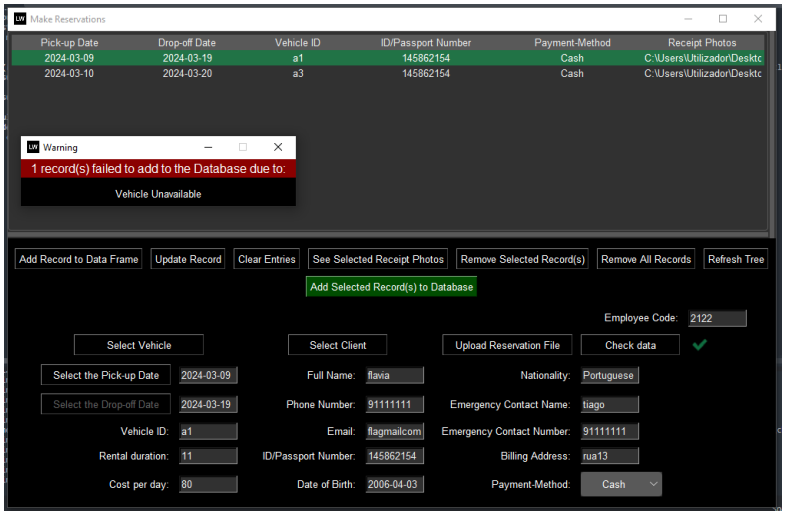
Duplicated license plates in the list of selected items warning:



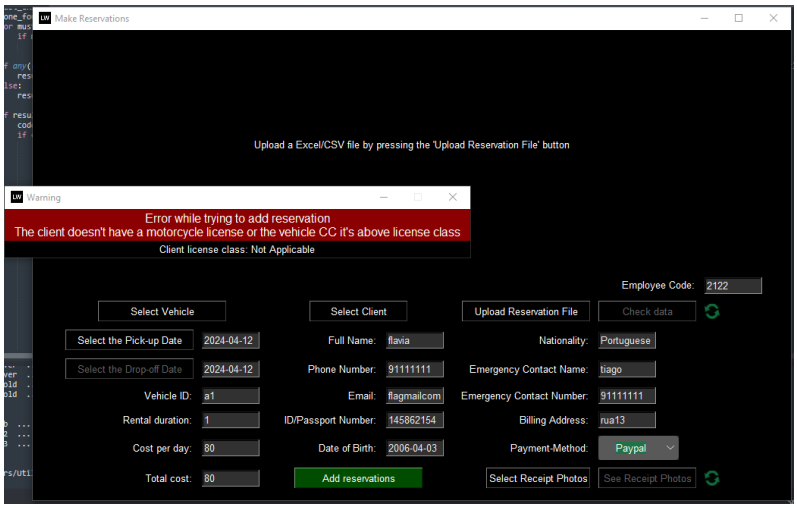
Client and/or Vehicle not found warning:



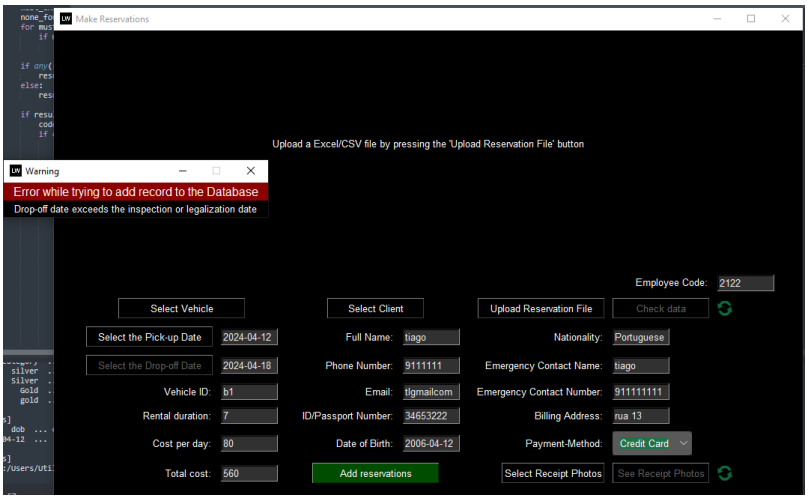
Unavailable vehicle warning:



Lack of client's qualification to drive the intended vehicle warning:



End date of reservation exceeds inspection/legalization date warning:



Changes in the database after inserting reservation

Unlike the other sections, which, upon inserting elements into the database, only modify the information in their respective tables, the reservation section modifies information in its own table, the vehicles table, the clients table, and the payments table.

<u>Reservations Table:</u>

reservation_state	date_confirm_completed_renting	code_confirm_completed_renting	date_cancel_reservation	code_cancel_reservation	last_update	code_last_update	insertion_date	code_insertion
Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter
In progress	Not Completed	Employee Code	Not Cancelled	Employee Code	No previous update	Employee Code	2024-04-12	2122

- reservation_state – state in which the reservation is (in progress/complete/canceled);
- date_confirm_completed_renting – date when the completion of the reservation was confirmed;
- code_confirm_completed_renting – code of the employee/user who confirmed the completion of the reservation;
- date_cancel_reservation – date when the reservation was canceled, if that happens;

- code_cancel_reservation – code of the employee/user who confirmed the cancellation of the reservation;
- last_update – date of the last data update of this reservation;
- code_last_update – code of the employee/user who made the last update;
- insertion_date – date when the reservation was inserted into the Database;
- code_insertion – code of the employee/user who inserted the reservation into the Database.

Vehicles Table:

availability	rented	code_rented
Filter	Filter	Filter
Available	No	Employee Code
Available	No	Employee Code
Unavailable	Yes	2122

When entering a reservation, the vehicle selected for that reservation will have the following changes in the columns above:

- availability – the availability of the vehicle will be changed from available to unavailable, an important piece of information that needs to be kept updated throughout the program because the program checks the availability of the vehicle before performing various actions;
- rented – this column specifies that the reason for the unavailability of the vehicle is due to a reservation that is still in progress;
- code_rented – code of the employee/ user who entered into the Database the reservation in which the vehicle was selected

Clients Table:

renting	code_renting
Filter	Filter
Yes	2122
No	Employee Code

When entering a reservation, the client selected for that reservation will have the following changes in the columns above:

- renting – this column is used by the program to confirm that the client has an ongoing reservation, something the program needs so it can display information about the current vehicles and reservations of that client in other windows;
- code_renting – code of the employee/user who entered into the Database the reservation in which the client was selected.

Payments Table:

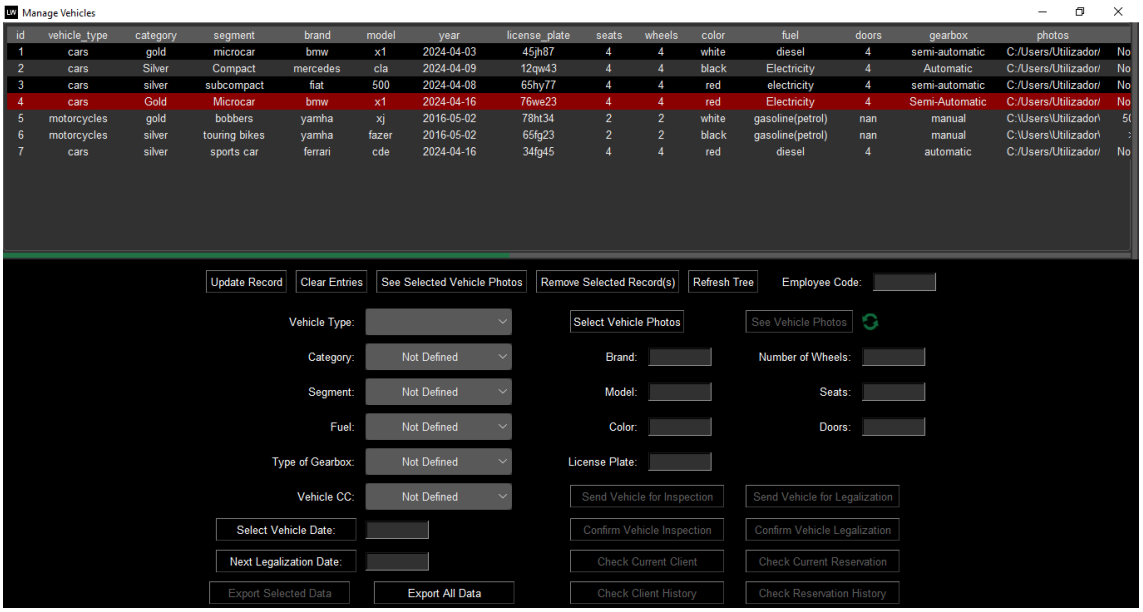
	id	total_pay	payment_method	f_name	p_number	id_number	bill_address	vehicle_id	photos	last_update	code_last_update	insertion_date	code_insertion
	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter
1	1	80	Paypal	tiago	9111111	34653222	rua 13	a1	C:/Users/Utilizador/Desktop/clw2.png	No previous update	Employee Code	2024-04-11	2122
2	2	1920	Credit Card	tiago	9111111	34653222	rua 13	a3	C:/Users/Utilizador/Desktop/date_picker.png	No previous update	Employee Code	2024-04-12	2122

When entering a reservation, in order to facilitate and automate the work of the employee/user, a payment record for the respective reservation is automatically created and inserted into the Database. This new record contains the following information:

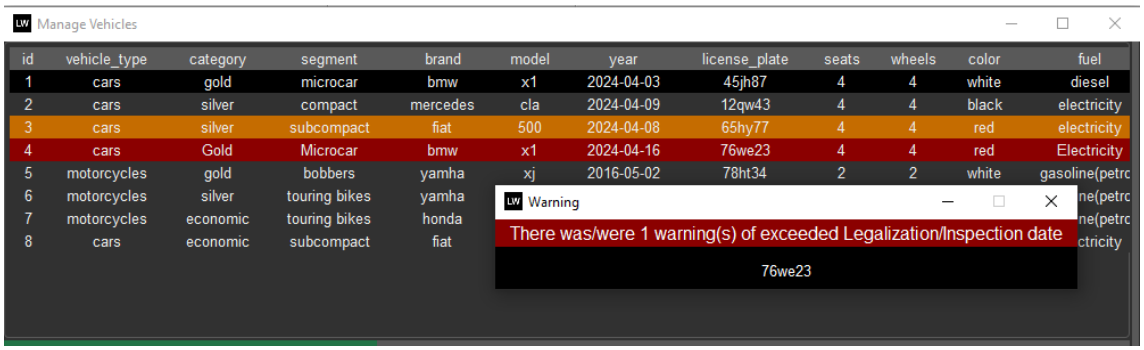
- total_pay – the total payment amount for that reservation;
- payment_method – payment method used by the client;
- f_name (Full name) – full name of the client;
- p_number (Phone number) – client's contact number;
- id_number – client's identification number;
- bill_address – billing address;
- vehicle_id – vehicle ID (license plate);
- photos – photos of the payment receipt;
- last_update – date of the last data update of this payment;
- code_last_update – code of the employee/user who made the last update;
- insertion_date – date when the payment record was inserted into the Database;
- code_insertion – code of the employee/user who inserted the record into the Database.

Sections for managing Vehicles/Clients/Reservations/Payments:

Manage Vehicles



From this section onwards, all actions are constantly synchronized with the information stored in the Database.



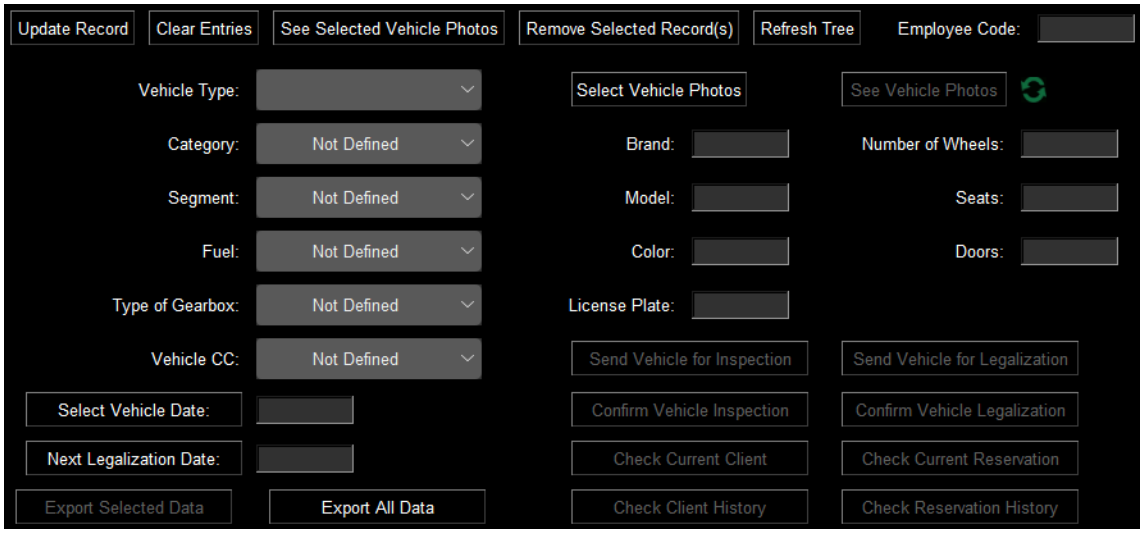
When opening the window, the employee/user, if a manager, is informed if any vehicle has exceeded the inspection and/or legalization date, or if the current day is the last day to do so.

The different colors observed in the table serve to alert the employee/user about the following possible situations:

- Normal color (id 2) – the color the table row has when there is no situation to highlight regarding that vehicle;
- Black color (id 1) – the color the table row has when the vehicle is unavailable due to ongoing rental/legalization/inspection;
- Red color (id 4) – the color the table row has when the legalization and/or inspection date of the vehicle has been exceeded;
- Orange color (id 3) – the color the table row has when the legalization and/or inspection date of the vehicle is less than 15 days away from being exceeded.

<u>Now let's analyze what happens with certain buttons when selecting the different vehicles above, with the different possible situations.</u>

Buttons state without any selection

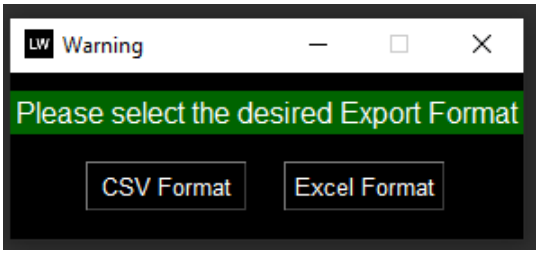


Buttons description:

Buttons to export table information

- Export all data - button to export all elements from the table, in CSV or Excel format;
- Export selected data - button that will become active when one or more elements of the table are selected, used to export the information of the selected ones, in CSV or Excel format.

When clicking on any of these buttons, the program uses the warning window so that the user can select the desired format.

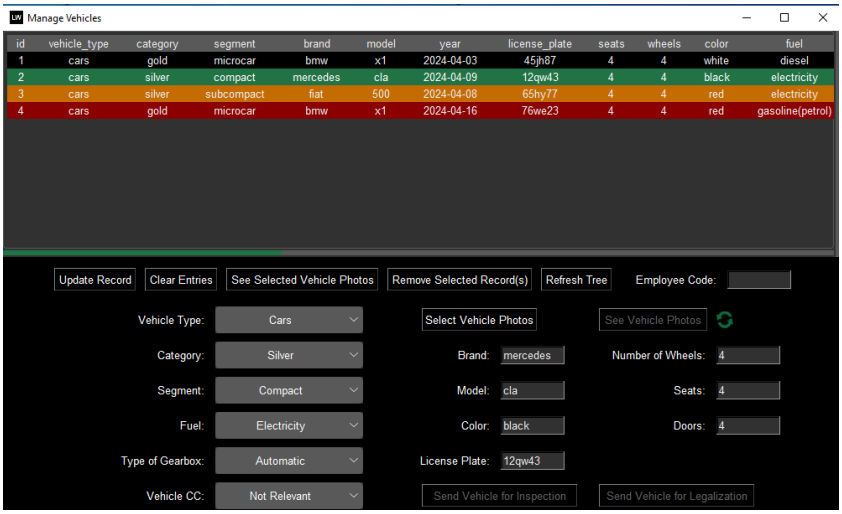


Buttons to send the vehicle for Inspection and Legalization

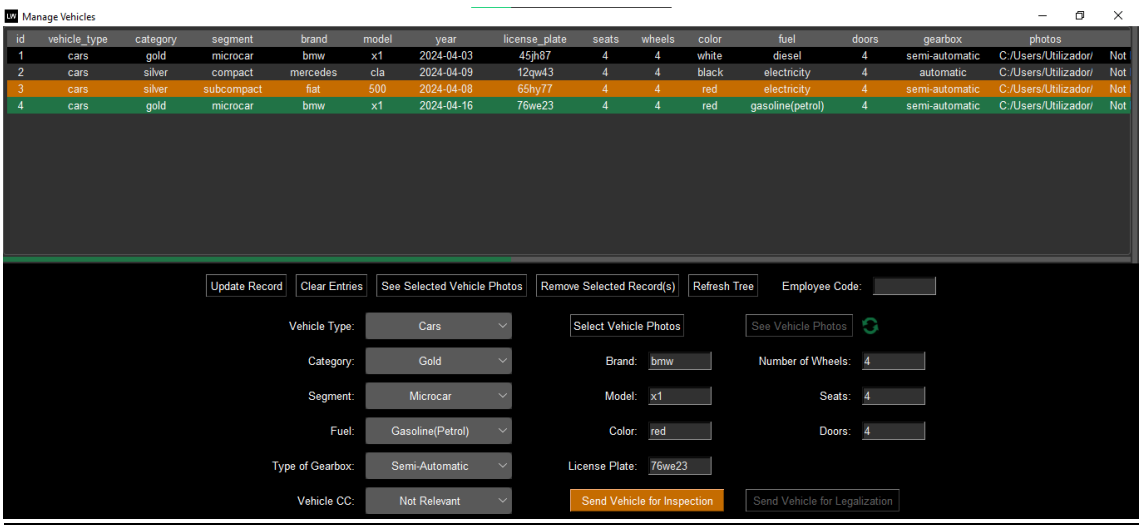
- Send vehicle for inspection – button to send the vehicle for inspection;
- Send vehicle for legalization – button to send the vehicle for legalization.

These two buttons are initially disabled, and they only become active if the selected vehicle requires inspection and/or legalization. Only the button relevant to the necessary action will be activated.

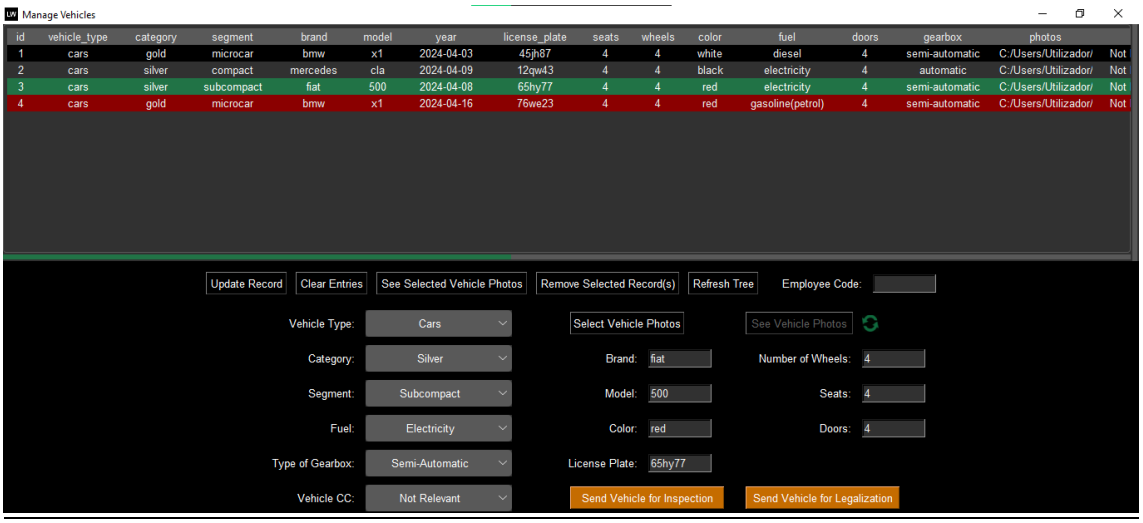
<u>When selecting the vehicle with ID 2, which does not requires inspection or legalization, the buttons remain inactive:</u>



When selecting the vehicle with ID 4, which requires inspection but not legalization, only the button to send for inspection becomes active:



When selecting the vehicle with ID 3, which requires both inspection and legalization, both buttons are activated:



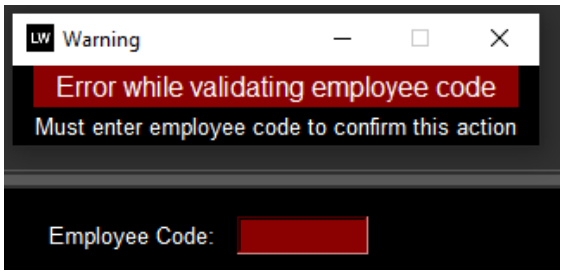
Now let's analyze what happens when you click on any of these buttons:

It will be used the vehicle with ID 4, which only requires inspection.

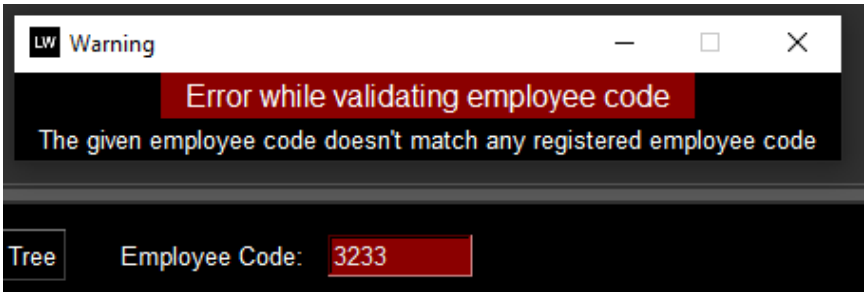
Verifications

Both actions require analyzing the employee code because only managers can perform these actions:

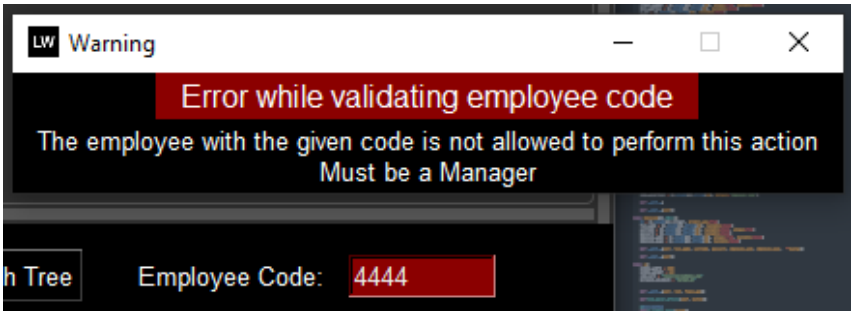
If an employee code is not entered:



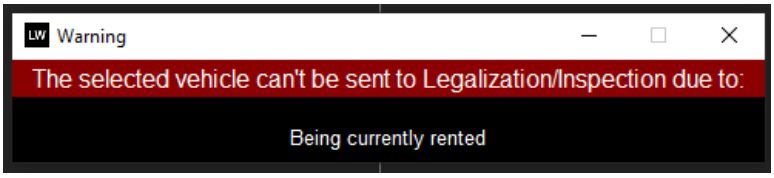
If the entered code does not match any employee registered in the database:



If the entered code belongs to an employee who is not a manager:



In addition to checking the entered employee/user code, it also verifies if the selected vehicle is currently rented. If it is, the action cannot be performed, and the user is notified:



Changes after validation

After successfully validating and executing the action, several changes occur. Let's observe each one:

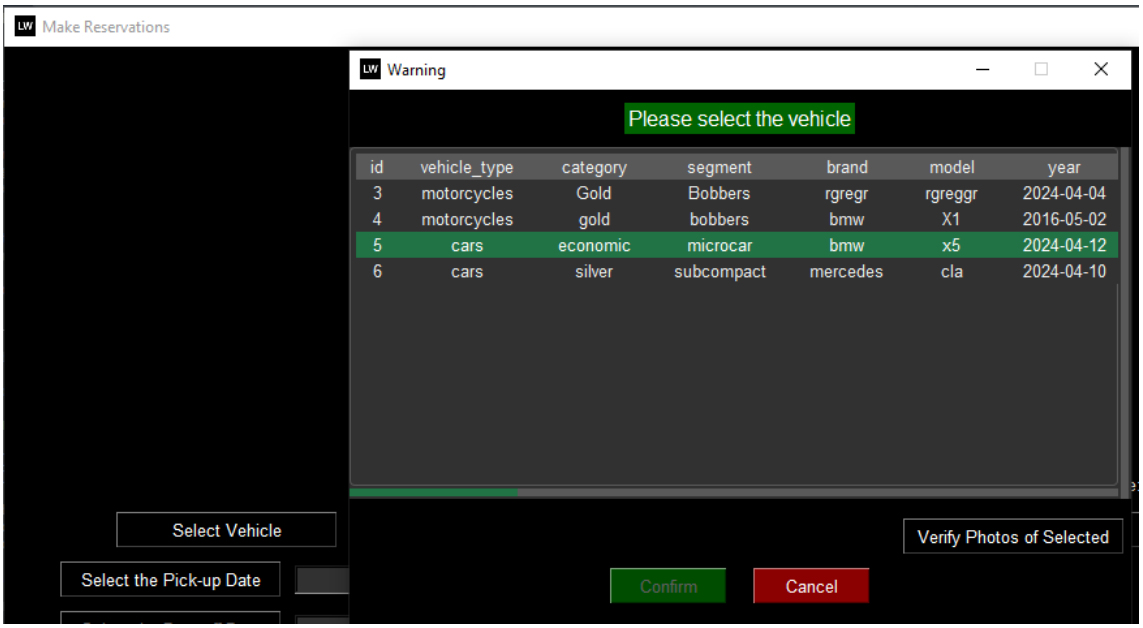
- The vehicle's availability is updated in the database, making it unavailable for any reservations;
- The information in the database columns related to this action is updated;
- The row of the vehicle that we sent for inspection changes from red to black because the vehicle is now unavailable.

Database table after sending the vehicle for inspection:

availability	rented	code_rented	for_inspection	code_inspection ▾1	for_legalization	code_legalization
Filter	Filter	Filter	Filter	Filter	Filter	Filter
Unavailable	No	Employee Code	Yes	2122	No	Employee Code

We can observe that the vehicle is unavailable, and the column "for_inspection" is the only one marked "Yes," confirming that the vehicle is unavailable because it was sent for inspection.

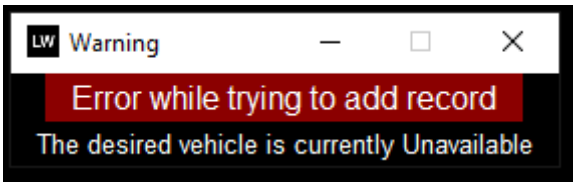
The availability column is extremely important, and the program relies on this information to perform various actions. For example, if you try to select this unavailable vehicle for a reservation using the vehicle selection button:



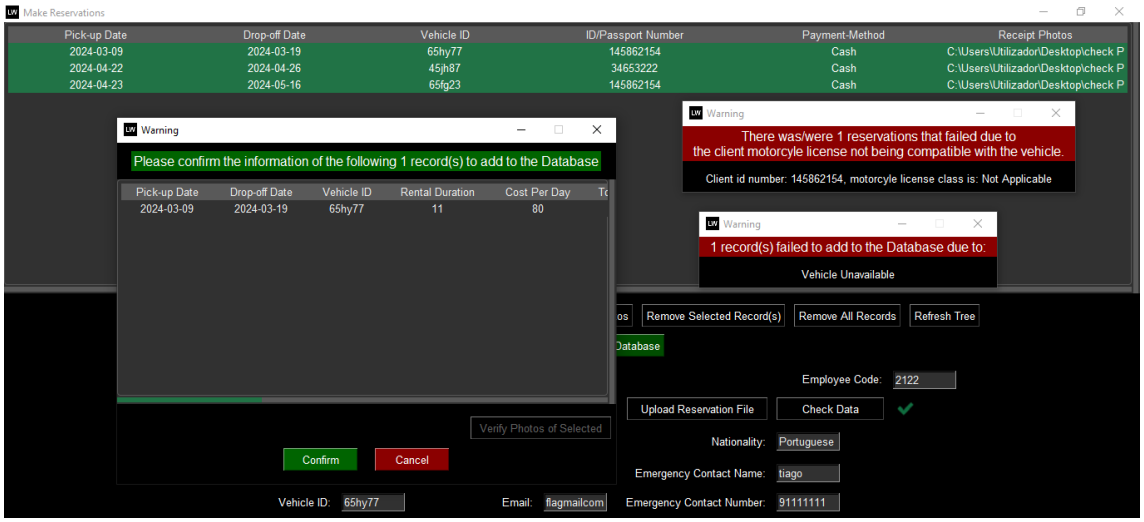
When we select a vehicle that is unavailable, the following happens:

- The program checks if the vehicle is available.
- If it's available, the "Confirm" button is activated.
- If it's unavailable, as is the case, the button remains inactive as a way to control and prevent the selection of an unavailable vehicle.

If we directly input the ID of the unavailable vehicle in the text entry, when trying to add the reservation to the database, we receive this warning:



If we are inputting reservations from an Excel/CSV document, when attempting to add the reservation to the database, we receive this warning:



In this case, coincidentally, we also received a warning that in one of the reservations the client does not have qualifications to drive motorcycles.

It's important to highlight that the vehicle that passed the verifications and awaits confirmation to be inserted into the database was validated despite being unavailable because the end date of the reservation is earlier than the date this reservation was being entered. Therefore, the program understands that we are only adding the record of a reservation that has already been completed previously.

Typically, when inserting/making a reservation, some information about the client and the vehicle would be updated to register that there is a lease in progress. In this case, as it is a reservation already concluded, these changes are not made.

If we have a vehicle that has expired or is about to expire its inspection and legalization, for example, sending it only for inspection will change the vehicle's status to unavailable. However, the vehicle's row will not turn black because, despite being sent for inspection, it also requires legalization. It will continue to display the color that applies to its situation, red if it has already expired or orange if it is within 15 days or less of expiring.

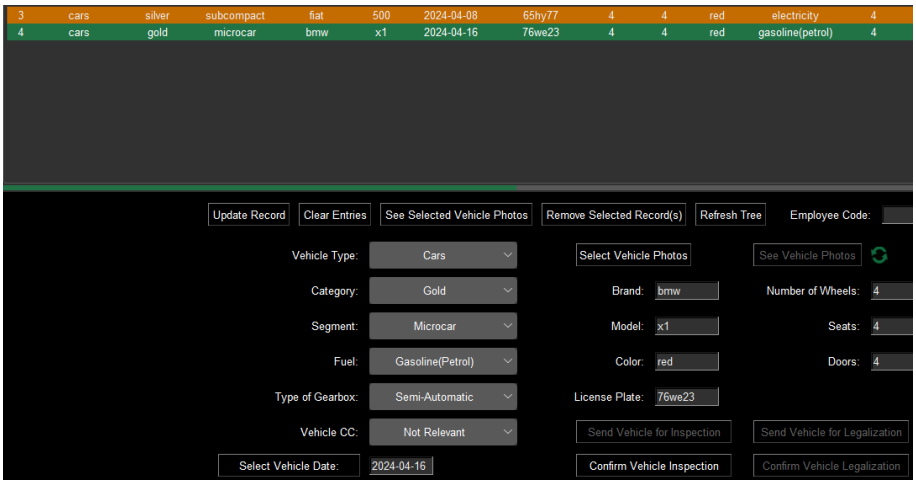
Buttons to confirm that the vehicle has undergone Inspection and/or Legalization

- Confirm Vehicle Inspection - button to confirm that the vehicle has undergone inspection;
- Confirm Vehicle Legalization - button to confirm that the vehicle has undergone legalization.

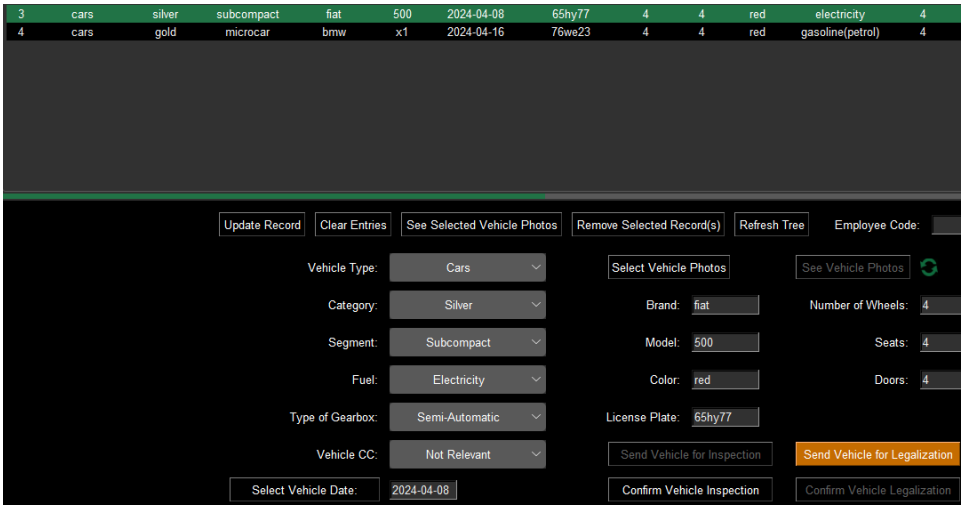
After the vehicle has been sent for inspection and/or legalization, the program requires confirmation from the user that the inspection and/or legalization has been completed.

When selecting a vehicle that was sent for inspection and/or legalization, the above buttons become active so that the user can confirm that vehicle has undergone inspection and/or legalization. The button to confirm inspection is activated if the vehicle was sent for inspection, the button to confirm legalization is activated if the vehicle was sent for legalization, or both if both actions were necessary.

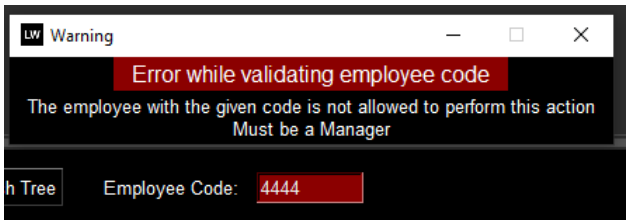
In the case of the selected vehicle below, it was only sent for inspection.



We can observe, in the image above, that the row of vehicle 3 continues to be orange because the vehicle was sent for inspection but also requires legalization, as we will see below when selecting vehicle 3.



Only managers can confirm that these actions have been performed.



Below, we will observe what happens to the color of the rows when confirming that vehicle 3 and vehicle 4 have undergone inspection.

3	cars	silver	subcompact	fiat	500	2024-04-08	65hy77	4	4	red	electricity	4	semi-automatic	C:/Users/Utilizador/	Not
4	cars	gold	microcar	bmw	x1	2024-04-16	76we23	4	4	red	gasoline(petrol)	4	semi-automatic	C:/Users/Utilizador/	Not

- The row of vehicle 4 changes its color to the normal color because the program confirmed that the vehicle is in conditions to become available again;
- The row of vehicle 3 remains orange because the vehicle still needs to be sent for legalization, despite having done the inspection.

Within the function for confirming these actions, before changing the vehicle's availability, the program analyzes if any of the following conditions apply:

- If the vehicle is leased, which the program would not allow because when making a reservation, the program checks if the deadline for legalization/inspection is before the reservation end date;
- If the vehicle has also been sent for another action (legalization/inspection), depending on the action we are confirming.
  - If we are confirming that the vehicle underwent legalization, the program checks if the vehicle was also sent for inspection.
- If the inspection/legalization date is before the current date, indicating that the vehicle has expired the inspection/legalization date.
  - If we are confirming that the vehicle underwent legalization, the program checks if the next inspection date has expired.

Below, we can observe the code responsible for this analysis:

```
vehicle = Vehicle.query.filter(Vehicle.license_plate.ilike(str(df.at[x, 'license_plate']).lower())).first()
next_inspection_date=self.calculate_date(str(df.at[x, 'next_inspection']), True)

vehicle.next_inspection = next_inspection_date
vehicle.for_inspection = "No"
vehicle.code_inspection = str(employee_code_entry.get())

legalization_date = datetime.strptime(vehicle.next_legalization, "%Y-%m-%d").date()
current_date = datetime.now().date()

if vehicle.rented == "Yes" or vehicle.for_legalization == "Yes" or legalization_date <= current_date:
    vehicle.availability = "Unavailable"
else:
    vehicle.availability = "Available"

db.session.commit()
```

If none of the above conditions are met, the vehicle becomes available.

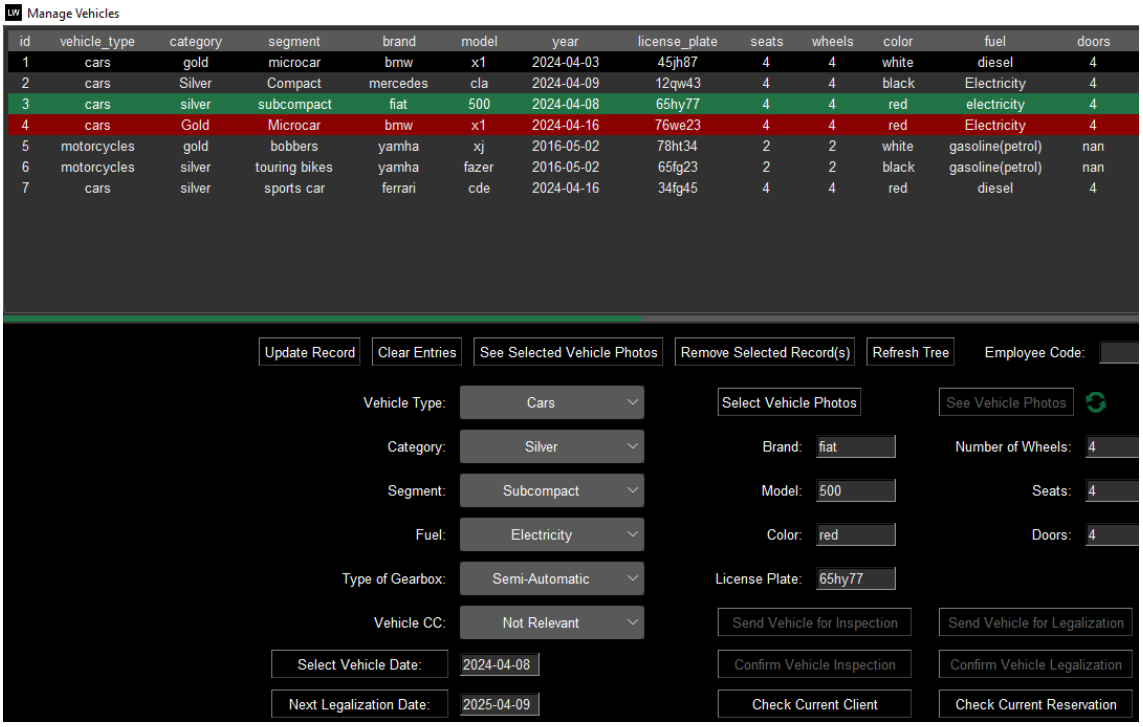
Whenever we perform any of the actions described so far, the table data is updated. With this update, the "verify_data" function is used, which is responsible for checking the availability of vehicles and the need for inspection/legalization. If this need is confirmed, it analyzes if the inspection/legalization date has expired or is about to expire.

Below, we see the piece of code responsible for this verification/alteration:

```
date_next_inspection = datetime.strptime(str(row['next_inspection']), "%Y-%m-%d")

current_date = datetime.now()
days_left_to_inspection = (date_next_inspection - current_date).days
days_left_to_inspection +=1

date_next_legalization = datetime.strptime(str(row['next_legalization']), "%Y-%m-%d")
days_left_to_legalization = (date_next_legalization - current_date).days

days_left_to_legalization += 1

if str(row['availability']) == "Unavailable":
    self.change_row_color(treeview, index, "#000000")
    if days_left_to_legalization <= 15 and str(row['for_legalization']) == "No":
        self.change_row_color(treeview, index, "#C56C00")
        if days_left_to_legalization <= 0:
            date_exceeded.append(str(row['license_plate']))
            self.change_row_color(treeview, index, "darkred")
    if days_left_to_inspection <= 15 and str(row['for_inspection']) == "No":
        self.change_row_color(treeview, index, "#C56C00")
        if days_left_to_inspection <= 0:
            date_exceeded.append(str(row['license_plate']))
            self.change_row_color(treeview, index, "darkred")
elif str(row['availability']) == "Available":
    self.change_row_color(treeview, index, "#313131")
    if days_left_to_legalization <= 15 or days_left_to_inspection <= 15:
        self.change_row_color(treeview, index, "#C56C00")
    if days_left_to_legalization <= 0 or days_left_to_inspection <= 0:
        date_exceeded.append(str(row['license_plate']))
        self.change_row_color(treeview, index, "darkred")
```

Buttons to view the client and the current reservation of the selected vehicle

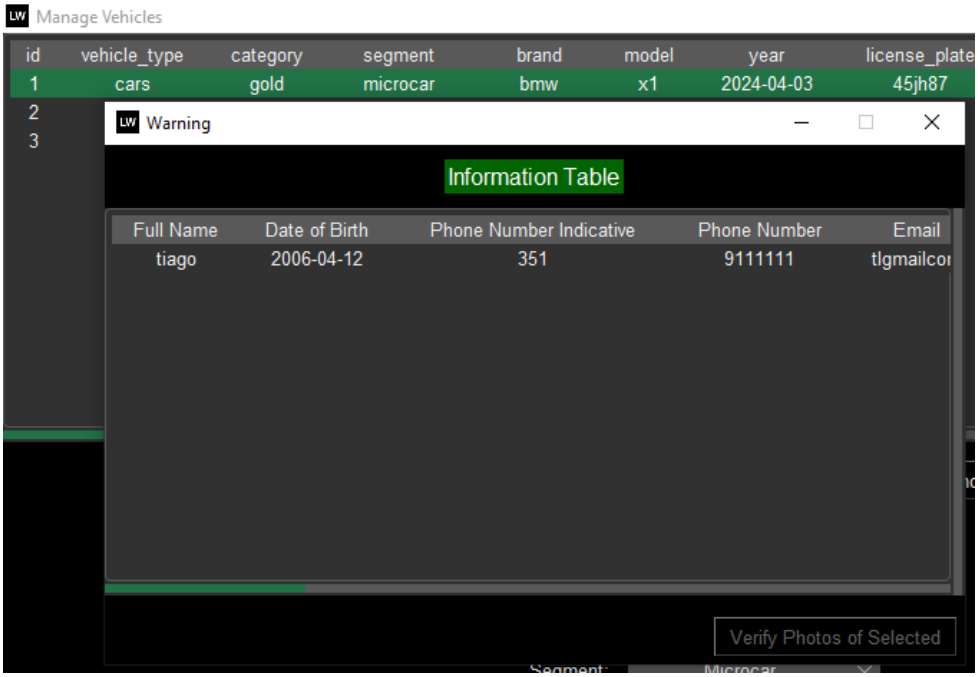
These buttons are only activated if the selected vehicle has a current reservation in progress.



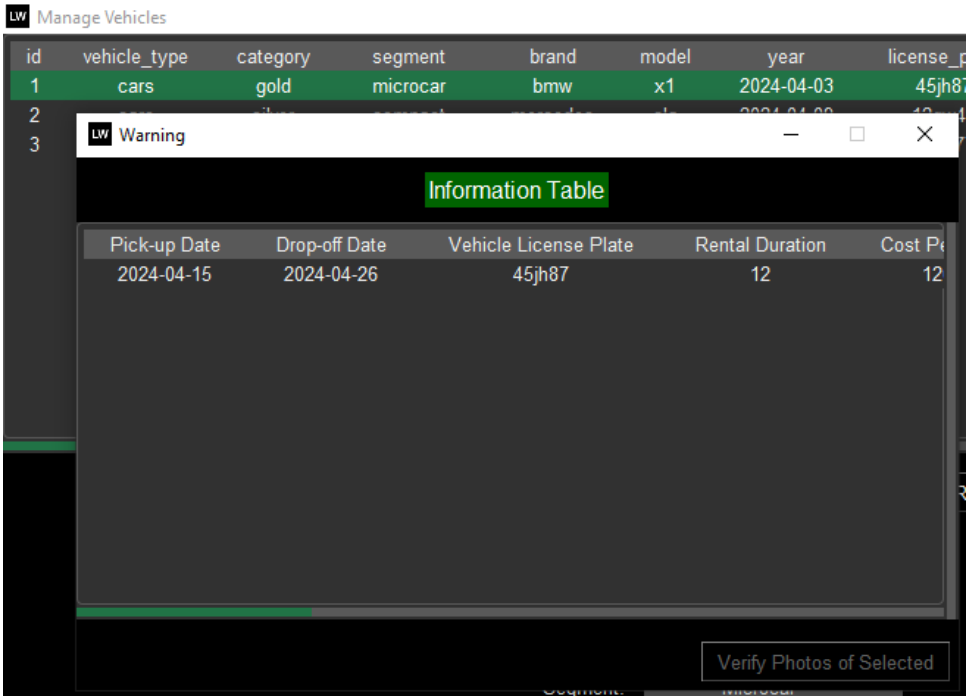
id	vehicle_type	category	segment	brand	model	year	license_plate	seats	wheels	color	fuel	doors
1	cars	gold	microcar	bmw	x1	2024-04-03	45jh87	4	4	white	diesel	4
2	cars	Silver	Compact	mercedes	cla	2024-04-09	12qw43	4	4	black	Electricity	4
3	cars	silver	subcompact	fiat	500	2024-04-08	65hy77	4	4	red	electricity	4
4	cars	Gold	Microcar	bmw	x1	2024-04-16	76we23	4	4	red	Electricity	4
5	motorcycles	gold	bobbers	yamha	xj	2016-05-02	78ht34	2	2	white	gasoline(petrol)	nan
6	motorcycles	silver	touring bikes	yamha	fazer	2016-05-02	65fg23	2	2	black	gasoline(petrol)	nan
7	cars	silver	sports car	ferrari	cde	2024-04-16	34fg45	4	4	red	diesel	4

When clicking on one of these buttons, the program utilizes the warning window to present the user with a table containing the information of the current client and the current reservation of the selected vehicle. This information is retrieved from the database and automatically collected by the program.

Current Client Information:

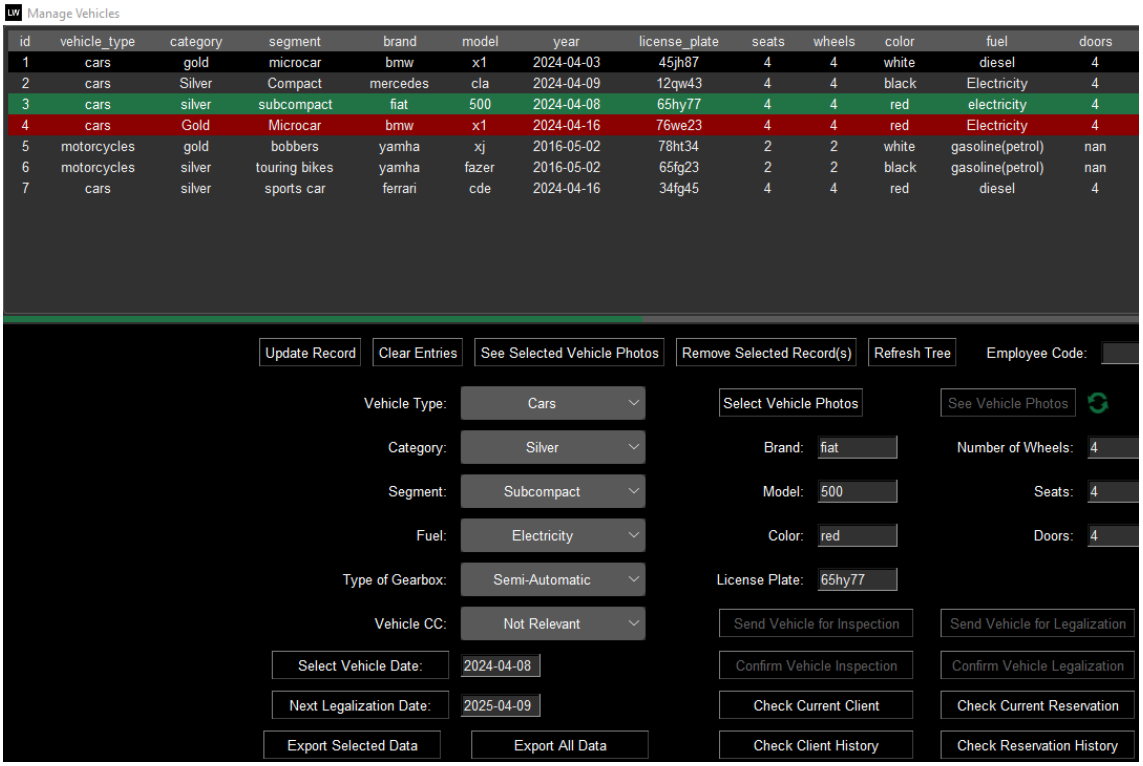


Current Reservation Information:



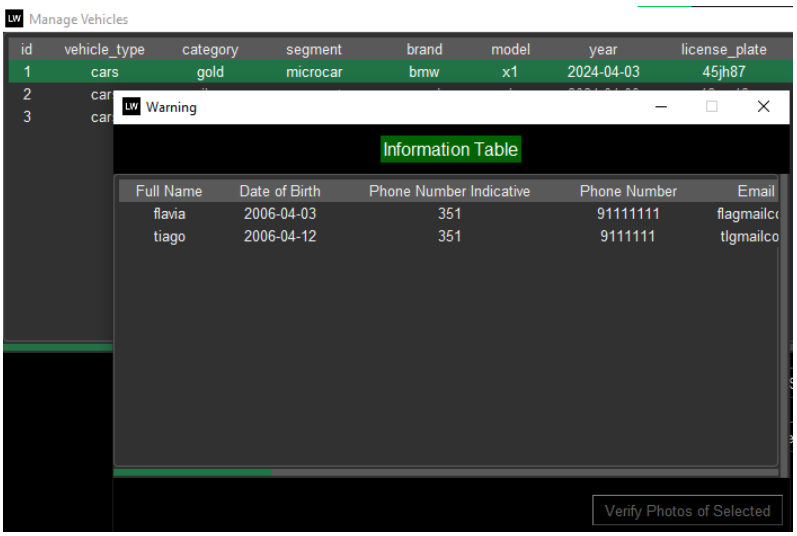
Buttons to view the history of clients and reservations for the selected vehicle

These buttons are only activated if the selected vehicle already has a history of reservations, meaning it has been reserved at least once before.

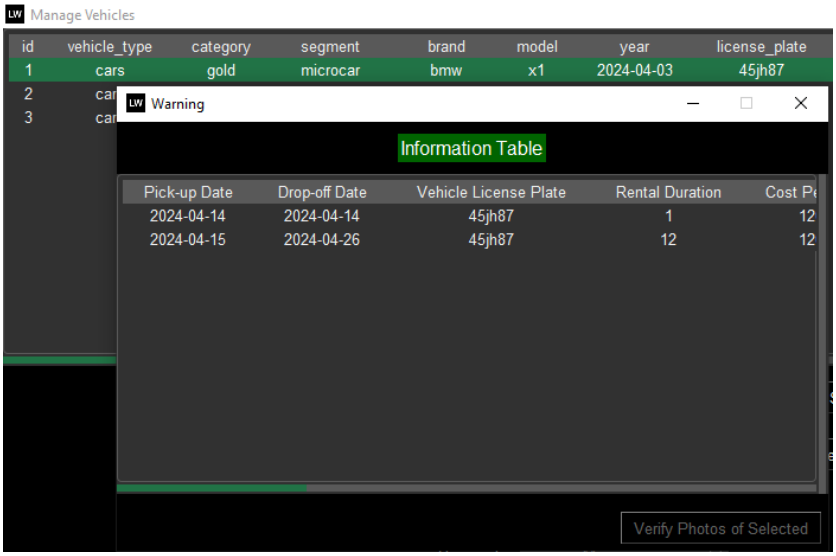


When clicking on one of these buttons, the program utilizes the alert window to present the user with a table containing the information of the client and reservation history of the selected vehicle. This information is retrieved from the database and automatically collected by the program.

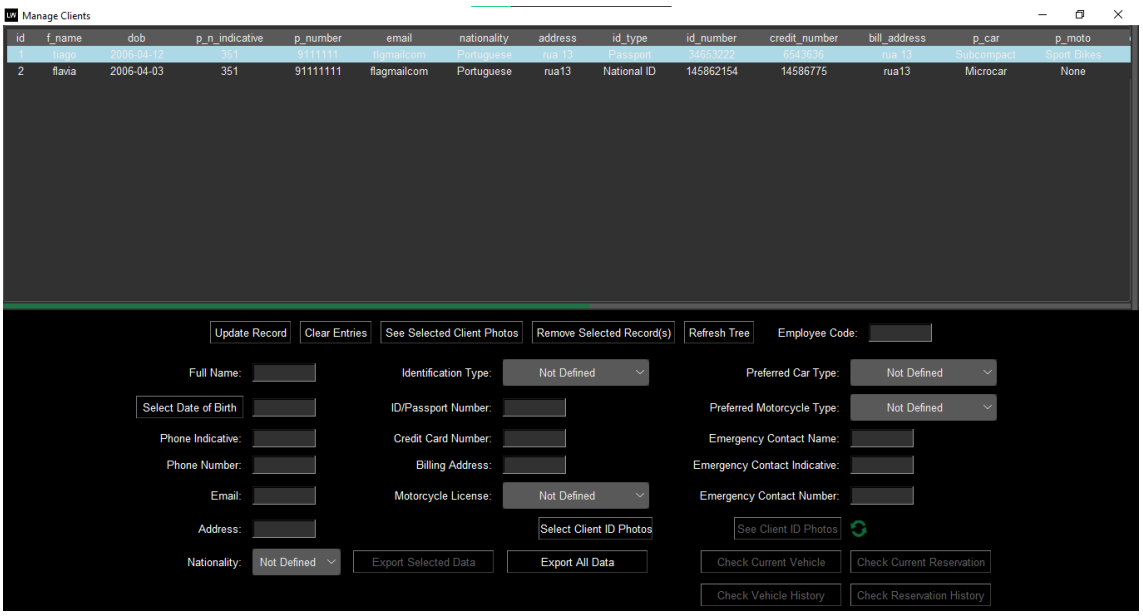
<u>Client History:</u>



Reservation History:

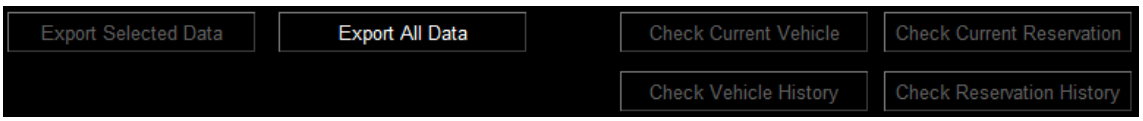


Manage Clients



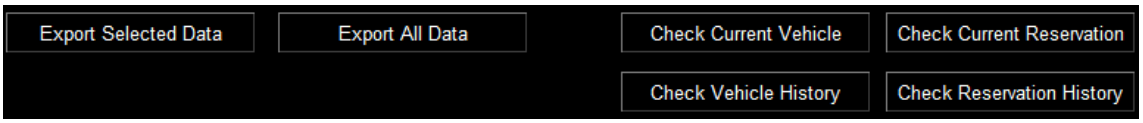
Clients who currently have a reservation in progress have their row highlighted in blue.

Similar to the previous section for managing vehicles, we have buttons for exporting information in CSV/Excel format and buttons for viewing information stored in the database.

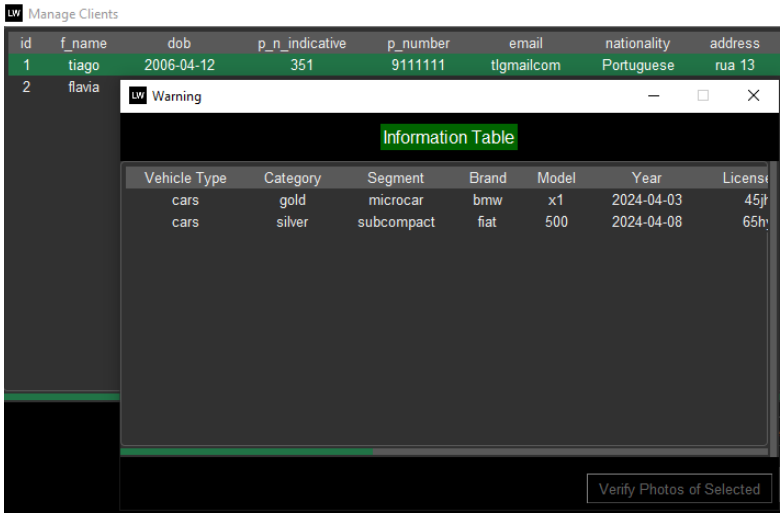


Buttons after selecting a client who has a history of reservations but does not have any reservations in progress:

Buttons after selecting a client who has a history of reservations and currently has reservations in progress:

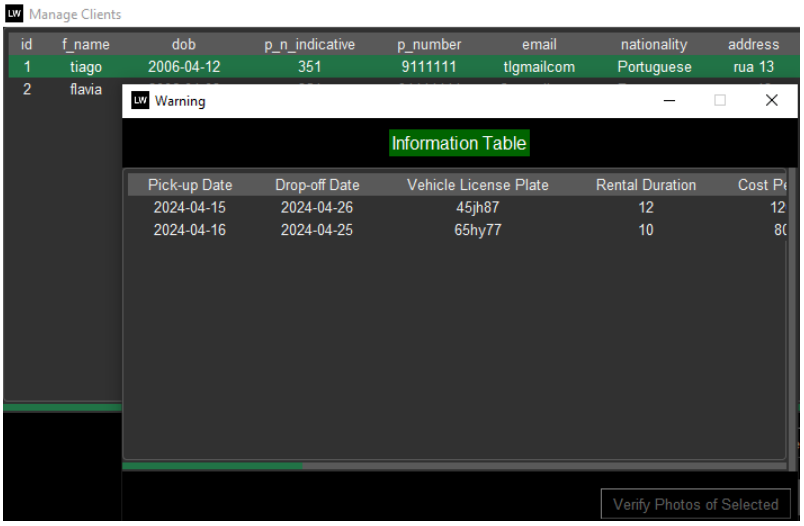


<u>Information about the current vehicle or vehicles of the client. A client may reserve more than one vehicle at the same time.</u>



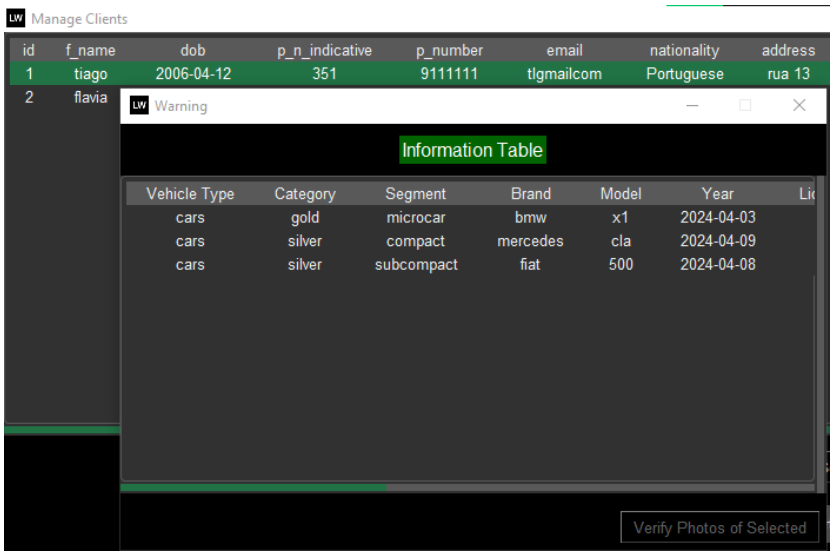
(In this case, the client has more than one reservation in progress.)

<u>Information about the current reservation or reservations. A client may have more than one reservation in progress simultaneously.</u>

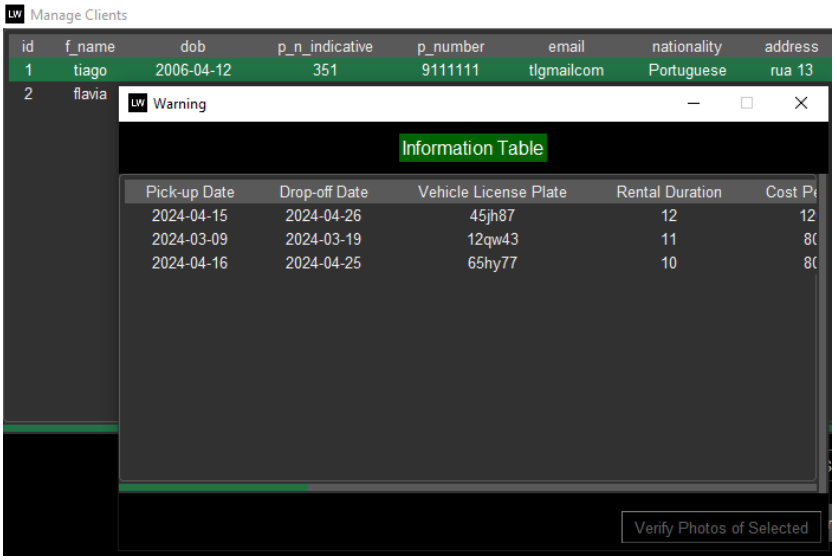


(In this case, the client has more than one reservation in progress.)

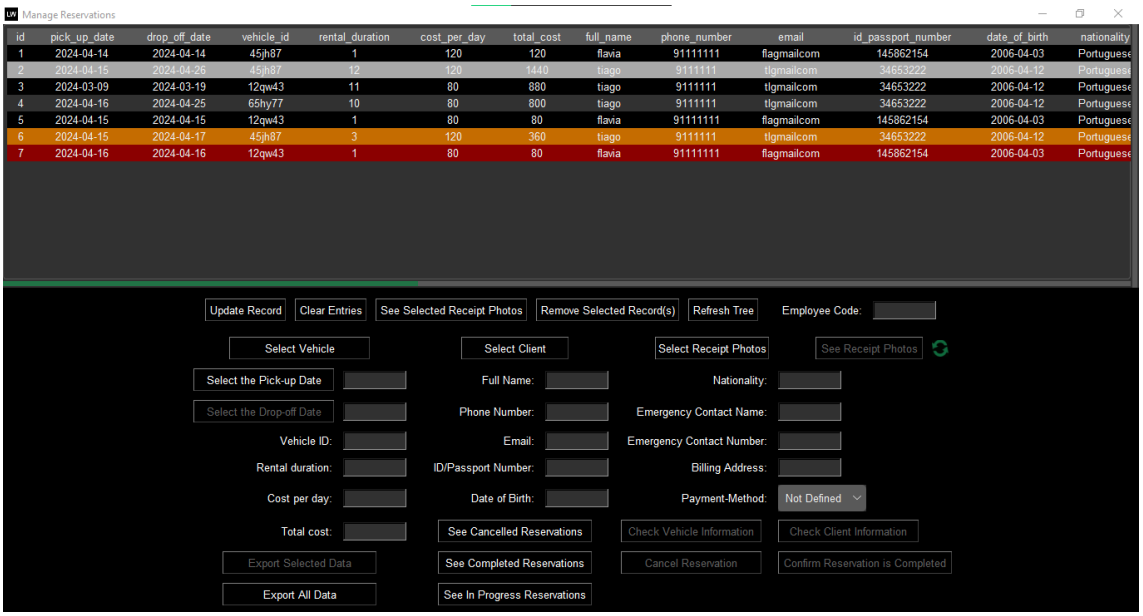
History of Vehicles rented by the Client:



History of Reservations made by the Client:



Manage Reservations



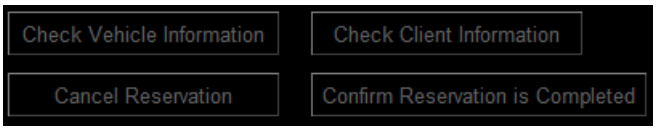
The different colors we observe in the table serve to alert the employee/user about the following possible situations:

- Normal color (id4) – the color that the table row has when the reservation is still ongoing;
- Black color (id1) – the color that the table row has when the reservation is complete;
- Light gray color (id2) – the color that the table row has when the reservation is canceled;
- Orange color (id6) – the color that the table row has when the reservation is at 3 or fewer days from ending;
- Red color (id7) – the color that the table row has when the reservation is at 0 days from ending or has already ended.

Similar to the previous sections, we have buttons to export information in CSV/Excel and buttons to view information stored in the Database.

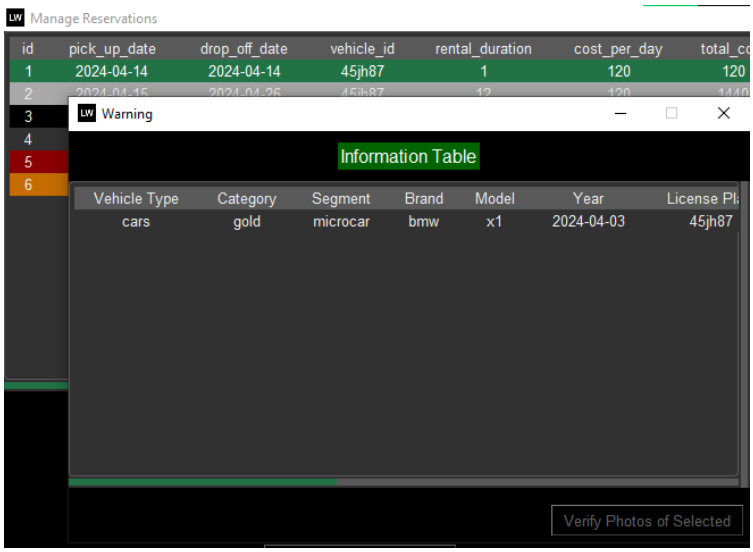
Now let's analyze what happens with certain buttons when selecting different reservations above, with the different possible situations.

Buttons state without any selection

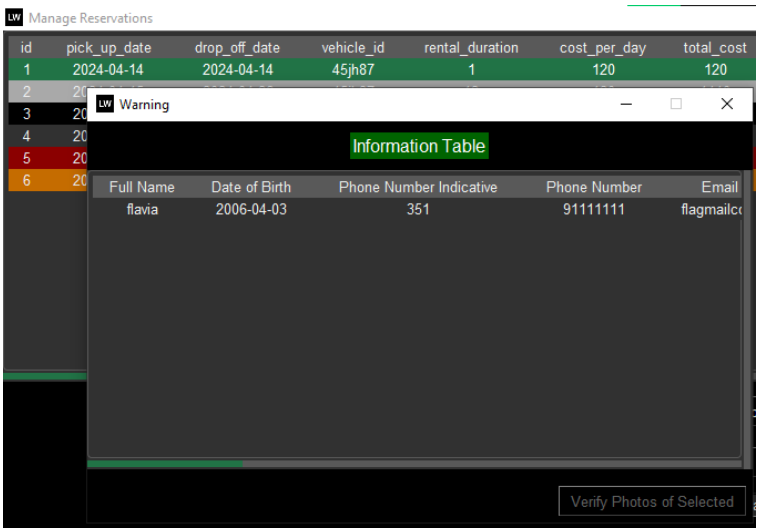


When selecting a completed or canceled reservation, only the buttons to view the information of the client ("Check Client Information") and vehicle ("Check Vehicle Information") inserted in that reservation are enabled.

Information of the vehicle inserted in the selected reservation:

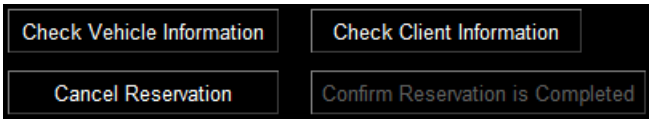


Information of the client inserted in the selected reservation:

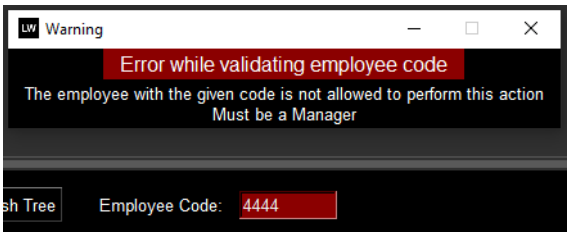


When selecting a reservation that is still ongoing and where the time until the reservation ends is greater than 0 days, the following buttons become active:

- Button to view client information ("Check Client Information").
- Button to view vehicle information ("Check Vehicle Information").
- Button that allows canceling the reservation ("Cancel Reservation").



This action requires a manager-level employee's authorization code to be validated.



Upon confirming this action, we update some information in the tables of the client, vehicle, and reservation in the database.

<u>Changes made to the client information in the table:</u>

renting	code_renting
Filter	Filter
Yes	2122

Regarding the client, the information to be updated is the one responsible for indicating if the client currently has a renting in progress, the column 'renting'. However, before updating this information, the program needs to check if it is necessary to proceed with this change. The program checks if the client has any other ongoing reservation.

For this verification, the program searches the Database if the client's identification number is in any reservation where the reservation status is 'In progress', indicating that the client has more reservations ongoing besides the one that was canceled/completed.

Below is the piece of code responsible for this verification:

```
client_reservations = Reservation.query.filter_by(id_passport_number=client.id_number, reservation_state="In progress").all()
if len(client_reservations) > 0:
    client.renting = "Yes"
else:
    client.renting = "No"
    client.code_renting = str(employee_code_entry.get())
```

- The value of the "renting" column is changed to "No" if the client no longer has ongoing reservations. In this case, the "code_renting" is updated to the code of the employee/user who made this change;
- If the client has ongoing reservations, the value of the "renting" column remains "Yes".

<u>Changes made to the vehicle information in the table:</u>

availability	rented	code_rented
Filter	Filter	Filter
Unavailable	Yes	2122

Regarding the vehicle, the information that will be updated is the information related to the vehicle's availability. Before making this change, the program checks if the vehicle needs to remain unavailable.

It checks if the vehicle has expired its legalization date and/or inspection.

Below is the piece of code responsible for this verification:

```
inspection_date = datetime.strptime(vehicle.next_inspection, "%Y-%m-%d").date()
legalization_date = datetime.strptime(vehicle.next_legalization, "%Y-%m-%d").date()
current_date = datetime.now().date()

vehicle.rented = "No"
vehicle.code_rented = str(employee_code_entry.get())

if vehicle.for_inspection == "Yes" or vehicle.for_legalization == "Yes" or legalization_date <= current_date or inspection_date <= current_date:
    vehicle.availability = "Unavailable"
else:
    vehicle.availability = "Available"
```

- The value of the "availability" column is changed to "Available" if none of the exceptions above are confirmed;
- The value of the "rented" column is changed to "No" and the "code_rented" is updated to the code of the employee/user who made this change.

<u>Changes made to the reservation information in the table:</u>

reservation_state	date_confirm_completed_renting	code_confirm_completed_renting	date_cancel_reservation	code_cancel_reservation
Filter	Filter	Filter	Filter	Filter
Cancelled	Not Applicable	Employee Code	2024-04-15	2122

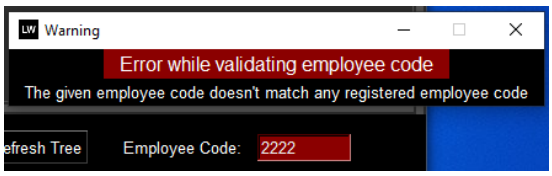
When canceling a reservation, we update the columns referring to the reservation status as follows:

- The value of the column "reservation_state" will change from "In progress" to "Cancelled";
- The value of the column "date_cancel_reservation" will be replaced by the date when this action was confirmed;
- The value of the column "code_cancel_reservation" will be replaced by the code of the employee/user who confirmed the cancellation of the reservation.

Finally, when selecting a reservation where the reservation time has ended or the current day is the last day of the reservation, the following buttons become active:

- Button to view client information ("Check Client Information");
- Button to view vehicle information ("Check Vehicle Information");
- Button to cancel the reservation ("Cancel Reservation");
- Button to confirm that the reservation is completed ("Confirm Reservation is Completed").

This action requires an employee code, but it does not require manager level validation.



When confirming that a reservation is complete, we update the same information as when canceling a reservation, with the exception of the reservation details. In this case, since we are confirming that the reservation is complete, the columns related to cancellation will not be filled.

Changes made to the reservation information in the table:

reservation_state	date_confirm_completed_renting	code_confirm_completed_renting ▾1	date_cancel_reservation	code_cancel_reservation
Filter	Filter	Filter	Filter	Filter
Completed	2024-04-14	2122	Not Applicable	Employee Code

When confirming that a reservation is complete, we update the columns referring to the reservation status as follows:

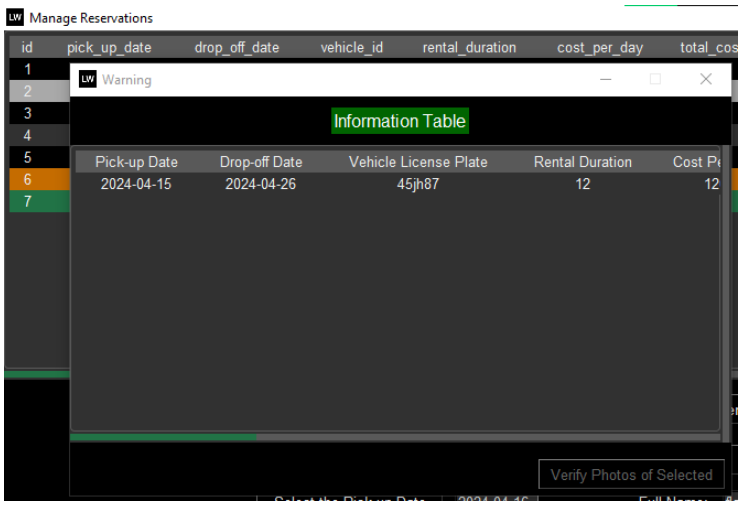
- The value of the column "reservation_state" will change from "In progress" to "Completed";
- The value of the column "date_confirm_completed_renting" will be replaced by the date when this action was confirmed;
- The value of the column "code_confirm_completed_renting" will be replaced by the code of the employee/user who confirmed that the rental was completed.

Besides the buttons presented so far, within the section to manage reservations, we also have three more buttons:

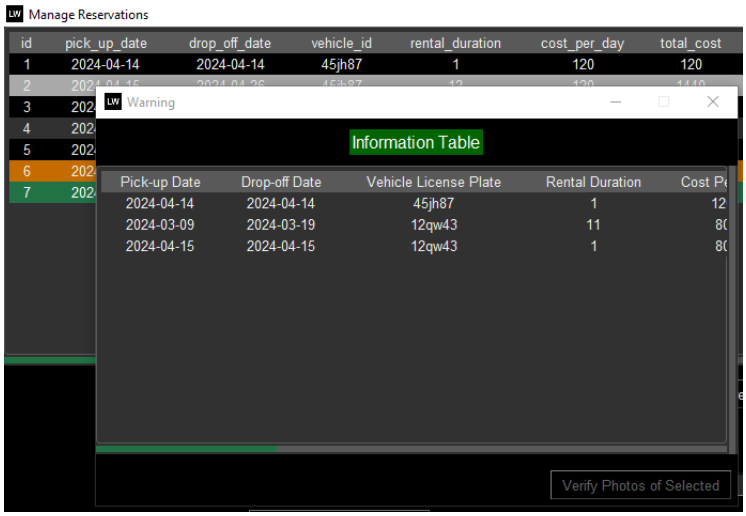
These buttons allow us to view only the reservations that are in the three possible states:

- See Cancelled Reservations – view all cancelled reservations;
- See Completed Reservations – view all completed reservations;
- See In Progress Reservations – view all reservations still in progress.

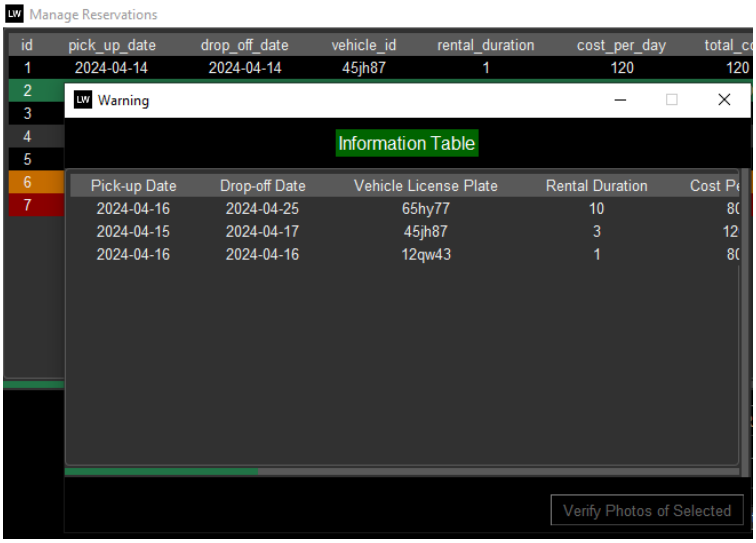
All Cancelled Reservations:



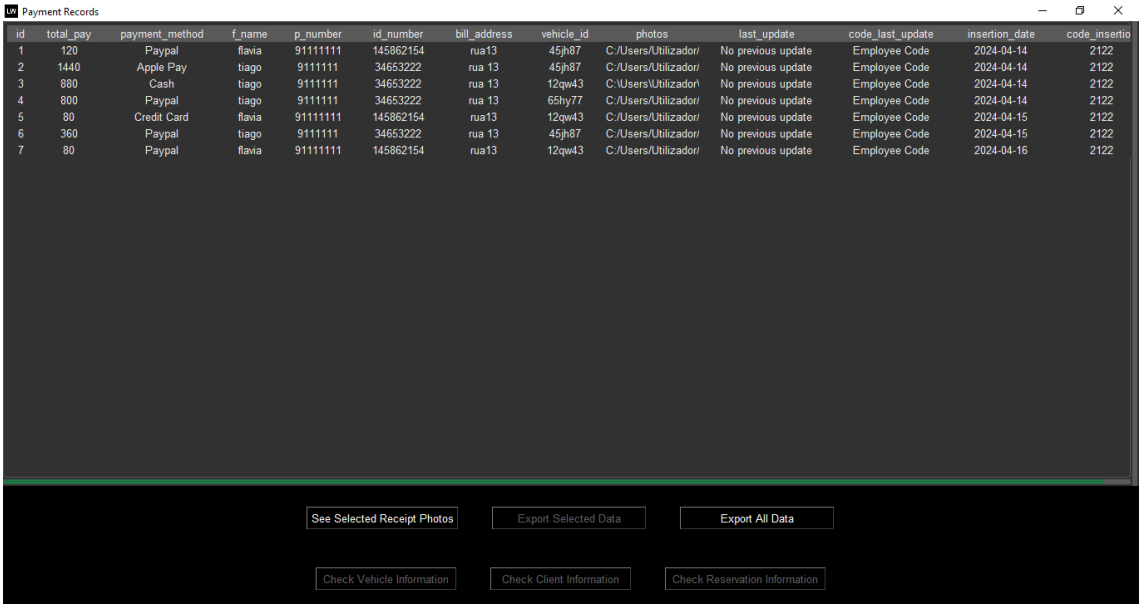
All Completed Reservations:



All Reservations Still In Progress:



In addition to the Sections for Inserting/Managing Vehicles/Clients/Reservations, we have a section for viewing payments.



id	total_pay	payment_method	f_name	p_number	id_number	bill_address	vehicle_id	photos	last_update	code_last_update	insertion_date	code_insertio
1	120	Paypal	flavia	91111111	145862154	rua13	45jh87	C:/Users/Utilizador/	No previous update	Employee Code	2024-04-14	2122
2	1440	Apple Pay	tiago	9111111	34653222	rua 13	45jh87	C:/Users/Utilizador/	No previous update	Employee Code	2024-04-14	2122
3	880	Cash	tiago	9111111	34653222	rua 13	12qw43	C:\Users\Utilizador\	No previous update	Employee Code	2024-04-14	2122
4	800	Paypal	tiago	9111111	34653222	rua 13	65hy77	C:/Users/Utilizador/	No previous update	Employee Code	2024-04-14	2122
5	80	Credit Card	flavia	91111111	145862154	rua13	12qw43	C:/Users/Utilizador/	No previous update	Employee Code	2024-04-15	2122
6	360	Paypal	tiago	9111111	34653222	rua 13	45jh87	C:/Users/Utilizador/	No previous update	Employee Code	2024-04-15	2122
7	80	Paypal	flavia	91111111	145862154	rua13	12qw43	C:/Users/Utilizador/	No previous update	Employee Code	2024-04-16	2122

We have the following buttons:

- See Selected Receipt Photos - button to view the photos of the selected item;
- Export Selected Data/Export All Data - buttons to export information in CSV/Excel;
- Check Vehicle Information - button to view the information of the vehicle inserted in the reservation related to this payment;
- Check Client Information - button to view the information of the client inserted in the reservation related to this payment;
- Check Reservation Information - button to view the information of the reservation related to this payment.

In this section, it is not possible to update/change the information, to change any information, we must use the reservation management section. Any changes made to any reservation will also be applied to the related payment record. The same applies if a reservation is deleted, the payment record for that reservation will also be deleted

Below is a part of the code responsible for deleting a reservation from the database. We use the reservation ID we intend to delete to obtain and also delete the payment record related to that reservation.

```
x = treeview.index(record)
reservation = Reservation.query.filter_by(id=int(df.at[x, 'id'])).first()
if reservation.reservation_state == "In progress":
    can_not_delete.append(reservation.id)
else:
    pay_record = Payment.query.filter_by(id=reservation.id).first()
    db.session.delete(reservation)
    db.session.delete(pay_record)
    db.session.commit()
```

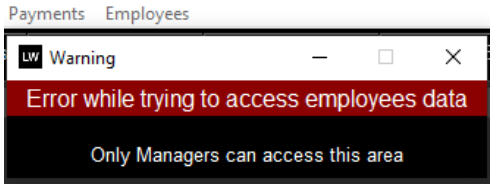
Finally, we have the section that allows viewing the information of the remaining employees/users. Access to this section is only permitted for users who are 'Managers'.

When the user logs in, the program uses the entered username to retrieve the user's data. From this data, it extracts the type of employee. If the employee type is 'Manager', it defines that clicking the button should open the window. Otherwise, clicking the button will prompt a notification stating that only 'Managers' can access that section.

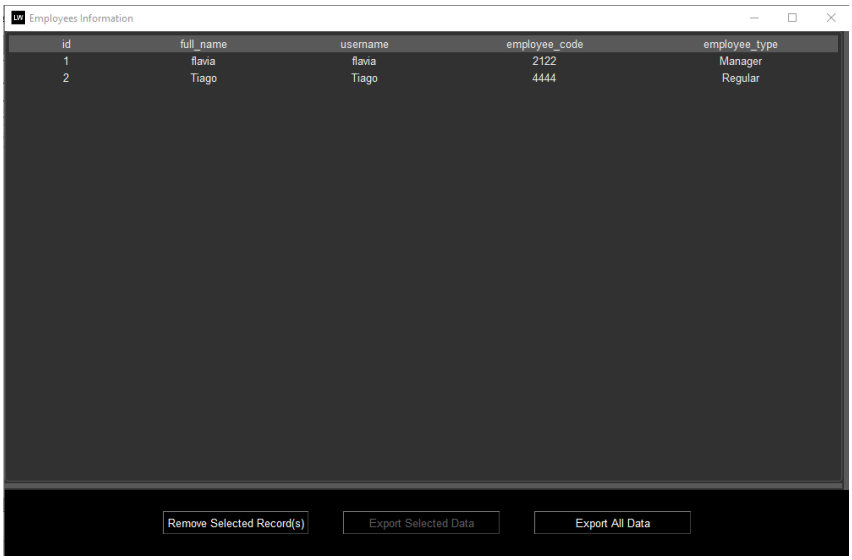
Below is the code responsible for this verification:

```
employee = Employee.query.filter_by(username=authenticated_username).first()

employee_menu = tk.Menu(menu_bar, tearoff=0)
employee_menu.add_command(label="Check Employees Information", command=lambda: self.create_section_window("Employees Information")
                          if employee.employee_type == "Manager" else self.pop_warning(self.root, "Only managers can access this area", "employeesdata"))
menu_bar.add_cascade(label="Employees", menu=employee_menu)
```

If the user is not a manager:



If the user is a manager:



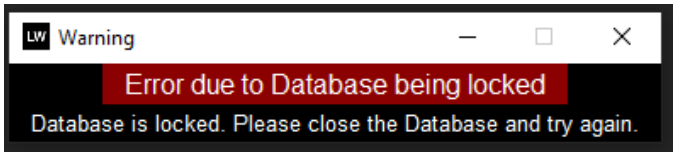
Within this section, only the following actions are permitted:

- Exporting information in CSV/Excel;
- Removing users/employees.

No employee code is required to complete any of the following actions because if the section has opened, it means the program has already established that the user is a 'Manager'.

In case we have our database open and we are changing information directly in the tables, for example in the 'DB Browser' program, if we try to insert a new element into a table or modify/update any data, we would encounter an error because the database would be locked.

To control the situation described above, the program receives this error and informs the user:



An example of this control in one of the possible situations, in this case, when trying to remove an element from the table:

```
def remove_selected():
    selected_items = treeview.selection()
    if len(selected_items) > 0:
        try:
            if len(selected_items) == Vehicle.query.count():
                warning = "Can not delete all the data from the Vehicle Database"
                self.pop_warning(new_window, warning, "cannotdeletealldb" )
            else:
                code_check=self.check_employee_code(str(employee_code_entry.get()), True)
                if code_check == "valid":
                    self.toggle_entry_colors(1, employee_code_entry)
                    can_not_delete = []
                    for record in selected_items:
                        x = treeview.index(record)
                        vehicle = Vehicle.query.filter(Vehicle.license_plate.ilike(str(df.at[x, 'license_plate']).lower())).first()
                        if vehicle.rented == "Yes":
                            can_not_delete.append(vehicle.license_plate)
                        else:
                            db.session.delete(vehicle)
                            db.session.commit()
                            treeview.delete(record)
                            df.drop(index=x, inplace=True)
                            df.reset_index(drop=True, inplace=True)
                    verify_data()
                    if len(can_not_delete) > 0:
                        self.pop_warning(new_window, can_not_delete, "cannotdelete")
                else:
                    self.toggle_entry_colors(0, employee_code_entry)
                    self.pop_warning(new_window, code_check, "wrongemployeecode")
        except OperationalError as e:
            warning = "Database is locked. Please close the Database and try again."
            self.pop_warning(new_window, warning, "databaselocked")
            db.session.rollback()
            print("Database is locked. Please try again later.")
    else:
        warning = "Must select at least one record to remove"
        self.pop_warning(new_window, warning, "noselectedtoremove")
```

With this, we conclude the analysis of the app.