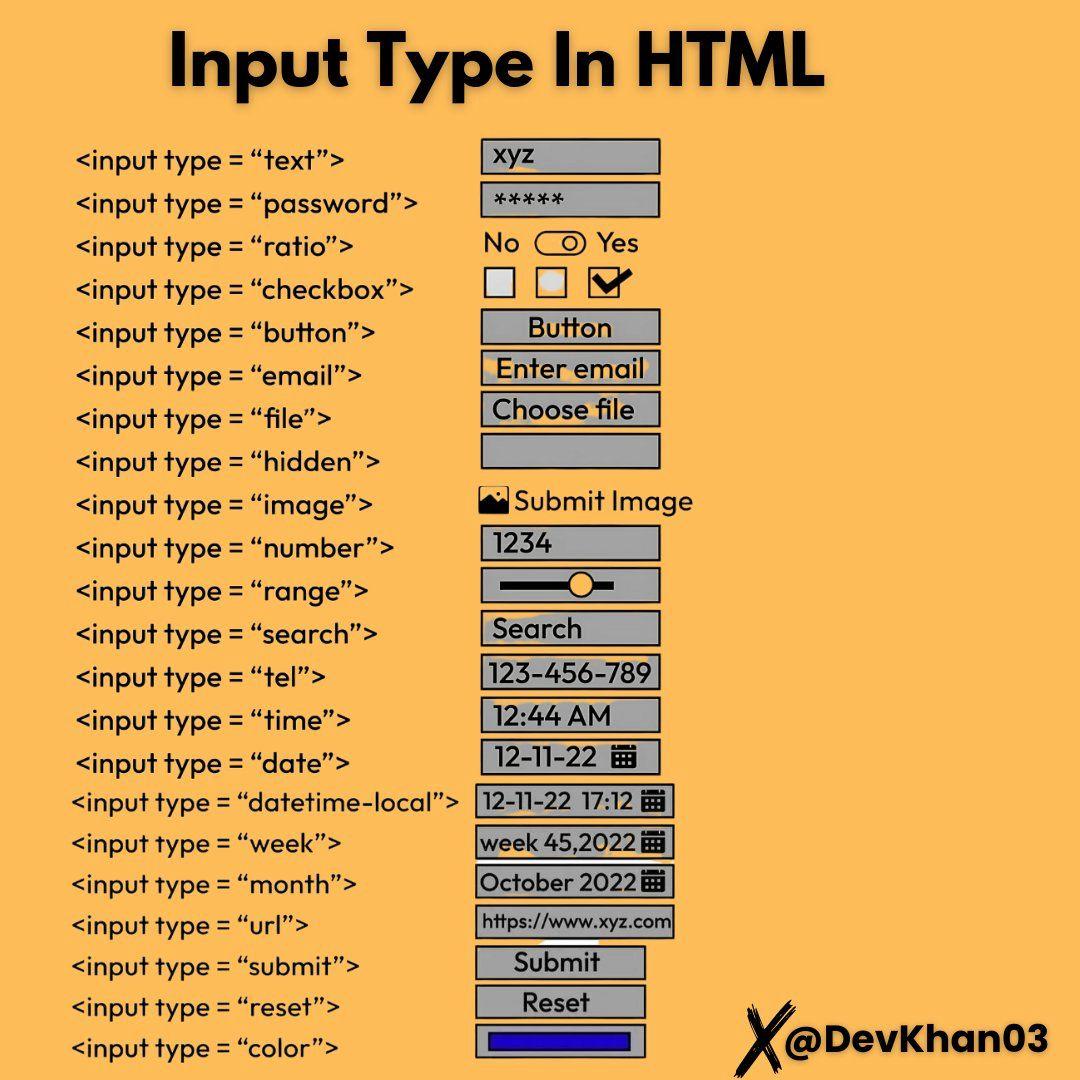
José Alfredo F. Costa – Outubro de 2024

Outras informações – complementares – sobre HMTL e uso do Python

Veja a imagem ->



Ela apresenta uma lista de diferentes tipos de campos de entrada de dados disponíveis no HTML. Esses campos são definidos com a tag <input>, acompanhada do atributo type, que define qual o tipo de dado que o campo vai capturar. Vou explicar cada um desses tipos e mostrar exemplos práticos de como você pode utilizá-los em formulários HTML.

**Explicação dos Tipos de Input**

1. **<input type="text">**  
   Usado para capturar texto simples.  
   Exemplo: Nome de usuário, endereço.
2. **<input type="password">**  
   Utilizado para capturar senhas, onde os caracteres são mascarados (exibidos como asteriscos ou pontos).  
   Exemplo: Senha de login.
3. **<input type="radio">**  
   Campos de opção onde o usuário pode escolher apenas uma entre várias alternativas.  
   Exemplo: Gênero (Masculino/Feminino/Outro).
4. **<input type="checkbox">**  
   Permite que o usuário marque várias opções.  
   Exemplo: Assinar uma newsletter.
5. **<input type="button">**  
   Exibe um botão que pode executar uma função definida por JavaScript quando clicado.
6. **<input type="email">**  
   Para captura de endereços de e-mail, com validação automática do formato.  
   Exemplo: E-mail de contato.
7. **<input type="file">**  
   Usado para fazer upload de arquivos a partir do dispositivo do usuário.
8. **<input type="hidden">**  
   Esconde um campo de entrada do usuário, mas seus dados ainda são enviados quando o formulário é submetido.
9. **<input type="image">**  
   Insere uma imagem que pode funcionar como botão de envio do formulário.
10. **<input type="number">**  
    Campo para números, com controle para aumentar ou diminuir o valor.  
    Exemplo: Quantidade de itens.
11. **<input type="range">**  
    Exibe um controle deslizante para selecionar um valor dentro de um intervalo.  
    Exemplo: Volume de um player.
12. **<input type="search">**  
    Campo específico para busca de texto.  
    Exemplo: Barra de pesquisa.
13. **<input type="tel">**  
    Campo para entrada de números de telefone, com validação de formato.  
    Exemplo: Número de contato.
14. **<input type="time">**  
    Campo para seleção de um horário específico (horas e minutos).
15. **<input type="date">**  
    Exibe um seletor de data para o usuário escolher um dia, mês e ano.
16. **<input type="datetime-local">**  
    Permite selecionar tanto uma data quanto uma hora em um único campo.
17. **<input type="week">**  
    Para selecionar uma semana específica do ano.
18. **<input type="month">**  
    Campo para escolher um mês e ano.
19. **<input type="url">**  
    Campo para inserir URLs, com validação automática do formato.  
    Exemplo: Endereço de um site.
20. **<input type="submit">**  
    Botão de envio do formulário, para submeter todos os dados preenchidos.
21. **<input type="reset">**  
    Botão que redefine o formulário, limpando todos os dados preenchidos.
22. **<input type="color">**  
    Abre uma paleta de cores para o usuário selecionar uma cor.  
    Exemplo: Selecionar uma cor de fundo.

**Exemplos de Formulários com Esses Inputs**

**Exemplo 1: Formulário Básico de Registro**

***html***

<form>

Nome: <input type="text" name="nome"><br>

E-mail: <input type="email" name="email"><br>

Senha: <input type="password" name="senha"><br>

Data de Nascimento: <input type="date" name="data\_nascimento"><br>

Cor favorita: <input type="color" name="cor"><br>

<input type="submit" value="Registrar">

</form>

**Exemplo 2: Seleção de Opções (Checkbox e Radio)**

***html***

<form>

Gênero:<br>

<input type="radio" name="genero" value="masculino"> Masculino<br>

<input type="radio" name="genero" value="feminino"> Feminino<br>

<input type="radio" name="genero" value="outro"> Outro<br>

Interesses:<br>

<input type="checkbox" name="interesses" value="esportes"> Esportes<br>

<input type="checkbox" name="interesses" value="musica"> Música<br>

<input type="checkbox" name="interesses" value="arte"> Arte<br>

<input type="submit" value="Enviar">

</form>

**Exemplo 3: Controle Deslizante e Númerico**

***html***

<form>

Escolha sua idade: <input type="number" name="idade" min="0" max="100"><br>

Escolha o volume: <input type="range" name="volume" min="0" max="100"><br>

<input type="submit" value="Enviar">

</form>

**Exemplo 4: Upload de Arquivo**

***html***

<form>

Envie seu currículo: <input type="file" name="curriculo"><br>

<input type="submit" value="Enviar">

</form>

**Exemplo 5: Seleção de Data e Hora**

***html***

<form>

Escolha uma data: <input type="date" name="data"><br>

Escolha uma hora: <input type="time" name="hora"><br>

<input type="submit" value="Agendar">

</form>

Esses são apenas alguns exemplos de como você pode utilizar os diferentes tipos de campos de entrada no HTML para capturar dados variados dos usuários. Cada tipo oferece uma forma única de interação, seja através de seletores, botões ou caixas de texto.

Vamos relembrar o projeto de gestão de consultas em uma clínica. Temos usos como um cadastro de um paciente e de um médico. Vamos desenvolver um formulário para cadastro de uma consulta, que terá um paciente, um médico, um dia e horário.

Vamos desenvolver um sistema simples de formulários HTML para o **cadastro de pacientes**, **cadastro de médicos** e o **agendamento de consultas** em uma clínica. Vamos discutir detalhes e a estilização usando CSS para melhorar o visual dos formulários.

**Estrutura dos Arquivos**

1. **index.html** – Será o arquivo principal onde colocaremos os formulários de cadastro de pacientes, médicos e consultas.
2. **style.css** – Será o arquivo de estilos (CSS) para melhorar o design e visual dos formulários.

**1. Cadastro de Pacientes**

Vamos começar com o formulário de **cadastro de pacientes**, onde coletamos informações básicas como nome, e-mail, telefone, data de nascimento e endereço.

**Formulário de Cadastro de Pacientes**

***html***

<!-- index.html -->

<!DOCTYPE html>

<html lang="pt-br">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Cadastro de Pacientes e Médicos</title>

<link rel="stylesheet" href="style.css">

</head>

<body>

<h2>Cadastro de Paciente</h2>

<form action="/submit\_patient" method="post">

<label for="nome">Nome:</label>

<input type="text" id="nome" name="nome" required><br><br>

<label for="email">E-mail:</label>

<input type="email" id="email" name="email" required><br><br>

<label for="telefone">Telefone:</label>

<input type="tel" id="telefone" name="telefone" required><br><br>

<label for="data\_nascimento">Data de Nascimento:</label>

<input type="date" id="data\_nascimento" name="data\_nascimento" required><br><br>

<label for="endereco">Endereço:</label>

<input type="text" id="endereco" name="endereco" required><br><br>

<input type="submit" value="Cadastrar Paciente">

</form>

</body>

</html>

**Explicação**

1. **Estrutura básica HTML**: O código contém um formulário simples com campos para nome, email, telefone, data de nascimento e endereço.
2. **Campo obrigatório (required)**: Todos os campos possuem o atributo required, o que significa que o usuário não pode enviar o formulário sem preenchê-los.
3. **Tipo de input**: Cada campo de entrada usa um tipo apropriado como email, tel (para telefone) e date (para data de nascimento).

**2. Cadastro de Médicos**

Agora vamos criar o formulário de **cadastro de médicos**, onde coletamos o nome, CRM (número de registro médico) e especialidade do médico.

**Formulário de Cadastro de Médicos**

***html***

<h2>Cadastro de Médico</h2>

<form action="/submit\_doctor" method="post">

<label for="nome\_medico">Nome:</label>

<input type="text" id="nome\_medico" name="nome\_medico" required><br><br>

<label for="crm">CRM:</label>

<input type="text" id="crm" name="crm" required><br><br>

<label for="especialidade">Especialidade:</label>

<input type="text" id="especialidade" name="especialidade" required><br><br>

<input type="submit" value="Cadastrar Médico">

</form>

**Explicação**

1. **Campo CRM**: Esse campo captura o número do registro do médico.
2. **Campo Especialidade**: Aqui o médico insere sua especialidade, como Cardiologia, Ortopedia, etc.

**3. Agendamento de Consultas**

Para o **agendamento de consultas**, criamos um formulário que associa um paciente a um médico, além de incluir a data e o horário da consulta.

**Formulário de Cadastro de Consulta**

***html***

<h2>Agendamento de Consulta</h2>

<form action="/submit\_appointment" method="post">

<label for="paciente">Paciente:</label>

<select id="paciente" name="paciente" required>

<option value="1">Paciente 1</option>

<option value="2">Paciente 2</option>

<!-- Mais pacientes aqui -->

</select><br><br>

<label for="medico">Médico:</label>

<select id="medico" name="medico" required>

<option value="1">Dr. Médico 1</option>

<option value="2">Dr. Médico 2</option>

<!-- Mais médicos aqui -->

</select><br><br>

<label for="data\_consulta">Data da Consulta:</label>

<input type="date" id="data\_consulta" name="data\_consulta" required><br><br>

<label for="hora\_consulta">Hora da Consulta:</label>

<input type="time" id="hora\_consulta" name="hora\_consulta" required><br><br>

<input type="submit" value="Agendar Consulta">

</form>

**Explicação**

1. **Campo select**: Utilizamos a tag <select> para listar os pacientes e médicos disponíveis para consulta. No caso real, essas listas viriam do banco de dados.
2. **Campo date e time**: Esses campos permitem que o usuário escolha a data e a hora da consulta.

**4. CSS para Melhorar o Estilo**

Agora, vamos adicionar o arquivo **style.css** para estilizar os formulários e melhorar a interface.

**Arquivo CSS: style.css**

***css***

/\* style.css \*/

body {

font-family: Arial, sans-serif;

background-color: #f4f4f4;

padding: 20px;

}

h2 {

color: #333;

}

form {

background-color: #fff;

padding: 20px;

border: 1px solid #ccc;

border-radius: 10px;

max-width: 500px;

margin: 20px 0;

}

label {

display: block;

margin-bottom: 10px;

font-weight: bold;

}

input[type="text"],

input[type="email"],

input[type="tel"],

input[type="date"],

input[type="time"],

select {

width: 100%;

padding: 10px;

margin-bottom: 20px;

border: 1px solid #ccc;

border-radius: 5px;

}

input[type="submit"] {

background-color: #28a745;

color: white;

padding: 10px 20px;

border: none;

border-radius: 5px;

cursor: pointer;

}

input[type="submit"]:hover {

background-color: #218838;

}

**Explicação do CSS**

1. **Estilos básicos**: O body tem um fundo claro e as fontes estão padronizadas com a fonte Arial. Isso melhora a legibilidade.
2. **Formulários**: Os formulários têm um fundo branco, borda arredondada, e margem para separá-los das bordas da página.
3. **Inputs e Labels**: Cada input tem um preenchimento adequado e uma largura de 100% para preencher o container, garantindo que o layout seja fluido.
4. **Botão de Submit**: O botão de envio tem uma cor de fundo verde e um efeito hover (quando o mouse passa por cima), mudando para uma cor mais escura.

**Obs:**

Agora você tem uma estrutura básica de cadastro de pacientes, médicos e consultas, com formulários funcionais em HTML e um estilo simples aplicado com CSS. Esses formulários podem ser expandidos e integrados com backend (servidor) para que os dados sejam armazenados em um banco de dados ou processados conforme necessário.

Agora, podemos criar um sistema de **cadastro de pacientes**, **cadastro de médicos** e **agendamento de consultas** usando **Django** e **Python.**

Seguiremos com uma estrutura clara, abordando desde a configuração inicial até a integração do HTML e CSS.

**Estrutura da Resposta**

1. **Configuração Inicial do Projeto Django**
   * Instalação do Django
   * Criação do Projeto e da Aplicação
2. **Modelos (Models)**
   * Definição dos modelos para Pacientes, Médicos e Consultas
3. **Formulários (Forms)**
   * Criação dos formulários usando Django Forms
4. **URLs e Views**
   * Criação das rotas e lógica de exibição (Views)
5. **Templates (HTML) e Integração com CSS**
   * Criação dos arquivos HTML e CSS para o frontend
6. **Banco de Dados**
   * Configuração do banco de dados e migrações
7. **Testes e Execução**

**1. Configuração Inicial do Projeto Django**

**1.1. Instalando o Django**

Se ainda não instalou o Django, pode fazê-lo via pip. No terminal:

***bash***

pip install django

**1.2. Criando o Projeto**

Após a instalação do Django, crie um novo projeto chamado clinica:

***bash***

django-admin startproject clinica

Isso cria a estrutura de diretórios básica para o projeto.

**1.3. Criando a Aplicação**

Dentro do projeto, crie uma aplicação chamada consultas, onde implementaremos a lógica de negócio:

***bash***

cd clinica

python manage.py startapp consultas

Agora, você precisa adicionar a nova aplicação ao arquivo de configurações do Django settings.py:

python

# clinica/settings.py

INSTALLED\_APPS = [

...

'consultas', # Adicione aqui

]

**2. Modelos (Models)**

Agora vamos definir os **modelos** (Models) que representarão as entidades do nosso sistema: **Paciente**, **Médico** e **Consulta**.

**2.1. Definindo os Modelos**

No arquivo models.py da aplicação consultas, defina os três modelos:

***python***

# consultas/models.py

from django.db import models

class Paciente(models.Model):

nome = models.CharField(max\_length=100)

email = models.EmailField()

telefone = models.CharField(max\_length=15)

data\_nascimento = models.DateField()

endereco = models.TextField()

def \_\_str\_\_(self):

return self.nome

class Medico(models.Model):

nome = models.CharField(max\_length=100)

crm = models.CharField(max\_length=15)

especialidade = models.CharField(max\_length=50)

def \_\_str\_\_(self):

return f'{self.nome} - {self.especialidade}'

class Consulta(models.Model):

paciente = models.ForeignKey(Paciente, on\_delete=models.CASCADE)

medico = models.ForeignKey(Medico, on\_delete=models.CASCADE)

data\_consulta = models.DateField()

hora\_consulta = models.TimeField()

def \_\_str\_\_(self):

return f'{self.paciente} - {self.medico} em {self.data\_consulta} às {self.hora\_consulta}'

**Explicação**

* **Paciente**: Contém os campos de informações pessoais do paciente.
* **Medico**: Registra os dados do médico, incluindo o CRM e especialidade.
* **Consulta**: Associa um paciente a um médico com data e hora.

**3. Formulários (Forms)**

Usaremos o Django Forms para criar os formulários baseados nos modelos.

**3.1. Definindo os Formulários**

Crie um arquivo forms.py dentro da pasta consultas:

***python***

# consultas/forms.py

from django import forms

from .models import Paciente, Medico, Consulta

class PacienteForm(forms.ModelForm):

class Meta:

model = Paciente

fields = ['nome', 'email', 'telefone', 'data\_nascimento', 'endereco']

class MedicoForm(forms.ModelForm):

class Meta:

model = Medico

fields = ['nome', 'crm', 'especialidade']

class ConsultaForm(forms.ModelForm):

class Meta:

model = Consulta

fields = ['paciente', 'medico', 'data\_consulta', 'hora\_consulta']

**Explicação**

* **PacienteForm**: Gera o formulário para cadastrar pacientes.
* **MedicoForm**: Gera o formulário para cadastrar médicos.
* **ConsultaForm**: Gera o formulário para agendar consultas.

**4. URLs e Views**

Agora vamos definir as **views** que processam os formulários e as rotas correspondentes.

**4.1. Definindo as Views**

No arquivo views.py, adicione a lógica para os cadastros e agendamentos:

**python**

# consultas/views.py

from django.shortcuts import render, redirect

from .forms import PacienteForm, MedicoForm, ConsultaForm

def cadastrar\_paciente(request):

if request.method == 'POST':

form = PacienteForm(request.POST)

if form.is\_valid():

form.save()

return redirect('lista\_pacientes')

else:

form = PacienteForm()

return render(request, 'consultas/cadastrar\_paciente.html', {'form': form})

def cadastrar\_medico(request):

if request.method == 'POST':

form = MedicoForm(request.POST)

if form.is\_valid():

form.save()

return redirect('lista\_medicos')

else:

form = MedicoForm()

return render(request, 'consultas/cadastrar\_medico.html', {'form': form})

def agendar\_consulta(request):

if request.method == 'POST':

form = ConsultaForm(request.POST)

if form.is\_valid():

form.save()

return redirect('lista\_consultas')

else:

form = ConsultaForm()

return render(request, 'consultas/agendar\_consulta.html', {'form': form})

**4.2. Definindo as URLs**

Agora, crie as rotas para essas views no arquivo urls.py:

***python***

# consultas/urls.py

from django.urls import path

from . import views

urlpatterns = [

path('cadastrar\_paciente/', views.cadastrar\_paciente, name='cadastrar\_paciente'),

path('cadastrar\_medico/', views.cadastrar\_medico, name='cadastrar\_medico'),

path('agendar\_consulta/', views.agendar\_consulta, name='agendar\_consulta'),

]

Adicione também no arquivo urls.py principal do projeto:

***python***

# clinica/urls.py

from django.contrib import admin

from django.urls import include, path

urlpatterns = [

path('admin/', admin.site.urls),

path('consultas/', include('consultas.urls')),

]

**5. Templates (HTML) e Integração com CSS**

Agora vamos criar os arquivos HTML e conectar o CSS para estilizar os formulários.

**5.1. Templates HTML**

Crie uma pasta chamada templates dentro da aplicação consultas e, dentro dela, crie a pasta consultas para armazenar os arquivos HTML.

**Formulário de Cadastro de Pacientes**

***html***

<!-- consultas/templates/consultas/cadastrar\_paciente.html -->

{% extends 'base.html' %}

{% block content %}

<h2>Cadastrar Paciente</h2>

<form method="post">

{% csrf\_token %}

{{ form.as\_p }}

<button type="submit">Cadastrar Paciente</button>

</form>

{% endblock %}

**Formulário de Cadastro de Médicos**

***html***

<!-- consultas/templates/consultas/cadastrar\_medico.html -->

{% extends 'base.html' %}

{% block content %}

<h2>Cadastrar Médico</h2>

<form method="post">

{% csrf\_token %}

{{ form.as\_p }}

<button type="submit">Cadastrar Médico</button>

</form>

{% endblock %}

**Formulário de Agendamento de Consultas**

***html***

<!-- consultas/templates/consultas/agendar\_consulta.html -->

{% extends 'base.html' %}

{% block content %}

<h2>Agendar Consulta</h2>

<form method="post">

{% csrf\_token %}

{{ form.as\_p }}

<button type="submit">Agendar Consulta</button>

</form>

{% endblock %}

**5.2. Arquivo Base para Templates**

Crie um arquivo base.html para servir como modelo para os outros templates:

***html***

<!-- consultas/templates/base.html -->

<!DOCTYPE html>

<html lang="pt-br">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Clínica</title>

<link rel="stylesheet" href="{% static 'css/style.css' %}">

</head>

<body>

<nav>

<a href="{% url 'cadastrar\_paciente' %}">Cadastrar Paciente</a> |

<a href="{% url 'cadastrar\_medico' %}">Cadastrar Médico</a> |

<a href="{% url 'agendar\_consulta' %}">Agendar Consulta</a>

</nav>

<div class="container">

{% block content %}{% endblock %}

</div>

</body>

</html>

**5.3. Estilos CSS**

Crie uma pasta static/css para os arquivos de estilo e adicione um arquivo style.css para estilizar o sistema:

***css***

/\* static/css/style.css \*/

body {

font-family: Arial, sans-serif;

background-color: #f8f9fa;

margin: 0;

padding: 20px;

}

nav {

margin-bottom: 20px;

}

nav a {

margin-right: 10px;

text-decoration: none;

color: #007bff;

}

.container {

max-width: 600px;

margin: 0 auto;

}

button {

background-color: #007bff;

color: white;

padding: 10px 15px;

border: none;

cursor: pointer;

}

button:hover {

background-color: #0056b3;

}

**6. Banco de Dados**

Antes de iniciar o servidor, faça as migrações para criar as tabelas no banco de dados:

***bash***

python manage.py makemigrations

python manage.py migrate

**7. Testes e Execução**

Por fim, inicie o servidor:

***bash***

python manage.py runserver

Acesse o servidor em http://127.0.0.1:8000/consultas/, onde você verá as páginas de cadastro de pacientes, médicos e agendamento de consultas.

Obs: O objetivo deste texto é introduzir e ajudar na criação do projeto até a implementação de um sistema completo, com integração de HTML e CSS, fornecendo um sistema funcional para cadastro e gerenciamento de consultas médicas.

Para aprimorar o sistema de **cadastro de pacientes, médicos e consultas**, e incluir funcionalidades mais sofisticadas, há diversas melhorias e novas funcionalidades que podemos implementar. Abaixo estão algumas ideias:

**Funcionalidades Faltantes ou Melhorias**

1. **Listagem de Médicos e Pacientes**
   * Exibir listas de médicos e pacientes cadastrados, com a possibilidade de editar e excluir registros.
2. **Listagem de Consultas**
   * Exibir todas as consultas já agendadas, com opções de filtro por médico, paciente, data, ou horário.
3. **Verificação de Disponibilidade de Médicos**
   * Implementar uma função para verificar a **disponibilidade de um médico** em um determinado horário. Podemos construir uma "grade de horários" para os médicos com base nas consultas agendadas.
4. **Criação de uma Agenda para os Médicos**
   * Uma funcionalidade que exiba a **agenda** completa de um médico com todas as consultas agendadas. O ideal é mostrar os horários disponíveis e ocupados.
5. **Restrições no Agendamento de Consultas**
   * Adicionar regras para **impedir o agendamento de consultas em horários que já estejam ocupados** para o médico em questão.
6. **Autenticação de Usuários**
   * Implementar um sistema de autenticação onde os médicos e secretárias possam se logar no sistema. Isso pode ser feito utilizando o sistema de **autenticação de usuários do Django**.
7. **Atribuição de Consultas a Médicos Logados**
   * Caso os médicos tenham sua própria conta no sistema, poderíamos listar as consultas que foram **agendadas para o médico logado**, permitindo que ele visualize facilmente sua agenda.
8. **Histórico de Consultas**
   * Exibir o **histórico de consultas** de um paciente, mostrando todas as consultas anteriores.
9. **Busca e Filtragem Avançada**
   * Adicionar um sistema de **busca e filtragem** para encontrar pacientes, médicos ou consultas com base em diversos critérios (nome, CRM, especialidade, data de nascimento, etc.).
10. **Exportação de Dados**
    * Criar a funcionalidade de **exportação de relatórios** para gerar PDFs ou planilhas com dados de pacientes, médicos ou consultas.
11. **Paginação e Ordenação**
    * Adicionar **paginação** para listas longas de pacientes ou médicos, bem como a possibilidade de ordenar essas listas por nome, CRM, especialidade, etc.
12. **Dashboard para Secretárias e Médicos**
    * Criar um **dashboard** com estatísticas sobre o número de consultas agendadas, médicos disponíveis, número de pacientes atendidos em determinado período, entre outras métricas úteis.

**Ideias Específicas para Implementar:**

**1. Listagem de Médicos e Pacientes**

* Podemos adicionar uma view que retorna todos os médicos e pacientes cadastrados. Exemplo de uma view para listar médicos:

***python***

# consultas/views.py

from .models import Medico, Paciente

def listar\_medicos(request):

medicos = Medico.objects.all()

return render(request, 'consultas/listar\_medicos.html', {'medicos': medicos})

def listar\_pacientes(request):

pacientes = Paciente.objects.all()

return render(request, 'consultas/listar\_pacientes.html', {'pacientes': pacientes})

Exemplo de template para listar médicos:

***html***

<!-- consultas/templates/consultas/listar\_medicos.html -->

<h2>Médicos Cadastrados</h2>

<ul>

{% for medico in medicos %}

<li>{{ medico.nome }} - {{ medico.especialidade }}</li>

{% endfor %}

</ul>

**2. Consulta de Disponibilidade de Horários**

Podemos criar uma função que recebe o ID do médico e retorna uma lista de horários disponíveis. Exemplo:

***python***

# consultas/views.py

from django.utils.timezone import localtime, now

def verificar\_disponibilidade(medico\_id, data\_consulta):

consultas = Consulta.objects.filter(medico\_id=medico\_id, data\_consulta=data\_consulta)

horarios\_ocupados = [consulta.hora\_consulta for consulta in consultas]

# Defina os horários de trabalho

horarios\_disponiveis = [

"08:00", "09:00", "10:00", "11:00", "14:00", "15:00", "16:00"

]

# Remova os horários ocupados da lista de horários disponíveis

horarios\_disponiveis = [h for h in horarios\_disponiveis if h not in horarios\_ocupados]

return horarios\_disponiveis

**3. Exibição da Agenda Completa de um Médico**

Uma view que mostra todas as consultas de um médico para o dia atual ou uma data específica:

***python***

# consultas/views.py

from django.shortcuts import get\_object\_or\_404

def agenda\_medico(request, medico\_id):

medico = get\_object\_or\_404(Medico, id=medico\_id)

consultas = Consulta.objects.filter(medico=medico).order\_by('data\_consulta', 'hora\_consulta')

return render(request, 'consultas/agenda\_medico.html', {'consultas': consultas, 'medico': medico})

Exemplo de template:

***html***

<!-- consultas/templates/consultas/agenda\_medico.html -->

<h2>Agenda do Dr. {{ medico.nome }}</h2>

<ul>

{% for consulta in consultas %}

<li>{{ consulta.data\_consulta }} às {{ consulta.hora\_consulta }} - Paciente: {{ consulta.paciente.nome }}</li>

{% endfor %}

</ul>

**4. Histórico de Consultas de Pacientes**

Para listar o histórico de consultas de um paciente, podemos criar uma view como esta:

***python***

# consultas/views.py

def historico\_paciente(request, paciente\_id):

paciente = get\_object\_or\_404(Paciente, id=paciente\_id)

consultas = Consulta.objects.filter(paciente=paciente).order\_by('data\_consulta')

return render(request, 'consultas/historico\_paciente.html', {'consultas': consultas, 'paciente': paciente})

Template para o histórico de consultas:

***html***

<!-- consultas/templates/consultas/historico\_paciente.html -->

<h2>Histórico de Consultas de {{ paciente.nome }}</h2>

<ul>

{% for consulta in consultas %}

<li>{{ consulta.data\_consulta }} - Médico: {{ consulta.medico.nome }}</li>

{% endfor %}

</ul>

**5. Filtragem e Paginação**

Adicionar filtragem e paginação para a listagem de médicos e consultas seria útil. O Django oferece um **paginator** embutido, que pode ser usado como exemplo na listagem de médicos:

***python***

# consultas/views.py

from django.core.paginator import Paginator

def listar\_medicos(request):

medicos\_list = Medico.objects.all()

paginator = Paginator(medicos\_list, 10) # Mostra 10 médicos por página

page\_number = request.GET.get('page')

medicos = paginator.get\_page(page\_number)

return render(request, 'consultas/listar\_medicos.html', {'medicos': medicos})

Template de exemplo com paginação:

***html***

<!-- consultas/templates/consultas/listar\_medicos.html -->

<h2>Médicos Cadastrados</h2>

<ul>

{% for medico in medicos %}

<li>{{ medico.nome }} - {{ medico.especialidade }}</li>

{% endfor %}

</ul>

<div>

<span class="step-links">

{% if medicos.has\_previous %}

<a href="?page=1">&laquo; primeira</a>

<a href="?page={{ medicos.previous\_page\_number }}">anterior</a>

{% endif %}

<span> Página {{ medicos.number }} de {{ medicos.paginator.num\_pages }} </span>

{% if medicos.has\_next %}

<a href="?page={{ medicos.next\_page\_number }}">próxima</a>

<a href="?page={{ medicos.paginator.num\_pages }}">última &raquo;</a>

{% endif %}

</span>

</div>

**Obs:** Estas são algumas ideias e funcionalidades adicionais que podem transformar o sistema básico de cadastro de consultas em uma solução completa e robusta, com recursos úteis para a gestão de clínicas, médicos e pacientes. A integração com CSS, autenticação, listagens, e a verificação de disponibilidade de horários são essenciais para tornar o sistema prático e eficiente.

Temos ainda algumas sugestões e ideias para melhorar ainda mais o sistema de gestão de consultas em Python e Django, considerando que nossa turma tem pouca experiência com Django.

**Sugestões para Melhorias**

1. **Validação de Formulários**
   * É uma boa prática garantir que, ao cadastrar ou atualizar informações, os dados inseridos estejam corretos. Podemos adicionar validações de dados em nossos **model forms** para garantir que, por exemplo, o horário da consulta seja válido, que o nome do paciente tenha um mínimo de caracteres ou que um médico com um CRM duplicado não possa ser cadastrado.
   * No Django, usamos o **Form validation**. Isso pode ser feito no próprio ModelForm:

***python***

# consultas/forms.py

from django import forms

from .models import Consulta

class ConsultaForm(forms.ModelForm):

class Meta:

model = Consulta

fields = ['paciente', 'medico', 'data\_consulta', 'hora\_consulta']

def clean\_data\_consulta(self):

data = self.cleaned\_data['data\_consulta']

if data < datetime.date.today():

raise forms.ValidationError("A data da consulta não pode ser no passado.")

return data

def clean\_hora\_consulta(self):

hora = self.cleaned\_data['hora\_consulta']

if hora < datetime.time(8, 0) or hora > datetime.time(18, 0):

raise forms.ValidationError("O horário da consulta deve ser entre 8:00 e 18:00.")

return hora

Essa validação ajuda a evitar erros comuns, como agendar consultas no passado ou fora do horário de atendimento.

1. **Integração com Email e Notificações**
   * Um recurso muito interessante seria o envio de emails para confirmar a consulta ou lembrar os pacientes do seu horário agendado.
   * O Django facilita isso com a função send\_mail. Podemos configurar nosso projeto para enviar um email quando uma nova consulta for marcada, avisando tanto o médico quanto o paciente.
   * Exemplo básico de uso do envio de email:

***python***

# consultas/views.py

from django.core.mail import send\_mail

def criar\_consulta(request):

if request.method == "POST":

form = ConsultaForm(request.POST)

if form.is\_valid():

consulta = form.save()

# Envia email de confirmação

send\_mail(

'Confirmação de Consulta',

f'Consulta agendada com Dr. {consulta.medico.nome} no dia {consulta.data\_consulta} às {consulta.hora\_consulta}.',

'admin@clinica.com',

[consulta.paciente.email],

fail\_silently=False,

)

return redirect('listar\_consultas')

else:

form = ConsultaForm()

return render(request, 'consultas/criar\_consulta.html', {'form': form})

1. **Sistema de Autenticação**
   * É importante que o sistema tenha um controle de quem pode acessar determinadas áreas. Por exemplo, secretárias podem agendar consultas, mas talvez os médicos só precisem visualizar suas próprias consultas.
   * O Django já possui um sistema de **autenticação e autorização** embutido. Podemos criar **grupos** de usuários, como **Médicos** e **Secretárias**, e limitar o que cada grupo pode fazer.
   * Para permitir apenas usuários logados, usamos o **@login\_required**. Além disso, podemos usar o **PermissionRequiredMixin** para definir permissões por grupo.

***python***

# consultas/views.py

from django.contrib.auth.decorators import login\_required

@login\_required

def listar\_consultas(request):

consultas = Consulta.objects.all()

return render(request, 'consultas/listar\_consultas.html', {'consultas': consultas})

1. **Filtragem de Consultas por Médico ou Data**
   * Para melhorar a usabilidade, podemos adicionar a opção de **filtrar consultas** por médico ou por uma data específica. Isso é útil, por exemplo, para médicos que querem ver apenas suas consultas ou para a secretária verificar quais consultas estão agendadas para um determinado dia.

***python***

# consultas/views.py

from django.db.models import Q

def listar\_consultas(request):

medico\_id = request.GET.get('medico')

data\_consulta = request.GET.get('data\_consulta')

consultas = Consulta.objects.all()

if medico\_id:

consultas = consultas.filter(medico\_id=medico\_id)

if data\_consulta:

consultas = consultas.filter(data\_consulta=data\_consulta)

return render(request, 'consultas/listar\_consultas.html', {'consultas': consultas})

1. **Melhorias na Interface com Bootstrap**
   * Para melhorar a experiência do usuário (UI/UX), podemos estilizar os formulários e páginas usando **Bootstrap**, que é um framework CSS muito utilizado.
   * Exemplo de como integrar Bootstrap em seu template:

***html***

<!-- consultas/templates/base.html -->

<!DOCTYPE html>

<html lang="pt-br">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Clínica</title>

<!-- Bootstrap CSS -->

<link href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css" rel="stylesheet">

</head>

<body>

<nav class="navbar navbar-expand-lg navbar-light bg-light">

<a class="navbar-brand" href="{% url 'home' %}">Clínica XYZ</a>

<div class="collapse navbar-collapse">

<ul class="navbar-nav ml-auto">

<li class="nav-item"><a class="nav-link" href="{% url 'listar\_consultas' %}">Consultas</a></li>

<li class="nav-item"><a class="nav-link" href="{% url 'listar\_medicos' %}">Médicos</a></li>

<li class="nav-item"><a class="nav-link" href="{% url 'listar\_pacientes' %}">Pacientes</a></li>

</ul>

</div>

</nav>

<div class="container mt-4">

{% block content %}

{% endblock %}

</div>

<!-- Bootstrap JS -->

<script src="https://code.jquery.com/jquery-3.5.1.slim.min.js"></script>

<script src="https://cdn.jsdelivr.net/npm/bootstrap@4.5.2/dist/js/bootstrap.bundle.min.js"></script>

</body>

</html>

1. **Logs de Atividades e Histórico**
   * Outra funcionalidade importante para um sistema desse tipo é ter um **log de atividades**. Isso pode ser útil para auditoria, ou seja, para saber **quem criou ou modificou uma consulta**, ou **quando uma consulta foi excluída**.
   * Podemos usar o modelo django.contrib.admin.models.LogEntry, que armazena logs de mudanças feitas por usuários no sistema administrativo do Django. Isso pode ser estendido para o nosso sistema.

***python***

# logs/models.py

from django.contrib.admin.models import LogEntry

from django.contrib.contenttypes.models import ContentType

def log\_activity(user, action\_flag, object\_repr):

LogEntry.objects.log\_action(

user\_id=user.id,

content\_type\_id=ContentType.objects.get\_for\_model(Consulta).pk,

object\_id=Consulta.pk,

object\_repr=object\_repr,

action\_flag=action\_flag

)

1. **Sistema de Backup e Segurança dos Dados**
   * Como estamos lidando com dados sensíveis (médicos, pacientes, consultas), é essencial garantir que o sistema seja seguro. Algumas práticas recomendadas incluem:
     + **Backup automático** da base de dados.
     + **Criptografia de dados** sensíveis (como informações pessoais).
     + Uso de **HTTPS** para todas as páginas.
2. **Automatizar Testes de Sistema**
   * Uma boa prática em qualquer sistema é garantir que as funcionalidades estejam sempre funcionando corretamente, mesmo depois de várias modificações no código. Podemos configurar **testes automatizados** em Django para verificar se as views, formulários e modelos estão funcionando como esperado.

***python***

# consultas/tests.py

from django.test import TestCase

from .models import Medico

class MedicoModelTest(TestCase):

def test\_criar\_medico(self):

medico = Medico.objects.create(nome='Dr. João', especialidade='Cardiologia', crm='123456')

self.assertEqual(medico.nome, 'Dr. João')

self.assertEqual(medico.especialidade, 'Cardiologia')

**Obs:**

Estas são apenas algumas sugestões de funcionalidades e melhorias que podem ser implementadas no seu sistema de gestão de consultas em Django. O principal objetivo dessas melhorias é proporcionar uma melhor experiência para os usuários (médicos, secretárias, pacientes) e garantir que o sistema seja robusto, seguro e eficiente.

**Questões adicionais importantes**

Para garantir **qualidade do código** em um sistema de gestão de consultas (ou em qualquer sistema), é importante seguir um conjunto de **boas práticas** que visam não apenas a funcionalidade correta do sistema, mas também a sua escalabilidade, manutenção e segurança ao longo do tempo. Abaixo vou detalhar várias áreas-chave que você pode trabalhar e organizar, utilizando **validação de dados**, **autenticação e autorização**, e **manutenção de logs**, além de outras boas práticas essenciais para o desenvolvimento em Django.

**1. Organização do Projeto e Estrutura de Código**

Uma boa estrutura de projeto facilita a manutenção e escalabilidade. No Django, o modelo "app" é ideal para modularizar o código. Divida funcionalidades distintas em apps separadas. Por exemplo:

* Um app para **pacientes** (cadastro, listagem, etc.).
* Um app para **médicos** (gerenciamento de médicos).
* Um app para **consultas** (agendamento e histórico de consultas).

bash

myproject/

├── pacientes/

├── medicos/

├── consultas/

├── templates/

├── static/

├── manage.py

└── myproject/

**Vantagens:**

* **Separação de responsabilidades**: Cada app trata de uma funcionalidade específica, facilitando a adição de novas funcionalidades sem "poluir" o código.
* **Escalabilidade**: Se precisar adicionar novas funcionalidades, você pode criar novos apps facilmente.
* **Testabilidade**: Testes ficam mais organizados quando o código está modularizado.

**2. Validação de Dados**

**a) Validação no Backend**

A primeira linha de defesa é a **validação no backend**. O Django já oferece validações automáticas com base nos campos dos modelos, mas podemos adicionar validações customizadas para garantir a consistência dos dados.

python

# consultas/models.py

from django.core.exceptions import ValidationError

class Consulta(models.Model):

paciente = models.ForeignKey(Paciente, on\_delete=models.CASCADE)

medico = models.ForeignKey(Medico, on\_delete=models.CASCADE)

data\_consulta = models.DateField()

hora\_consulta = models.TimeField()

def clean(self):

if self.data\_consulta < timezone.now().date():

raise ValidationError('A data da consulta não pode ser no passado.')

# Verificar disponibilidade do médico

if Consulta.objects.filter(medico=self.medico, data\_consulta=self.data\_consulta, hora\_consulta=self.hora\_consulta).exists():

raise ValidationError('O médico já tem uma consulta nesse horário.')

def save(self, \*args, \*\*kwargs):

self.clean()

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

**b) Validação no Frontend (JavaScript)**

Embora as validações no backend sejam essenciais, melhorar a experiência do usuário com **validações no frontend** (via JavaScript) também é importante. Isso proporciona feedback imediato para o usuário, evitando recargas desnecessárias da página.

**c) Validações Adicionais**

* **Campos obrigatórios**: Use blank=False nos modelos Django para garantir que os campos obrigatórios sejam preenchidos.
* **Formulários**: Utilize **ModelForm** e **Form validation** para garantir que o usuário preencha corretamente os dados no formulário.

**3. Autenticação e Autorização**

**a) Autenticação**

No Django, o sistema de autenticação já vem embutido e é muito fácil de usar. Com o **Django Auth**, podemos criar diferentes níveis de acesso para usuários da clínica (secretárias, médicos, administradores, etc.).

* **Login e Logout**: A autenticação é importante para garantir que apenas usuários autorizados acessem o sistema. A função **@login\_required** pode ser usada para garantir que apenas usuários logados acessem certas páginas.

python

from django.contrib.auth.decorators import login\_required

@login\_required

def listar\_consultas(request):

consultas = Consulta.objects.all()

return render(request, 'consultas/listar\_consultas.html', {'consultas': consultas})

**b) Autorização**

Além da autenticação, é importante limitar o que diferentes tipos de usuários podem fazer. O Django permite criar **grupos e permissões** que controlam o que cada tipo de usuário pode acessar.

* **Exemplo de Grupos**:
  + **Secretárias**: Podem criar, editar e cancelar consultas.
  + **Médicos**: Apenas visualizam suas próprias consultas.
  + **Administradores**: Têm acesso a tudo.

python

from django.contrib.auth.models import Group, User

# Adicionar usuários ao grupo 'Médicos'

medico\_grupo, created = Group.objects.get\_or\_create(name='Medicos')

user.groups.add(medico\_grupo)

* **PermissionRequiredMixin**: Para usar permissões nas views, o PermissionRequiredMixin pode ser aplicado para controlar o acesso a certas páginas ou funcionalidades.

python

from django.contrib.auth.mixins import PermissionRequiredMixin

class ConsultasListView(PermissionRequiredMixin, ListView):

permission\_required = 'consultas.view\_consulta'

model = Consulta

**c) Proteção de Dados e Páginas**

Além de autenticar e autorizar corretamente, é importante **proteger rotas e dados**. Um exemplo é usar **cabeçalhos de segurança** para garantir que o sistema esteja protegido contra vulnerabilidades comuns (como ataques de CSRF).

* **HTTPS**: Garantir que o site esteja rodando com HTTPS (SSL) para criptografar a comunicação entre o cliente e o servidor.

**4. Manutenção de Logs**

Ter um sistema de logs robusto é fundamental para auditoria e depuração. No Django, podemos usar o próprio **framework de logs** embutido para capturar eventos importantes como criação, edição e exclusão de consultas, além de possíveis erros no sistema.

**a) Configurando Logs no Django**

python

# settings.py

LOGGING = {

'version': 1,

'disable\_existing\_loggers': False,

'handlers': {

'file': {

'level': 'DEBUG',

'class': 'logging.FileHandler',

'filename': 'logs/debug.log',

},

},

'loggers': {

'django': {

'handlers': ['file'],

'level': 'DEBUG',

'propagate': True,

},

},

}

**b) Registrar Ações no Sistema**

Podemos adicionar logs sempre que um usuário realizar uma ação importante, como criar ou deletar uma consulta.

python

import logging

logger = logging.getLogger(\_\_name\_\_)

def criar\_consulta(request):

if request.method == "POST":

form = ConsultaForm(request.POST)

if form.is\_valid():

consulta = form.save()

logger.info(f'Consulta criada: {consulta}')

return redirect('listar\_consultas')

* **Monitoramento em Tempo Real**: Outra prática recomendada é utilizar ferramentas como **Sentry** para monitoramento de erros em tempo real.

**5. Testes Automatizados**

Manter a qualidade do código também significa garantir que novas funcionalidades não quebrem o que já funciona. O Django facilita a criação de **testes automatizados** para verificar se as funções e views estão operando como esperado.

* **Testes Unitários**: Verificam pequenas partes do código, como funções e modelos.

python

from django.test import TestCase

from .models import Paciente

class PacienteModelTest(TestCase):

def test\_criar\_paciente(self):

paciente = Paciente.objects.create(nome='Maria', data\_nascimento='1990-01-01')

self.assertEqual(paciente.nome, 'Maria')

* **Testes de Integração**: Verificam a interação entre várias partes do sistema, como views e formulários.

python

from django.urls import reverse

class ConsultaViewTest(TestCase):

def test\_acesso\_pagina\_consultas(self):

response = self.client.get(reverse('listar\_consultas'))

self.assertEqual(response.status\_code, 200)

**6. Refatoração e Manutenção**

Com o tempo, é natural que o código precise ser refatorado para se adequar a novas funcionalidades e melhorias. Seguir boas práticas de **refatoração** ajuda a manter o código limpo e organizado:

* **DRY (Don’t Repeat Yourself)**: Evite duplicar código. Reutilize funções e componentes.
* **Modularização**: Divida o código em partes menores e reutilizáveis.
* **Comentários e Documentação**: Documente bem o código para facilitar o entendimento de outros desenvolvedores.

**Conclusão**

Ao seguir essas boas práticas – **validação de dados**, **autenticação e autorização**, **manutenção de logs**, **testes automatizados** e **organização modular do projeto** – você garante que o sistema de gestão de consultas seja **robusto, escalável e seguro**. Com isso, ele será mais fácil de manter e estender, além de garantir uma melhor experiência para todos os usuários envolvidos no sistema (médicos, secretárias, pacientes e administradores). Essas práticas são essenciais para o crescimento profissional de quem está começando a desenvolver em Django e devem ser aplicadas desde os primeiros projetos.