UNIVERSIDADE DE FORTALEZA
CURSO DE ANÁLISE E DESENVOLVIMENTO DE SISTEMAS
DISCIPLINA: N704 – PROGRAMAÇÃO FUNCIONAL

ATIVIDADE PARCIAL/TRABALHO EM GRUPO

INTEGRANTES DA EQUIPE:

ALUÍSIO RODRIGUES JÚNIOR – 2326179
ELAYNE NASCIMENTO LIMA – 2326596
FRANCISCO EUDES RODRIGUES DA SILVA– 2314695
GILSSILANY VALENTINO CHAVES - 2318460
IGOR MARCELO DE SOUSA FREIRE – 2326293
JOÃO PAULO GOMES DOS SANTOS - 2323778
MARCUS VINICIUS MONTEIRO DA SILVA COSTA – 2326313

Fortaleza, Setembro de 2025

Documento de Especificação do Sistema de Agendamento

1. Descrição Geral do Sistema

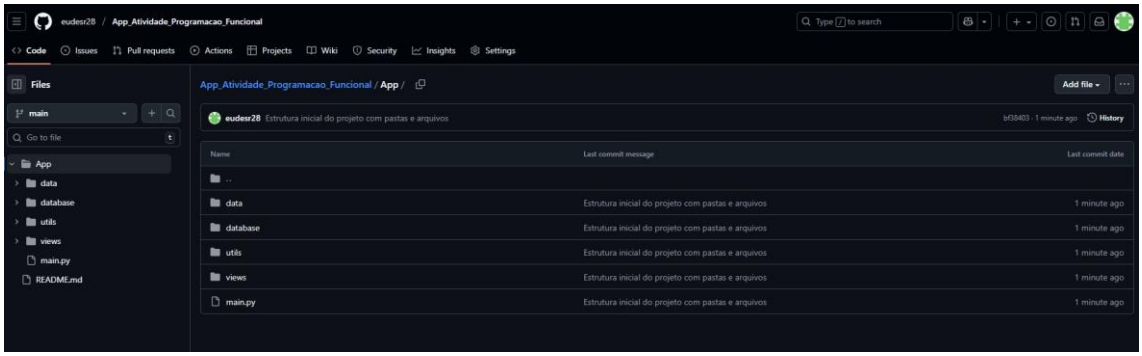
O sistema desenvolvido é um aplicativo desktop em Python utilizando Tkinter para interface gráfica e SQLite para persistência de dados.
Ele possibilita o cadastro de usuários, realização de login seguro, agendamento de serviços, além de permitir que o administrador gerencie usuários e agendamentos.

O aplicativo é voltado para organizações que necessitam de controle de agendamentos de forma simples e acessível.

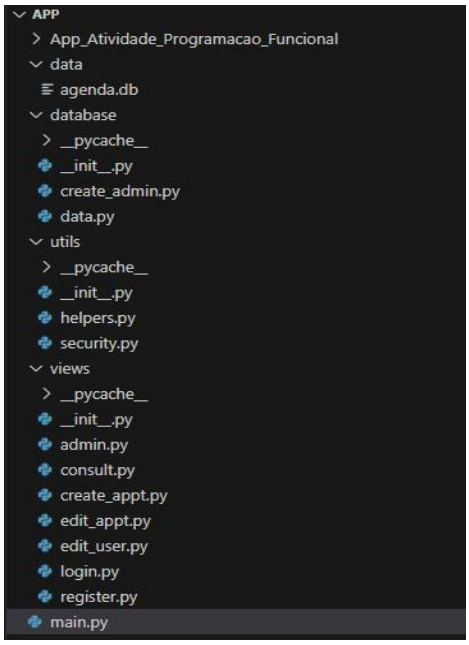
Repositório criado no Github:

Link: https://github.com/eudesr28/App_Atividade_Programacao_Funcional/tree/main

Obs: Todo o conteúdo deste documento está especificado também no README do repositório, bem como o arquivo PDF foi incluso na pasta do projeto no repositório



2. Estrutura de pastas do projeto



3. Funcionalidades do Sistema (por módulo/código)

main.py

- Estrutura principal da aplicação.
- Inicializa o Tkinter, gerencia as diferentes telas (frames) e troca entre elas (LoginFrame, RegisterFrame, ConsultFrame, AdminFrame, etc.).
- Função show_frame recebe parâmetros extras (ex.: user, appt, return_to) para permitir navegação contextualizada.

database/data.py

- Implementa a comunicação com o banco SQLite.
- Funções principais:
- create_tables() → Cria as tabelas users e appointments.

```
def init_db(conn):
cur = conn.cursor()
cur.execute("""
      CREATE TABLE IF NOT EXISTS users (...)
   """)
   cur.execute("""
      CREATE TABLE IF NOT EXISTS appointments (...)
   """)
```

- creat_user() → Insere novo usuário.

```
def create_user(conn, name, email, phone, dob, password, is_admin=0):
ok, msg = validate_password(password)    if not ok:
      raise ValueError(msg)
   salt, h = hash_password(password)
   cur = conn.cursor()
cur.execute("""
      INSERT INTO users (name, email, phone, dob, pwd_salt, pwd_hash, is_admin)
      VALUES (?, ?, ?, ?, ?, ?, ?)
   """, (name.strip(), email.strip().lower(), phone.strip(), dob, salt, h, is_admin))    return
cur.lastrowid
```

- get_user_by_login () → Busca usuário por e-mail ou telefone (usado no login).

```
def get_user_by_login(conn, login, password):
   cur = conn.cursor()
   norm = lambda s: s.strip().lower()   # <- lambda (programação funcional)

   if "@" in login:
      cur.execute("SELECT id, name, email, phone, dob, pwd_salt, pwd_hash, is_admin FROM users WHERE
email = ?",
(norm(login),))
else:
      cur.execute("SELECT id, name, email, phone, dob, pwd_salt, pwd_hash, is_admin FROM users WHERE
phone = ?",
(login.strip(),))
```

- salt + hash_password()→ Garantem a segurança das credenciais.

```
from utils.security import hash_password,
verify_password,
```

- add_appointment(), update_appointment()→ Operações de CRUD para agendamentos.

```
def create_appointment(conn, user_id,
service, appt_date, appt_time):
… def update_appointment(conn, appt_id,
service, appt_date, appt_time):
```

- get_appointment() → Busca agendamento de um usuário.
- def get_appointment(conn, user_id):

- get_all_users() / get_all_appointments() → Usadas pela interface de administração.

views/login.py

- Tela de login.
- Recebe credenciais do usuário, consulta no banco e verifica senha criptografada.

```
    user =
get_user_by_login(login_value,
password_value)       if not
user:
        messagebox.showerror("Erro",
"Usuário ou senha incorretos")          return
```

- Direciona usuário comum para ConsultFrame e administrador para AdminFrame.

```
self.controller.current_user =
user          # admin?          if
user.get("is_admin"):
       self.controller.show_frame("AdminFrame",
user=user)      else:
       self.controller.show_frame("ConsultFrame",
user=user)
```

views/register.py

- Tela de cadastro de usuário.
- Campos: nome, email, telefone, data de nascimento, senha.

```
      labels = ["Nome", "Email", "Telefone", "Data Nascimento
(DD/MM/YYYY)", "Senha"]       self.entries = {}
```

- Insere usuário no banco via create_user.

```
          user_id = create_user(nome, email, telefone, dob, senha)
```

```
        messagebox.showinfo("Sucesso", f"Usuário cadastrado
com sucesso! ID: {user_id}")
self.controller.show_frame("LoginFrame")
```

views/consult.py

- Tela para usuário comum.
- Exibe informações pessoais e agendamentos.

```
    self.title_label.config(text=f"Bem-vindo {user['name']}")


dob_str =
user['dob
']       try:
         dob_fmt = datetime.strptime(dob_str, "%Y-%m-
%d").strftime("%d/%m/%Y")       except Exception:
dob_fmt = dob_str

     ttk.Label(self.info_frame, text=f"Email: {user['email']}").pack()
ttk.Label(self.info_frame, text=f"Telefone: {user['phone']}").pack()
ttk.Label(self.info_frame, text=f"Nascimento: {user['dob']}").pack()
```

- Permite criar ou alterar agendamento existente.

```
      ttk.Button(self.info_frame, text="Criar Agendamento",

                 command=lambda:
self.controller.show_frame("CreateApptFrame",
user=user,return_to="ConsultFrame")).pack(pady=5)
ttk.Button(self.info_frame, text="Alterar Agendamento",
                 command=lambda: self.controller.show_frame("EditApptFrame", user=user,
appt=appt, return_to="ConsultFrame")).pack(pady=5)
```

- Exibe mensagens diferentes caso não exista agendamento:

```
else:
          ttk.Label(self.info_frame, text="Você não possui
                              serviço agendado").pack(pady=5)
```

views/edit_appt.py e views/create_appt.py

- Telas de edição e criação de agendamentos.

```
from database.data import update_appointment…
update_appointment(self.appt['id'], service, date, time)
messagebox.showinfo("Sucesso", "Agendamento atualizado!")
```

```
from database.data import create_appointment…
create_appointment(self.user['id'], service, date, time)
messagebox.showinfo("Sucesso", "Agendamento criado!")
```

- Atualiza ou insere dados no banco.

views/edit_user.py

- Tela de edição de dados do usuário.
- Usada pelo admin para atualizar informações de outros usuários.

```
def update_user(conn, user_id, name, email,
phone, dob):
```

- Implementa retorno dinâmico (return_to) para voltar ao frame correto após edição.

```
update_user(self.user["id"], name, email,
phone, dob)
messagebox.showinfo("Sucesso", "Dados do
usuário atualizados!")
self.controller.show_frame("AdminFrame")
```

views/admin.py

- Tela exclusiva do administrador.
- Exibe lista de usuários e agendamentos

```
@with_db
  def fetch_users(conn):    cur = conn.cursor()
   cur.execute("SELECT id, name, email,
phone, dob FROM users")    return
cur.fetchall()
```

- Permite editar usuários, editar agendamentos e excluir registros.

```
from database.data import get_user_by_id, get_appointment,
with_db def edit_user(self):… def edit_appt(self):… def
delete_user(self):… def delete_appt(self):…
```

4. Requisitos Funcionais (RF)

1. **Cadastro de usuários**
    1. O sistema deve permitir que novos usuários se cadastrem.
    2. Implementação: função add_user() em data.py; interface RegisterFrame.
2. **Login com credenciais seguras**
    1. O sistema deve permitir que usuários façam login.
    2. Implementação: função get_user_by_login() + verificação em LoginFrame.
3. **Segurança de credenciais**

1. O sistema deve armazenar senhas criptografadas.
2. Implementação: salt + hash_password()→ em data.py.

4. **Agendamento de serviços**
    1. O usuário deve poder criar agendamento.
    2. Implementação: função add_appointment() em data.py; tela CreateApptFrame.
5. **Edição de agendamento**
    1. O usuário deve poder alterar data/hora do serviço.
    2. Implementação: função update_appointment() em data.py; tela EditApptFrame.
6. **Cancelamento de agendamento**
    1. O usuário deve poder excluir um agendamento.
    2. Implementação: função delete_appt() em admin.py.
7. **Consulta de informações pessoais**
    1. O usuário deve poder visualizar seus dados cadastrados.
    2. Implementação: tela ConsultFrame.
8. **Gerenciamento de usuários (Admin)**
    1. O administrador deve listar e editar usuários.
    2. Implementação: get_all_users(), EditUserFrame.
9. **Gerenciamento de agendamentos (Admin)**
    1. O administrador deve listar, editar e excluir agendamentos.
    2. Implementação: get_all_appointments(), update_appointment(), delete_appointment().

4. Requisitos Não Funcionais (RNF)

1. **Persistência de dados**

- Implementada com Tkinter e ttk (uso de botões, labels e organização em frames).

2. **Persistência de dados**
    - O sistema deve manter dados salvos em banco SQLite (data.py).
3. **Validação de dados**
    - Datas devem seguir padrão ISO (%Y-%m-%d) e senhas não podem ser armazenadas em texto plano.
4. **Portabilidade**

- Sistema funciona em qualquer máquina com Python 3.x sem necessidade de servidor externo.

5. **Separação de responsabilidades**
    - Código organizado em módulos (views/, database/ e main.py).

6. Funções:

1. **Funções lambda:**

```
# database\data.py norm =
 lambda s: s.strip().lower()
```

- **Arquivo: database/data.py, dentro da função get_user_by_login**
- Serve para normalizar o login.

Outro exemplo:

```
# views\consult.py
command=lambda: self.controller.show_frame("EditApptFrame", user=user, appt=appt,
return_to="ConsultFrame")
```

- Aqui o lambda é usado para passar uma função rápida ao botão do Tkinter.

2. **List comprehension**

No projeto, aparecem em consultas/filtragens, por exemplo:

```
# database\data.py rows
= cur.fetchall()
appointments = [
    {"id": r[0], "user_id": r[1], "service": r[2], "date": r[3], "time": r[4]}
for r in rows
]
```

3. **Closure**

Closure é quando uma função interna usa variáveis da função externa. Um exemplo está no utils/security.py (onde você tem criptografia de senha):

```
def make_password_checker(salt, hash_):     def
check(password):         return
verify_password(password, salt, hash_)     return
check
```

Aqui:

- check é uma função interna.
- Ela fecha sobre (captura) as variáveis salt e hash_ da função externa.
- Quando retornamos check, temos uma closure.

4. **Função de alta ordem**

Uma função de alta ordem é aquela que:

- Recebe outra função como argumento ou
- Retorna outra função.

Dois exemplos claros:

1. O decorador @with_db:

```
def with_db(func):    def
wrapper(*args, **kwargs):
      conn = get_connection()
try
return func(conn, *args, **kwargs)
finally
          conn.close()     return wrapper
```

Arquivo: database/data.py.

- with_db recebe uma função (func) como parâmetro → alta ordem.

2. O exemplo de closure que foi citado anteriormente (make_password_checker) também é alta ordem, porque retorna uma função (check).

7. Casos de Testes

CT01 – Cadastro de Usuário com dados válidos

- **Objetivo**: Verificar se o sistema cadastra um novo usuário corretamente.
- **Entradas**: Nome, Email, Telefone, Data de Nascimento (DD/MM/YYYY), Senha válida.
- **Resultado Esperado**: Mensagem de sucesso e redirecionamento para tela de login.

CT02 – Cadastro de Usuário com senha inválida

- **Objetivo**: Validar a rejeição de senhas que não atendem aos critérios.
- **Entradas**: Senha sem caractere especial ou número.
- **Resultado Esperado**: Mensagem de erro informando os critérios não atendidos.

CT03 – Login com credenciais corretas

- **Objetivo**: Verificar se o login funciona com dados válidos.
- **Entradas**: Email ou telefone + senha correta.
- **Resultado Esperado**: Acesso ao sistema e redirecionamento para tela apropriada (Admin ou Consult).

CT04 – Login com credenciais incorretas

- **Objetivo**: Validar o tratamento de erro no login.

- **Entradas**: Email/telefone inexistente ou senha incorreta.
- **Resultado Esperado**: Mensagem de erro “Usuário ou senha incorretos”.

CT05 – Criação de agendamento

- **Objetivo**: Verificar se o usuário consegue agendar um serviço.
- **Entradas**: Serviço, Data (DD-MM-YYYY), Hora (HH:MM).
- **Resultado Esperado**: Mensagem de sucesso e exibição do agendamento na tela ConsultFrame.

CT06 – Edição de agendamento existente

- **Objetivo**: Validar a alteração de dados de um agendamento.
- **Entradas**: Novo serviço, data e hora.
- **Resultado Esperado**: Mensagem de sucesso e atualização visível na tela.

CT07 – Exclusão de agendamento

- **Objetivo**: Verificar se o administrador consegue excluir um agendamento.
- **Entradas**: Seleção de usuário com agendamento.
- **Resultado Esperado**: Mensagem de sucesso e remoção do registro.

CT08 – Edição de dados do usuário

- **Objetivo**: Validar a edição de dados cadastrais por parte do administrador.
- **Entradas**: Nome, Email, Telefone, Data de Nascimento.
- **Resultado Esperado**: Mensagem de sucesso e dados atualizados.

CT09 – Exclusão de usuário

- **Objetivo**: Verificar se o administrador consegue excluir um usuário.
- **Entradas**: Seleção de usuário.
- **Resultado Esperado**: Mensagem de sucesso e remoção do usuário e seus agendamentos.

CT10 – Consulta de informações pessoais

- **Objetivo**: Validar se o usuário visualiza corretamente seus dados e agendamento.
- **Entradas**: Login válido.
- **Resultado Esperado**: Exibição de nome, email, telefone, nascimento e agendamento.

8. Responsabilidades por Colaborador

Durante o desenvolvimento do sistema de agendamento, a equipe dividiu as tarefas levando em consideração as habilidades e disponibilidade de cada integrante. Abaixo está a descrição das responsabilidades assumidas por cada colaborador:

Francisco Eudes Rodrigues

Responsável pela estrutura principal da aplicação (main.py) e pela interface administrativa (admin.py). Implementou a lógica de navegação entre telas e os métodos que permitem ao administrador visualizar, editar e excluir usuários e agendamentos.

Elayne Nascimento

Ficou encarregada da camada de dados (data.py) e da tela de edição de usuários (edit_user.py). Implementou as funções de acesso ao banco SQLite, como criação de usuários, agendamentos e autenticação, além da lógica de atualização dos dados cadastrais.

Gilssilany Valentino

Desenvolveu o script de criação do usuário administrador (creat_admin.py) e a tela de cadastro de novos usuários (register.py). Foi responsável por validar os dados de entrada e garantir que o processo de registro estivesse integrado ao banco de dados.

Igor Marcelo Freire

Trabalhou na tela de consulta de informações (consult.py) e na edição de agendamentos (edit_appt.py). Implementou a exibição dos dados do usuário e a lógica que permite alterar data, hora e tipo de serviço agendado.

Aluísio Rodrigues

Ficou responsável pelos módulos auxiliares (helpers.py) e de segurança (securit.py). Implementou funções de apoio e a criptografia de senhas, garantindo que os dados sensíveis fossem armazenados de forma segura.

Marcus Vinícius Monteiro

Desenvolveu a tela de criação de agendamentos (creat_appt.py) e a tela de login (login.py). Implementou a lógica de autenticação e a interface que permite ao usuário agendar serviços com data e hora específicas.

Com essa divisão, a equipe inteira participou da implementação do código.