Back-and

podologia/core/admin.py

```
from django.contrib import admin
from .models import (
    Perfil,
    Usuario,
    Disponibilidade,
    ProfissionalDePodologia,
    TratamentoPodologico,
    Agendamento,
    Feedback,
    ContaUser,
    Anamnese
)


# ********** PERFIL **********
@admin.register(Perfil)
class PerfilAdmin(admin.ModelAdmin):
    list_display = ('email', 'nome', 'telefone', 'podologo')
    search_fields = ('email', 'nome')
    list_filter = ('podologo',)
    ordering = ('email',)


# ********** USUÁRIO **********
@admin.register(Usuario)
class UsuarioAdmin(admin.ModelAdmin):
    list_display = ('id','nome', 'email', 'telefone', 'cpf', 'idade')
    search_fields = ('nome', 'email', 'cpf')
    list_filter = ('data_nasc',)
    ordering = ('nome',)


# ********** DISPONIBILIDADE **********
@admin.register(Disponibilidade)
class DisponibilidadeAdmin(admin.ModelAdmin):
    list_display = ('dia', 'horario_inicio', 'horario_fim')
    list_filter = ('dia',)
    ordering = ('dia', 'horario_inicio')
```

```
# ********** PROFISSIONAL DE PODOLOGIA **********
@admin.register(ProfissionalDePodologia)
class ProfissionalDePodologiaAdmin(admin.ModelAdmin):
  list_display = ('id', 'nome', 'especializacao', 'email', 'telefone_whatsapp',
'aprovado')
  search_fields = ('nome', 'email', 'telefone_whatsapp', 'especializacao')
  list_filter = ('aprovado',)
  ordering = ('nome',)
  filter_horizontal = ('disponibilidade',)


# ********** TRATAMENTO PODOLOGICO **********
@admin.register(TratamentoPodologico)
class TratamentoPodologicoAdmin(admin.ModelAdmin):
  list_display = ('id', 'nome', 'tipo', 'duracao')
  # list_display = ('id', 'nome', 'tipo', 'preco', 'duracao')

  search_fields = ('nome', 'tipo')
  list_filter = ('tipo',)
  ordering = ('nome',)


# ********** AGENDAMENTO **********
@admin.register(Agendamento)
class AgendamentoAdmin(admin.ModelAdmin):
  list_display = ('id', 'data', 'usuario', 'profissional', 'status')
  search_fields = ('usuario__nome', 'profissional__nome')
  list_filter = ('status', 'data')
  ordering = ('data',)
  filter_horizontal = ('servicos',)


# ********** FEEDBACK **********
@admin.register(Feedback)
class FeedbackAdmin(admin.ModelAdmin):
  list_display = ('usuario', 'agendamento', 'nota', 'data', 'respondido')
  search_fields = ('usuario__nome', 'agendamento__usuario__nome')
  list_filter = ('nota', 'data')
  ordering = ('-data',)
```

```
# ********** CONTA DO USUÁRIO **********
@admin.register(ContaUser)
class ContaUserAdmin(admin.ModelAdmin):
  list_display = ('id', 'usuario', 'nome', 'pontuacao', 'avatar')
  search_fields = ('usuario__nome', 'nome')
  ordering = ('pontuacao',)


admin.site.register(Anamnese)
```

podologia/core/forms.py

```
from django import forms
from .models import Perfil
from django.core.exceptions import ValidationError



class UsuarioForms(forms.ModelForm):
  password = forms.CharField(label='Senha', required=True,
widget=forms.PasswordInput())
  password2 = forms.CharField(label='Confirmar Senha', required=True,
widget=forms.PasswordInput())
  concorda_termos = forms.BooleanField(label='Concordo com os termos de uso
e privacidade', required=True)

  class Meta:
    model = Perfil
    fields = ['nome', 'telefone', 'email', 'password', 'password2', 'concorda_termos']

  def clean_password2(self):
    """Valida se as senhas são iguais."""
    password = self.cleaned_data.get("password")
    password2 = self.cleaned_data.get("password2")

    if password and password2 and password != password2:
      raise ValidationError("As senhas não coincidem!")
    return password2
```

```
  def clean_concorda_termos(self):
    """Valida se o usuário concordou com os termos."""
    concorda_termos = self.cleaned_data.get("concorda_termos")
    if not concorda_termos:
      raise ValidationError("Você deve concordar com os termos de uso e
privacidade.")
    return concorda_termos

  def save(self, commit=True):
    """Salva o usuário com senha criptografada."""
    user = super().save(commit=False)
    user.set_password(self.cleaned_data["password"])
    if commit:
      user.save()
    return user
```

podologia/core/models.py

```
from django.db import models
from datetime import date
from django.core.exceptions import ValidationError
import random, re
from django.contrib.auth.models import AbstractUser
from .utils import validate_cpf, get_coordinates_from_cep, gerar_link_whatsapp


class Perfil(AbstractUser):

  username = models.CharField(max_length=100)
  email = models.EmailField(unique=True)
  telefone = models.CharField("Telefone", max_length=11)
  nome = models.CharField("Nome Completo", max_length=100)
  concorda_termos = models.BooleanField(default=False)
  podologo = models.BooleanField(default=False)

  USERNAME_FIELD = "email"
  REQUIRED_FIELDS = ["username"]

  def __str__(self):
```

```
        return self.email

    def gerar_username(self):
        email = self.email
        if '@' in email:
            base_username = email.split('@')[0]
        else:
            base_username = email

        while True:
            numeros_aleatorios = ''.join(random.choices('0123456789', k=4))
            username = f"{base_username}{numeros_aleatorios}"
            if not Perfil.objects.filter(username=username).exists():
                break

        return username

    def save(self, *args, **kwargs):
        if not self.username:
            self.username = self.gerar_username()
        super().save(*args, **kwargs)

    class Meta:
        verbose_name = "Perfil de Login"
        verbose_name_plural = "Perfis de Login"


class Usuario(models.Model):
    nome = models.CharField('Nome', max_length=255)
    data_nasc = models.DateField('Data de Nascimento', null=True, blank=True)
    foto = models.ImageField("Foto", upload_to='avatares', blank=True, null=True)
    email = models.EmailField('E-mail')
    telefone = models.CharField("Telefone", max_length=15, null=True, blank=True)
    cpf = models.CharField("CPF", max_length=15, null=True, blank=True)
    usuario = models.OneToOneField(
        Perfil, verbose_name='Usuario', on_delete=models.CASCADE, blank=True,
null=True, related_name='usuario_perfil'
    )

    @property
    def idade(self):
```

```
        if self.data_nasc:
            hoje = date.today()
            diferenca = hoje - self.data_nasc
            return round(diferenca.days // 365.25)
        return None

    def save(self, *args, **kwargs):
        if self.cpf:
            self.cpf = validate_cpf(self.cpf)
        super().save(*args, **kwargs)

    def __str__(self):
        return self.nome

    class Meta:
        verbose_name = "Usuário"
        verbose_name_plural = "Usuários"


class Disponibilidade(models.Model):
    DIAS_DA_SEMANA = [
        ('segunda', 'Segunda-feira'),
        ('terca', 'Terça-feira'),
        ('quarta', 'Quarta-feira'),
        ('quinta', 'Quinta-feira'),
        ('sexta', 'Sexta-feira'),
        ('sabado', 'Sábado'),
        ('domingo', 'Domingo'),
    ]

    dia = models.CharField(max_length=10, choices=DIAS_DA_SEMANA,
verbose_name="Dia da Semana")
    horario_inicio = models.TimeField(verbose_name="Horário de Início")
    horario_fim = models.TimeField(verbose_name="Horário de Fim")

    class Meta:
        verbose_name = "Disponibilidade"
        verbose_name_plural = "Disponibilidades"
        ordering = ['dia', 'horario_inicio']

    def __str__(self):
```

```
        return f"{self.get_dia_display()} ({self.horario_inicio} às {self.horario_fim})"

    def clean(self):
        if self.horario_inicio >= self.horario_fim:
            raise ValidationError("O horário de fim deve ser posterior ao horário de
início.")


class ProfissionalDePodologia(models.Model):
    nome = models.CharField("Nome Completo", max_length=100)
    especializacao = models.CharField(max_length=300,
                                      help_text="Especialização, como atendimento infantil,
TEA/TDAH etc.")
    user = models.OneToOneField(Perfil, verbose_name='Usuario',
on_delete=models.CASCADE, blank=True, null=True,
                                related_name='profissional_podologia')
    foto = models.ImageField("Foto", upload_to='avatares', blank=True, null=True)
    email = models.EmailField("E-mail", help_text="E-mail de contato do
profissional")
    cpf = models.CharField("CPF", max_length=14, unique=True)
    telefone_whatsapp = models.CharField("Telefone/WhatsApp", max_length=15,
help_text="Telefone de contato")
    rede_social = models.CharField("Link de Rede Social", max_length=255,
null=True, blank=True)

    disponibilidade = models.ManyToManyField(Disponibilidade,
verbose_name="Disponibilidades",
                                             related_name="profissionais", blank=True)
    cep = models.CharField("CEP", max_length=10)
    endereco = models.CharField("Endereço", max_length=255, null=True,
blank=True, help_text="Endereço")
    bairro = models.CharField("Bairro", max_length=255, null=True, blank=True,
help_text="Bairro")
    cidade = models.CharField('Cidade', max_length=50, null=True, blank=True)
    estado = models.CharField("UF", max_length=2, blank=True, null=True)
    especialidade = models.TextField("Especialidade",
                                     help_text="Descrição da especialidade do profissional e sua
experiência", blank=True, null=True)
    aprovado = models.BooleanField("Aprovado", default=False)
    latitude = models.FloatField("Latitude", null=True, blank=True)
    longitude = models.FloatField("Longitude", null=True, blank=True)
```

```
  link_whatsapp = models.URLField("Link do WhatsApp", null=True, blank=True)


  class Meta:
    verbose_name = "Profissional de Podologia"
    verbose_name_plural = "Profissionais de Podologia"

  def save(self, *args, **kwargs):
    if self.cep:
      self.cep = re.sub(r'\D', '', self.cep)
    if self.cpf:
      self.cpf = validate_cpf(self.cpf)
    if self.cep:
      latitude, longitude = get_coordinates_from_cep(self.cep)
      if latitude and longitude:
        self.latitude = latitude
        self.longitude = longitude
    if self.telefone_whatsapp:
      self.link_whatsapp = gerar_link_whatsapp(self.telefone_whatsapp)

    super().save(*args, **kwargs)

  def __str__(self):
    return self.nome


class TratamentoPodologico(models.Model):
  TIPOS_TRATAMENTO = [
    ('Preventivo', 'Preventivo'),
    ('Estético', 'Estético'),
    ('Clínico', 'Clínico'),
    ('Reabilitação', 'Reabilitação'),
  ]

  user = models.ForeignKey(ProfissionalDePodologia,
on_delete=models.CASCADE, blank=True, null=True)
  nome = models.CharField(max_length=100, verbose_name="Nome do
Tratamento")
  descricao = models.TextField(verbose_name="Descrição", help_text="Breve
descrição do tratamento.")
  duracao = models.PositiveIntegerField(verbose_name="Duração (minutos)",
```

```
help_text="Duração média do tratamento.")
    # preco = models.DecimalField(max_digits=10, decimal_places=2,
verbose_name="Preço", blank=True, null=True)
    tipo = models.CharField(max_length=20, choices=TIPOS_TRATAMENTO,
verbose_name="Tipo do Tratamento")

    class Meta:
        verbose_name = "Tratamento Podológico"
        verbose_name_plural = "Tratamentos Podológicos"
        ordering = ['nome']

    def __str__(self):
        return self.nome




class Anamnese(models.Model):

    NIVEL_DE_SENSIBILIDADE = [
        ('nenhuma', 'Nenhuma'),
        ('pouco', 'Pouca'),
        ('suportavel', 'Suportável'),
        ('muito', 'Muita')
    ]

    medicamentos = models.CharField("Medicamentos que usa",max_length=50,
blank=True, null=True)
    gravidez = models.BooleanField("Grávido?", default=False)
    pratica_esporte = models.BooleanField("Pratica algum esporte?", default=False)
    esporte = models.CharField("Qual esporte pratica?", max_length=50,null=True,
blank=True)
    fez_cirurgia_no_pe = models.BooleanField("Já fez algum cirurgia nos membros
inferiores?", default=False)
    cirurgia_no_pe = models.CharField("Qual cirurgia nos membros inferiores?",
max_length=50, null=True, blank=True)
    sensibilidade_a_dor = models.CharField("Nível de sensibilidade a
dor",max_length=10, choices=NIVEL_DE_SENSIBILIDADE, blank=True, null=True)
    marca_passo = models.BooleanField("Possui marca passo?", default=False)
    pressao_alta = models.BooleanField("Possui pressão alta?", default=False)
    antecedentes_cancerigenos = models.BooleanField("Tem antecedente
```

```
familiares com cancer?", default=False)
    diabetes = models.BooleanField("Tem diabetes?", default=False)
    convulsoes = models.BooleanField("Tem problemas de convulsões?",
default=False)
    problema_circulatorio = models.BooleanField("Tem problemas de
circulatórios?", default=False)

    def __str__(self):
        return str(self.id)

    class Meta:
        verbose_name = "Anamnese"
        verbose_name_plural = "Anamneses"


class ContaUser(models.Model):
    AVATAR_CHOICES = [
        ('avatar1.png', 'Avatar 1'),
        ('avatar2.png', 'Avatar 2'),
        ('avatar3.png', 'Avatar 3'),
        ('avatar4.png', 'Avatar 4'),
        ('avatar5.png', 'Avatar 5'),
        ('avatar6.png', 'Avatar 6'),
    ]

    usuario = models.ForeignKey(Usuario, on_delete=models.CASCADE,
verbose_name='Perfil de Usuário')
    anamnese = models.OneToOneField(Anamnese, on_delete=models.CASCADE,
verbose_name='Anamnese do paciente', blank=True, null=True)
    nome = models.CharField('Nome', max_length=100, default='Perfil')
    data_nasc = models.DateField('Data de Nascimento', null=True, blank=True)
    pontuacao = models.BigIntegerField(verbose_name='Pontuação', blank=True,
null=True)
    avatar = models.CharField(choices=AVATAR_CHOICES, max_length=50,
default='avatar1.png')

    @property
    def idade(self):
        if self.data_nasc:
            hoje = date.today()
            diferenca = hoje - self.data_nasc
```

```
            return round(diferenca.days // 365.25)
        return None



    def __str__(self):
        return f'{self.nome} - {self.pontuacao}'

    class Meta:
        verbose_name = "Conta"
        verbose_name_plural = "Contas"




class Agendamento(models.Model):
    STATUS_CHOICES = [
        ('pendente', 'Pendente'),
        ('confirmado', 'Confirmado'),
        ('concluido', 'Concluído'),
        ('cancelado', 'Cancelado'),
    ]

    usuario = models.ForeignKey(Usuario, on_delete=models.CASCADE,
verbose_name="Usuario/Conta", related_name="agendamentos")
    conta = models.ForeignKey(ContaUser, on_delete=models.CASCADE,
verbose_name="Cliente", blank=True, null=True)
    profissional = models.ForeignKey(
        ProfissionalDePodologia, on_delete=models.CASCADE,
verbose_name="Profissional", related_name="agendamentos"
    )
    servicos = models.ManyToManyField(TratamentoPodologico,
verbose_name="Serviços", related_name="agendamentos")
    data = models.DateField(verbose_name="Data do Agendamento",
default=date.today)
    status = models.CharField(max_length=10, choices=STATUS_CHOICES,
default='pendente',
                              verbose_name="Status do Agendamento")
    justificativa = models.TextField("Justificativa", max_length=200,
                                     help_text="Em casos de cancelamento é preciso justificar",
blank=True, null=True)
```

```
    class Meta:
        verbose_name = "Agendamento"
        verbose_name_plural = "Agendamentos"
        ordering = ['data']

    def __str__(self):
        return f"Agendamento em {self.data} - {self.usuario.nome}"


class Feedback(models.Model):
    NOTA_CHOICES = [
        (1, '1 - Muito ruim'),
        (2, '2 - Ruim'),
        (3, '3 - Regular'),
        (4, '4 - Bom'),
        (5, '5 - Excelente'),
    ]

    usuario = models.ForeignKey(Usuario, on_delete=models.CASCADE,
verbose_name="Cliente", related_name="feedbacks")
    agendamento = models.OneToOneField(Agendamento,
on_delete=models.CASCADE, verbose_name="Agendamento",
                                       related_name="feedback")
    nota = models.PositiveSmallIntegerField(choices=NOTA_CHOICES,
verbose_name="Nota", blank=True, null=True)
    comentario = models.TextField(verbose_name="Comentário", blank=True,
null=True,
                                  help_text="Comentários adicionais sobre o atendimento")
    data = models.DateTimeField(verbose_name="Data do Feedback",
auto_now_add=True)
    respondido = models.BooleanField(default=False)

    class Meta:
        verbose_name = "Feedback"
        verbose_name_plural = "Feedbacks"
        ordering = ['-data']

    def __str__(self):
```

```
        return f"Feedback de {self.usuario.nome} - Nota: {self.nota}"
```

podologia/core/serializers.py

```
from rest_framework import serializers
from .models import Usuario, Anamnese, Perfil, Disponibilidade,
ProfissionalDePodologia, TratamentoPodologico, Agendamento, Feedback,
ContaUser
from django.contrib.auth.password_validation import validate_password
from .utils import validate_unique_email
from django.contrib.auth import authenticate
from django.contrib.auth.models import User

class PasswordResetSerializer(serializers.Serializer):
    email = serializers.EmailField()

    def validate_email(self, value):
        if not User.objects.filter(email=value).exists():
            raise serializers.ValidationError("Usuário com este e-mail não encontrado.")
        return value


class UsuarioSerializer(serializers.ModelSerializer):

    class Meta:
        model = Usuario
        fields = ['id', 'nome', 'data_nasc', 'foto', 'email', 'telefone', 'cpf']


class DisponibilidadeSerializer(serializers.ModelSerializer):
    """
    Serializer para o modelo Disponibilidade.
    """
    class Meta:
        model = Disponibilidade
        fields = ['id', 'dia', 'horario_inicio', 'horario_fim']
```

```
class ProfissionalDePodologiaSerializerCreate(serializers.ModelSerializer):
    """
    Serializer para o modelo ProfissionalDePodologia,
    com validação personalizada para os campos obrigatórios.
    """
    disponibilidade = DisponibilidadeSerializer(many=True, required=False)

    class Meta:
        model = ProfissionalDePodologia
        fields = [
            'id',
            'nome',
            'especializacao',
            'email',
            'cpf',
            'telefone_whatsapp',
            'disponibilidade',
            'cep',
            'rede_social',  # Incluído o campo 'rede_social'
        ]

    def validate(self, data):
        # Validação adicional para garantir a presença de todos os campos
obrigatórios
        required_fields = ['nome', 'especializacao', 'email', 'cpf', 'telefone_whatsapp',
'cep']
        for field in required_fields:
            if field not in data or not data[field]:
                raise serializers.ValidationError({field: "Este campo é obrigatório."})
        return data

    def create(self, validated_data):
        # Verificar se disponibilidades foram fornecidas
        disponibilidades_data = validated_data.pop('disponibilidade', [])
        profissional = ProfissionalDePodologia.objects.create(**validated_data)

        # Criar disponibilidades apenas se existirem no payload
        for disponibilidade_data in disponibilidades_data:
            Disponibilidade.objects.create(**disponibilidade_data,
profissionais=profissional)
```

```
        return profissional


class TratamentoPodologicoSerializer(serializers.ModelSerializer):
    class Meta:
        model = TratamentoPodologico
        fields = ['id', 'nome', 'descricao', 'duracao',  'tipo']
        # fields = ['id', 'nome', 'descricao', 'duracao', 'preco', 'tipo']


class TratamentoPodologicoSerializerCreate(serializers.ModelSerializer):
    class Meta:
        model = TratamentoPodologico
        fields = ['id', 'user', 'nome', 'descricao', 'duracao', 'tipo']
        # fields = ['id', 'user', 'nome', 'descricao', 'duracao', 'preco', 'tipo']

        read_only_fields = ['id']

class ContaUserSerializer(serializers.ModelSerializer):
    class Meta:
        model = ContaUser
        fields = ['id', 'usuario', 'nome', 'pontuacao', 'avatar', 'data_nasc']



class ProfissionalDePodologiaSerializer(serializers.ModelSerializer):
    disponibilidade = DisponibilidadeSerializer(many=True)
    tratamentos = TratamentoPodologicoSerializer(many=True,
source='tratamentopodologico_set')

    class Meta:
        model = ProfissionalDePodologia
        fields = [
            'id', 'nome', 'especializacao', 'foto', 'email', 'cep',
            'telefone_whatsapp', 'rede_social', 'disponibilidade', 'especialidade', 'bairro',
'cidade', 'estado', 'tratamentos', 'link_whatsapp'
        ]

class AgendamentoSerializer(serializers.ModelSerializer):
    usuario = UsuarioSerializer()
```

```
    profissional = ProfissionalDePodologiaSerializer()
    servicos = TratamentoPodologicoSerializer(many=True)
    conta_nome = serializers.CharField(source='conta.nome', read_only=True)  #
Use 'conta.nome'

    class Meta:
        model = Agendamento
        fields = ['id', 'usuario', 'conta_nome', 'profissional', 'servicos', 'data', 'status']



class AgendamentoSerializerPodologo(serializers.ModelSerializer):
    class Meta:
        model = Agendamento
        fields = ['usuario', 'conta' ,'servicos', 'data', 'status', 'profissional']
        read_only_fields = ['status']

class StatusAgendamentoSerializer(serializers.Serializer):
    status = serializers.ChoiceField(choices=['pendente', 'confirmado', 'concluido',
'cancelado'])



class AgendamentoSerializerList(serializers.ModelSerializer):
    usuario_nome = serializers.CharField(source='usuario.nome', read_only=True)
    profissional_nome = serializers.CharField(source='profissional.nome',
read_only=True)
    conta_nome = serializers.CharField(source='conta.nome', read_only=True)
    id_conta_user = serializers.CharField(source='conta.id', read_only=True)
    usuario_telefone = serializers.CharField(source='usuario.telefone',
read_only=True)

    class Meta:
        model = Agendamento
        fields = ['id', 'id_conta_user', 'data', 'status', 'usuario_nome', 'conta_nome',
'usuario_telefone' ,'profissional_nome', 'servicos', 'justificativa']
        depth = 1



class CancelarAgendamentoSerializer(serializers.ModelSerializer):
    class Meta:
        model = Agendamento
```

```
        fields = [
            'id',
            'usuario',
            'profissional',
            'servicos',
            'data',
            'status',
            'justificativa',
        ]
        depth = 1


class FeedbackSerializer(serializers.ModelSerializer):
    usuario = UsuarioSerializer()
    agendamento = AgendamentoSerializer()

    class Meta:
        model = Feedback
        fields = ['id', 'usuario', 'agendamento', 'nota', 'comentario', 'data']


class FeedbackSerializerTeste(serializers.ModelSerializer):
    class Meta:
        model = Feedback
        fields = ['id', 'usuario', 'agendamento', 'nota', 'comentario', 'data', 'respondido']




class FeedbackSerializerResposta(serializers.ModelSerializer):
    class Meta:
        model = Feedback
        fields = ['id', 'usuario', 'agendamento', 'nota', 'comentario', 'data', 'respondido']
        read_only_fields = ['id', 'usuario', 'agendamento', 'data', 'respondido']

    def validate_nota(self, value):
        if value is None or not (1 <= value <= 5):
            raise serializers.ValidationError("A nota deve ser um número entre 1 e 5.")
        return value

    def validate_comentario(self, value):
```

```
        if not value or value.strip() == "":
            raise serializers.ValidationError("O comentário não pode estar vazio.")
        return value


class PerfilSerializer(serializers.ModelSerializer):
    password = serializers.CharField(write_only=True, min_length=6, required=True,
validators=[validate_password])
    password2 = serializers.CharField(write_only=True, min_length=6,
required=True, validators=[validate_password])
    email = serializers.EmailField(validators=[validate_unique_email])

    class Meta:
        model = Perfil
        fields = ['nome', 'email', 'telefone', 'concorda_termos', 'password', 'password2']

    def validate(self, attrs):
        if attrs["password"] != attrs["password2"]:
            raise serializers.ValidationError({"password": "As senhas digitadas não
correspondem"})
        return attrs

    def validate_concorda_termos(self, value):
        if not value:
            raise serializers.ValidationError("Você precisa concordar com os termos para
se cadastrar.")
        return value

    def create(self, validated_data):
        validated_data.pop('password2')
        password = validated_data.pop('password')
        perfil = Perfil.objects.create(**validated_data)
        perfil.set_password(password)
        perfil.save()
        return perfil

class LoginSerializer(serializers.Serializer):
    email = serializers.EmailField()
    password = serializers.CharField(write_only=True)

    def validate(self, attrs):
```

```
        email = attrs.get('email')
        password = attrs.get('password')

        # Validação do e-mail
        user = Perfil.objects.filter(email=email).first()
        if user is None:
            raise serializers.ValidationError("E-mail ou senha inválidos.")

        # Autenticação do usuário com e-mail e senha
        user = authenticate(email=email, password=password)
        if user is None:
            raise serializers.ValidationError("E-mail ou senha inválidos.")

        attrs['user'] = user
        return attrs

class ChangePasswordSerializer(serializers.Serializer):
    old_password = serializers.CharField(required=True, write_only=True)
    new_password = serializers.CharField(required=True, write_only=True)
    confirm_new_password = serializers.CharField(required=True, write_only=True)

    def validate_old_password(self, value):
        user = self.context['request'].user
        if not user.check_password(value):
            raise serializers.ValidationError("A senha antiga está incorreta.")
        return value

    def validate(self, data):
        if data['new_password'] != data['confirm_new_password']:
            raise serializers.ValidationError({"new_password": "As novas senhas não
coincidem."})

        # Adicione validações adicionais, se necessário
        if len(data['new_password']) < 8:
            raise serializers.ValidationError({"new_password": "A nova senha deve ter
pelo menos 8 caracteres."})

        return data

class AgendamentoSerializerCreateUser(serializers.ModelSerializer):
    servicos = serializers.PrimaryKeyRelatedField(
```

```
        queryset=TratamentoPodologico.objects.all(),
        many=True
    )

    class Meta:
        model = Agendamento
        fields = ['id', 'usuario', 'conta', 'profissional', 'servicos', 'data', 'status']

    def create(self, validated_data):
        # Extraindo os dados necessários
        usuario = validated_data.pop('usuario')
        conta = validated_data.pop('conta')
        servicos = validated_data.pop('servicos')
        agendamento = Agendamento.objects.create(
            usuario=usuario,
            conta=conta,
            **validated_data
        )
        agendamento.servicos.set(servicos)
        return agendamento


class AgendamentoSerializerCreate(serializers.ModelSerializer):
    class Meta:
        model = Agendamento
        fields = ['id', 'usuario', 'profissional', 'servicos', 'data', 'status', 'justificativa']
        read_only_fields = ['id', 'status', 'justificativa']


class AnamneseSerializer(serializers.ModelSerializer):
    class Meta:
        model = Anamnese
        fields = [
            'id',
            'medicamentos',
            'gravidez',
            'pratica_esporte',
            'esporte',
            'fez_cirurgia_no_pe',
            'cirurgia_no_pe',
            'sensibilidade_a_dor',
```

```
        'marca_passo',
        'pressao_alta',
        'antecedentes_cancerigenos',
        'diabetes',
        'convulsoes',
        'problema_circulatorio',
    ]
```

podologia/core/signais.py

```
from django.db.models.signals import post_save, pre_save
from django.dispatch import receiver
from .models import ProfissionalDePodologia, Perfil, Usuario, Agendamento,
Feedback, ContaUser, Anamnese
from django.contrib.auth.hashers import make_password
import requests
from django.core.mail import send_mail
from django.conf import settings


@receiver(post_save, sender=ProfissionalDePodologia)
def criar_perfil_para_profissional(sender, instance, created, **kwargs):
    """
    Quando um ProfissionalDePodologia é criado, associa um Perfil ao profissional.
    Define a senha do Perfil como o CPF do profissional.
    """
    if created and instance.user is None:
        # Cria um Perfil associado
        perfil = Perfil.objects.create(
            nome=instance.nome,
            email=instance.email,
            podologo=True,
            concorda_termos=True,
            telefone=instance.telefone_whatsapp,
            username=instance.email,  # Usa o email como username
            password=make_password(instance.cpf),  # Define o CPF como senha
        )
        instance.user = perfil
        instance.save()
```

```
@receiver(post_save, sender=ProfissionalDePodologia)
def preencher_dados_viacep(sender, instance, created, **kwargs):
    """
    Preenche automaticamente os dados de endereço usando a API ViaCEP
    após salvar um novo ProfissionalDePodologia.
    """
    if created and instance.cep and not instance.endereco:
        try:
            url = f"https://viacep.com.br/ws/{instance.cep}/json/"
            response = requests.get(url)
            if response.status_code == 200:
                dados = response.json()
                instance.endereco = dados.get("logradouro", "")
                instance.bairro = dados.get("bairro", "")
                instance.cidade = dados.get("localidade", "")
                instance.estado = dados.get("uf", "")
                instance.save()  # Salva as mudanças no banco de dados
        except requests.exceptions.RequestException as e:
            print(f"Erro ao obter dados do CEP: {e}")


@receiver(post_save, sender=Perfil)
def criar_usuario_para_perfil(sender, instance, created, **kwargs):
    """
    Quando um Perfil é criado, cria um objeto Usuario associado,
    desde que não exista um ProfissionalDePodologia com o mesmo e-mail.
    """
    if created:
        # Verifica se existe um ProfissionalDePodologia com o mesmo e-mail
        if not ProfissionalDePodologia.objects.filter(email=instance.email).exists():
            # Cria o Usuario associado ao Perfil
            Usuario.objects.create(
                nome=instance.nome,
                email=instance.email,
                telefone=instance.telefone,
                usuario=instance
            )
```

```
@receiver(post_save, sender=Agendamento)
def criar_feedback_ao_concluir_agendamento(sender, instance, created,
**kwargs):
  # Verificar se o agendamento foi concluído
  if instance.status == 'concluido':
    # Verificar se já existe um feedback para este agendamento
    if not hasattr(instance, 'feedback'):
      # Criar um feedback automaticamente
      Feedback.objects.create(
        usuario=instance.usuario,
        agendamento=instance,
        nota=None,  # A nota ficará em branco inicialmente
        comentario=None,  # O comentário ficará em branco inicialmente
      )

# @receiver(post_save, sender=Usuario)
# def criar_contaUser_ao_criar_usuario(sender, instance, created, **kwargs):
#   if created:
#     ContaUser.objects.bulk_create([
#       ContaUser(usuario=instance, nome="Perfil 1"),
#       ContaUser(usuario=instance, nome="Perfil 2"),
#       ContaUser(usuario=instance, nome="Perfil 3"),
#     ])


# @receiver(post_save, sender=Usuario)
# def criar_contaUser_ao_criar_usuario(sender, instance, created, **kwargs):
#   if created:
#     contas = [
#       ContaUser(usuario=instance, nome="Perfil 1"),
#       ContaUser(usuario=instance, nome="Perfil 2"),
#       ContaUser(usuario=instance, nome="Perfil 3"),
#     ]
#     ContaUser.objects.bulk_create(contas)
#     for conta in contas:
#       anamnese = Anamnese.objects.create()
#       conta.anamnese = anamnese
#       conta.save()  # Salva a relação com Anamnese

@receiver(post_save, sender=Usuario)
```

```
def criar_contaUser_ao_criar_usuario(sender, instance, created, **kwargs):
  if created:
    for nome in ["Perfil 1", "Perfil 2", "Perfil 3"]:
      conta_user = ContaUser(usuario=instance, nome=nome)
      conta_user.save()  # Isto dispara o post_save para ContaUser


@receiver(post_save, sender=ContaUser)
def criar_anamnese_para_conta_user(sender, instance, created, **kwargs):
  if created:
    anamnese = Anamnese.objects.create()
    instance.anamnese = anamnese
    instance.save()




@receiver(post_save, sender=ProfissionalDePodologia)
def enviar_email_confirmacao(sender, instance, created, **kwargs):
  if created:
    assunto = 'Confirmação de Cadastro'

    mensagem = (
      f'Você foi cadastrado em nossa plataforma! '
      f'Seu e-mail de acesso é ({instance.email}) e use seu CPF (somente
números) '
      f'para login! Pedimos que, para sua segurança, troque sua senha
imediatamente!'
    )

    remetente = 'email@email.com.br'
    destinatario = [instance.email]

    send_mail(assunto, mensagem, remetente, destinatario)


@receiver(pre_save, sender=Agendamento)
def enviar_email_atualizacao_status(sender, instance, **kwargs):
  """
  Envia um email para o usuário e o podólogo sempre que o status de um
```

```
agendamento for alterado.
    """
    try:
        # Verifica se o agendamento já existe no banco de dados
        agendamento_anterior = Agendamento.objects.get(pk=instance.pk)
        if agendamento_anterior.status != instance.status:
            # Status foi alterado, prepara o envio de email
            usuario_email = instance.usuario.email
            profissional_email = instance.profissional.email

            # Formata a data no padrão DD/MM/AAAA
            data_formatada = instance.data.strftime('%d/%m/%Y')

            assunto = f"Atualização no status do seu agendamento"
            mensagem = (
                f"Olá,\n\n"
                f"O status do agendamento marcado para {data_formatada} foi alterado
para: {instance.get_status_display()}.\n\n"
                f"Detalhes do agendamento:\n"
                f"Usuário: {instance.usuario.nome}\n"
                f"Profissional: {instance.profissional.nome}\n"
                f"Status Atual: {instance.get_status_display()}\n\n"
                f"Atenciosamente,\nEquipe de Podologia"
            )

            # Envia email para o usuário
            send_mail(
                assunto,
                mensagem,
                settings.DEFAULT_FROM_EMAIL,
                [usuario_email],
                fail_silently=False,
            )

            # Envia email para o profissional
            send_mail(
                assunto,
                mensagem,
                settings.DEFAULT_FROM_EMAIL,
                [profissional_email],
                fail_silently=False,
```

```
        )
    except Agendamento.DoesNotExist:
        # O agendamento é novo, nenhuma alteração de status
        pass



@receiver(post_save, sender=Feedback)
def enviar_email_feedback(sender, instance, created, **kwargs):
    """
    Envia um email ao usuário cadastrado no feedback, solicitando que ele avalie o atendimento.
    """
    if created:
        usuario_email = instance.usuario.email
        profissional_nome = instance.agendamento.profissional.nome
        data_agendamento = instance.agendamento.data.strftime('%d/%m/%Y')

        assunto = "Avaliação do Atendimento com seu Podólogo"
        mensagem = (
            f"Olá {instance.usuario.nome},\n\n"
            f"Esperamos que seu atendimento com o podólogo {profissional_nome}, realizado em {data_agendamento}, tenha sido satisfatório.\n\n"
            f"Gostaríamos de saber sua opinião! Por favor, avalie o atendimento respondendo ao formulário de avaliação disponível no nosso sistema.\n\n"
            f"Agradecemos pela sua colaboração.\n\n"
            f"Atenciosamente,\nEquipe de Podologia"
        )

        # Envia o email
        send_mail(
            assunto,
            mensagem,
            settings.DEFAULT_FROM_EMAIL,
            [usuario_email],
            fail_silently=False,
        )
```

podologia/core/urls.py

```
from django.urls import path, re_path
from django.contrib.auth import views as auth_views
from .views import (
    ProfissionalDePodologiaListView,
    ListarAgendamentosView,
    UsuarioLogadoView,
    CadastroPerfilView,
    LoginView,
    LogoutView,
    AtualizarSenha,
    ContaUserListView,
    CadastrarTratamentoView,
    GerenciarTratamentosView,
    CriarProfissionalDePodologiaView,
    DisponibilidadeView,
    ListaAgendamentosView,
    CriarAgendamentoView,
    CancelarAgendamentoView,
    FeedbackNaoRespondidoView,
    ResponderFeedbackView,
    CriarAgendamentoViewUser,
    ContaUserUpdate,
    FeedbacksDoProfissionalView,
    AtualizarStatusAgendamentoView,
    ListaContaUsersAgendamentosView,
    AtualizarAnamneseView,
    ListarTratamentosPorPodologo,
    DetalharPodologo,
    MeusAgendamentosPorData,
    ListarMeusClientes,
    ContasUserPorUsuario,
    BuscarClienteView,
    PasswordResetView,
    NovoTrabalho,
    PasswordResetView as CustomPasswordResetView
)
from rest_framework.permissions import AllowAny
from drf_yasg.views import get_schema_view
```

```
from drf_yasg import openapi

schema_view = get_schema_view(
   openapi.Info(
      title="Sistema para Podologas",
      default_version='v1',
      description="API para agendamento e gerenciamento de atendimento com
podologas",
   ),
   public=True,
   permission_classes=(AllowAny,),
)

urlpatterns = [
   # Views Públicas
   path('cadastro/', CadastroPerfilView.as_view(), name='cadastro-usuario'),
   path('login/', LoginView.as_view(), name='login-usuario'),
   path('logout/', LogoutView.as_view(), name='logout-usuario'),
   path('trocar-senha/', AtualizarSenha.as_view(), name='trocar-senha'),
   path('usuario-logado/', UsuarioLogadoView.as_view(), name='usuario-logado'),

   # Views do SuperUsuário
   path('criar-podologo/', CriarProfissionalDePodologiaView.as_view(), name='criar-
podologo'),
   path('disponibilidade/', DisponibilidadeView.as_view(),
name='consultar_disponibilidade'),
   path('podologo/<int:pk>/tratamentos/',
ListarTratamentosPorPodologo.as_view(),
name='listar_tratamentos_por_podologo'),

   # Views do Usuário Cliente
   path('podologos/', ProfissionalDePodologiaListView.as_view(),
name='podologos'),
   path('podologos/<int:pk>/', DetalharPodologo.as_view(), name='detail-
podologo-por-id'),
   path('meus-agendamentos/', ListaAgendamentosView.as_view(),
name='agendamentos-usuarios'),
   path('meus-agendamentos/criar/', CriarAgendamentoViewUser.as_view(),
name='criar_agendamento'),
   path('meus-agendamentos/agendamentos/<int:agendamento_id>/cancelar/',
CancelarAgendamentoView.as_view(), name='cancelar-agendamento'),
```

```
    path('feedbacks/nao-respondidos/', FeedbackNaoRespondidoView.as_view(),
name='feedbacks-nao-respondidos'),
    path('feedbacks/nao-respondidos/<int:feedback_id>/responder/',
ResponderFeedbackView.as_view(), name='responder-feedback'),
    path('contas/', ContaUserListView.as_view(), name='listar-contas'),
    path('contas/atualizar/<int:pk>/', ContaUserUpdate.as_view(), name='atualizar-
conta'),

    # Views do Profissional de Podologia
    path('tratamentos/', GerenciarTratamentosView.as_view(),
name='listar_tratamentos'),
    path('tratamentos/<int:pk>/', GerenciarTratamentosView.as_view(),
name='gerir_tratamento'),
    path('tratamentos/cadastrar/', CadastrarTratamentoView.as_view(),
name='cadastrar-tratamento'),
    path('meus-feedbacks/', FeedbacksDoProfissionalView.as_view(),
name='feedbacks-do-profissional'),
    path('clientes/buscar/<str:email>/', BuscarClienteView.as_view(),
name='buscar_cliente'),
    path('agendamentos/', ListarAgendamentosView.as_view(), name='lista-
agendamentos-podologo'),
    path('agendamentos/criar/', NovoTrabalho.as_view(), name='criar-agendamento-
podologo'),
    path('agendamentos/<str:data>/', MeusAgendamentosPorData.as_view(),
name='lista-agendamentos-podologo-por-data'),
    path('agendamentos/<int:agendamento_id>/atualizar-status/',
AtualizarStatusAgendamentoView.as_view(), name='atualizar-status-
agendamento'),
    path('agendamentos/clientes/', ListarMeusClientes.as_view(), name='listando-
meus-cliente'),
    path('agendamentos/clientes/<int:conta_id>/contasuser/',
ContasUserPorUsuario.as_view(), name='listando-contausers-dos-meus-cliente'),
    path('agendamentos/clientes/anamneses/',
ListaContaUsersAgendamentosView.as_view(), name='anamneses-clientes'),
    path('agendamentos/clientes/anamneses/<int:conta_user_id>/atualizar/',
AtualizarAnamneseView.as_view(), name='atualizar-anamneses-clientes'),

    # Documentação da API
    path('swagger/', schema_view.with_ui('swagger', cache_timeout=0),
name='schema-swagger-ui'),
```

```
  # Reset de Senha
  # path('api/password-reset/', CustomPasswordResetView.as_view(),
name='password-reset'),
  path('password-reset/', CustomPasswordResetView.as_view(),
name='password_reset'),
  path('password-reset/done/', auth_views.PasswordResetDoneView.as_view(),
name='password_reset_done'),
  path('reset/<uidb64>/<token>/',
auth_views.PasswordResetConfirmView.as_view(),
name='password_reset_confirm'),
  path('reset/done/', auth_views.PasswordResetCompleteView.as_view(),
name='password_reset_complete'),

  path('api/password-reset/', PasswordResetView.as_view(), name='password-
reset'),
]
```

podologia/core/utils.py

```
from rest_framework import serializers
from django.core.exceptions import ValidationError
from validate_docbr import CPF
import re, requests

def validate_password(value):
  if len(value) < 6:
    raise serializers.ValidationError("A senha deve ter pelo menos 6 caracteres.")
  return value

def validate_unique_email(value):
  # Importação atrasada para evitar circularidade
  from django.apps import apps
  Perfil = apps.get_model('core', 'Perfil')  # Obtém o modelo dinamicamente
  if Perfil.objects.filter(email=value).exists():
    raise serializers.ValidationError("Este e-mail já está cadastrado.")
  return value

def validate_cpf(cpf: str):
  cpf = re.sub(r'\D', '', cpf)  # Remove caracteres não numéricos
  cpf_validator = CPF()
  if not cpf_validator.validate(cpf):
    raise ValidationError('CPF inválido!')
```

```
    return cpf

def get_coordinates_from_cep(cep):
    """
    Função que recebe um CEP e retorna a latitude e longitude correspondentes.
    """
    url = f"https://www.cepaberto.com/api/v3/cep?cep={cep}"
    headers = {'Authorization': 'Token b0c307f1fc665088f801579e6cdd34b8'}
    try:
        response = requests.get(url, headers=headers)
        if response.status_code == 200:
            data = response.json()
            latitude = data.get('latitude')
            longitude = data.get('longitude')
            return latitude, longitude
        else:
            return None, None
    except requests.exceptions.RequestException as e:
        print(f"Erro ao buscar dados do CEP: {e}")
        return None, None


def gerar_link_whatsapp(contato):
    numero = re.sub(r'\D', '', contato)
    return f"https://wa.me/{numero}"
```

podologia/core/views.py

```
from rest_framework.generics import ListAPIView, CreateAPIView, UpdateAPIView,
RetrieveAPIView
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status
from rest_framework.permissions import IsAuthenticated
from rest_framework_simplejwt.tokens import RefreshToken
from django.shortcuts import get_object_or_404
from django.db.models import Q
from geopy.distance import geodesic
from rest_framework.exceptions import PermissionDenied
from django.utils import timezone
```

```
from django.contrib.auth.models import User
from django.core.mail import send_mail
from django.utils.crypto import get_random_string
from django.contrib.auth.views import PasswordResetView
from .serializers import PasswordResetSerializer
from django.contrib.auth.tokens import default_token_generator
from django.core.exceptions import ObjectDoesNotExist
from django.conf import settings
import datetime
import logging

from .models import (
    ProfissionalDePodologia,
    TratamentoPodologico,
    Usuario,
    Agendamento,
    ContaUser,
    Disponibilidade,
    Feedback
)
from .serializers import (
    AnamneseSerializer,
    UsuarioSerializer,
    StatusAgendamentoSerializer,
    AgendamentoSerializerPodologo,
    FeedbackSerializerTeste,
    AgendamentoSerializerCreateUser,
    FeedbackSerializerResposta,
    ChangePasswordSerializer,
    ProfissionalDePodologiaSerializer,
    CancelarAgendamentoSerializer,
    AgendamentoSerializer,
    AgendamentoSerializerList,
    TratamentoPodologicoSerializer,
    TratamentoPodologicoSerializerCreate,
    PerfilSerializer,
    LoginSerializer,
    ContaUserSerializer,
    ProfissionalDePodologiaSerializerCreate,
    DisponibilidadeSerializer,
    FeedbackSerializer
```

```
)

# Resetar Senha:

from django.contrib.auth.tokens import default_token_generator
from django.core.exceptions import ObjectDoesNotExist
from django.conf import settings

class CustomPasswordResetView(PasswordResetView):
  """
  Classe personalizada para redefinição de senha usando templates específicos.
  """
  email_template_name = 'registration/password_reset_email.html'
  subject_template_name = 'registration/password_reset_subject.txt'
  success_url = '/password-reset/done/'


class PasswordResetView(APIView):
  """
  Endpoint para solicitação de redefinição de senha.
  """
  def post(self, request):
    serializer = PasswordResetSerializer(data=request.data)
    if serializer.is_valid():
      email = serializer.validated_data['email']

      try:
        # Obter o usuário pelo e-mail
        user = User.objects.get(email=email)

        # Gerar um token seguro de redefinição de senha
        token = default_token_generator.make_token(user)

        # Construir a URL de redefinição
        reset_url = f"{settings.FRONTEND_URL}/resetar-
senha?uid={user.id}&token={token}"

        # Enviar o e-mail com o link de redefinição
        send_mail(
          subject='Redefinição de Senha',
          message=f'Clique no link para redefinir sua senha: {reset_url}',
```

```
            from_email=settings.DEFAULT_FROM_EMAIL,
            recipient_list=[email],
        )

        return Response(
            {"message": "E-mail de redefinição enviado com sucesso."},
            status=status.HTTP_200_OK
        )

    except ObjectDoesNotExist:
        # Resposta genérica para evitar exposição de dados sensíveis
        return Response(
            {"error": "Se um usuário com este e-mail existir, você receberá um e-mail com instruções."},
            status=status.HTTP_200_OK
        )

  return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)


# ********** Para Superusuários **********

class CriarProfissionalDePodologiaView(APIView):
  """
  View para criar profissionais de podologia.
  Somente superusuários têm permissão para criar.
  """
  permission_classes = [IsAuthenticated]

  def post(self, request, *args, **kwargs):
    # Verificar se o usuário autenticado é superusuário
    if not request.user.is_superuser:
      raise PermissionDenied("Apenas superusuários podem criar profissionais de podologia.")

    # Serializar os dados recebidos
    serializer = ProfissionalDePodologiaSerializerCreate(data=request.data)
    if serializer.is_valid():
      # Salvar o novo profissional de podologia, incluindo disponibilidades
      serializer.save()
      return Response(serializer.data, status=status.HTTP_201_CREATED)
```

```
        else:
            return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

class DisponibilidadeView(APIView):
    """
    Endpoint para consultar todos os horários de disponibilidade.
    Apenas superusuários têm acesso.
    """
    permission_classes = [IsAuthenticated]  # Garante que o usuário esteja autenticado

    def get(self, request):
        # Verifica se o usuário é superusuário
        if not request.user.is_superuser:
            return Response({"detail": "Você não tem permissão para acessar este recurso."}, status=status.HTTP_403_FORBIDDEN)

        # Consulta todos os registros de Disponibilidade
        disponibilidades = Disponibilidade.objects.all()

        # Serializa os dados
        serializer = DisponibilidadeSerializer(disponibilidades, many=True)

        return Response(serializer.data, status=status.HTTP_200_OK)


class ListarTratamentosPorPodologo(APIView):
    """
    Exibe a lista de tratamentos relacionados a um podólogo específico.
    """
    permission_classes = [IsAuthenticated]  # Apenas usuários autenticados podem acessar.

    def get(self, request, pk):
        # Obtém o podólogo pelo ID fornecido na URL.
        podologo = get_object_or_404(ProfissionalDePodologia, id=pk)

        # Filtra os tratamentos associados ao podólogo.
        tratamentos = TratamentoPodologico.objects.filter(user=podologo)

        # Verifica se existem tratamentos para o podólogo.
```

```
        if not tratamentos.exists():
            return Response(
                {"mensagem": "Nenhum tratamento encontrado para este podólogo."},
                status=status.HTTP_404_NOT_FOUND
            )

        # Serializa os tratamentos para o formato JSON.
        serializer = TratamentoPodologicoSerializer(tratamentos, many=True)

        # Retorna os tratamentos no formato JSON.
        return Response(serializer.data, status=status.HTTP_200_OK)


# ********** ContaUser **********

class ContaUserListView(APIView):
    """
    API para listar todas as ContasUser.
    """
    def get(self, request, format=None):
        usuario = request.user

        # Verificar se o usuário logado é um perfil de usuário
        if not hasattr(usuario, 'usuario_perfil'):
            return Response(
                {"erro": "Acesso não permitido. Apenas usuários podem criar
agendamentos."},
                status=status.HTTP_403_FORBIDDEN
            )

        # Obter a instância do modelo Usuario

        usuario_perfil = usuario.usuario_perfil

        contas = ContaUser.objects.filter(usuario=usuario_perfil)
        serializer = ContaUserSerializer(contas, many=True)
        return Response(serializer.data, status=status.HTTP_200_OK)

class ContaUserUpdate(UpdateAPIView):
    """
    Atualiza uma ContaUser.
```

```
  """
  queryset = ContaUser.objects.all()
  serializer_class = ContaUserSerializer
  permission_classes = [IsAuthenticated]

  def put(self, request, pk):
    try:
      # Verificar se o usuário atual tem perfil associado
      if hasattr(request.user, 'usuario_perfil'):
        usuario = request.user.usuario_perfil

        # Buscar a conta associada ao usuário e ao ID fornecido
        perfil = ContaUser.objects.get(id=pk, usuario=usuario)

        # Serializar e validar os dados
        serializer = ContaUserSerializer(perfil, data=request.data, partial=True)
        if serializer.is_valid():
          serializer.save()
          return Response(serializer.data, status=status.HTTP_200_OK)

        return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)

      return Response({"erro": "Apenas usuários podem atualizar tratamentos."},
status=status.HTTP_403_FORBIDDEN)
    except ContaUser.DoesNotExist:
      return Response({"erro": "ContaUser não encontrada ou não pertence ao
usuário."}, status=status.HTTP_404_NOT_FOUND)


# ********** Usuários e Profissionais de Podologia **********

class CadastroPerfilView(APIView):
  """
  Endpoint para cadastro de novos usuários.
  """
  def post(self, request):
    serializer = PerfilSerializer(data=request.data)
    if serializer.is_valid():
      serializer.save()
      return Response({"message": "Usuário cadastrado com sucesso!"},
```

```
status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)


class LoginView(APIView):
    """
    Endpoint para login de usuários e geração de tokens JWT.
    """
    def post(self, request):
        serializer = LoginSerializer(data=request.data)
        if serializer.is_valid():
            user = serializer.validated_data['user']
            refresh = RefreshToken.for_user(user)

            # Verificar se o usuário é cliente ou profissional
            is_cliente = hasattr(user, 'usuario_perfil') and user.usuario_perfil is not None
            is_profissional = hasattr(user, 'profissional_podologia') and
user.profissional_podologia is not None

            return Response({
                'refresh': str(refresh),
                'access': str(refresh.access_token),
                'nome': user.nome,
                'email': user.email,
                'tipo_usuario': 'cliente' if is_cliente else 'profissional' if is_profissional else
'indefinido',
            }, status=status.HTTP_200_OK)

        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

class LogoutView(APIView):
    """
    Endpoint para logout de usuários, revogando o token de refresh.
    """
    permission_classes = [IsAuthenticated]

    def post(self, request):
        refresh_token = request.data.get('refresh')
        if not refresh_token:
            return Response({"error": "Token de refresh não fornecido."},
status=status.HTTP_400_BAD_REQUEST)
```

```
        try:
            token = RefreshToken(refresh_token)
            token.blacklist()  # Revoga o token adicionando-o à blacklist
            return Response({"message": "Logout realizado com sucesso!"},
status=status.HTTP_200_OK)
        except Exception as e:
            return Response({"error": "Falha ao revogar o token."},
status=status.HTTP_400_BAD_REQUEST)


class AtualizarSenha(APIView):
    """
    Endpoint para troca de senha.
    """
    permission_classes = [IsAuthenticated]

    def post(self, request):
        serializer = ChangePasswordSerializer(data=request.data, context={'request':
request})
        if serializer.is_valid():
            user = request.user
            user.set_password(serializer.validated_data['new_password'])
            user.save()
            return Response({"success": "Senha alterada com sucesso."},
status=status.HTTP_200_OK)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)


class UsuarioLogadoView(APIView):
    """
    Retorna informações sobre o usuário autenticado e permite atualização de
dados.
    """
    permission_classes = [IsAuthenticated]

    def get(self, request):
        """
        Retorna os dados do usuário logado.
        """
        usuario = request.user
```

```
    # Verifica se o usuário possui um perfil de cliente
    if hasattr(usuario, 'usuario_perfil'):
        usuario_perfil = usuario.usuario_perfil
        contas = ContaUser.objects.filter(usuario=usuario_perfil)
        contas_usuarios = ContaUserSerializer(contas, many=True).data

        # Serializa os dados do perfil e inclui contas
        usuario_detalhado = UsuarioSerializer(usuario_perfil).data
        usuario_detalhado['contas_usuario'] = contas_usuarios
        return Response(usuario_detalhado, status=status.HTTP_200_OK)

    # Verifica se o usuário é um profissional de podologia
    elif hasattr(usuario, 'profissional_podologia'):
        usuario_perfil = usuario.profissional_podologia
        usuario_detalhado = ProfissionalDePodologiaSerializer(usuario_perfil).data
        return Response(usuario_detalhado, status=status.HTTP_200_OK)

    # Retorna erro caso o usuário não tenha um perfil associado
    return Response(
        {"detail": "Acesso negado!"},
        status=status.HTTP_403_FORBIDDEN
    )

def put(self, request):
    """
    Atualiza os dados do usuário logado.
    """
    usuario = request.user

    # Atualiza dados do perfil de cliente
    if hasattr(usuario, 'usuario_perfil'):
        usuario_perfil = usuario.usuario_perfil
        return self._update_perfil(
            serializer_class=UsuarioSerializer,
            perfil=usuario_perfil,
            data=request.data
        )

    # Atualiza dados do profissional de podologia
    elif hasattr(usuario, 'profissional_podologia'):
```

```
            usuario_perfil = usuario.profissional_podologia
            return self._update_perfil(
                serializer_class=ProfissionalDePodologiaSerializer,
                perfil=usuario_perfil,
                data=request.data
            )

        # Retorna erro caso o usuário não tenha um perfil editável
        return Response(
            {"detail": "Acesso negado!"},
            status=status.HTTP_403_FORBIDDEN
        )

    def _update_perfil(self, serializer_class, perfil, data):
        """
        Atualiza os dados de um perfil utilizando o serializer apropriado.
        """
        serializer = serializer_class(perfil, data=data, partial=True)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_200_OK)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)


# ********** Usuários **********

class ListaAgendamentosView(APIView):
    """
    Exibe todos os agendamentos pendentes, confirmados e concluídos do usuário autenticado.
    Apenas usuários do tipo cliente podem acessar esta View.
    """
    permission_classes = [IsAuthenticated]

    def get(self, request):
        # Verifica se o usuário é do tipo cliente
        usuario = request.user
        if not hasattr(usuario, 'usuario_perfil'):
            return Response(
                {"detail": "Apenas clientes podem acessar esta funcionalidade."},
                status=status.HTTP_403_FORBIDDEN
```

```
            )

        usuario_perfil = usuario.usuario_perfil

        # Filtra os agendamentos pelo usuário autenticado e pelo status
        agendamentos_pendentes = Agendamento.objects.filter(
            usuario=usuario_perfil, status='pendente'
        )
        agendamentos_confirmados = Agendamento.objects.filter(
            usuario=usuario_perfil, status='confirmado'
        )
        agendamentos_concluidos = Agendamento.objects.filter(
            usuario=usuario_perfil, status='concluido'
        )

        # Serializa os agendamentos
        pendentes_serializados = AgendamentoSerializer(agendamentos_pendentes,
many=True).data
        confirmados_serializados =
AgendamentoSerializer(agendamentos_confirmados, many=True).data
        concluidos_serializados = AgendamentoSerializer(agendamentos_concluidos,
many=True).data

        # Retorna os dados organizados
        return Response(
            {
                "pendentes": pendentes_serializados,
                "confirmados": confirmados_serializados,
                "concluidos": concluidos_serializados,
            },
            status=status.HTTP_200_OK
        )


class CancelarAgendamentoView(APIView):
    """
    Permite que o usuário cancele um agendamento com status 'pendente' ou
'confirmado'.
    O cancelamento exige uma justificativa.
    Somente o usuário que criou o agendamento pode cancelá-lo.
    """
```

```
  permission_classes = [IsAuthenticated]

  def put(self, request, agendamento_id):
    usuario = request.user

    if not hasattr(usuario, 'usuario_perfil'):
      return Response(
        {"detail": "Apenas clientes podem acessar esta funcionalidade."},
        status=status.HTTP_403_FORBIDDEN
      )

    usuario_perfil = usuario.usuario_perfil

    # Busca o agendamento relacionado ao usuário
    agendamento = get_object_or_404(Agendamento, id=agendamento_id)

    # Verifica se o agendamento pertence ao usuário logado
    if agendamento.usuario != usuario.usuario_perfil:
      return Response(
        {"detail": "Você não pode cancelar um agendamento que não pertence a
você."},
        status=status.HTTP_403_FORBIDDEN
      )

    # Verifica se o status permite cancelamento
    if agendamento.status not in ['pendente', 'confirmado']:
      return Response(
        {"detail": "Apenas agendamentos pendentes ou confirmados podem ser
cancelados."},
        status=status.HTTP_400_BAD_REQUEST
      )

    # Obtém a justificativa do corpo da requisição
    justificativa = request.data.get('justificativa')
    if not justificativa:
      return Response(
        {"detail": "É necessário fornecer uma justificativa para cancelar o
agendamento."},
        status=status.HTTP_400_BAD_REQUEST
      )
```

```
        # Atualiza o status para 'cancelado' e salva a justificativa
        agendamento.status = 'cancelado'
        agendamento.justificativa = justificativa
        agendamento.save()

        # Retorna a confirmação
        return Response(
            {
                "detail": "Agendamento cancelado com sucesso.",
                "agendamento": CancelarAgendamentoSerializer(agendamento).data,
            },
            status=status.HTTP_200_OK
        )


class ProfissionalDePodologiaListView(APIView):
    """
    Lista os profissionais de podologia mais próximos com base na localização do usuário.
    Apenas acessível para usuários autenticados que não são profissionais.
    """
    permission_classes = [IsAuthenticated]

    def get(self, request):
        # Verifica se o usuário é um profissional de podologia
        if hasattr(request.user, 'profissional_podologia'):
            return Response(
                {"detail": "Apenas usuários não profissionais podem acessar este recurso."},
                status=status.HTTP_403_FORBIDDEN
            )

        # Obtém as coordenadas da requisição
        latitude = request.query_params.get('latitude')
        longitude = request.query_params.get('longitude')

        if not latitude or not longitude:
            return Response(
                {"detail": "Coordenadas latitude e longitude são obrigatórias."},
                status=status.HTTP_400_BAD_REQUEST
            )
```

```
        try:
            user_location = (float(latitude), float(longitude))
        except ValueError:
            return Response(
                {"detail": "Coordenadas inválidas."},
                status=status.HTTP_400_BAD_REQUEST
            )

        # Lista os profissionais de podologia próximos
        profissionais = ProfissionalDePodologia.objects.exclude(latitude__isnull=True,
longitude__isnull=True)
        profissionais_proximos = []

        for profissional in profissionais:
            profissional_location = (profissional.latitude, profissional.longitude)
            distance = geodesic(user_location, profissional_location).kilometers
            profissionais_proximos.append((profissional, distance))

        # Ordena por proximidade e aplica limite
        profissionais_proximos.sort(key=lambda x: x[1])
        limit = int(request.query_params.get('limit', 10))
        profissionais_proximos = profissionais_proximos[:limit]

        # Serializa os dados
        serializer = ProfissionalDePodologiaSerializer(
            [profissional[0] for profissional in profissionais_proximos], many=True
        )
        return Response(serializer.data, status=status.HTTP_200_OK)

class FeedbackNaoRespondidoView(APIView):
    """
    Lista todos os feedbacks não respondidos do usuário autenticado.
    """
    permission_classes = [IsAuthenticated]

    def get(self, request):
        usuario = request.user  # Obtém o usuário autenticado

        # Verifica se o usuário tem um perfil de cliente
        if not hasattr(usuario, 'usuario_perfil'):  # Verifica se o usuário tem o perfil de
```

```
cliente
        return Response(
          {"detail": "Apenas clientes podem acessar esta funcionalidade."},
          status=status.HTTP_403_FORBIDDEN
        )

      # Acessa o perfil do usuário autenticado
      usuario_perfil = usuario.usuario_perfil

      # Filtra feedbacks não respondidos para o usuário autenticado
      feedbacks_nao_respondidos = Feedback.objects.filter(usuario=usuario_perfil,
respondido=False)

      # Caso não haja feedbacks pendentes, retornamos uma mensagem
apropriada
      if not feedbacks_nao_respondidos.exists():
        return Response(
          {"detail": "Você não possui feedbacks pendentes de resposta."},
          status=status.HTTP_200_OK
        )

      # Serializa os feedbacks encontrados
      serializer = FeedbackSerializer(feedbacks_nao_respondidos, many=True)

      return Response(serializer.data, status=status.HTTP_200_OK)

class ResponderFeedbackView(APIView):
    """
    Permite ao usuário responder feedbacks não respondidos.
    """
    permission_classes = [IsAuthenticated]

    def post(self, request, feedback_id):
      usuario = request.user  # Obtém o usuário autenticado

      # Verifica se o usuário tem o perfil de cliente
      if not hasattr(usuario, 'usuario_perfil'):
        return Response(
          {"detail": "Apenas clientes podem acessar esta funcionalidade."},
          status=status.HTTP_403_FORBIDDEN
        )
```

```
    # Acessa diretamente o perfil do usuário
    usuario_perfil = usuario.usuario_perfil

    # Tenta buscar o feedback correspondente ao ID e ao usuário
    try:
        feedback = Feedback.objects.get(id=feedback_id, usuario=usuario_perfil,
respondido=False)
    except Feedback.DoesNotExist:
        return Response(
            {"detail": "Feedback não encontrado ou já respondido."},
            status=status.HTTP_404_NOT_FOUND
        )

    # Obtém os dados de nota e comentário da requisição
    nota = request.data.get('nota')
    comentario = request.data.get('comentario')

    # Verifica se nota e comentário foram enviados
    if nota is None or comentario is None:
        return Response(
            {"detail": "A nota e o comentário são obrigatórios para responder ao
feedback."},
            status=status.HTTP_400_BAD_REQUEST
        )

    # Valida a nota, garantindo que seja um número entre 1 e 5
    try:
        nota = int(nota)
    except ValueError:
        return Response(
            {"detail": "A nota deve ser um número inteiro entre 1 e 5."},
            status=status.HTTP_400_BAD_REQUEST
        )

    if not (1 <= nota <= 5):
        return Response(
            {"detail": "A nota deve ser entre 1 e 5."},
            status=status.HTTP_400_BAD_REQUEST
        )
```

```
        # Atualiza o feedback com a nota, comentário e marca como respondido
        feedback.nota = nota
        feedback.comentario = comentario
        feedback.respondido = True
        feedback.data = timezone.now()  # Atualiza a data para o momento da resposta
        feedback.save()

        # Serializa a resposta do feedback e retorna
        serializer = FeedbackSerializerResposta(feedback)
        return Response(serializer.data, status=status.HTTP_200_OK)


class CriarAgendamentoViewUser(APIView):
    permission_classes = [IsAuthenticated]

    def post(self, request):
        usuario = request.user

        # Verificar se o usuário logado é um perfil de usuário
        if not hasattr(usuario, 'usuario_perfil'):
            return Response(
                {"erro": "Acesso não permitido. Apenas usuários podem criar
agendamentos."},
                status=status.HTTP_403_FORBIDDEN
            )

        # Obter a instância do modelo Usuario

        usuario_perfil = usuario.usuario_perfil

        # Receber os dados do corpo da requisição
        conta_id = request.data.get("conta_id")
        podologo_id = request.data.get("podologo_id")
        servicos_ids = request.data.get("servicos_ids", [])
        data_agendamento = request.data.get("data")

        if not conta_id or not podologo_id or not servicos_ids or not
data_agendamento:
            return Response(
                {"erro": "Os campos 'conta_id', 'podologo_id', 'servicos_ids' e 'data' são
obrigatórios."},
```

```
            status=status.HTTP_400_BAD_REQUEST
        )

    # Verificar se a ContaUser pertence ao usuário logado
    conta = get_object_or_404(ContaUser, id=conta_id, usuario=usuario_perfil)

    if not podologo_id or not servicos_ids or not data_agendamento:
        return Response(
            {"erro": "Os campos 'podologo_id', 'servicos_ids' e 'data' são obrigatórios."},
            status=status.HTTP_400_BAD_REQUEST
        )

    # Verificar se o podólogo existe
    podologo = get_object_or_404(ProfissionalDePodologia, id=podologo_id)

    # Verificar se os serviços pertencem ao podólogo selecionado
    servicos = TratamentoPodologico.objects.filter(id__in=servicos_ids,
user=podologo)
    if len(servicos) != len(servicos_ids):
        return Response(
            {"erro": "Alguns serviços selecionados não pertencem ao podólogo
escolhido."},
            status=status.HTTP_400_BAD_REQUEST
        )

    agendamento_data = {
        "usuario": usuario_perfil.id,
        "conta": conta.id,
        "profissional": podologo.id,
        "servicos": servicos_ids,
        "data": data_agendamento,
        "status": "pendente",
    }
    agendamento_serializer =
AgendamentoSerializerCreateUser(data=agendamento_data, context={'request':
request})
    if agendamento_serializer.is_valid():
        agendamento = agendamento_serializer.save()
        return Response(
            {
                "mensagem": "Agendamento criado com sucesso!",
```

```
                "agendamento": agendamento_serializer.data
            },
            status=status.HTTP_201_CREATED
        )

    return Response(agendamento_serializer.errors,
status=status.HTTP_400_BAD_REQUEST)


class FeedbacksDoProfissionalView(APIView):
  """
  Exibe todos os feedbacks dos agendamentos do profissional de podologia
autenticado.
  """
  permission_classes = [IsAuthenticated]

  def get(self, request):
    usuario = request.user  # Obtém o usuário autenticado

    if not hasattr(usuario, 'profissional_podologia'):
      return Response(
        {"erro": "Acesso não permitido. Apenas usuários podem criar
agendamentos."},
        status=status.HTTP_403_FORBIDDEN
      )

    podologo = usuario.profissional_podologia.cpf

    # Verifica se o usuário tem o perfil de profissional de podologia
    try:
      profissional = ProfissionalDePodologia.objects.get(cpf=podologo)  #
Obtemos o profissional pelo usuário autenticado
    except ProfissionalDePodologia.DoesNotExist:
      return Response(
        {"detail": "Você não tem permissão de acesso!"},
        status=status.HTTP_403_FORBIDDEN
      )

    # Filtra os feedbacks relacionados aos agendamentos do profissional
autenticado
    feedbacks = Feedback.objects.filter(agendamento__profissional=profissional,
```

```
respondido=True)

    # Caso não haja feedbacks, retornamos uma mensagem
    if not feedbacks.exists():
        return Response(
            {"detail": "Não há feedbacks registrados para os agendamentos deste
profissional."},
            status=status.HTTP_404_NOT_FOUND
        )

    # Serializa os feedbacks encontrados
    serializer = FeedbackSerializer(feedbacks, many=True)

    # Retorna a resposta com os dados serializados
    return Response(serializer.data, status=status.HTTP_200_OK)


class CadastrarTratamentoView(APIView):
    """
    Endpoint para cadastro de tratamentos podológicos pelo profissional
autenticado.
    """
    permission_classes = [IsAuthenticated]

    def post(self, request):
        if hasattr(request.user, 'profissional_podologia'):
            profissional = request.user.profissional_podologia
            data = request.data

            # Aqui já atribuímos o usuário autenticado ao campo 'user'
            data['user'] = profissional.id  # Certifique-se de que está atribuindo o id
corretamente

            # Passando os dados para o serializer
            serializer = TratamentoPodologicoSerializerCreate(data=data)
            if serializer.is_valid():
                serializer.save()  # O 'user' será salvo automaticamente como o
profissional
                return Response({"message": "Tratamento cadastrado com sucesso!"},
status=status.HTTP_201_CREATED)
```

```
            return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

        return Response({"error": "Apenas profissionais de podologia podem cadastrar tratamentos."}, status=status.HTTP_403_FORBIDDEN)

class GerenciarTratamentosView(APIView):
    """
    Endpoint para listar, atualizar e deletar tratamentos do podólogo autenticado.
    """
    permission_classes = [IsAuthenticated]

    def get(self, request):
        """
        Lista todos os tratamentos do podólogo autenticado.
        """
        if hasattr(request.user, 'profissional_podologia'):
            profissional = request.user.profissional_podologia
            tratamentos = TratamentoPodologico.objects.filter(user=profissional)
            serializer = TratamentoPodologicoSerializer(tratamentos, many=True)
            return Response(serializer.data, status=status.HTTP_200_OK)
        return Response({"erro": "Apenas profissionais podem acessar tratamentos."}, status=status.HTTP_403_FORBIDDEN)


    def put(self, request, pk=None):
        """
        Atualiza um tratamento específico.
        """
        try:
            if hasattr(request.user, 'profissional_podologia'):
                profissional = request.user.profissional_podologia
                # Verificar se o tratamento pertence ao profissional
                tratamento = TratamentoPodologico.objects.get(pk=pk, user=profissional)
                serializer = TratamentoPodologicoSerializer(tratamento, data=request.data, partial=True)
                if serializer.is_valid():
                    serializer.save()
                    return Response(serializer.data, status=status.HTTP_200_OK)
                return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
            return Response({"erro": "Apenas profissionais podem atualizar
```

```
tratamentos."}, status=status.HTTP_403_FORBIDDEN)
      except TratamentoPodologico.DoesNotExist:
        return Response({"erro": "Tratamento não encontrado ou não pertence ao
profissional."}, status=status.HTTP_404_NOT_FOUND)

  def delete(self, request, pk=None):
    """
    Deleta um tratamento específico.
    """
    try:
      if hasattr(request.user, 'profissional_podologia'):
        profissional = request.user.profissional_podologia
        tratamento = TratamentoPodologico.objects.get(pk=pk, user=profissional)
        tratamento.delete()
        return Response({"mensagem": "Tratamento deletado com sucesso."},
status=status.HTTP_204_NO_CONTENT)
    except TratamentoPodologico.DoesNotExist:
      return Response({"erro": "Tratamento não encontrado ou não pertence ao
profissional."}, status=status.HTTP_404_NOT_FOUND)


class AtualizarStatusAgendamentoView(APIView):
  permission_classes = [IsAuthenticated]

  def put(self, request, agendamento_id):
    usuario = request.user

    # Verifica se o usuário é um profissional de podologia
    if not hasattr(usuario, 'profissional_podologia'):
      return Response(
        {"erro": "Acesso não permitido!"},
        status=status.HTTP_403_FORBIDDEN
      )

    podologo = usuario.profissional_podologia

    # Busca o agendamento associado ao ID
    agendamento = get_object_or_404(Agendamento, id=agendamento_id,
profissional=podologo)

    # Valida o novo status
```

```
        serializer = StatusAgendamentoSerializer(data=request.data)
        if not serializer.is_valid():
            return Response(
                {"detail": "Status inválido."},
                status=status.HTTP_400_BAD_REQUEST
            )

        # Atualiza o status
        agendamento.status = serializer.validated_data['status']
        agendamento.save()

        return Response(
            {"detail": "Status do agendamento atualizado com sucesso."},
            status=status.HTTP_200_OK
        )


# ********** Agendamentos **********

# class CriarAgendamentoView(CreateAPIView):
#     """
#     Endpoint para criar novos agendamentos.
#     """
#     permission_classes = [IsAuthenticated]
#     serializer_class = AgendamentoSerializer

#     def perform_create(self, serializer):
#         usuario = self.request.user
#         if usuario.podologo:
#             profissional = ProfissionalDePodologia.objects.get(user=usuario)
#             serializer.save(profissional=profissional, status="confirmado")
#         else:
#             paciente = Usuario.objects.get(usuario=usuario)
#             serializer.save(usuario=paciente, status="pendente")


class CriarAgendamentoView(APIView):
    """
    Permite que o podólogo autenticado crie um agendamento.
    O podólogo será automaticamente associado ao agendamento.
    """
```

```
    permission_classes = [IsAuthenticated]

    def post(self, request):
        """
        Cria um novo agendamento com o podólogo autenticado.
        """
        usuario = request.user  # Obtém o usuário autenticado

        # Verifica se o usuário é um profissional de podologia
        if not hasattr(usuario, 'profissional_podologia'):
            return Response(
                {"erro": "Acesso não permitido! Apenas podólogos podem criar
agendamentos."},
                status=status.HTTP_403_FORBIDDEN
            )

        podologo = usuario.profissional_podologia

        # Adiciona o podólogo ao payload e valida os dados
        data = request.data.copy()
        data['profissional'] = podologo.id

        # Verifica se a conta está relacionada ao usuário cliente
        conta_id = data.get('conta')
        if not conta_id:
            return Response(
                {"erro": "A conta é obrigatória para criar um agendamento."},
                status=status.HTTP_400_BAD_REQUEST
            )

        # Verifica se a conta pertence ao cliente
        try:
            conta = ContaUser.objects.get(id=conta_id)
        except ContaUser.DoesNotExist:
            return Response(
                {"erro": "A conta selecionada não existe."},
                status=status.HTTP_404_NOT_FOUND
            )

        # Garante que a conta esteja associada ao usuário cliente
        if conta.usuario != data.get('usuario'):
```

```
            return Response(
                {"erro": "A conta selecionada não pertence ao usuário cliente informado."},
                status=status.HTTP_400_BAD_REQUEST
            )

        # Valida e cria o agendamento
        serializer = AgendamentoSerializerPodologo(data=data)
        if serializer.is_valid():
            agendamento = serializer.save()
            return Response(
                {
                    "detail": "Agendamento criado com sucesso.",
                    "agendamento": AgendamentoSerializerPodologo(agendamento).data,
                },
                status=status.HTTP_201_CREATED
            )

        return Response(
            {"detail": "Erro ao criar o agendamento.", "errors": serializer.errors},
            status=status.HTTP_400_BAD_REQUEST
        )

class BuscarClienteView(APIView):
    """
    Endpoint dedicado para buscar clientes por e-mail.
    """
    permission_classes = [IsAuthenticated]

    def get(self, request, email):
        if not email:
            return Response({"error": "O parâmetro 'email' é obrigatório."},
status=status.HTTP_400_BAD_REQUEST)

        try:
            # Busca o cliente pelo e-mail
            cliente = Usuario.objects.get(email=email)

            # Busca as contas associadas ao cliente
            contas = ContaUser.objects.filter(usuario=cliente)

            # Serializa os dados do cliente e das contas
```

```
            cliente_data = UsuarioSerializer(cliente).data
            contas_data = ContaUserSerializer(contas, many=True).data

            # Combina os dados em uma única resposta
            response_data = {
                "usuario": cliente_data,
                "contas": contas_data
            }

            return Response(response_data, status=status.HTTP_200_OK)
        except Usuario.DoesNotExist:
            return Response({"error": "Cliente não encontrado."},
status=status.HTTP_404_NOT_FOUND)
```

```
class ConfirmarAgendamentoView(UpdateAPIView):
    """
    Endpoint para confirmar agendamentos (apenas podólogos).
    """
    permission_classes = [IsAuthenticated]
    serializer_class = AgendamentoSerializer

    def get_queryset(self):
        return Agendamento.objects.filter(profissional__user=self.request.user)

    def perform_update(self, serializer):
        serializer.save(status="confirmado")


class ListarAgendamentosView(APIView):
    """
    Endpoint para listar agendamentos do podólogo autenticado.
    """
    permission_classes = [IsAuthenticated]
```

```
  def get(self, request):
    """
    Lista todos os agendamentos do podólogo autenticado.
    """
    usuario = request.user
    if not hasattr(usuario, 'profissional_podologia'):
      return Response(
        {"erro": "Acesso não permitido!"},
        status=status.HTTP_403_FORBIDDEN
      )


    profissional = usuario.profissional_podologia
    agendamentos = Agendamento.objects.filter(profissional=profissional)
    serializer = AgendamentoSerializerList(agendamentos, many=True)
    return Response(serializer.data, status=status.HTTP_200_OK)




class AtualizarAnamneseView(APIView):
  """
  Endpoint para atualizar a Anamnese de uma ContaUser.
  """


  # permission_classes = [IsAuthenticated]

  def get(self, request, conta_user_id):
    conta_user = get_object_or_404(ContaUser, id=conta_user_id)
    anamnese = conta_user.anamnese
    serializer = AnamneseSerializer(anamnese)
    return Response(serializer.data, status=status.HTTP_200_OK)


  def put(self, request, conta_user_id):

    usuario = request.user

    if not hasattr(usuario, 'profissional_podologia'):
```

```
            return Response(
                {"erro": "Acesso não permitido!"},
                status=status.HTTP_403_FORBIDDEN
            )

        # Busca a ContaUser pelo ID
        conta_user = get_object_or_404(ContaUser, id=conta_user_id)

        # Verifica se a ContaUser tem uma Anamnese associada
        anamnese = conta_user.anamnese
        if not anamnese:
            return Response({"detail": "Nenhuma anamnese associada a esta conta."},
status=status.HTTP_404_NOT_FOUND)

        # Serializa os dados recebidos
        serializer = AnamneseSerializer(anamnese, data=request.data)

        # Valida os dados e salva
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_200_OK)

        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

    def patch(self, request, conta_user_id):
        """
        Atualiza parcialmente os dados da Anamnese.
        """
        usuario = request.user


        if not hasattr(usuario, 'profissional_podologia'):
            return Response(
                {"erro": "Acesso não permitido!"},
                status=status.HTTP_403_FORBIDDEN
            )


        conta_user = get_object_or_404(ContaUser, id=conta_user_id)
        anamnese = conta_user.anamnese
        if not anamnese:
```

```
        return Response({"detail": "Nenhuma anamnese associada a esta conta."},
status=status.HTTP_404_NOT_FOUND)

    serializer = AnamneseSerializer(anamnese, data=request.data, partial=True)
    if serializer.is_valid():
      serializer.save()
      return Response(serializer.data, status=status.HTTP_200_OK)

    return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)


class ListaContaUsersAgendamentosView(APIView):
  """
  Endpoint para listar ContaUsers dos agendamentos de um Podólogo
autenticado.
  """
  permission_classes = [IsAuthenticated]

  def get(self, request):
    # Verifica se o usuário autenticado é um podólogo
    if hasattr(request.user, 'profissional_podologia'):
      profissional = request.user.profissional_podologia

      # Busca os agendamentos associados ao podólogo
      agendamentos = Agendamento.objects.filter(profissional=profissional)

      # Extrai os ContaUsers únicos dos agendamentos
      conta_users =
ContaUser.objects.filter(agendamento__in=agendamentos).distinct()

      # Serializa os ContaUsers
      serializer = ContaUserSerializer(conta_users, many=True)
      return Response(serializer.data, status=status.HTTP_200_OK)

    return Response({"detail": "Apenas profissionais têm acesso a este recurso."},
status=status.HTTP_403_FORBIDDEN)



class DetalharPodologo(APIView):
```

```
  def get(self, request, pk):
    try:
      podologo = ProfissionalDePodologia.objects.get(id=pk)
      serializer = ProfissionalDePodologiaSerializer(podologo)
      return Response(serializer.data, status=status.HTTP_200_OK)
    except ProfissionalDePodologia.DoesNotExist:
      return Response(
        {"error": "Profissional de Podologia não encontrado."},
        status=status.HTTP_404_NOT_FOUND
      )


class MeusAgendamentosPorData(APIView):
  permission_classes = [IsAuthenticated]

  def get(self, request, data):
    if hasattr(request.user, 'profissional_podologia'):
      profissional = request.user.profissional_podologia
      data = datetime.datetime.strptime(data, '%Y-%m-%d').date()
      agendamentos = Agendamento.objects.filter(data=data,
profissional=profissional)
      serializer = AgendamentoSerializerList(agendamentos, many=True)
      return Response(serializer.data, status=status.HTTP_200_OK)

class ListarMeusClientes(APIView):
  permission_classes = [IsAuthenticated]

  def get(self, request):
    profissional = request.user.profissional_podologia
    agendamentos = Agendamento.objects.filter(profissional=profissional)
    clientes = Usuario.objects.filter(agendamentos__in=agendamentos).distinct()
    serializer = UsuarioSerializer(clientes, many=True)
    return Response(serializer.data)


class ContasUserPorUsuario(APIView):
  permission_classes = [IsAuthenticated]

  def get(self, request, conta_id):
    if hasattr(request.user, 'profissional_podologia'):
      cliente = Usuario.objects.get(id=conta_id)
```

```
        contas = ContaUser.objects.filter(usuario=cliente)
        serializer = ContaUserSerializer(contas, many=True)
        return Response(serializer.data, status=status.HTTP_200_OK)


from django.contrib.auth import get_user_model
from django.contrib.auth.tokens import default_token_generator
from django.utils.crypto import get_random_string

User = get_user_model()



class PasswordResetView(APIView):
  """
  View to handle forgotten password.
  Sends a temporary password to the user's registered email.
  """
  def post(self, request, *args, **kwargs):
    email = request.data.get("email")
    if not email:
      return Response({"error": "Email field is required."},
status=status.HTTP_400_BAD_REQUEST)

    try:
      user = User.objects.get(email=email)
    except User.DoesNotExist:
      return Response({"error": "No user found with this email address."},
status=status.HTTP_404_NOT_FOUND)

    temporary_password = get_random_string(length=7,
allowed_chars='0123456789')
    user.set_password(temporary_password)
    user.save()

    subject = "Redefinição de Senha"
    message = f"Olá, {user.nome}. Sua senha temporária é:
{temporary_password}\nPor favor, altere sua senha após o próximo login."
    from_email = "podologia@somostodosnerds.com.br"

    send_mail(subject, message, from_email, [user.email])
```

```
    return Response({"message": "Uma senha temporária foi enviada para o seu email."}, status=status.HTTP_200_OK)


logger = logging.getLogger(__name__)

class NovoTrabalho(APIView):
  """
  Permite que o podólogo autenticado crie um agendamento.
  O podólogo será automaticamente associado ao agendamento.
  """
  permission_classes = [IsAuthenticated]

  def post(self, request):
    logger.info("Recebido POST para criar agendamento com dados: %s", request.data)
    usuario = request.user

    # Verifica se o usuário é um profissional de podologia
    if not hasattr(usuario, 'profissional_podologia'):
      logger.warning("Acesso negado: Usuário não é um podólogo.")
      return Response(
        {"erro": "Acesso não permitido! Apenas podólogos podem criar agendamentos."},
        status=status.HTTP_403_FORBIDDEN
      )

    podologo = usuario.profissional_podologia
    data = request.data.copy()
    data['profissional'] = podologo.id

    # Verifica se a conta está relacionada ao usuário cliente
    conta_id = data.get('conta')
    if not conta_id:
      logger.error("Conta não fornecida no payload.")
      return Response(
        {"erro": "A conta é obrigatória para criar um agendamento."},
        status=status.HTTP_400_BAD_REQUEST
      )
```

```
    try:
        conta = ContaUser.objects.get(id=conta_id)
    except ContaUser.DoesNotExist:
        logger.error("Conta com ID %s não existe.", conta_id)
        return Response(
            {"erro": "A conta selecionada não existe."},
            status=status.HTTP_404_NOT_FOUND
        )

    usuario_id = data.get('usuario')
    logger.info("Comparando conta.usuario.id (%s) com usuario_id (%s)",
conta.usuario.id, usuario_id)
    if conta.usuario.id != int(usuario_id):
        logger.error("Conta pertence ao usuário %s, mas foi fornecido usuário %s.",
conta.usuario.id, usuario_id)
        return Response(
            {"erro": "A conta selecionada não pertence ao usuário cliente informado."},
            status=status.HTTP_400_BAD_REQUEST
        )

    # Valida e cria o agendamento
    serializer = AgendamentoSerializerPodologo(data=data)
    if serializer.is_valid():
        agendamento = serializer.save()
        logger.info("Agendamento criado com sucesso: %s", agendamento.id)
        return Response(
            {
                "detail": "Agendamento criado com sucesso.",
                "agendamento": AgendamentoSerializerPodologo(agendamento).data,
            },
            status=status.HTTP_201_CREATED
        )

    logger.error("Validação do serializer falhou: %s", serializer.errors)
    return Response(
        {"detail": "Erro ao criar o agendamento.", "errors": serializer.errors},
        status=status.HTTP_400_BAD_REQUEST
    )
```

podologia/podologia/settings.py

```
import os
from pathlib import Path
from datetime import timedelta


# Build paths inside the project like this: BASE_DIR / 'subdir'.
BASE_DIR = Path(__file__).resolve().parent.parent


# Quick-start development settings - unsuitable for production
# See https://docs.djangoproject.com/en/5.0/howto/deployment/checklist/

# SECURITY WARNING: keep the secret key used in production secret!
SECRET_KEY = 'django-insecure-@opsgoke(_)9h9sje+#ya@%$kmslhin)-
ed%s=2j!w5!h(^zn_'

# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = True

ALLOWED_HOSTS = ['localhost', '127.0.0.1', '216.39.249.41',
'podologia.somostodosnerds.com.br']


# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'core',
    'widget_tweaks',
    "corsheaders",
    'rest_framework',
    'rest_framework_simplejwt',
    'rest_framework.authtoken',
    'rest_framework_simplejwt.token_blacklist',
```

```
    'drf_yasg',
]

MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    "corsheaders.middleware.CorsMiddleware",
    "django.middleware.common.CommonMiddleware",
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]

ROOT_URLCONF = 'podologia.urls'

TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [BASE_DIR / 'templates'],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]

WSGI_APPLICATION = 'podologia.wsgi.application'

# REST_FRAMEWORK = {
#     'DEFAULT_RENDERER_CLASSES': [
#         'rest_framework.renderers.JSONRenderer',
#     ],
#     'DEFAULT_AUTHENTICATION_CLASSES': [
#         'rest_framework_simplejwt.authentication.JWTAuthentication',
#     ],
# }
```

```
REST_FRAMEWORK = {
  'DEFAULT_AUTHENTICATION_CLASSES': (
    'rest_framework_simplejwt.authentication.JWTAuthentication',
  ),
}


SIMPLE_JWT = {
  'ACCESS_TOKEN_LIFETIME': timedelta(days=30),
  'REFRESH_TOKEN_LIFETIME': timedelta(days=90),
  # 'ROTATE_REFRESH_TOKENS': False,
  'AUTH_TOKEN_CLASSES': ('rest_framework_simplejwt.tokens.AccessToken',),
  'BLACKLIST_AFTER_ROTATION': True,
}

# Database
# https://docs.djangoproject.com/en/5.0/ref/settings/#databases

DATABASES = {
  'default': {
    'ENGINE': 'django.db.backends.postgresql',
    'NAME': 'banco13db',
    'USER': 'user13',
    'PASSWORD': 'Samuca10x',
    'HOST': 'localhost',
    'PORT': '5432',
  }
}


# DATABASES = {
#    "default": {
#       "ENGINE": "django.db.backends.sqlite3",
#       "NAME": "mydatabase",
#    }
# }

# Password validation
# https://docs.djangoproject.com/en/5.0/ref/settings/#auth-password-validators
```

```
AUTH_PASSWORD_VALIDATORS = [
  {
    'NAME':
'django.contrib.auth.password_validation.UserAttributeSimilarityValidator',
  },
  {
    'NAME': 'django.contrib.auth.password_validation.MinimumLengthValidator',
  },
  {
    'NAME':
'django.contrib.auth.password_validation.CommonPasswordValidator',
  },
  {
    'NAME': 'django.contrib.auth.password_validation.NumericPasswordValidator',
  },
]


AUTH_USER_MODEL = 'core.Perfil'

# Internationalization
# https://docs.djangoproject.com/en/5.0/topics/i18n/

LANGUAGE_CODE = 'pt-br'

TIME_ZONE = 'America/Sao_Paulo'

USE_I18N = True

USE_TZ = True



# Diretório onde o Django buscará arquivos estáticos
STATIC_URL = '/static/'

# Diretório onde os arquivos estáticos serão coletados para produção
STATIC_ROOT = os.path.join(BASE_DIR, 'staticfiles')

# Diretórios adicionais onde o Django buscará arquivos estáticos durante o
desenvolvimento
```

```
STATICFILES_DIRS = [
   os.path.join(BASE_DIR, 'static'),  # Defina a pasta onde seus arquivos estáticos estão localizados
]

MEDIA_URL = '/media/'
MEDIA_ROOT = BASE_DIR / 'media'

# URL de redirecionamento para login
LOGIN_URL = '/login/'

#AUTH_USER_MODEL = 'core.Usuario'

# Default primary key field type
# https://docs.djangoproject.com/en/5.0/ref/settings/#default-auto-field

# configuração do e-mail para recuperação
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
EMAIL_HOST = 'mail.somostodosnerds.com.br'
EMAIL_PORT = 587
EMAIL_USE_TLS = True
EMAIL_HOST_USER = 'podologia@somostodosnerds.com.br'
EMAIL_HOST_PASSWORD = 'podologia@10x'
DEFAULT_FROM_EMAIL = 'Podologia <podologia@somostodosnerds.com.br>'


DEFAULT_AUTO_FIELD = 'django.db.models.BigAutoField'


CORS_ALLOW_ALL_ORIGINS = True
```

podologia/podologia/urls.py

```
from django.conf.urls.static import static
from django.contrib import admin
from django.urls import path, include
from podologia import settings

# Padrões de URL para o projeto
urlpatterns = [
```

```
    path('admin/', admin.site.urls),
    path('api/', include('core.urls')),

]

urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
urlpatterns += static(settings.STATIC_URL, document_root=settings.STATIC_ROOT)
```

podologia/static/css/style.css

```
/* Configuração global para o corpo da página */
body {
    font-family: Arial, sans-serif; /* Define a fonte padrão como Arial, com fallback sans-serif */
    margin: 0;                /* Remove margens padrão */
    padding: 0;               /* Remove preenchimentos padrão */
    background-color: #f4f4f9;    /* Define uma cor de fundo suave para o corpo da página */
}

/* Estilo para o cabeçalho da página */
header {
    background-color: #007BFF;    /* Define o fundo do cabeçalho como azul */
    color: white;             /* Define a cor do texto como branca */
    padding: 1rem;            /* Adiciona preenchimento em torno do conteúdo do cabeçalho */
    text-align: center;       /* Centraliza o texto */
}

/* Estilo para os links de navegação dentro do cabeçalho */
header nav a {
    color: white;             /* Define a cor dos links como branca */
    margin: 0 1rem;           /* Adiciona espaçamento horizontal entre os links */
    text-decoration: none;    /* Remove sublinhado dos links */
}

/* Estilo para o rodapé da página */
footer {
    background-color: #333;       /* Define o fundo do rodapé como cinza escuro */
    color: white;             /* Define a cor do texto como branca */
```

```
  padding: 1rem;              /* Adiciona preenchimento ao redor do conteúdo do
rodapé */
  text-align: center;         /* Centraliza o texto do rodapé */
  margin-top: 2rem;           /* Adiciona espaço acima do rodapé */
}

/* Estilo para contêineres de páginas específicas (login, dashboard, relatórios,
feedbacks, agendamentos) */
.login-container, .dashboard-container, .relatorios-container, .feedbacks-
container, .agendamentos-container {
  max-width: 600px;           /* Limita a largura máxima dos contêineres */
  margin: 2rem auto;          /* Centraliza o contêiner e adiciona margem vertical */
  padding: 2rem;              /* Adiciona preenchimento interno */
  background-color: white;      /* Define o fundo do contêiner como branco */
  box-shadow: 0 0 10px rgba(0, 0, 0, 0.1); /* Adiciona uma leve sombra para
destaque */
}

/* Estilo para botões */
button {
  background-color: #007BFF;    /* Define a cor de fundo do botão como azul */
  color: white;               /* Define a cor do texto do botão como branca */
  padding: 0.5rem 1rem;       /* Adiciona preenchimento ao redor do texto do
botão */
  border: none;               /* Remove borda padrão */
  cursor: pointer;            /* Define o cursor como "pointer" para indicar
interatividade */
}

/* Estilo de hover para botões */
button:hover {
  background-color: #0056b3;    /* Muda o fundo para um azul mais escuro ao
passar o mouse */
}
```

podologia/static/main.js

```
// Aguarda o carregamento completo do DOM antes de executar o código
document.addEventListener('DOMContentLoaded', function () {
  // Recupera o token de autenticação JWT do armazenamento local
```

```
  const token = localStorage.getItem('access');

  /**
   * Função assíncrona para carregar os relatórios de progresso do usuário.
   * Faz uma requisição à API para obter dados de progresso e exibe os resultados
   * na página, dentro do elemento com o ID 'relatorios-list'.
   */
  async function loadRelatorios() {
    const response = await fetch('/api/relatorios/', {
      headers: { 'Authorization': `Bearer ${token}` } // Inclui o token de
autenticação
    });
    const data = await response.json(); // Extrai os dados JSON da resposta
    const relatoriosList = document.getElementById('relatorios-list');

    // Mapeia os dados recebidos para HTML e os insere no elemento
    relatoriosList.innerHTML = data.map(item =>
      `<div class="list-group-item">
        <h5>Data: ${item.data}</h5>
        <p>Progresso: ${item.progresso}%</p>
        <p>Recomendações: ${item.recomendacoes}</p>
      </div>`
    ).join(''); // join('') remove as vírgulas entre os itens do array
  }

  /**
   * Função assíncrona para carregar os feedbacks fornecidos pelos responsáveis.
   * Faz uma requisição à API para obter os feedbacks e exibe os resultados
   * dentro do elemento com o ID 'feedbacks-list'.
   */
  async function loadFeedbacks() {
    const response = await fetch('/api/feedbacks/', {
      headers: { 'Authorization': `Bearer ${token}` } // Inclui o token de
autenticação
    });
    const data = await response.json(); // Extrai os dados JSON da resposta
    const feedbacksList = document.getElementById('feedbacks-list');

    // Mapeia os dados recebidos para HTML e os insere no elemento
    feedbacksList.innerHTML = data.map(item =>
      `<div class="list-group-item">
```

```
            <h5>Data: ${item.data}</h5>
            <p>${item.conteudo}</p>
          </div>`
      ).join('');
    }

    /**
     * Função assíncrona para carregar a lista de agendamentos do usuário.
     * Faz uma requisição à API para obter dados de agendamentos e exibe os resultados
     * dentro do elemento com o ID 'agendamentos-list'.
     */
    async function loadAgendamentos() {
      const response = await fetch('/api/agendamentos/', {
        headers: { 'Authorization': `Bearer ${token}` } // Inclui o token de autenticação
      });
      const data = await response.json(); // Extrai os dados JSON da resposta
      const agendamentosList = document.getElementById('agendamentos-list');

      // Mapeia os dados recebidos para HTML e os insere no elemento
      agendamentosList.innerHTML = data.map(item =>
        `<div class="list-group-item">
            <h5>Data do Agendamento: ${item.data_agendamento}</h5>
            <p>Notificação em: ${item.data_notificacao}</p>
          </div>`
      ).join('');
    }

    // Verifica a presença dos elementos no DOM e carrega os dados específicos para cada página
    if (document.getElementById('relatorios-list')) loadRelatorios();
    if (document.getElementById('feedbacks-list')) loadFeedbacks();
    if (document.getElementById('agendamentos-list')) loadAgendamentos();
});
```