07-06-2024

Gestão de Oficina

Disciplina de Programação Avançada

Pedro Espírito Santo nº 82264
Rodrigo Lucindo nº 82265
João Bagueixo Ferreira nº 82268
João Pinheiro Ferreira nº 82269

EXMO. PROFESSOR FERNANDO BARROS

Índice

Resumo

Este projeto consiste no desenvolvimento de um sistema de gestão para uma oficina, com funcionalidades como: a gestão de utilizadores com reconhecimento facial, gestão de clientes, gestão de manutenções e colisões com suporte a calendário, e gestão de pagamentos. O sistema foi criado com o objetivo de automatizar e facilitar a administração das atividades diárias da oficina, proporcionando uma interface intuitiva e funcionalidades avançadas para otimizar processos, melhorar a eficiência operacional e consequentemente aumentar a satisfação dos clientes. Utilizando tecnologias modernas como o reconhecimento facial para a autenticação de utilizadores, o sistema assegura segurança e precisão. A integração de um calendário facilita a organização das manutenções e colisões, garantindo que os serviços sejam realizados de forma oportuna. Além disso, a gestão centralizada de clientes e pagamentos contribui para um controlo financeiro preciso e um melhor relacionamento com os clientes.

Introdução

A administração eficiente de uma oficina envolve diversas tarefas complexas e desafiadoras, desde a gestão de clientes e serviços até ao controlo de pagamentos e manutenção de registos. Com o objetivo de simplificar e otimizar estes processos, desenvolvemos um sistema de gestão abrangente que integra funcionalidades essenciais para a operação diária de uma oficina.

Este sistema foi projetado para fornecer uma interface intuitiva e funcionalidades robustas, utilizando tecnologias modernas como o reconhecimento facial para a autenticação de utilizadores, garantindo assim segurança e precisão. A integração de um calendário para a gestão de manutenções e colisões permite uma organização clara e eficiente, assegurando que todos os serviços sejam realizados de forma oportuna e dentro dos prazos estipulados.

A centralização da gestão de clientes e pagamentos facilita o controlo financeiro e melhora o relacionamento com os clientes, proporcionando uma visão abrangente das atividades da oficina. Além disso, a utilização de uma estrutura de diretórios bem organizada garante a manutenção e expansão futura do sistema, permitindo adaptações e melhorias contínuas conforme as necessidades evoluem.

Neste relatório, detalhamos cada componente do sistema, abordando a sua estrutura, funcionalidades específicas e a lógica de implementação. Discutimos também os resultados obtidos e as possíveis melhorias futuras para expandir e aprimorar ainda mais o sistema, garantindo uma solução eficiente e fiável para a gestão de oficinas.

Descrição do problema

O desafio apresentado é a gestão eficiente de uma oficina, que implica a coordenação de várias atividades simultâneas e a manutenção de um elevado nível de organização para assegurar a eficiência e a satisfação dos clientes. No entanto, muitas oficinas enfrentam desafios significativos ao tentarem manter os seus processos operacionais bem organizados e sob controlo.

Os principais problemas incluem a gestão de utilizadores e segurança, a dificuldade na gestão de clientes devido a registos imprecisos e desatualizados, a complexidade na gestão de manutenções e colisões, e a falta de controlo financeiro devido a sistemas de pagamento desatualizados. Para resolver esses problemas, propomos o desenvolvimento de um sistema abrangente de gestão que integra o reconhecimento facial para autenticação de utilizadores, um calendário para a gestão de manutenções e colisões, e funcionalidades centralizadas para a gestão de clientes e pagamentos.

Este sistema visa automatizar e otimizar as operações diárias da oficina, melhorando a eficiência, segurança e satisfação dos clientes.

Arquitetura do Sistema

A arquitetura do sistema é dividida em três camadas principais:

- Camada de Apresentação:

  Interface gráfica construída com Tkinter para a interação do utilizador.

- Camada de Lógica de Negócio:

  Implementação das funcionalidades principais do sistema.

- Camada de Dados:

  Base de dados MySQL para armazenamento persistente de dados.

Diagrama da base de dados

Clients	**Services**	**Payments**	**Users**
client_id (PK) client_name client_address client_nif client_mobile client_email client_created	service_id (PK) service_client_id (FK) service_type service_description service_start_date service_end_date service_state	payment_id (PK) payment_service_id (FK) payment_date payment_value payment_state payment_type	user_id (PK) user_name user_fullname user_password user_role

Diretórios

```
.
├── api_tests
│   ├── api_test_get_clients.py
│   ├── api_test_get_payments.py
│   └── api_test_get_services.py
├── app.py
├── assets
│   └── logo.png
├── clients.py
├── images
├── LICENSE
├── main.py
├── payments.py
├── __pycache__
│   ├── app.cpython-312.pyc
│   ├── clients.cpython-312.pyc
│   ├── main.cpython-312.pyc
│   ├── payments.cpython-312.pyc
│   ├── quit_login.cpython-312.pyc
│   ├── services.cpython-312.pyc
│   ├── user_admin.cpython-312.pyc
│   └── users.cpython-312.pyc
├── README.md
├── requirements.txt
├── services.py
├── src
│   ├── api
│   │   └── server.py
│   ├── database
│   │   ├── clients.py
│   │   ├── database.py
│   │   ├── payments.py
│   │   ├── __pycache__
│   │   │   ├── clients.cpython-312.pyc
│   │   │   ├── database.cpython-312.pyc
│   │   │   ├── payments.cpython-312.pyc
│   │   │   ├── services.cpython-312.pyc
│   │   │   └── users.cpython-312.pyc
│   │   ├── services.py
│   │   └── users.py
│   ├── faceRec.py
│   ├── __pycache__
│   │   ├── faceRec.cpython-312.pyc
│   │   ├── takePic.cpython-312.pyc
│   │   └── utils.cpython-312.pyc
│   └── takePic.py
└── users.py
```

Implementação do *Back-End*

Face Recognition

Este código realiza autenticação facial. Ele carrega imagens de uma pasta, extrai codificações faciais, e usa a webcam para capturar e comparar faces em tempo real. Se uma face for reconhecida, abre a janela principal correspondente ao utilizador.

```
import cv2
import numpy as np
import face_recognition
import os
from main import main_window
from src.database.users import verify_login, login_fullname

# Define path to images
path = 'images'
images = []
classNames = []

# Ensure the path exists
if not os.path.exists(path):
    raise FileNotFoundError(f"The directory '{path}' does not exist.")

# List images in the directory
myList = os.listdir(path)


# Load images and class names
for cl in myList:
    curImg = cv2.imread(os.path.join(path, cl))
    if curImg is None:
        print(f"Warning: Could not load image {cl}. Skipping.")
        continue
    images.append(curImg)
    classNames.append(os.path.splitext(cl)[0])

# Function to find encodings for all images
def findEncodings(images):
    encodeList = []
    for img in images:
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        encodings = face_recognition.face_encodings(img)
        if encodings:
            encodeList.append(encodings[0])
        else:
            print("Warning: No faces found in an image. Skipping.")
    return encodeList

def faceRec():
    # Get known encodings
    encodeListKnown = findEncodings(images)
    print('Encoding Complete')

    # Start webcam
```

```
    cap = cv2.VideoCapture(0)

    if not cap.isOpened():
        raise RuntimeError("Error: Could not open webcam.")

    try:
        while True:
            success, img = cap.read()
            if not success:
                print("Warning: Failed to capture image. Retrying...")
                continue

            imgS = cv2.resize(img, (0, 0), None, 0.25, 0.25)
            imgS = cv2.cvtColor(imgS, cv2.COLOR_BGR2RGB)

            facesCurFrame = face_recognition.face_locations(imgS)
            encodesCurFrame = face_recognition.face_encodings(imgS,
facesCurFrame)

            for encodeFace, faceLoc in zip(encodesCurFrame,
facesCurFrame):
                matches = face_recognition.compare_faces(encodeListKnown,
encodeFace)
                faceDis = face_recognition.face_distance(encodeListKnown,
encodeFace)
                matchIndex = np.argmin(faceDis)

                if matches[matchIndex]:
                    name = classNames[matchIndex].upper()
                    y1, x2, y2, x1 = faceLoc
                    y1, x2, y2, x1 = y1 * 4, x2 * 4, y2 * 4, x1 * 4
                    cv2.rectangle(img, (x1, y1), (x2, y2), (0, 255, 0),
2)
                    cv2.rectangle(img, (x1, y2 - 35), (x2, y2), (0, 255,
0), cv2.FILLED)
                    cv2.putText(img, name, (x1 + 6, y2 - 6),
cv2.FONT_HERSHEY_COMPLEX, 1, (255, 255, 255), 2)

                    # Show the name on the image for a brief period
before calling main_window
                    cv2.imshow('Webcam', img)
                    cv2.waitKey(2500)  # Wait for 2.5 seconds
                    cap.release()
                    cv2.destroyAllWindows()
                    if verify_login(name) == "Operador":
                        main_window("Operador", login_fullname(name))
                    else:
                        main_window("Contabilidade",
login_fullname(name))
                    return  # Exit the function after calling main_window

            cv2.imshow('Webcam', img)
            if cv2.waitKey(1) & 0xFF == ord('q'):
                break
    finally:
        cap.release()
        cv2.destroyAllWindows()

if __name__ == "__main__":
    faceRec()
```

Configuração da base de dados

Este código configura a base de dados MySQL para a aplicação de gestão de oficina. Tenta conectar-se à base de dados existente e, se não existir, cria-a. De seguida, cria quatro tabelas: ”clients”, “services”, “payments” e “users”, para guardar dados sobre clientes, serviços, pagamentos e utilizadores, respetivamente.

```
import mysql.connector
from dotenv import load_dotenv
import os

load_dotenv()
PASSWORD = os.getenv('PASSWORD')

def connect():
    mysql.connector.connect(
        host="localhost",
        user="root",
        passwd=PASSWORD,
        database="trabalho_final"
    )
    print('Database connected!')

def create_db():
        mydb = mysql.connector.connect(
            host="localhost",
            user="root",
            passwd=PASSWORD,
        )

        mycursor = mydb.cursor()
        mycursor.execute('CREATE DATABASE trabalho_final')

        print('Database created!')

def create_tables():
    mydb = mysql.connector.connect(
        host="localhost",
        user="root",
        passwd=PASSWORD,
        database="trabalho_final"
    )

    mycursor = mydb.cursor()

    # creating tables

    mycursor.execute('CREATE TABLE clients (client_id INT AUTO_INCREMENT
PRIMARY KEY, \
    client_name VARCHAR(256), \
    client_address VARCHAR(256), \
    client_nif VARCHAR(9), \
    client_mobile VARCHAR(9), \
    client_email VARCHAR(50), \
    client_created DATE)')

    mycursor.execute('CREATE TABLE services (service_id INT
```

```
AUTO_INCREMENT PRIMARY KEY, \
    service_client_id INT, \
    service_type VARCHAR(50), \
    service_description VARCHAR(1024), \
    service_start_date DATE, \
    service_end_date DATE, \
    service_state VARCHAR(50), \
    FOREIGN KEY(service_client_id) REFERENCES clients(client_id))')

    mycursor.execute('CREATE TABLE payments (payment_id INT
AUTO_INCREMENT PRIMARY KEY, \
    payment_service_id INT, \
    payment_date DATE, \
    payment_value FLOAT, \
    payment_state VARCHAR(50), \
    payment_type VARCHAR(50), \
    FOREIGN KEY(payment_service_id) REFERENCES services(service_id))')

    mycursor.execute('CREATE TABLE users (user_id INT AUTO_INCREMENT
PRIMARY KEY, \
    user_name VARCHAR(50), \
    user_fullname VARCHAR(256), \
    user_password VARCHAR(64), \
    user_role VARCHAR(50))')
```

Gestão de Clientes

Este código faz operações na base de dados para clientes. Inclui funções para verificar, criar, atualizar, eliminar e mostrar clientes, usa o mysql.connector para se ligar ao MySQL e fazer as operações pedidas pelos utilizadores.

```python
# Importação dos módulos e bibliotecas
import mysql.connector
from dotenv import load_dotenv
import os
from mysql.connector import Error
from datetime import datetime

# Carregamento das variáveis de ambiente
load_dotenv()
PASSWORD = os.getenv("PASSWORD")

# Função para verificar se um cliente com um dado NIF já existe
def existe_cliente(nif):
    try:
        connection = mysql.connector.connect(
            host='localhost',
            database='trabalho_final',
            user='root',
            password=PASSWORD
        )
        if connection.is_connected():
            cursor = connection.cursor()
```

```
            cursor.execute('SELECT client_nif FROM clients WHERE
client_nif=%s', (nif,))
            result = cursor.fetchone()
            return result is None
    except Error as err:
        print(f"Erro ao conectar ao MySQL: {err}")
    finally:
        if connection.is_connected():
            cursor.close()
            connection.close()
            print("Conexão ao MySQL encerrada.")

# Função para criar um novo cliente
def create_client(client_name, client_address, client_nif, client_mobile,
client_email):
    try:
        connection = mysql.connector.connect(
            host='localhost',
            database='trabalho_final',
            user='root',
            password=PASSWORD
        )
        if connection.is_connected():
            cursor = connection.cursor()
            client_created = datetime.now().strftime('%Y-%m-%d')
            insert_query = """
                INSERT INTO clients (client_name, client_address,
client_nif, client_mobile, client_email, client_created)
                VALUES (%s, %s, %s, %s, %s, %s)
            """
            record = (client_name, client_address, client_nif,
client_mobile, client_email, client_created)
            cursor.execute(insert_query, record)
            connection.commit()
            print("Cliente inserido com sucesso.")
    except Error as err:
        print(f"Erro ao conectar ao MySQL: {err}")
    finally:
        if connection.is_connected():
            cursor.close()
            connection.close()
            print("Conexão ao MySQL encerrada.")

# Função para atualizar as informações de um cliente
def update_client(client_address, client_mobile, client_email,
client_id):
    try:
        connection = mysql.connector.connect(
            host='localhost',
            database='trabalho_final',
            user='root',
            password=PASSWORD
        )
        if connection.is_connected():
            cursor = connection.cursor()
            update_query = """
                UPDATE clients
                SET client_address = %s, client_mobile = %s, client_email
= %s
                WHERE client_id = %s
            """
```

```
            record = (client_address, client_mobile, client_email,
client_id)
            cursor.execute(update_query, record)
            connection.commit()
            print("Cliente atualizado com sucesso.")
    except Error as err:
        print(f"Erro ao conectar ao MySQL: {err}")
    finally:
        if connection.is_connected():
            cursor.close()
            connection.close()
            print("Conexão ao MySQL encerrada.")

# Função para excluir um cliente
def delete_client(client_id):
    try:
        connection = mysql.connector.connect(
            host='localhost',
            database='trabalho_final',
            user='root',
            password=PASSWORD
        )
        if connection.is_connected():
            cursor = connection.cursor()
            cursor.execute("DELETE FROM services WHERE service_client_id
= %s", (client_id,))
            delete_query = "DELETE FROM clients WHERE client_id = %s"
            cursor.execute(delete_query, (client_id,))
            connection.commit()
            print("Cliente excluído com sucesso.")
    except Error as err:
        print(f"Erro ao conectar ao MySQL: {err}")
    finally:
        if connection.is_connected():
            cursor.close()
            connection.close()
            print("Conexão ao MySQL encerrada.")

# Função para listar todos os clientes
def list_clients():
    try:
        connection = mysql.connector.connect(
            host='localhost',
            database='trabalho_final',
            user='root',
            password=PASSWORD
        )
        if connection.is_connected():
            cursor = connection.cursor()
            select_query = """SELECT client_id, client_name,
client_address, client_nif, client_mobile, client_email,
DATE_FORMAT(client_created, "%d/%m/%Y") FROM clients"""
            cursor.execute(select_query)
            clients = cursor.fetchall()
            return clients
    except Error as err:
        print(f"Erro ao conectar ao MySQL: {err}")
        return None
    finally:
        if connection.is_connected():
            cursor.close()
```

```
            connection.close()
            print("Conexão ao MySQL encerrada.")

# Função para verificar se um cliente existe com base no ID
def verify_client(client_id):
    try:
        connection = mysql.connector.connect(
            host='localhost',
            database='trabalho_final',
            user='root',
            password=PASSWORD
        )
        if connection.is_connected():
            cursor = connection.cursor()
            cursor.execute('SELECT client_id FROM clients WHERE client_id
= %s', (client_id,))
            result = cursor.fetchone()
            return result is not None
    except Error as err:
        print(f'Erro ao conectar a base de dados: {err}')
    finally:
        if connection.is_connected():
            cursor.close()
            connection.close()
            print("Conexão ao MySQL encerrada.")
```

Gestão de Serviços

Este código faz operações na base de dados para serviços. Inclui funções para criar, atualizar, eliminar e mostrar serviços, usa o mysql.connector para se ligar ao MySQL e fazer as operações pedidas pelos utilizadores.

```
import mysql.connector
from dotenv import load_dotenv
import os
from mysql.connector import Error
from datetime import datetime

load_dotenv()
PASSWORD = os.getenv("PASSWORD")

def create_service(service_client_id, service_type, service_description,
service_price):
    try:
        connection = mysql.connector.connect(
            host='localhost',
            database='trabalho_final',
            user='root',
            password=PASSWORD
        )

        if connection.is_connected():
            cursor = connection.cursor()
```

```
            insert_query = """
                INSERT INTO services (service_client_id, service_type,
service_description,
                service_start_date, service_state)
                VALUES (%s, %s, %s, %s, %s)
            """
            record = (service_client_id, service_type,
service_description,
                      datetime.now().strftime('%Y-%m-%d'), "Começado")

            insert_query_2 = """
                INSERT INTO payments (payment_service_id, payment_value,
payment_state)
                VALUES ((SELECT service_id FROM services ORDER BY
service_id DESC LIMIT 1), %s, %s)
            """
            record_2 = (service_price, "Por pagar")

            cursor.execute(insert_query, record)
            cursor.execute(insert_query_2, record_2)
            connection.commit()
            print("Serviço inserido com sucesso.")

    except Error as err:
        print("Erro ao conectar ao MySQL:", err)

    finally:
        if connection.is_connected():
            cursor.close()
            connection.close()
            print("Conexão ao MySQL encerrada.")

def update_service(service_id):
    try:
        connection = mysql.connector.connect(
            host='localhost',
            database='trabalho_final',
            user='root',
            password=PASSWORD
        )

        if connection.is_connected():
            cursor = connection.cursor()

            update_query = """
                UPDATE services
                SET service_end_date = %s,
                    service_state = %s
                WHERE service_id = %s
            """
            record = (datetime.now().strftime('%Y-%m-%d'), "Terminado",
service_id)

            cursor.execute(update_query, record)
            connection.commit()
            print("Serviço atualizado com sucesso.")
            return cursor.rowcount  # Return the number of affected rows

    except Error as err:
        print("Erro ao conectar ao MySQL:", err)
```

```
        return 0

    finally:
        if connection.is_connected():
            cursor.close()
            connection.close()
            print("Conexão ao MySQL encerrada.")

def delete_service(service_id):
    try:
        connection = mysql.connector.connect(
            host='localhost',
            database='trabalho_final',
            user='root',
            password=PASSWORD
        )

        if connection.is_connected():
            cursor = connection.cursor()

            delete_query = """
                DELETE FROM services
                WHERE service_id = %s
            """
            record = (service_id,)

            cursor.execute(delete_query, record)
            connection.commit()
            print("Serviço excluído com sucesso.")
            return cursor.rowcount  # Return the number of affected rows

    except Error as err:
        print("Erro ao conectar ao MySQL:", err)
        return 0

    finally:
        if connection.is_connected():
            cursor.close()
            connection.close()
            print("Conexão ao MySQL encerrada.")

def list_services():
    try:
        connection = mysql.connector.connect(
            host='localhost',
            database='trabalho_final',
            user='root',
            password=PASSWORD
        )

        if connection.is_connected():
            cursor = connection.cursor()
            query = """
            SELECT service_id, service_client_id, service_type,
service_description, DATE_FORMAT(service_start_date, "%d/%m/%Y"),
DATE_FORMAT(service_end_date, "%d/%m/%Y"), service_state FROM services
            """
            cursor.execute(query)
            services = cursor.fetchall()
            return services
```

```
    except mysql.connector.Error as err:
        print("Erro ao conectar ao MySQL:", err)
        return []

    finally:
        if connection.is_connected():
            cursor.close()
            connection.close()
            print("Conexão ao MySQL encerrada.")

def list_services_by_date(date):
    try:
        connection = mysql.connector.connect(
            host='localhost',
            database='trabalho_final',
            user='root',
            password=PASSWORD
        )

        if connection.is_connected():
            cursor = connection.cursor()
            query = """
            SELECT service_id, service_client_id, service_type,
service_description,
                   DATE_FORMAT(service_start_date, "%d/%m/%Y"),
DATE_FORMAT(service_end_date, "%d/%m/%Y"), service_state
            FROM services
            WHERE DATE(service_start_date) = %s
            """
            cursor.execute(query, (date,))
            services = cursor.fetchall()
            return services

    except mysql.connector.Error as err:
        print("Erro ao conectar ao MySQL:", err)
        return []

    finally:
        if connection.is_connected():
            cursor.close()
            connection.close()
            print("Conexão ao MySQL encerrada.")
```

Gestão de Pagamentos

Este código faz operações na base de dados para pagamentos. Inclui funções para criar, atualizar, eliminar e mostrar pagamentos, usa o mysql.connector para se ligar ao MySQL e fazer as operações pedidas pelos utilizadores.

```
import mysql.connector
from dotenv import load_dotenv
import os
from mysql.connector import Error
```

```
from datetime import datetime

# Load environment variables
load_dotenv()
PASSWORD = os.getenv("PASSWORD")

def create_payment(payment_date, payment_value, payment_state,
payment_type):
    try:
        # Connect to the database
        connection = mysql.connector.connect(
            host='localhost',
            database='trabalho_final',
            user='root',
            password=PASSWORD
        )

        if connection.is_connected():
            cursor = connection.cursor()

            # SQL query to insert a new payment
            insert_query = """
            INSERT INTO payments (payment_date, payment_value,
payment_state, payment_type)
            VALUES (%s, %s, %s, %s)
            """
            record = (payment_date, payment_value, payment_state,
payment_type)

            # Execute the query
            cursor.execute(insert_query, record)
            connection.commit()
            print("Pagamento inserido com sucesso.")
            return cursor.rowcount

    except Error as erro:
        print(f"Erro ao conectar ao MySQL: {erro}")
        return None
    finally:
        if connection.is_connected():
            cursor.close()
            connection.close()
            print("Conexão ao MySQL encerrada.")

def update_payment(payment_service_id, payment_type):
    try:
        # Connect to the database
        connection = mysql.connector.connect(
            host='localhost',
            database='trabalho_final',
            user='root',
            password=PASSWORD
        )

        if connection.is_connected():
            cursor = connection.cursor()

            # SQL query to update a payment
            update_query = """
            UPDATE payments
            SET
```

```
                payment_date = %s,
                payment_state = %s,
                payment_type = %s
            WHERE
                payment_service_id = %s
            """
            record = (datetime.now().strftime('%Y-%m-%d'), "Pago",
payment_type, payment_service_id)

            # Execute the query
            cursor.execute(update_query, record)
            connection.commit()
            print("Pagamento atualizado com sucesso.")
            return cursor.rowcount

    except Error as erro:
        print(f"Erro ao conectar ao MySQL: {erro}")
        return None
    finally:
        if connection.is_connected():
            cursor.close()
            connection.close()
            print("Conexão ao MySQL encerrada.")

def delete_payment(payment_id):
    try:
        # Connect to the database
        connection = mysql.connector.connect(
            host='localhost',
            database='trabalho_final',
            user='root',
            password=PASSWORD
        )

        if connection.is_connected():
            cursor = connection.cursor()

            # SQL query to delete a payment
            delete_query = """
            DELETE FROM payments
            WHERE payment_id = %s
            """
            record = (payment_id,)

            # Execute the query
            cursor.execute(delete_query, record)
            connection.commit()
            print("Pagamento eliminado com sucesso.")
            return cursor.rowcount

    except Error as erro:
        print(f"Erro ao conectar ao MySQL: {erro}")
        return None
    finally:
        if connection.is_connected():
            cursor.close()
            connection.close()
            print("Conexão ao MySQL encerrada.")

def list_payments():
    try:
```

```
        # Connect to the database
        connection = mysql.connector.connect(
            host='localhost',
            database='trabalho_final',
            user='root',
            password=PASSWORD
        )

        if connection.is_connected():
            cursor = connection.cursor()

            # SQL query to select all payments
            select_query = """SELECT payment_service_id,
DATE_FORMAT(payment_date, "%d/%m/%Y"), payment_value, payment_state,
payment_type FROM payments"""

            # Execute the query
            cursor.execute(select_query)

            # Fetch all rows
            payments = cursor.fetchall()

            return payments

    except Error as erro:
        print(f"Erro ao conectar ao MySQL: {erro}")
        return None
    finally:
        if connection.is_connected():
            cursor.close()
            connection.close()
            print("Conexão ao MySQL encerrada.")

def verify_payment(payment_service_id):
    try:
        connection = mysql.connector.connect(
            host='localhost',
            database='trabalho_final',
            user='root',
            password=PASSWORD
        )

        if connection.is_connected():
            cursor = connection.cursor()

            cursor.execute('SELECT payment_service_id FROM payments WHERE
payment_service_id = %s', (payment_service_id,))
            result = cursor.fetchone()

            if result is not None:
                return True
            else:
                return False

    except Error as err:
        print(f'Erro ao conectar a base de dados: {err}')

    finally:
        if connection.is_connected():
            cursor.close()
```

```
            connection.close()
            print("Conexão ao MySQL encerrada.")
```

Gestão de Utilizadores

Este código lida com a autenticação de utilizadores no sistema, utilizando MySQL para guardar os dados de login. Inclui funções para criar, eliminar e atualizar utilizadores, além de verificar credenciais de login e obter informações sobre os utilizadores.

```
# Importação dos módulos e bibliotecas
import mysql.connector
import bcrypt
from main import *
from dotenv import load_dotenv
import os

# Carregamento das variáveis de ambiente
load_dotenv()
PASSWORD = os.getenv("PASSWORD")

# Função para criar um novo utilizador
def create_user(user_name, user_fullname, user_password, user_role):
    try:
        # Conexão com a base de dados
        connection = mysql.connector.connect(
            host='localhost',
            database='trabalho_final',
            user='root',
            password=PASSWORD
        )

        if connection.is_connected():
            cursor = connection.cursor()

            # Consulta SQL para inserir um novo utilizador
            insert_query = 'INSERT INTO users (user_name, user_fullname,
user_password, user_role) \
                VALUES (%s, %s, %s, %s)'

            # Hash da password utilizando bcrypt
            hashed_password = bcrypt.hashpw(user_password.encode('utf-
8'), bcrypt.gensalt())
            record = (user_name, user_fullname,
hashed_password.decode('utf-8'), user_role)

            cursor.execute(insert_query, record)
            connection.commit()
            print("Utilizador inserido com sucesso.")
    except mysql.connector.Error as err:
        print(f'Erro ao criar o utilizador: {err}')
    finally:
        if connection.is_connected():
            cursor.close()
```

```
            connection.close()
            print("Conexão ao MySQL encerrada.")


# Função para remover um utilizador
def delete_user(user_name):
    try:
        # Conexão com a base de dados
        connection = mysql.connector.connect(
            host='localhost',
            database='trabalho_final',
            user='root',
            password=PASSWORD
        )

        if connection.is_connected():
            cursor = connection.cursor()

            # Consulta SQL para excluir um utilizador
            delete_query = 'DELETE FROM users WHERE user_name = %s'
            record = (user_name,)

            # Executar a consulta
            cursor.execute(delete_query, record)
            connection.commit()

            # Remover a imagem do utilizador se existir
            image_path = os.path.join("images/", f"{user_name}.jpg")
            if os.path.isfile(image_path):
                os.remove(image_path)

            print("Utilizador eliminado com sucesso.")
    except mysql.connector.Error as erro:
        print(f"Erro ao conectar ao MySQL: {erro}")
    finally:
        if connection.is_connected():
            cursor.close()
            connection.close()
            print("Conexão ao MySQL encerrada.")

# Função para atualizar a password de um utilizador
def update_password(user_name, user_password):
    try:
        # Conexão com a base de dados
        connection = mysql.connector.connect(
            host='localhost',
            database='trabalho_final',
            user='root',
            password=PASSWORD
        )

        if connection.is_connected():
            cursor = connection.cursor()

            # Hash da nova password utilizando bcrypt
            hashed_password = bcrypt.hashpw(user_password.encode('utf-
8'), bcrypt.gensalt())
            update_query = 'UPDATE users SET user_password = %s WHERE
user_name = %s'
            record = (hashed_password.decode('utf-8'), user_name)
```

```
            # Executar a consulta
            cursor.execute(update_query, record)
            connection.commit()
            print("Utilizador atualizado com sucesso.")
            return cursor.rowcount
    except mysql.connector.Error as erro:
        print(f"Erro ao conectar ao MySQL: {erro}")
        return None
    finally:
        if connection.is_connected():
            cursor.close()
            connection.close()
            print("Conexão ao MySQL encerrada.")

# Função para verificar a existência de um utilizador
def verify_user(user_name):
    try:
        connection = mysql.connector.connect(
            host='localhost',
            database='trabalho_final',
            user='root',
            password=PASSWORD
        )

        if connection.is_connected():
            cursor = connection.cursor()

            # Consulta SQL para verificar a existência de um utilizador
            cursor.execute('SELECT user_name FROM users WHERE user_name =
%s', (user_name,))
            result = cursor.fetchone()

            return result is not None
    except mysql.connector.Error as err:
        print(f'Erro ao conectar à base de dados: {err}')
    finally:
        if connection.is_connected():
            cursor.close()
            connection.close()
            print("Conexão ao MySQL encerrada.")

# Função para verificar o login de um utilizador
def login(user_name, user_password):
    try:
        connection = mysql.connector.connect(
            host='localhost',
            database='trabalho_final',
            user='root',
            password=PASSWORD
        )

        if connection.is_connected():
            cursor = connection.cursor()

            # Consulta SQL para obter a password do utilizador
            cursor.execute('SELECT user_password FROM users WHERE
user_name = %s', (user_name,))
            result = cursor.fetchone()

            if result is not None:
                stored_password = result[0]
```

```
                # Verificação da password com bcrypt
                if bcrypt.checkpw(user_password.encode('utf-8'),
stored_password.encode('utf-8')):
                    print("Login bem-sucedido.")
                    return True
                else:
                    print("Username ou password incorretos")
                    return False
            else:
                print("Utilizador não encontrado.")
                return False
    except mysql.connector.Error as err:
        print(f'Erro ao conectar à base de dados: {err}')
    finally:
        if connection.is_connected():
            cursor.close()
            connection.close()
            print("Conexão ao MySQL encerrada.")

# Função para verificar o cargo de um utilizador (Operador ou
Contabilidade)
def verify_login(user_name):
    try:
        connection = mysql.connector.connect(
            host='localhost',
            database='trabalho_final',
            user='root',
            password=PASSWORD
        )

        if connection.is_connected():
            cursor = connection.cursor()

            # Consulta SQL para verificar o cargo do utilizador
            cursor.execute('SELECT user_name FROM users WHERE user_name =
%s AND user_role = "Operador"', (user_name,))
            result = cursor.fetchone()

            return "Operador" if result is not None else "Contabilidade"
    except mysql.connector.Error as err:
        print(f'Erro ao conectar à base de dados: {err}')
    finally:
        if connection.is_connected():
            cursor.close()
            connection.close()
            print("Conexão ao MySQL encerrada.")


# Função para obter o nome completo de um utilizador
def login_fullname(user_name):
    try:
        connection = mysql.connector.connect(
            host='localhost',
            database='trabalho_final',
            user='root',
            password=PASSWORD
        )

        if connection.is_connected():
            cursor = connection.cursor()
```

```
            # Consulta SQL para obter o nome completo do utilizador
            cursor.execute('SELECT user_fullname FROM users WHERE
user_name = %s', (user_name,))
            result = cursor.fetchone()

            return result[0] if result else None
    except mysql.connector.Error as err:
        print(f'Erro ao conectar à base de dados: {err}')
    finally:
        if connection.is_connected():
            cursor.close()
            connection.close()
            print("Conexão ao MySQL encerrada.")
```

Implementação do *Front-End*

Menu Principal

Este código cria a janela principal da aplicação, com um menu ajustado ao papel do utilizador (*Admin*, Operador, Contabilidade). Dependendo do papel, permite gerir utilizadores, clientes, pagamentos e serviços. Exibe um logotipo, uma mensagem de boas-vindas, e inclui uma opção para sair da aplicação.

```
from users import *
from payments import *
from services import *
from clients import *
import tkinter as tk
from tkinter import PhotoImage
import sys

def main_window(role, user_fullname):
    root = tk.Tk()
    root.title("Norauto")
    root.geometry("600x400")
    root.configure(bg="#1f286d")

    menu_bar = tk.Menu(root, tearoff=0)

    if role == "Admin":
        menu_utilizadores = tk.Menu(menu_bar, tearoff=0)
        menu_bar.add_cascade(label="Utilizadores",
menu=menu_utilizadores)
        menu_utilizadores.add_command(label="Adicionar",
command=add_user)
        menu_utilizadores.add_command(label="Remover",
command=remove_user)
        menu_utilizadores.add_command(label="Mudar palavra-passe",
command=change_password)
        menu_utilizadores.add_command(label="Atualizar fotografia",
command=update_photo)

    if role == "Admin" or role == "Operador":
        menu_clientes = tk.Menu(menu_bar, tearoff=0)
        menu_bar.add_cascade(label="Clientes", menu=menu_clientes)
        menu_clientes.add_command(label="Adicionar", command=add_client)
        menu_clientes.add_command(label="Remover", command=remove_client)
        menu_clientes.add_command(label="Lista de Clientes",
command=list_all_clients)

    if role == "Admin" or role == "Contabilidade":
        menu_pagamentos = tk.Menu(menu_bar, tearoff=0)
        menu_bar.add_cascade(label="Pagamentos", menu=menu_pagamentos)
        menu_pagamentos.add_command(label="Finalizar",
command=finish_payment)
        menu_pagamentos.add_command(label="Lista de Pagamentos",
command=list_all_payments)

    menu_services = tk.Menu(menu_bar, tearoff=0)
    menu_bar.add_cascade(label="Serviços", menu=menu_services)
    if role == "Admin" or role == "Operador":
```

```
        menu_services.add_command(label="Começar",
command=create_service_page)
        menu_services.add_command(label="Terminar",
command=update_service_page)
    menu_services.add_command(label="Lista de Serviços",
command=list_service_page)

    menu_bar.add_command(label="Sair", command=lambda: sys.exit())

    root.config(menu=menu_bar)

    logo = PhotoImage(file="assets/logo.png", master=root)
    label_logo = tk.Label(root, image=logo, bg="#1f286d")
    label_logo.place(relx=0.5, rely=0.5, anchor=tk.CENTER)

    bem_vindo = tk.Label(root, text=f"Bem-vindo, {user_fullname}",
bg="#1f286d", fg="white", font=("Helvetica", 20, "bold"))
    bem_vindo.place(relx=0.5, rely=0.15, anchor=tk.CENTER)

    root.mainloop()
```

Log In

Este código cria uma interface de login para a aplicação. Tenta conectar-se à base de dados e, se falhar, cria-a com um utilizador *admin*. A interface oferece ainda duas opções de login: reconhecimento facial para utilizadores "comuns" e um login com senha para administradores.

```
# Importação dos módulos e bibliotecas
from src.faceRec import *
from src.database.database import *
from main import *
from src.database.users import create_user, login
from tkinter import messagebox
import tkinter as tk

# Tenta conectar-se à base de dados, se falhar cria a base de dados e as
tabelas, e adiciona um utilizador admin.
try:
    connect()
except:
    create_db()
    create_tables()
    create_user("admin", "Admin", "admin", "Admin")

def quit_login():
    root.destroy()

# Função para a interface de login do administrador
def admin():
    admin_login = tk.Toplevel()
    admin_login.title("Login Administrador")
    tk.Label(admin_login, text="Nome de utilizador:").grid(row=0,
column=0)
```

```
    tk.Label(admin_login, text="Password:").grid(row=1, column=0)
    eUsername = tk.Entry(admin_login)
    ePassword = tk.Entry(admin_login, show="*")
    eUsername.grid(row=0, column=1)
    ePassword.grid(row=1, column=1)

    def aLogin():
        username = eUsername.get()
        password = ePassword.get()
        if username and password:
            if login(username, password):
                messagebox.showinfo("Sucesso", "Login efetuado com
sucesso!")
                admin_login.destroy()
                root.destroy()
                main_window("Admin", login_fullname(username))
            else:
                messagebox.showerror("Erro", "Nome de utilizador ou
palavra-passe incorretos.")
        else:
            messagebox.showerror("Erro", "Por favor, preencha todos os
campos.")

    tk.Button(admin_login, text="Login", command=aLogin).grid(row=4,
columnspan=2)
    admin_login.mainloop()

# Configuração da janela principal
root = tk.Tk()
root.title("Norauto - Login")
root.geometry("600x400")
root.configure(bg="#1f286d")

# Carregar a logo
logo = PhotoImage(file="assets/logo.png")

# Configuração do canvas para exibir a logo
canvas = tk.Canvas(root, width=logo.width(), height=logo.height(),
bg="#1f286d", highlightthickness=0)
canvas.pack(expand=True)
canvas.create_image(0, 0, anchor="nw", image=logo)

# Botões de login
button_login = tk.Button(root, text="Login", command=faceRec)
button_admin_login = tk.Button(root, text="Admin Login", command=admin)

# Posicionamento dos botões no canvas
canvas.create_window(logo.width() // 2, (logo.height() // 2) + 125,
anchor="center", window=button_login)
canvas.create_window(logo.width() // 2, (logo.height() // 2) + 160,
anchor="center", window=button_admin_login)

root.mainloop()
```

Gestão de Clientes

Este código apresenta uma interface gráfica simples em Tkinter para interagir com a base de dados de clientes. Inclui funções para adicionar um novo cliente, remover um cliente existente e mostrar todos os clientes guardando-os na base de dados.

```
# Importação dos módulos e bibliotecas
import tkinter as tk
from tkinter import ttk, messagebox
import re
from src.database.clients import create_client, delete_client,
list_clients, existe_cliente, verify_client


# Função para adicionar um novo cliente
def add_client():
    adicionar_cliente = tk.Toplevel()
    adicionar_cliente.title("Criar cliente")

    # Labels e campos de entrada para os detalhes do cliente
    tk.Label(adicionar_cliente, text="Nome:").grid(row=0, column=0)
    tk.Label(adicionar_cliente, text="Morada:").grid(row=1, column=0)
    tk.Label(adicionar_cliente, text="NIF:").grid(row=2, column=0)
    tk.Label(adicionar_cliente, text="Número de telefone:").grid(row=3,
column=0)
    tk.Label(adicionar_cliente, text="Email:").grid(row=4, column=0)
    eName = tk.Entry(adicionar_cliente)
    eAddress = tk.Entry(adicionar_cliente)

    # Função para validar o comprimento de 9 caracteres
    def validar9char(P):
        return len(P) <= 9

    # Configuração da validação de entrada para NIF e número de telefone
    valNif = (adicionar_cliente.register(validar9char), "%P")
    eNif = tk.Entry(adicionar_cliente, validate="key",
validatecommand=valNif)
    valMobile = (adicionar_cliente.register(validar9char), "%P")
    eMobile = tk.Entry(adicionar_cliente, validate="key",
validatecommand=valMobile)
    eMail = tk.Entry(adicionar_cliente)
    eName.grid(row=0, column=1)
    eAddress.grid(row=1, column=1)
    eNif.grid(row=2, column=1)
    eMobile.grid(row=3, column=1)
    eMail.grid(row=4, column=1)

    # Função para validar o formato do email
    def validar_email(email):
        pattern = r'^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$'
        return re.match(pattern, email) is not None

    # Função para adicionar o cliente à base de dados
    def adicionar():
        name = eName.get()
        address = eAddress.get()
        nif = eNif.get()
```

```
        mobile = eMobile.get()
        mail = eMail.get()
        if name and address and nif and mobile and mail:
            if not validar_email(mail):
                messagebox.showerror("Erro", "Email inválido.")
                return
            if not (nif.isdigit() and len(nif) == 9):
                messagebox.showerror("Erro", "NIF deve conter exatamente
9 dígitos.")
                return
            if not (mobile.isdigit() and len(mobile) == 9):
                messagebox.showerror("Erro", "Número de telefone deve
conter exatamente 9 dígitos.")
                return
            if existe_cliente(nif):
                messagebox.showerror("Erro", "Cliente já existe.")
                return
            create_client(name, address, nif, mobile, mail)
            messagebox.showinfo("Sucesso", "Cliente criado com sucesso!")
            adicionar_cliente.destroy()
        else:
            messagebox.showerror("Erro", "Por favor, preencha todos os
campos.")

    # Botão para adicionar o cliente
    tk.Button(adicionar_cliente, text="Adicionar",
command=adicionar).grid(row=5, columnspan=2)
    adicionar_cliente.mainloop()


# Função para remover um cliente existente
def remove_client():
    remover_cliente = tk.Toplevel()
    remover_cliente.title("Remover cliente")

    # Label e campo de entrada para o ID do cliente
    tk.Label(remover_cliente, text="ID do cliente:").grid(row=0,
column=0)
    eID = tk.Entry(remover_cliente)
    eID.grid(row=0, column=1)

    # Função para remover o cliente da base de dados
    def remover():
        id = eID.get()
        if id:
            if verify_client(id):
                delete_client(id)
                messagebox.showinfo("Sucesso", "Cliente removido com
sucesso!")
                remover_cliente.destroy()
            else:
                messagebox.showerror("Erro", "Cliente não existe.")
        else:
            messagebox.showerror("Erro", "Por favor, preencha todos os
campos.")

    # Botão para remover o cliente
    tk.Button(remover_cliente, text="Remover",
command=remover).grid(row=3, columnspan=2)
    remover_cliente.mainloop()
```

```
# Função para listar todos os clientes
def list_all_clients():
    listar_clients = tk.Toplevel()
    listar_clients.title("Lista de Clientes")

    # Labels para os cabeçalhos das colunas
    tk.Label(listar_clients, text="ID").grid(row=0, column=0)
    tk.Label(listar_clients, text="Nome").grid(row=0, column=1)
    tk.Label(listar_clients, text="Morada").grid(row=0, column=2)
    tk.Label(listar_clients, text="NIF").grid(row=0, column=3)
    tk.Label(listar_clients, text="Telefone").grid(row=0, column=4)
    tk.Label(listar_clients, text="Email").grid(row=0, column=5)
    tk.Label(listar_clients, text="Criado a").grid(row=0, column=6)

    # Obtém a lista de clientes da base de dados
    clients = list_clients()

    # Verificação se há clientes para mostrar
    if not clients:
        messagebox.showinfo("Info", "Não há clientes para mostrar.")
        listar_clients.destroy()
        return

    # Preenchimento dos dados dos clientes nas colunas correspondentes
    for i, element in enumerate(clients, start=1):
        id, name, address, nif, phone, email, created = element
        tk.Label(listar_clients, text=id).grid(row=i, column=0)
        tk.Label(listar_clients, text=name).grid(row=i, column=1)
        tk.Label(listar_clients, text=address).grid(row=i, column=2)
        tk.Label(listar_clients, text=nif).grid(row=i, column=3)
        tk.Label(listar_clients, text=phone).grid(row=i, column=4)
        tk.Label(listar_clients, text=email).grid(row=i, column=5)
        tk.Label(listar_clients, text=created).grid(row=i, column=6)

    listar_clients.mainloop()
```

Gestão de Serviços

Este código apresenta uma interface gráfica simples em Tkinter para interagir com a base de dados de serviços. Inclui funcionalidades para começar um novo serviço, finalizar um serviço existente e mostrar todos os serviços filtrando por data.

```
import tkinter as tk
from tkinter import ttk
from tkinter import messagebox
from tkcalendar import DateEntry
from src.database.services import *
from src.database.clients import verify_client

def create_service_page():
    add_window = tk.Toplevel()
    add_window.title("Começar Serviço")
```

```
    labels = ["Cliente", "Tipo de serviço", "Descrição", "Preço"]
    entries = []
    for i, label in enumerate(labels):
        tk.Label(add_window, text=label).grid(row=i, column=0)
        if label == "Tipo de serviço":
            entry = ttk.Combobox(add_window, values=["Manutenção",
"Colisão"])
        elif label == "Preço":
            entry = tk.Entry(add_window)
            entry.insert(0, "0.0")  # Define o valor inicial como 0.0
        else:
            entry = tk.Entry(add_window)
        entry.grid(row=i, column=1)
        entries.append(entry)

    def on_submit():
        client_id = entries[0].get().strip()
        service_type = entries[1].get().strip()
        description = entries[2].get().strip()
        price = entries[3].get().strip()

        if not client_id or not service_type or not description or not
price:
            messagebox.showerror("Erro", "Por favor, preencha todos os
campos.")
            return

        if service_type not in ["Manutenção", "Colisão"]:
            messagebox.showerror("Erro", "Tipo de serviço inválido.")
            return

        try:
            price = float(price)
            if price < 0:
                raise ValueError
        except ValueError:
            messagebox.showerror("Erro", "Preço inválido.")
            return

        if verify_client(client_id):
            try:
                create_service(client_id, service_type, description,
price)
                messagebox.showinfo("Sucesso", "Serviço criado com
sucesso!")
                add_window.destroy()
            except Exception as e:
                messagebox.showerror("Erro", f"Falha ao criar serviço:
{e}")
        else:
            messagebox.showerror("Erro", "ID do cliente não existe.")

    tk.Button(add_window, text="Começar",
command=on_submit).grid(row=len(labels), column=1)
    add_window.mainloop()

def update_service_page():
    update_window = tk.Toplevel()
    update_window.title("Finalizar Serviço")
    tk.Label(update_window, text="ID do serviço:").grid(row=0, column=0)
```

```
    eId = tk.Entry(update_window)
    eId.grid(row=0, column=1)

    def terminar():
        service_id = eId.get().strip()
        if service_id:
            try:
                update_service(service_id)
                messagebox.showinfo("Sucesso", "Serviço finalizado com
sucesso!")
                update_window.destroy()
            except Exception as e:
                messagebox.showerror("Erro", f"Falha ao finalizar
serviço: {e}")
        else:
            messagebox.showerror("Erro", "Por favor, preencha todos os
campos.")

    tk.Button(update_window, text="Terminar",
command=terminar).grid(row=3, columnspan=2)
    update_window.mainloop()

def list_service_page():
    list_window = tk.Toplevel()
    list_window.title("Lista de Serviços")

    tk.Label(list_window, text="Filtrar por data:").grid(row=0, column=2,
columnspan=3)
    date_entry = DateEntry(list_window, date_pattern="dd/mm/yyyy")
    date_entry.grid(row=0, column=4, columnspan=4)
    tk.Button(list_window, text="Filtrar", command=lambda:
filter_services(date_entry.get_date())).grid(row=0, column=6,
columnspan=3)

    columns = ["ID", "ID Cliente", "Tipo", "Descrição", "Data de início",
"Data de fim", "Estado"]
    for i, column in enumerate(columns):
        tk.Label(list_window, text=column).grid(row=1, column=i)

    def filter_services(date):
        for widget in list_window.grid_slaves():
            if int(widget.grid_info()["row"]) > 1:
                widget.grid_forget()

        services = list_services_by_date(date)
        if not services:
            messagebox.showinfo("Info", "Não há serviços para mostrar.")
            return

        for i, element in enumerate(services, start=2):
            id, client, type, desc, date_inic, date_fim, state = element
            tk.Label(list_window, text=id).grid(row=i, column=0)
            tk.Label(list_window, text=client).grid(row=i, column=1)
            tk.Label(list_window, text=type).grid(row=i, column=2)
            tk.Label(list_window, text=desc).grid(row=i, column=3)
            tk.Label(list_window, text=date_inic).grid(row=i, column=4)
            tk.Label(list_window, text=date_fim).grid(row=i, column=5)
            tk.Label(list_window, text=state).grid(row=i, column=6)

    list_window.mainloop()
```

Gestão de Pagamentos

Este código proporciona uma interface gráfica que permite finalizar pagamentos e mostrar todos os pagamentos registados na base de dados.

```
from datetime import datetime
import tkinter as tk
from tkinter import ttk
from src.database.payments import *
from tkinter import messagebox

def finish_payment():
    finalizar_pagamento = tk.Toplevel()
    finalizar_pagamento.title("Finalizar pagamento")
    tk.Label(finalizar_pagamento, text="ID serviço:").grid(row=0,
column=0)
    eId = tk.Entry(finalizar_pagamento)
    eId.grid(row=0, column=1)
    def procurar():
        id = eId.get()
        if id:
            if verify_payment(id):
                pedir_metodo = tk.Toplevel()
                pedir_metodo.title("Método de pagamento")
                tk.Label(pedir_metodo, text="Método de
pagamento:").grid(row=0, column=0)
                eMetodo = ttk.Combobox(pedir_metodo, values=["Numerário",
"Multibanco", "MB Way", "Apple Pay", "Transferencia Bancária", "Cheque"])
                eMetodo.grid(row=0, column=1)
                def guardar():
                    metodo = eMetodo.get()
                    if metodo:
                        update_payment(id, metodo)
                        messagebox.showinfo("Sucesso", "Pagamento
finalizado com sucesso!")
                        pedir_metodo.destroy()
                        finalizar_pagamento.destroy()
                    else:
                        messagebox.showerror("Erro", "Por favor, preencha
todos os campos.")
                tk.Button(pedir_metodo, text="Guardar",
command=guardar).grid(row=4, columnspan=2)
                pedir_metodo.mainloop()
            else:
                messagebox.showerror("Erro", "Serviço não existe.")
        else:
            messagebox.showerror("Erro", "Por favor, preencha todos os
campos.")
    tk.Button(finalizar_pagamento, text="Procurar",
command=procurar).grid(row=3, columnspan=2)
    finalizar_pagamento.mainloop()

def list_all_payments():
    listar_pagamentos = tk.Toplevel()
    listar_pagamentos.title("Lista de Pagamentos")

    tk.Label(listar_pagamentos, text="ID Serviço").grid(row=0, column=0)
    tk.Label(listar_pagamentos, text="Data").grid(row=0, column=1)
```

```
    tk.Label(listar_pagamentos, text="Valor").grid(row=0, column=2)
    tk.Label(listar_pagamentos, text="Estado").grid(row=0, column=3)
    tk.Label(listar_pagamentos, text="Tipo de Pagamento").grid(row=0,
column=4)

    payments = list_payments()

    if not payments:
        messagebox.showinfo("Info", "Não há pagamentos para mostrar.")
        listar_pagamentos.destroy()
        return

    for i, element in enumerate(payments, start=1):
        service_id, date, value, state, typePayment = element
        tk.Label(listar_pagamentos, text=service_id).grid(row=i,
column=0)
        tk.Label(listar_pagamentos, text=date).grid(row=i, column=1)
        tk.Label(listar_pagamentos, text=value).grid(row=i, column=2)
        tk.Label(listar_pagamentos, text=state).grid(row=i, column=3)
        tk.Label(listar_pagamentos, text=typePayment).grid(row=i,
column=4)

    listar_pagamentos.mainloop()
```

Gestão de Utilizadores

Este código proporciona uma interface para adicionar, remover e atualizar utilizadores no sistema, incluindo a funcionalidade para alterar a fotografia de reconhecimento facial associada a um utilizador.

```
# Importação dos módulos e bibliotecas
from src.database.users import *
from src.takePic import takePic
from tkinter import messagebox
import tkinter as tk
from tkinter import ttk

# Função para adicionar um novo utilizador
def add_user():
    # Criação de uma nova janela para adicionar utilizadores
    adicionar_utilizador = tk.Toplevel()
    adicionar_utilizador.title("Criar utilizador")

    # Labels e campos de entrada para os detalhes do utilizador
    tk.Label(adicionar_utilizador, text="Nome de
utilizador:").grid(row=0, column=0)
    tk.Label(adicionar_utilizador, text="Nome completo:").grid(row=1,
column=0)
    tk.Label(adicionar_utilizador, text="Cargo:").grid(row=2, column=0)
    eName = tk.Entry(adicionar_utilizador)
    eFullName = tk.Entry(adicionar_utilizador)
    eRole = ttk.Combobox(adicionar_utilizador, values=["Admin",
"Operador", "Contabilidade"])
    eName.grid(row=0, column=1)
```

```
    eFullName.grid(row=1, column=1)
    eRole.grid(row=2, column=1)

    # Função para adicionar o utilizador
    def adicionar():
        name = eName.get()
        fullName = eFullName.get()
        role = eRole.get()

        # Verificação se todos os campos estão preenchidos
        if name and fullName and role:
            if verify_user(name):
                messagebox.showerror("Erro", "Nome de utilizador já
existe.")
            else:
                if role == "Admin":
                    # Se o cargo for "Admin", solicitar a palavra-passe
                    pedir_password = tk.Toplevel()
                    pedir_password.title("Introduza palavra-passe")
                    tk.Label(pedir_password, text="Palavra-
passe:").grid(row=0, column=0)
                    tk.Label(pedir_password, text="Repita palavra-
passe:").grid(row=1, column=0)
                    ePassword = tk.Entry(pedir_password, show="*")
                    eRePassword = tk.Entry(pedir_password, show="*")
                    ePassword.grid(row=0, column=1)
                    eRePassword.grid(row=1, column=1)

                    # Função para guardar a palavra-passe
                    def guardar():
                        password = ePassword.get()
                        rePassword = eRePassword.get()
                        if password and rePassword:
                            if password == rePassword:
                                create_user(name, fullName, password,
role)
                                messagebox.showinfo("Sucesso",
"Utilizador criado com sucesso!")
                                pedir_password.destroy()
                                adicionar_utilizador.destroy()
                            else:
                                messagebox.showerror("Erro", "Palavras-
passe não correspondem.")
                        else:
                            messagebox.showerror("Erro", "Por favor,
preencha todos os campos.")

                    tk.Button(pedir_password, text="Guardar",
command=guardar).grid(row=4, columnspan=2)
                    pedir_password.mainloop()
                else:
                    # Captura de fotografia se o cargo não for "Admin"
                    takePic(name)
                    create_user(name, fullName, "", role)
                    messagebox.showinfo("Sucesso", "Utilizador criado com
sucesso!")
                    adicionar_utilizador.destroy()
        else:
            messagebox.showerror("Erro", "Por favor, preencha todos os
campos.")
```

```
    tk.Button(adicionar_utilizador, text="Adicionar",
command=adicionar).grid(row=4, columnspan=2)
    adicionar_utilizador.mainloop()


# Função para remover um utilizador existente
def remove_user():
    # Criação de uma nova janela para remover utilizadores
    remover_utilizador = tk.Toplevel()
    remover_utilizador.title("Remover utilizador")

    # Label e campo de entrada para o nome do utilizador
    tk.Label(remover_utilizador, text="Nome de utilizador:").grid(row=0,
column=0)
    eName = tk.Entry(remover_utilizador)
    eName.grid(row=0, column=1)

    # Função para remover o utilizador
    def remover():
        name = eName.get()
        if name:
            if verify_user(name):
                delete_user(name)
                messagebox.showinfo("Sucesso", "Utilizador removido com
sucesso!")
                remover_utilizador.destroy()
            else:
                messagebox.showerror("Erro", "Utilizador não existe.")
        else:
            messagebox.showerror("Erro", "Por favor, preencha todos os
campos.")

    tk.Button(remover_utilizador, text="Remover",
command=remover).grid(row=3, columnspan=2)
    remover_utilizador.mainloop()


# Função para mudar a palavra-passe de um utilizador existente
def change_password():
    # Criação de uma nova janela para mudar a palavra-passe
    mudar_password = tk.Toplevel()
    mudar_password.title("Mudar palavra-passe")

    # Label e campo de entrada para o nome do utilizador
    tk.Label(mudar_password, text="Nome de utilizador:").grid(row=0,
column=0)
    eName = tk.Entry(mudar_password)
    eName.grid(row=0, column=1)

    # Função para mudar a palavra-passe
    def mudar():
        name = eName.get()
        if name:
            if verify_user(name):
                # Criação de uma janela para introduzir a nova palavra-
passe
                pedir_password = tk.Toplevel()
                pedir_password.title("Introduza palavra-passe")
                tk.Label(pedir_password, text="Palavra-
passe:").grid(row=0, column=0)
                tk.Label(pedir_password, text="Repita palavra-
```

```
passe:").grid(row=1, column=0)
                ePassword = tk.Entry(pedir_password, show="*")
                eRePassword = tk.Entry(pedir_password, show="*")
                ePassword.grid(row=0, column=1)
                eRePassword.grid(row=1, column=1)

                # Função para guardar a nova palavra-passe
                def guardar():
                    password = ePassword.get()
                    rePassword = eRePassword.get()
                    if password and rePassword:
                        if password == rePassword:
                            update_password(name, password)
                            messagebox.showinfo("Sucesso", "Palavra-passe
alterada com sucesso!")
                            pedir_password.destroy()
                            mudar_password.destroy()
                        else:
                            messagebox.showerror("Erro", "Palavras-passe
não correspondem.")
                    else:
                        messagebox.showerror("Erro", "Por favor, preencha
todos os campos.")

                tk.Button(pedir_password, text="Guardar",
command=guardar).grid(row=4, columnspan=2)
                pedir_password.mainloop()
            else:
                messagebox.showerror("Erro", "Utilizador não existe.")
        else:
            messagebox.showerror("Erro", "Por favor, preencha todos os
campos.")

    tk.Button(mudar_password, text="Mudar", command=mudar).grid(row=3,
columnspan=2)
    mudar_password.mainloop()

# Função para atualizar a fotografia de um utilizador
def update_photo():
    # Criação de uma nova janela para atualizar a fotografia
    atualizar_foto = tk.Toplevel()
    atualizar_foto.title("Atualizar fotografia")

    # Label e campo de entrada para o nome do utilizador
    tk.Label(atualizar_foto, text="Nome de utilizador:").grid(row=0,
column=0)
    eName = tk.Entry(atualizar_foto)
    eName.grid(row=0, column=1)

    # Função para atualizar a fotografia
    def atualizar():
        name = eName.get()
        if name:
            if verify_user(name):
                takePic(name)
                messagebox.showinfo("Sucesso", "Fotografia atualizada com
sucesso!")
                atualizar_foto.destroy()
            else:
                messagebox.showerror("Erro", "Utilizador não existe.")
        else:
```

```
            messagebox.showerror("Erro", "Por favor, preencha todos os
campos.")

    tk.Button(atualizar_foto, text="Atualizar",
command=atualizar).grid(row=1, columnspan=2)
    atualizar_foto.mainloop()
```

Implementação de API (Serviço Web)

Criação do Servidor e End-Points

Este código Flask cria uma Application Program Interface (API) para interagir com a base de dados MySQL, fornecendo rotas para aceder a informações sobre clientes, pagamentos e serviços armazenados na base de dados. Utiliza consultas SQL para procurar os dados de cada tabela e devolve os resultados em formato JSON.

```python
# importa as biobliotecas
from flask import Flask, jsonify
from dotenv import load_dotenv
import mysql.connector
import os

load_dotenv()
PASSWORD = os.getenv("PASSWORD")
# verifica se a password esta definida no ficheiro .env
if not PASSWORD:
    raise ValueError("No PASSWORD environment variable set")

app = Flask(__name__)
# conneca-se ao mysql
def connect_db():
    connection = mysql.connector.connect(
        host="localhost",
        user="root",
        passwd=PASSWORD,
        database="trabalho_final"
    )
    print('Database connected!')
    return connection

# define a rota principal
@app.route("/")
def home():
    routes ={
        "clients": "/norauto/api/clients",
        "payments": "/norauto/api/payments",
        "services": "/norauto/api/services"
    }
    return jsonify(routes)

# define a rota dos clientes
@app.route("/norauto/api/clients")
```

```
def get_clients():
    try:
        # Conexão com o banco de dados
        connection = connect_db()

        if connection.is_connected():
            cursor = connection.cursor(dictionary=True)

            cursor.execute("SELECT * FROM clients")
            clients = cursor.fetchall()

            clients_dict = {client['client_id']: {
                "client_address": client['client_address'],
                "client_created": client['client_created'],
                "client_email": client['client_email'],
                "client_mobile": client['client_mobile'],
                "client_name": client['client_name'],
                "client_nif": client['client_nif']
            } for client in clients}
            return jsonify(clients_dict)

    except mysql.connector.Error as err:
        return jsonify({"error": str(err)}), 500

    finally:
        if connection.is_connected():
            cursor.close()
            connection.close()
            print("conexão ao mysql encerrada.")

    return jsonify({"error": "Nao foi possivel obter dados"}), 500
# define a rota dos pagamentos
@app.route("/norauto/api/payments")
def payments():
    try:
        # Conexão com o banco de dados
        connection = connect_db()

        if connection.is_connected():
            cursor = connection.cursor(dictionary=True)

            cursor.execute("SELECT * FROM payments")
            payments = cursor.fetchall()
            payments_dict = {payment['payment_id']: {
                "payment_date": payment['payment_date'],
                "payment_value": payment['payment_value'],
                "payment_state": payment['payment_state'],
                "payment_type": payment['payment_type']
            } for payment in payments}
            return jsonify(payments_dict)

    except mysql.connector.Error as err:
        return jsonify({"error": str(err)}), 500

    finally:
        if connection.is_connected():
            cursor.close()
            connection.close()
            print("conexão ao mysql encerrada.")

    return jsonify({"error": "Nao foi possivel obter dados"}), 500
```

```python
# define a rota dos servicos
@app.route("/norauto/api/services")
def services():
    try:
        # Conexão com o banco de dados
        connection = connect_db()

        if connection.is_connected():
            cursor = connection.cursor(dictionary=True)

            cursor.execute("SELECT * FROM services")
            services = cursor.fetchall()
            services_dict = {service['service_id']: {
                "service_client_id": service['service_client_id'],
                "service_created": service['service_created'],
                "service_description": service['service_description'],
                "service_end_date": service['service_end_date'],
                "service_price": service['service_price'],
                "service_start_date": service['service_start_date'],
                "service_state": service['service_state'],
                "service_type": service['service_type'],
                "service_updated": service['service_updated']
            } for service in services}
            return jsonify(services_dict)

    except mysql.connector.Error as err:
        return jsonify({"error": str(err)}), 500

    finally:
        if connection.is_connected():
            cursor.close()
            connection.close()
            print("conexão ao mysql encerrada.")

    return jsonify({"error": "Nao foi possivel obter dados"}), 500

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5005, debug=True)
```

Teste para Serviços

Este código faz uma solicitação GET para uma API Flask local para obter informações sobre um serviço com um ID especificado pelo utilizador em linha de comandos. Este imprime os detalhes do serviço se encontrado, caso não encontre, indica que nenhum serviço foi encontrado ou houve uma falha na obtenção de dados.

```python
import requests
import sys
import json

# Verifica se o script foi executado com um argumento
if len(sys.argv) != 2:
```

```
    print("Uso: python script.py <id>")
    sys.exit(1)

# Verifica se o argumento é uma string
id = sys.argv[1]

# Envia a solicitação GET para o endpoint da API
response = requests.get('http://127.0.0.1:5005/norauto/api/services')

# Verifica se a solicitação foi bem-sucedida
if response.status_code == 200:
    data = response.json()

    # Verifica se o ID está presente nos dados
    if id in data:
        # Formata os dados em JSON antes de imprimir
        formatted_data = json.dumps(data[id], indent=4)
        print(formatted_data)
    else:
        print(f'Nenhum item encontrado com o id: {id}')
else:
    print(f'Falha ao obter dados: {response.status_code}')
```

Teste para Pagamentos

Este código faz uma solicitação GET para uma API Flask local para obter informações sobre um pagamento com um ID especificado pelo utilizador em linha de comandos. Este imprime os detalhes do pagamento se encontrado, caso não encontre, indica que nenhum serviço foi encontrado ou houve uma falha na obtenção de dados.

```
# importa as bibliotecas necessarias
import requests
import sys
import json

# Verifica se o script foi executado com um argumento
if len(sys.argv) != 2:
    print("Uso: python script.py <id>")
    sys.exit(1)

# Verifica se o argumento é uma string
id = sys.argv[1]

# Envia a solicitação GET para o endpoint da API
response = requests.get('http://127.0.0.1:5005/norauto/api/payments')

# Verifica se a solicitação foi bem-sucedida
if response.status_code == 200:
    data = response.json()

    # Verifica se o ID está presente nos dados
    if id in data:
        # Formata os dados em JSON antes de imprimir
```

```
        formatted_data = json.dumps(data[id], indent=4)
        print(formatted_data)
    else:
        print(f'Nenhum item encontrado com o id: {id}')
else:
    print(f'Falha ao obter dados: {response.status_code}')
```

Teste para Clientes

Este código faz uma solicitação GET para uma API Flask local para obter informações sobre um cliente com um ID especificado pelo utilizador em linha de comandos. Este imprime os detalhes do cliente se encontrado, caso não encontre, indica que nenhum serviço foi encontrado ou houve uma falha na obtenção de dados.

```
# importa as bibliotecas necessárias
import requests
import sys
import json

# Verifica se o script foi executado com um argumento
if len(sys.argv) != 2:
    print("Uso: python script.py <id>")
    sys.exit(1)


id = sys.argv[1]

# Envia a solicitação GET para o endpoint da API
response = requests.get('http://127.0.0.1:5005/norauto/api/clients')

# Verifica se a solicitação foi bem-sucedida
if response.status_code == 200:
    data = response.json()

    # Verifica se o ID está presente nos dados
    if id in data:
        # Formata os dados em JSON antes de imprimir
        formatted_data = json.dumps(data[id], indent=4)
        print(formatted_data)
    else:
        print(f'Nenhum item encontrado com o id: {id}')
else:
    print(f'Falha ao obter dados: {response.status_code}')
```

Conclusão

O sistema de gestão para a oficina foi desenvolvido com o objetivo de melhorar a eficiência operacional e a satisfação dos clientes. Com uma arquitetura bem estruturada e funcionalidades avançadas, o sistema proporciona uma administração centralizada e eficiente das atividades da oficina. A integração bem-sucedida com uma interface de utilizador amigável em Tkinter e uma API robusta em Flask oferece uma experiência completa ao utilizador final.

Os objetivos iniciais propostos pelo Professor foram plenamente atendidos, refletindo uma sólida compreensão dos conceitos abordados em sala de aula. Além disso, a inclusão bem-sucedida dos serviços da web como uma tarefa adicional evidencia a capacidade de ampliar o escopo do projeto para incorporar novas funcionalidades, demonstrando uma abordagem proativa e um compromisso com a excelência no desenvolvimento do projeto.

As possíveis melhorias futuras incluem a expansão das funcionalidades, como a introdução de análises avançadas de dados para melhor tomada de decisão, e ainda aprimorar a experiência do utilizador, tornando-a mais intuitiva e agradável.

Este trabalho demonstra não apenas a aplicação prática de conhecimentos de programação em Python, mas também o impacto positivo que uma solução tecnológica bem desenvolvida pode ter no ambiente empresarial.