



DS

<https://divisaosaude.herokuapp.com/>

# Relatório Final

## CES-22 - Programação Orientada a Objeto

Adriano Soares, Matheus Vidal e Pedro Alves

Julho, 2019

Turma COMP 21

Prof.<sup>o</sup> Yano

Instituto Tecnológico de Aeronáutica (ITA)

São José dos Campos, São Paulo, Brasil.

{sadrianorod, matheusvidaldemenezes, alvesouza.pedro97}@gmail.com

git: <https://github.com/vidalmatheus/DS.com>

## I. Motivação

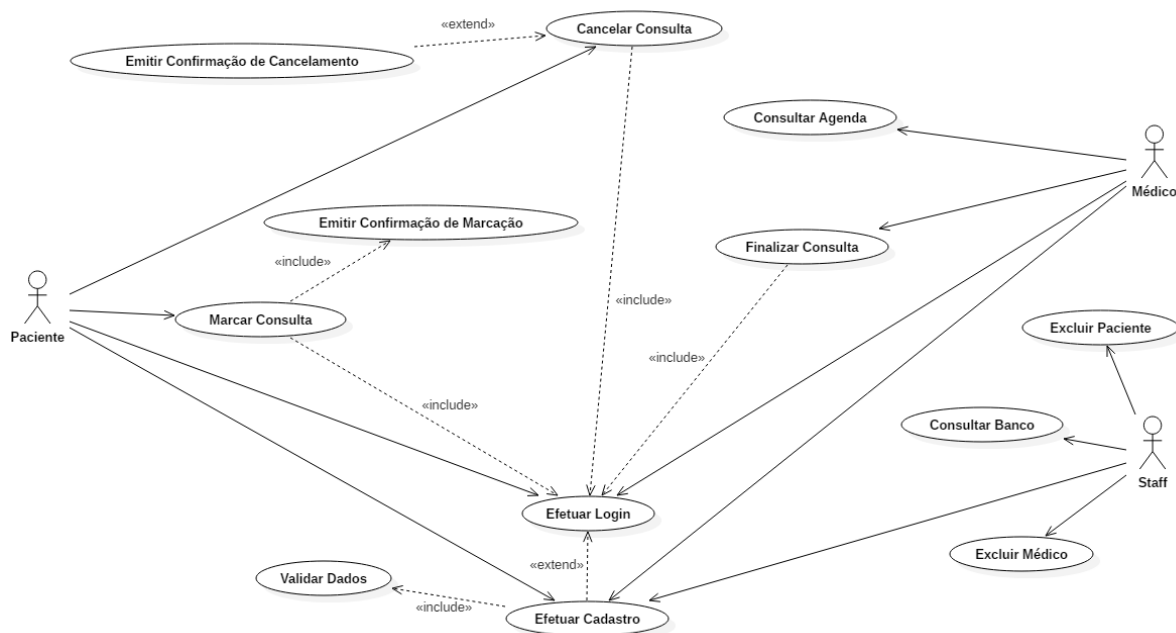
O grupo Adriano Soares Rodrigues, Matheus Vidal de Menezes e Pedro Alves de Souza Neto decidiu construir uma *web application* de marcação de consultas para a Divisão de Saúde (DS) do Departamento de Ciência e Tecnologia Aeroespacial (DCTA).

A principal motivação foi o fato da DS não possuir um sistema automatizado de marcação de consultas, fazendo com que um morador militar ou aluno do ITA tenha que se dirigir até a DS apenas para marcar sua consulta.

## II. Análise de Requisitos

Primeiramente, tivemos dificuldades de termos uma reunião presencial com os responsáveis adequados na DS, devido a desencontro de horários. Assim, o sistema teve seu desenvolvimento iniciado com base na experiência de vários alunos do ITA que passaram informações relevantes sobre o processo de marcação de consultas na Divisão de Saúde.

Almejando sempre boas práticas de Engenharia de Software, incorporamos diagramas da UML para nos ajudar a modelar tal aplicação real. Tais diagramas foram sofrendo modificações ao longo de mais informações consistentes que se obtinham. **Por isso, os diagramas presentes neste relatório estão um pouco diferentes daqueles apresentados em aula.** Segue o Diagrama de Caso de Uso.



**Figura 1.** Diagrama de Casos de Uso do sistema.

Para entender melhor como se dará o processo, segue os principais Fluxos de Evento, tanto do paciente, quanto do médico no sistema:

### <Marcar Consulta>

**Atores:** Paciente e Funcionário

**Pré-Condição:** O paciente não tem cadastro ainda.

#### **Fluxo de Eventos Principal:**

1. O paciente seleciona Cadastro.
2. O paciente fornece dados: CPF, senha, SARAM, nome, data de nascimento, sexo, endereço, telefone, e-mail, posto/graduação militar ou caso especial (aluno do ITA).
3. O funcionário valida o número SARAM, confirmando o cadastro do paciente.
4. O paciente recebe confirmação por e-mail que o cadastro foi confirmado.

### <Marcar Consulta>

**Atores:** Paciente

**Pré-Condição:** Paciente já cadastrado.

#### **Fluxo de Eventos Principal:**

1. O paciente efetua *login* com CPF e senha.
2. O paciente seleciona Marcar Consulta.
3. O paciente escolhe o dia da semana que quer marcar consulta.
4. O paciente escolhe a especialidade do médico.
5. O paciente escolhe o médico.
6. O paciente escolhe o horário da consulta de 30 min, dentre os 15 slots diários.
7. O paciente confirma seus dados.
8. O paciente seleciona Enviar.
9. O sistema exibe mensagem de operação foi concluída.
10. O sistema envia uma confirmação por e-mail.

### <Cancelar Consulta>

**Atores:** Paciente, Médico e Funcionário

**Pré-Condição:** O paciente deve estar com consulta marcada e ter efetuado login.

#### **Fluxo de Eventos Principal:**

1. O sistema exibe as consultas marcadas.
2. O paciente seleciona Cancelar Consulta.

### <Consultar Agenda>

**Atores:** Médico

**Pré-Condição:** O médico deve estar cadastrado.

#### **Fluxo de Eventos Principal:**

1. O sistema exibe a agenda semanal
2. O médico seleciona o dia desejado.
3. O médico consulta aos slots de consultas marcadas ou livres do dia.

### <Finalizar Consulta>

**Atores:** Médico

**Pré-Condição:** O médico deve estar cadastrado e ter terminado uma consulta.

**Fluxo de Eventos Principal:**

1. O sistema exibe os slots de consultas marcas no dia.
2. O médico seleciona Finalizar Consulta na consulta que já foi terminada.
3. O sistema retira o slot da tela.
4. O sistema torna o horário disponível de novo para marcação de consulta.

### <Efetuar Cadastro>

**Atores:** Médico e Funcionário

**Pré-Condição:** O médico não tem cadastro ainda.

**Fluxo de Eventos Principal:**

1. O médico seleciona Cadastro.
2. O médico fornece dados: CPF, senha, SARAM, nome, especialidade, posto/graduação militar.
3. O funcionário valida o número SARAM, confirmando o cadastro do médico.
4. O médico recebe confirmação por e-mail que o cadastro foi confirmado.

### <Consultar Banco>

**Atores:** Funcionário

**Fluxo de Eventos Principal:**

1. O funcionário faz login como administrador.
2. O sistema exibe todas as consultas marcadas da semana.

### <Excluir Médico>

**Atores:** Funcionário

**Fluxo de Eventos Principal:**

1. O funcionário faz login como administrador.
2. O funcionário seleciona Excluir Médico.
3. O sistema exibe todos os médicos.
4. O funcionário exclui o médico desejado.

### <Excluir Paciente>

**Atores:** Funcionário

**Fluxo de Eventos Principal:**

1. O funcionário faz login como administrador.
2. O funcionário seleciona Excluir Paciente.
3. O sistema exibe todos os pacientes.

4. O funcionário exclui o paciente desejado.

### III. Divisão de Trabalho

A equipe foi subdividida em três áreas principais: *Front-End* (*Interação da Aplicação Web com o usuário*), *Back-End* (Banco de Dados) e *Back-End* (Integração), com seus respectivos líderes, Adriano Soares, Matheus Vidal e Pedro Alves. O motivo disso foi o fato de dar mais autonomia para cada integrante, tal como uma situação real de trabalho. Essa divisão, porém, não foi excludente, a fim de que cada integrante conseguiu dialogar e até mesmo ajudar na codificação dos demais.

### IV. Tecnologias Utilizadas

<i>Front-end</i> : HTML5, CSS3, Bootstrap, JavaScript	Diagramas: StarUML
<i>Back-end</i> (Banco de Dados): PostgreSQL, pgAdmin 4	IDE: vscode
<i>Back-end</i> (Integração): Flask (python)	

### V. *Front-End*: Adriano Soares

Diante da responsabilidade de toda a interação usuário com o website, foi necessário o estudo das principais ferramentas de *Front-End*: HTML5, CSS3, *Bootstrap* além para uma interação inicial o JavaScript que gera certa dinâmica que ainda se encontra no campo do *Front-End*.

É importante salientar que o *Bootstrap* é um *framework* que agiliza a criação do website, uma vez que já apresenta padrões de materiais para utilização, sendo de extrema importância para a criação do DS.com uma vez que o tempo de projeto é curto, então necessitou-se otimizar o tempo utilizando padrões já disponibilizados.

Diante de todas essas ferramentas, o site DS.com tomou forma e todas as metas das páginas HTML necessárias foram criadas assim toda a interface tanto do paciente como do médico foram feitas: página inicial do DS.com, de cadastro, home do paciente onde ele pode visualizar suas consultas ou mudar dados cadastrais, e finalmente a página de visualização do médico que apenas visualiza seus horários marcados.

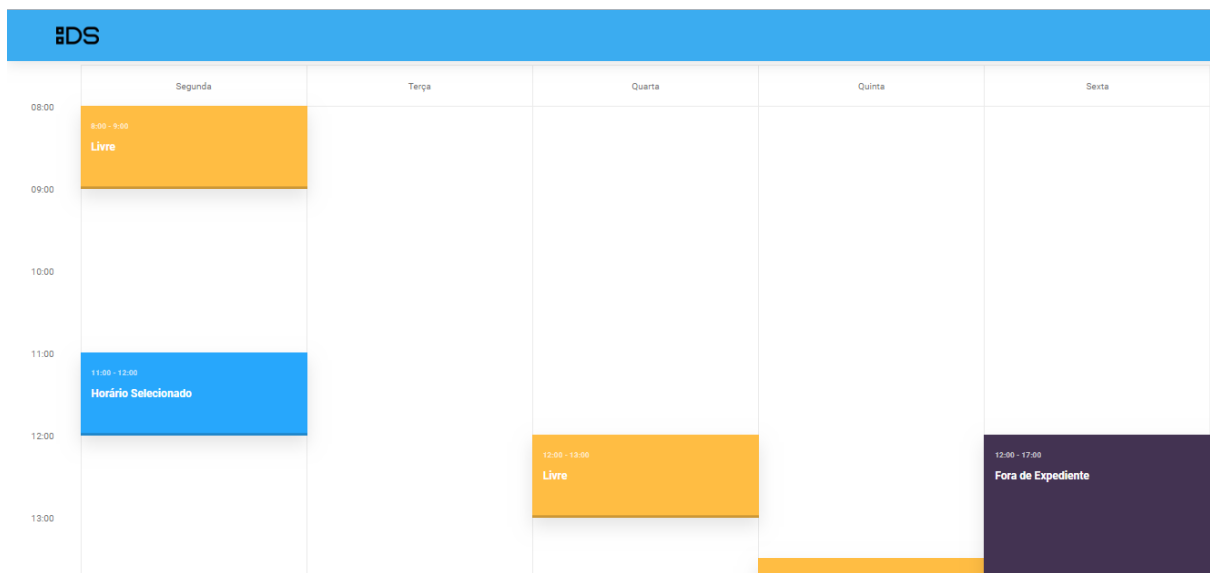
Nesse projeto o maior desafio foi a criação da agenda semanal que aparece os horários livres de cada médico, assim foi criado as *tags* HTML e com CSS3 foi feita a agenda, além disso pelo front implementei certa lógica onde ao paciente clicar em um horário livre a caixa muda sua cor e seu nome de “Livre” para “Horário Selecionado” e não consegue selecionar outro a não ser que ele clique novamente e desmarque o horário dele. Após selecionado o horário, o paciente

possui um campo para escrever algo caso queira e para finalizar ele clica no botão Enviar para finalmente marcar sua consulta.

Assim, foi criado o script `carregaBanco()` responsável por inserir os horários na agenda.

```
function carregaBanco(){  
    for(let i =0; i<horarios.length;i++){  
        let diaDaSemana = document.getElementById(dias[i]);  
        horarios[i].forEach((element)=>{  
            let novo_li = document.createElement('li');  
            novo_li.classList.add("cd-schedule__event", "horario_livre");  
  
            diaDaSemana.appendChild(novo_li);  
            novo_li.innerHTML = "<a data-start="+element.data_inicio+" data-end="+element.data_fim+" data-event=event-4> <div class=cd-schedule__name>Livre</div></a>";  
        });  
    }  
};
```

**Figura 2.** Função criada em JavaScript que carrega os dados da agenda para preencher os horários na agenda, a lógica é a mesma tanto para agenda dos médicos quanto para a dos pacientes, o que difere é a informação que ele lê, para os médicos são seus horários marcados, para os pacientes os horários livres.



**Figura 3.** Imagem da agenda na home do paciente que detalha os horários livres do médico (cor amarelo) e o horário selecionado pelo paciente (cor azul).

```

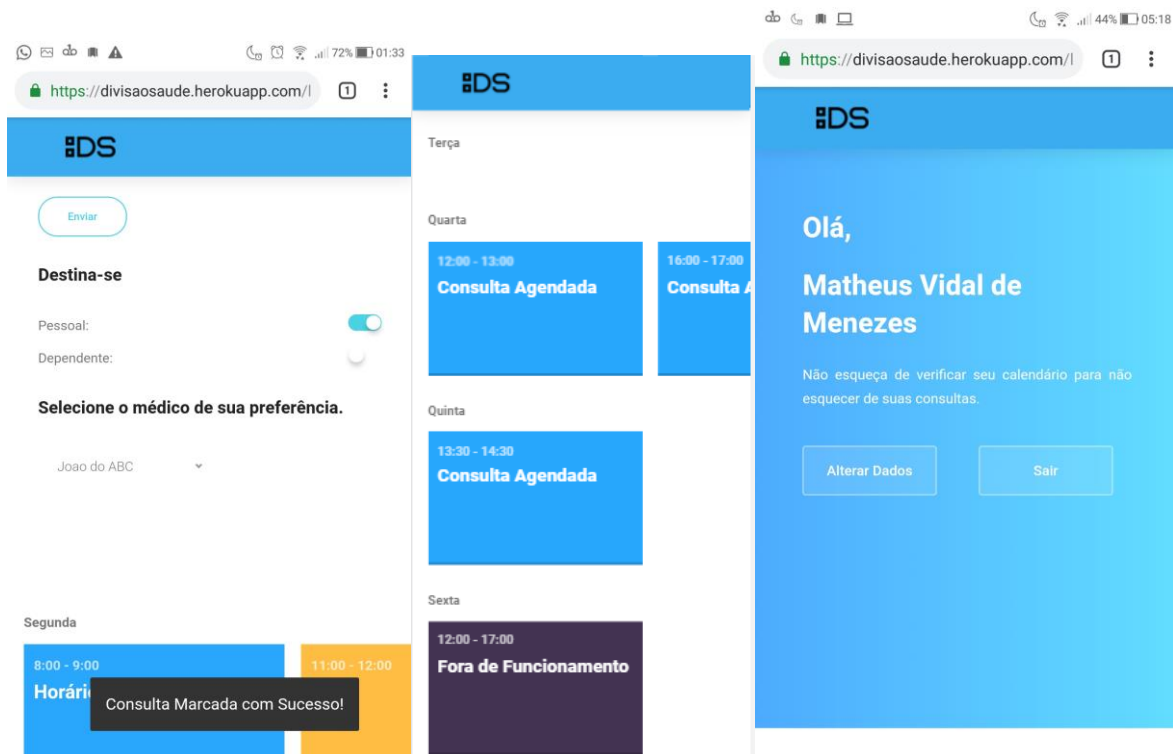
horariosLivres.forEach(function(horario){
  horario.addEventListener("click",function(event){
    event.preventDefault();
    if(!escolheuHorario){
      aux = this.getElementsByTagName('a')[0].getAttributeNode('data-event').value = "event-1";
      escolheuHorario = true;
      this.classList.add("Escolhido");
      this.getElementsByTagName('div')[0].textContent = "Horário Selecionado";
      console.log(this.getElementsByTagName('div')[0].textContent);
    }else if(this.classList.contains("Escolhido")){
      aux = this.getElementsByTagName('a')[0].getAttributeNode('data-event').value = "event-4";
      this.classList.remove("Escolhido");
      this.getElementsByTagName('div')[0].textContent = "Livre";
      escolheuHorario = false;
    }
  });
});

```

**Figura 4.** Script que cria a lógica de preenchimento de horários livres e da unicidade na escolha do horário.

**Figura 5.** Home do paciente para marcar consultas, escolhendo a especialidade o nome do médico e para quem se destina a consulta, ao militar ou algum dependente seu.

Outro desafio por parte do *Front-End* foi deixar a aplicação responsiva a fim de que tanto o médico quanto o paciente tivessem a opção de acessar suas consultas tanto pelo computador quanto por aparelhos móveis como celulares ou tablets.



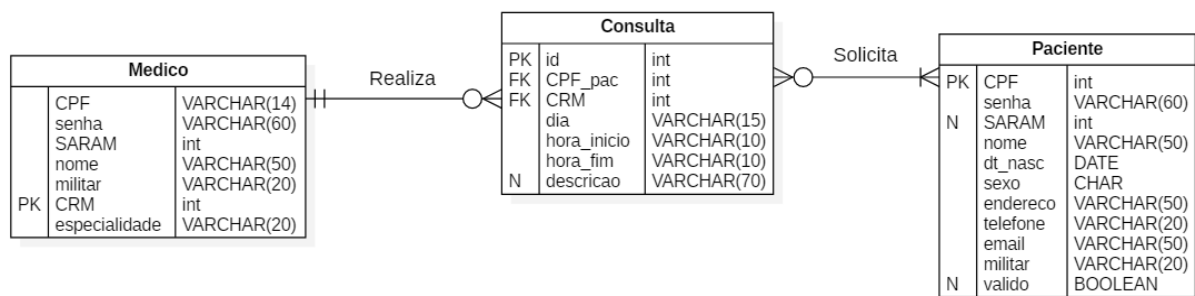
**Figura 6.** Acesso via celular da área de consultas dos médicos, ao deslizar os blocos ele consegue visualizar suas consultas agendadas no dia.

## VI. *Back-End:* Matheus Vidal

Como uma aplicação real, trouxe ao grupo conhecimentos e boas práticas de Engenharia de Software, utilizando a UML. Fiquei responsável principalmente pelo Modelo Conceitual do projeto, Análise de Requisitos (Diagramas de Casos de Uso e Fluxo de Eventos) e Modelagem dos Dados (criação do Banco de Dados).

Quanto a Análise de Requisitos, está bem descrita no tópico II.

A fim de modelar o sistema em mais baixo nível, a partir do Diagrama de Casos de Uso, foi montado o Diagrama de Entidade-Relacionamento do sistema:



**Figura 7.** Diagrama de Entidade-Relacionamento do sistema.



Note o uso correto das cardinalidades que originaram a tabela Consulta com as chaves estrangeiras da tabela Médico e Paciente:

- a) “1 médico realiza nenhuma ou muitas consultas.”
- b) “1 consulta é realizada por apenas 1 médico.”
- c) “1 paciente solicita nenhuma ou muitas consultas.”
- d) “1 consulta é solicitada por, no mínimo, 1 e, no máximo, muitos pacientes.”

Além disso, para que os usuários logados sejam reconhecidos em “nuvem” foi implementada a tabela Logado que guarda o CPF do médico ou paciente logado na aplicação. Ainda, ela guarda um valor *session\_hash* que funciona sendo uma criptografia, utilizando o *bcrypt*, para identificar o *cookie* da sessão, *i.e.*, de onde o usuário está efetuando login, de uma mesma máquina ou outra. Isso foi uma estratégia adotada vista como boa prática e a fim de evitar conflitos com variáveis locais em cada sessão de um usuário (observamos este conflito, quando a aplicação estava hospedada no Heroku). A tabela Logado “independente” do banco é vista como abaixo:

Logado		
PK	CPF session_hash	VARCHAR(14) VARCHAR(60)

**Figura 8.** Tabela Logado para o controle de acesso dos usuários do sistema.

Assim, foi gerado o script em .sql, para criar o Banco de Dados “ds”. O script é mostrado abaixo:

```
-- Database: ds
-- DROP DATABASE ds;
CREATE DATABASE ds
WITH
OWNER = postgres
ENCODING = 'UTF8'
LC_COLLATE = 'Portuguese_Brazil.1252'
LC_CTYPE = 'Portuguese_Brazil.1252'
TABLESPACE = pg_default
CONNECTION LIMIT = -1;

DROP TABLE IF EXISTS Medico CASCADE;
DROP TABLE IF EXISTS Paciente CASCADE;
DROP TABLE IF EXISTS Consulta CASCADE;
DROP TABLE IF EXISTS Logado CASCADE;
```

```
-- -----  
-- Table `ds`.`medico`  
-- -----
```

```
DROP TABLE IF EXISTS "ds".medico ;
```

```
CREATE TABLE Medico (  
    CPF varchar(14) unique NOT NULL,  
    senha varchar NOT NULL,  
    SARAM int unique NOT NULL,  
    Nome VARCHAR(50) NOT NULL,  
    militar VARCHAR(20) NOT NULL,  
    CRM int unique NOT NULL,  
    Especialidade VARCHAR(20) NOT NULL,  
    PRIMARY KEY (CRM)  
);
```

```
-- -----  
-- Table `ds`.`paciente`  
-- -----
```

```
DROP TABLE IF EXISTS "ds".paciente ;
```

```
CREATE TABLE Paciente (  
    CPF varchar(14) unique NOT NULL,  
    senha varchar NOT NULL,  
    SARAM int unique,  
    Nome VARCHAR(50) NOT NULL,  
    dt_nasc DATE NOT NULL,  
    sexo CHAR NOT NULL,  
    endereco VARCHAR(50) NOT NULL,  
    telefone VARCHAR(20) NOT NULL,  
    email VARCHAR(50) NOT NULL,  
    militar VARCHAR(20) NOT NULL,  
    valido BOOLEAN,  
    PRIMARY KEY (CPF)  
);
```

```
-- -----  
-- Table `ds`.`consulta`  
-- -----
```

```
DROP TABLE IF EXISTS "ds".consulta ;
```

```
CREATE TABLE Consulta (  
    id SERIAL,  
    CPF_pac varchar(14) unique NOT NULL,  
    CRM int unique NOT NULL,  
    dia varchar(15) NOT NULL,  
    hora_inicio varchar(10) NOT NULL,  
    hora_fim varchar(10) NOT NULL,  
    descricao varchar(70),  
    PRIMARY KEY (id)  
);
```

```
-- -----  
-- Table `ds`.`logado`  
-- -----
```

```
DROP TABLE IF EXISTS "ds".logado ;
```

```
CREATE TABLE Logado (  
    CPF varchar(14) unique NOT NULL,  
    session_hash varchar(60) unique NOT NULL,  
    PRIMARY KEY (CPF)  
);
```

```
ALTER TABLE Consulta ADD FOREIGN KEY (CRM) REFERENCES Medico(CRM);  
ALTER TABLE Consulta ADD FOREIGN KEY (CPF_pac) REFERENCES  
Paciente(CPF);
```

Não me restringi, porém, somente ao Banco de Dados, implementei tanto rotas em *Back-End*, como as de *register.py* e *changerRegister.py*, quanto, desenvolvimento do *Front-End*, como a proteção de dados indevidos nessas rotas por meio de *alerts* feitos com JavaScript, CSS e HTML.

```

<!--Função de Máscaras-->
<script>
    function formatar(mascara, documento) {
        var i = documento.value.length;
        var saida = mascara.substring(0, 1);
        var texto = mascara.substring(i)
        if (texto.substring(0, 1) != saida) {
            documento.value += texto.substring(0, 1);
        }
    }

    function validate() {
        var x = document.forms["register-form"]["military"].value;
        if (x != "alunoITA" && x != "Marechal-do-Ar" && x != "Tenente-Brigadeiro-do-Ar" && x != "Major-Brigadeiro-do-Ar" && x != "Brigadeiro" &&
            x != "Coronel" && x != "Tenente-Coronel" && x != "Major" && x != "Capitão" && x != "Primeiro Tenente" &&
            x != "Segundo Tenente" && x != "Aspirante" && x != "Suboficial" && x != "Primeiro Sargento" && x != "Segundo Sargento" &&
            x != "Terceiro Sargento" && x != "Cabo" && x != "Taifeiro-Mor" && x != "Soldado Primeira classe" && x != "Soldado Segunda Classe" &&
            x != "Taifeiro Segunda Classe") {
            alert("Patente '" + x + "' INVÁLIDA!");
            return false;
        }
    }
}
</script>

```

Figura 9. Funções JavaScript para tratamento de dados de CPF e “Posto/Graduação Militar”.

```

# register
@register_api.route('/register', methods=['GET', 'POST'])
def register():
    global usersDataOnline
    if request.method == 'POST':
        # Fetch form data
        userDetails = request.form
        cpf = userDetails['cpf']
        psd = userDetails['psd']
        saram = userDetails['saram']
        name = userDetails['name']
        birth_date = userDetails['birth_date']
        sex = userDetails['sex']
        adress = userDetails['adress']
        phone = userDetails['phone']
        email = userDetails['email']
        military = userDetails['military']
        #cursor
        cur = connectionData.getConnector().cursor()
        #print(cpf + " " + psd + " " + saram + " " + name + " " + birth_date + " " + sex + " " + adress + " " + phone +
        hashed = bcrypt.hashpw(psd.encode(),bcrypt.gensalt(12))
        hashedDecoded = hashed.decode('utf-8')
        cur.execute("INSERT INTO paciente VALUES(%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s)",(cpf,hashed
        #commit the transcation
        connectionData.getConnector().commit()

```

Figura 10. Rota de cadastro de paciente.

Outra implementação que vale ressaltar foi a de carregamento automatizado: quando o paciente escolhe a especialidade do médico, a lista de médicos é definida conforme sua escolha prévia, de modo que o banco está sendo acessado a cada seleção:

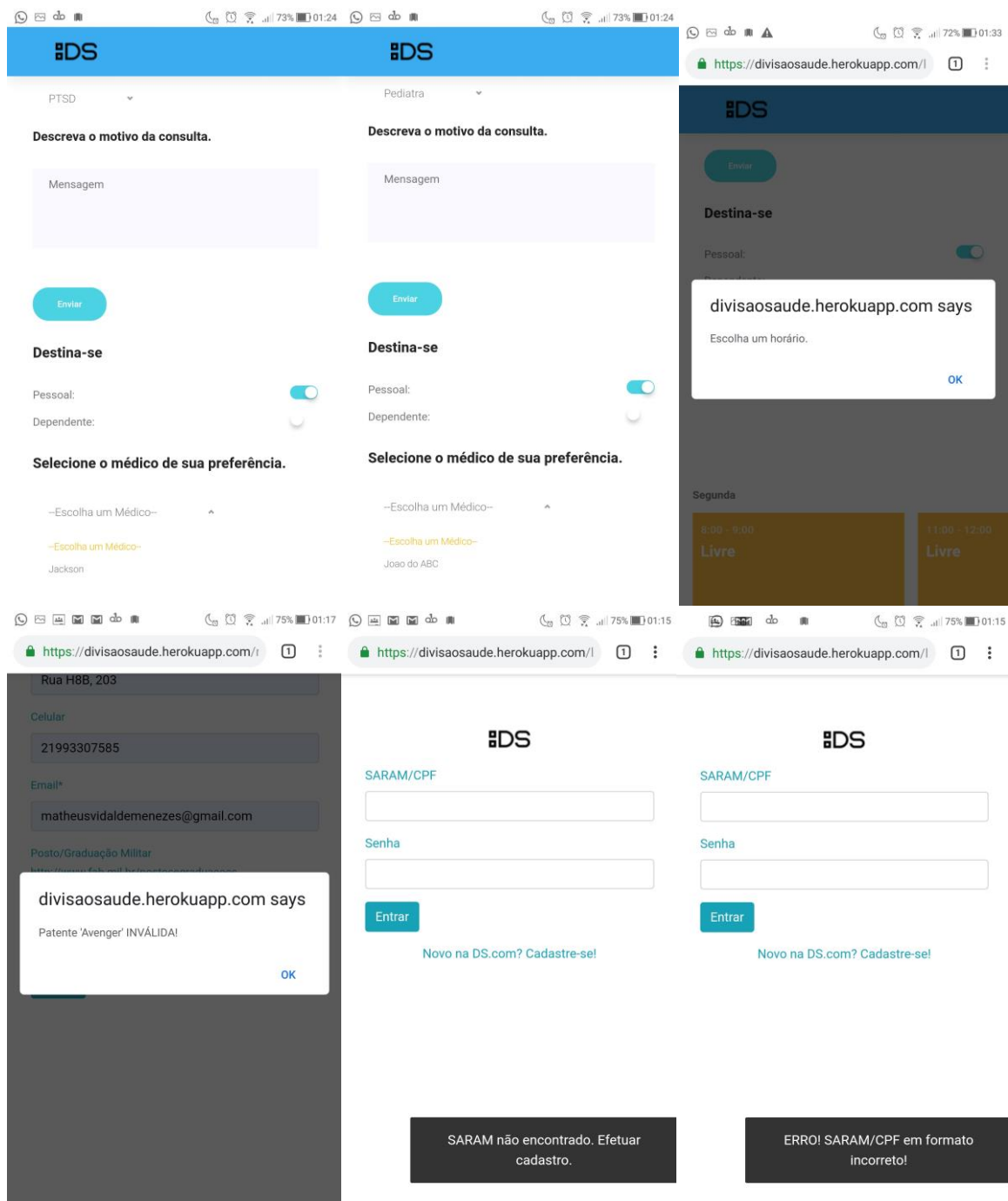
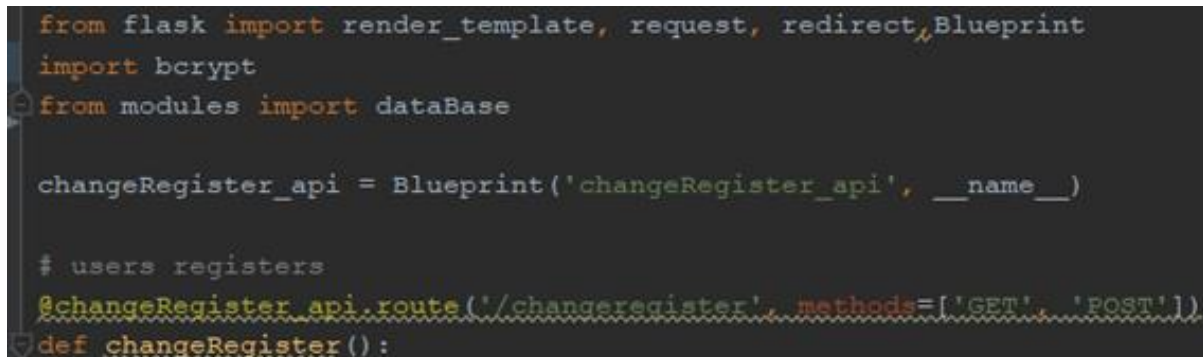


Figura 11. Telas de acesso por celular exemplificando a automação e *alerts*.

## VII. *Back-end*: Pedro Alves

- *Routes*

As routes servem para o *back-end* saber o que tem que fazer quando o usuário está em determinado endereço do site, no começo elas estavam todas determinadas dentro do “*app.py*”, mas isso estava fadado a sofrer com conflitos na hora de se fazer merges pelo GitHub logo, eu armazenei todos os routes em uma pasta específica chamada “*routes*”, onde eles seriam registrados através de *blueprints*, vide imagem abaixo:



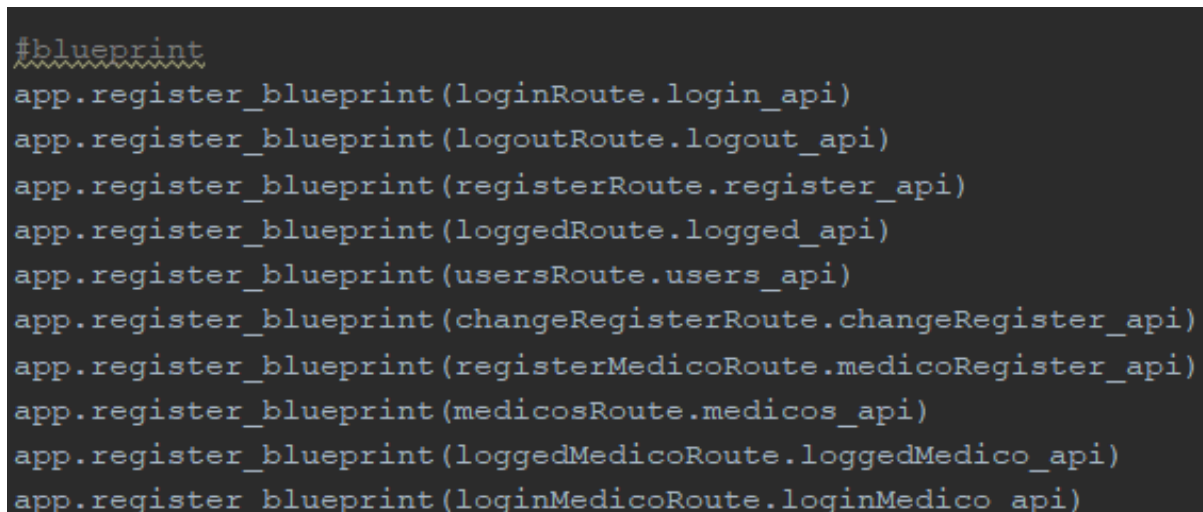
```
from flask import render_template, request, redirect, Blueprint
import bcrypt
from modules import DataBase

changeRegister_api = Blueprint('changeRegister_api', __name__)

# users registers
@changeRegister_api.route('/changepassword', methods=['GET', 'POST'])
def changeRegister():
```

Figura 12. Mostrando como se cria uma route usando blueprint

Abaixo, mostra como se conecta uma *route* de outro script com o app:



```
#blueprint
app.register_blueprint(loginRoute.login_api)
app.register_blueprint(logoutRoute.logout_api)
app.register_blueprint(registerRoute.register_api)
app.register_blueprint(loggedRoute.logged_api)
app.register_blueprint(usersRoute.users_api)
app.register_blueprint(changeRegisterRoute.changeRegister_api)
app.register_blueprint(registerMedicoRoute.medicoRegister_api)
app.register_blueprint(medicosRoute.medicos_api)
app.register_blueprint(loggedMedicoRoute.loggedMedico_api)
app.register_blueprint(loginMedicoRoute.loginMedico_api)
```

Figura 13. Mostrando como se conecta uma *route* ao *app* usando *blueprint*. Obs.: cada linha é para uma *route* diferente.

Assim fica mais fácil dos usuários alterarem *routes* diferentes sem gerar conflito e mais organizado.

- *Password*

Para a proteção dos dados dos usuários foi utilizado o método *hashpw* do module **bcrypt** que é uma função injetora, mas não possui inversa, logo para poder ser decodificado seria necessário utilizar força bruta para descobrir a senha do usuário a partir da *hash*. Uma vantagem

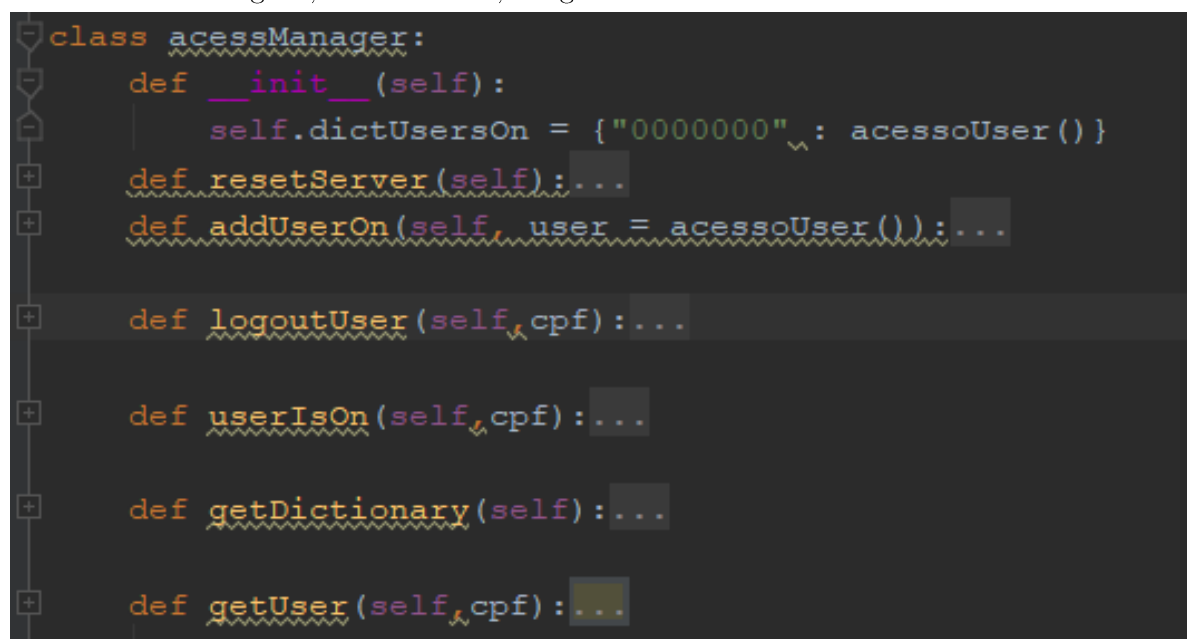
da **hashpw** é que ela é umas das mais lentas, ou seja, caso alguém consiga os *hashs* das senhas dos usuários aumentaria significativamente o tempo para descobrir as senhas, mas para caso de uma tentativa (acesso a partir do login) seria um tempo muito pequeno.

- *Sessions*

*Sessions* é o sistema de “cookies” do Flask, que recebe e armazena variáveis dentro do browser do computador do usuário em forma de dicionário.

## 1. Primeiramente Versão

Eu desenvolvi eles a partir de que o usuário não pudesse ser acessado por dois computadores ao mesmo tempo (por motivos de segurança), durante o acesso das *routes* era verificado se existia o `session['user']` para garantir que o usuário ficasse logado. Em caso de login verificava a partir de uma variável global que era um objeto da classe (`acessManager`) que verifica se o usuário está logado, caso estivesse, o login seria evitado.

A screenshot of a code editor showing the definition of a Python class named `acessManager`. The class has several methods: `__init__` which initializes a dictionary `dictUsersOn` with a single entry for CPF "0000000" pointing to `acessoUser()`; `resetServer`; `addUserOn` which adds a new user to the dictionary; `logoutUser` which removes a user; `userIsOn` which checks if a user is logged in; `getDictionary` which returns the dictionary; and `getUser` which returns the user object for a given CPF. The code is written in a dark-themed editor with syntax highlighting.

```
class acessManager:
    def __init__(self):
        self.dictUsersOn = {"0000000": acessoUser()}
    def resetServer(self):...
    def addUserOn(self, user = acessoUser()):...
    def logoutUser(self, cpf):...
    def userIsOn(self, cpf):...
    def getDictionary(self):...
    def getUser(self, cpf):...
```

**Figura 14.** Mostrando os métodos e a variável contida num `acessManager`, espero que os nomes dos métodos sejam autoexplicativos, as ‘keys’ do dicionário é o cpf do respectivo user.

Mas esse método tinha problemas primeiro além de má prática de programação variáveis globais não funcionam no *Back-End* como funcionariam em um programa em desktop, logo o que acontecia era que o valor dessa variável de `acessManager` não se mantinha entre as *routes*, uma hora ela tinha em si todos os usuários que estivessem logados outra hora ele perdia na lista (dicionário mais precisamente) o usuário logo o usuário era retirado automaticamente da sessão, mas ao tentar o login novamente o usuário voltava para a lista de usuários logados, logo ele não poderia fazer o login novamente fazendo ele ficar preso do “lado de fora”.

## 2. Segunda Versão

Eu queria ter evitado variáveis globais desde o início e operar os *routes* através de máquinas de estado, mas isso não era possível. Logo fiz uma mudança prática de que ao invés do computador não poder acessar um usuário caso este usuário já esteja logado fiz com que o computador anterior fizesse logout assim fazia mais sentido para um caso real.

Para resolver o problema da variável global foi feito com a lista de usuários logados foi feito uma tabela no banco de dados que armazena os usuários que estão logados, onde ela armazena o cpf do usuário e o *session\_hash* da última vez que o usuário “logou” pelo “login”.

```
if passwordCorrect:
    print("PASSWORD IS CORRECT")
    session["userName"] = userData.getName()
    session["userID"] = userData.getCPF()
    session["loginHash"] = bcrypt.hashpw((userData.getName()+userData.getCPF()+str(datetime.datetime.now()))).encode(),_bcrypt.gensalt(12))
    session["userType"] = "p"
```

**Figura 15.** Mostrando como é gerado os cookies de um usuário, caso este usuário já esteja no banco de dados logado ele será alterado para se alocar este novo login e adicionar este usuário ao banco de dados de usuários logados.

E todo acesso à uma *route* será verificado se o usuário não logou por outro computador e *browser* se sim, o usuário será desconectando.

```
# trabalha com a sessão e verifica se esta logado

if "userID" in session:
    cpf = session['userID']
    dataName = "cpf"

    dataAchou, tupleLogado = baseData.getDataInfo("logado", dataName, session['userID'])

    if dataAchou:
        dataAchou = (tupleLogado[0][1] == session['loginHash'])

        if dataAchou:
            if session['userType'] == 'M':
                return redirect('/loggedMedicos')

    if not dataAchou:
        session.pop("loginHash", None)
        session.pop("userName", None)
        session.pop("userID", None)
        session.pop("userType", None)
else:
    session.pop("loginHash", None)
    session.pop("userName", None)
    session.pop("userID", None)
    session.pop("userType", None)
```

**Figura 16.** Mostrando como os *cookies* são gerenciados para ao acesso de cada *route*, dependendo de qual *route* ele esteja o usuário será redirecionado para um *route* adequado.