

Django Framework na prática

Home

Todo material aqui apresentado serve de guia para o desenvolvimento do curso [Django framework na prática](#) e pode ser utilizado como material complementar às aulas disponíveis na plataforma Udemy. Apesar disto, todo o material referente ao livro "Django framework na prática" é de uso aberto e pode ser utilizado no aprendizado livre sem a necessidade de qualquer pagamento, contribuição ou até mesmo compra do curso na plataforma mencionada.

O objetivo do curso é explorar as principais funcionalidades do framework Django por meio da construção de uma dashboard para registro e administração de visitantes de um determinado condomínio. Vamos desenvolver um projeto real e aprender ao longo de seu desenvolvimento o que são as ferramentas que o framework nos dá e como utilizá-las. Seguindo o roteiro aqui apresentado, você vai poder iniciar a sua carreira como desenvolvedor web utilizando a linguagem Python e desenvolver aplicações seguras e escaláveis em tempo recorde.

Inicie no mundo do desenvolvimento fullstack utilizando a linguagem Python!

Conhecendo o processo

O projeto [controle-visitantes](#) tem o objetivo informatizar o processo de registro e administração de visitantes do condomínio Montanhas Azuis.

Hoje o condomínio conta com um processo manual e por meio de cadernos que são utilizados para registrar as informações referentes aos visitantes e informatizar esse processo é importante para ganhar tempo no processo, melhorar a experiência de trabalho dos porteiros e ainda armazenar as informações de forma segura e confiável.

O projeto consiste em uma ou mais páginas web em que seja possível registrar visitantes e visualizar suas informações. Além disso, precisamos disponibilizar algumas funcionalidades que seguem os fluxos executados pelo porteiro assim que um visitante chega à portaria e quando o mesmo deixa as dependências do condomínio.

Na imagem abaixo é possível visualizar o fluxo executado pelos porteiros e suas etapas:



Imagem com etapas do processo de registro de visitantes, onde temos quatro etapas: chegada na portaria, aguardando autorização, realizando visita e visita finalizada

Principais funcionalidades

Registro de visitantes

O formulário de registro de visitantes deve abstrair a etapa 01 do processo, onde o visitante informa **nome completo**, **CPF**, **data de nascimento**, o **número da casa** que deseja visitar e ainda a **placa do veículo**, se estiver utilizando durante a visita. Além desta informações, o formulário salva o **horário de chegada** do visitante automaticamente.

Registrar visitante

Formulário para registro de novo visitante

O asterisco (*) indica que o campo é obrigatório

Nome completo *

CPF *

Data de nascimento *

Número da casa a ser visitada *

Placa do veículo

Cancelar

Registrar visitante

Tela de registro de visitante com formulário para inserção das informações

Listagem de visitantes

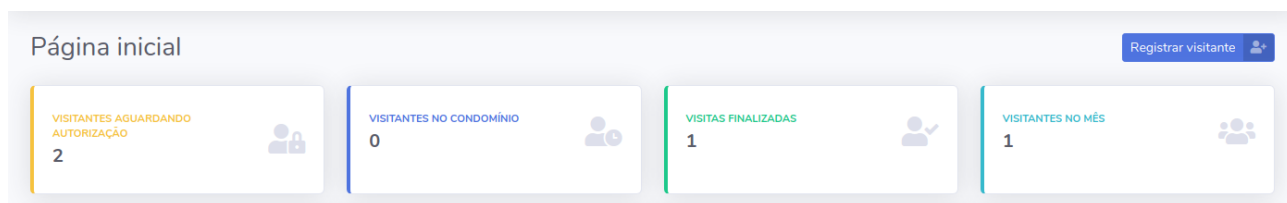
A listagem de visitantes exibe, por meio de uma tabela, os visitantes recentes classificados por horário de chegada, do mais recente para o mais antigo.

Visitantes recentes						
Exibindo 15 registros						
Nome	CPF	Horário de chegada	Status	Horário da autorização	Autorizado por	Mais informações
Décimo quinto visitante	151.515.151-51	5 de Maio de 2020 às 19:15	Aguardando autorização	Visitante aguardando autorização	Visitante aguardando autorização	Ver informações
Décimo quarto visitante	141.414.141-41	5 de Maio de 2020 às 19:15	Aguardando autorização	Visitante aguardando autorização	Visitante aguardando autorização	Ver informações
Décimo terceiro visitante	131.313.131-31	5 de Maio de 2020 às 19:14	Aguardando autorização	Visitante aguardando autorização	Visitante aguardando autorização	Ver informações
Décimo segundo visitante	121.212.121-21	5 de Maio de 2020 às 19:14	Aguardando autorização	Visitante aguardando autorização	Visitante aguardando autorização	Ver informações
Décimo primeiro visitante	011.111.111-11	5 de Maio de 2020 às 19:14	Aguardando autorização	Visitante aguardando autorização	Visitante aguardando autorização	Ver informações
Décimo visitante	101.010.101-01	5 de Maio de 2020 às 19:14	Aguardando autorização	Visitante aguardando autorização	Visitante aguardando autorização	Ver informações
Nono visitante	999.999.999-99	5 de Maio de 2020 às 19:13	Aguardando autorização	Visitante aguardando autorização	Visitante aguardando autorização	Ver informações
Oitavo visitante	888.888.888-88	5 de Maio de 2020 às 19:13	Aguardando autorização	Visitante aguardando autorização	Visitante aguardando autorização	Ver informações
Sétimo visitante	777.777.777-77	5 de Maio de 2020 às 19:11	Aguardando autorização	Visitante aguardando autorização	Visitante aguardando autorização	Ver informações
Sexto visitante	666.666.666-66	5 de Maio de 2020 às 19:10	Aguardando autorização	Visitante aguardando autorização	Visitante aguardando autorização	Ver informações
Página 1 de um total de 2						
1 2 Próxima página						

Tela com lista de visitantes recentes

Widgets para resumo de informações

O widgets da página inicial da dashboard têm a função de exibir um resumo dos números referentes aos visitantes em cada status e ainda o número total de visitantes registrados no mês.



Captura de tela de widgets para resumo de informações de visitantes

Visualização de informações de visitante

A partir da tabela que lista os visitantes recentes, é possível acessar a página que exibe as informações detalhadas de cada visitante. No exemplo abaixo um visitante com a visita já finalizada:

A tela de informações de visitante é dividida em duas seções principais: 'Informações gerais' e 'Informações pessoais'.

Informações gerais

Horário de chegada	Número da casa a ser visitada	Status
27 de Abril de 2020 às 22:16	61	Visita finalizada
Horário de autorização de entrada	Entrada autorizada por	Horário de saída
30 de Abril de 2020 às 19:53	Seu Jair	30 de Abril de 2020 às 19:59

Informações pessoais

Nome completo	CPF
Segundo visitante	222.222.222-22
Data de nascimento	Placa do veículo
27 de Agosto de 1995	Veículo não registrado

Visitante registrado em 27 de Abril de 2020 às 22:16 por Thiago Rodrigues Brasil

Voltar

Tela de informações de visitante

Autorização de entrada

A tela de informações de visitante é importante pois a partir dela é possível utilizar as funcionalidades de autorização de entrada e finalização de visita.

Quando um visitante está aguardando autorização, o botão para autorizar a entrada fica disponível na tela de informações deste visitante e, para autorizar a entrada, basta clicar no botão para abrir o formulário que deve ser preenchido com o nome do morador que autorizou a entrada do visitante e confirmar a ação.

No exemplo a seguir o visitante está aguardando autorização:

Informações de visitante

Autorizar entrada

Informações gerais

Horário de chegada

5 de Maio de 2020 às 19:15

Número da casa a ser visitada

17

Status

Aguardando autorização

Horário de autorização de entrada

Visitante aguardando autorização

Entrada autorizada por

Visitante aguardando autorização

Horário de saída

Horário de saída não registrado

Informações pessoais

Nome completo

Décimo quinto visitante

CPF

151.515.151-51

Data de nascimento

27 de Agosto de 1995

Placa do veículo

ABBA1550

Visitante registrado em 5 de Maio de 2020 às 19:15 por Thiago Rodrigues Brasil

Voltar

Tela de informações de visitante com botão para autorizar a entrada

Assim como quando o porteiro anotava o nome do morador responsável por autorizar a entrada e o horário de contato com esse morador, a funcionalidade recebe o nome do morador através de um formulário e salva o horário de contato e autorização de forma automática ao concluir a ação.

The image shows a web application interface with a modal dialog box in the foreground. The background is a blurred form titled 'Informações de visitante' (Visitor Information). The modal dialog is titled 'Autorizar entrada de visitante' (Authorize visitor entry) and contains a text input field with the name 'Jack' entered. Below the input field are two buttons: 'Cancelar' (Cancel) and 'Autorizar entrada' (Authorize entry). The background form has sections for 'Informações gerais' (General information) and 'Informações pessoais' (Personal information). The 'Informações gerais' section includes fields for 'Horário de chegada' (Arrival time) with the value '5 de Maio de 2020 às', 'Horário de autorização de entrada' (Entry authorization time) with the value 'Visitante aguardando autorização', 'Entrada autorizada por' (Authorized by) with the value 'Visitante aguardando autorização', and 'Status' (Status) with the value 'Aguardando autorização'. The 'Informações pessoais' section includes fields for 'Nome completo' (Full name) with the value 'Décimo quinto visitante', 'CPF' (CPF) with the value '151.515.151-51', 'Data de nascimento' (Date of birth) with the value '27 de Agosto de 1995', and 'Placa do veículo' (Vehicle plate) with the value 'ABBA1550'. At the bottom of the background form, there is a note: 'Visitante registrado em 5 de Maio de 2020 às 19:15 por Thiago Rodrigues Brasil' and a 'Voltar' (Back) button.

Tela de informações de visitante ao fundo com alerta exibindo formulário de cadastro de morador responsável

Finalização de visita

Assim como conseguimos utilizar a funcionalidade para autorizar a entrada do visitante, podemos finalizar sua visita. A funcionalidade funciona de forma parecida, com a diferença que é necessário apenas confirmar a ação, sem necessidade de informações adicionais.

Ao clicar no botão para finalizar uma visita, um alerta é exibido solicitando que o porteiro confirme a ação.

Informações de visitante

Finalizar visita

Informações gerais

Horário de chegada

28 de Abril de 2020 às 20:52

Número da casa a ser visitada

58

Status

Em visita

Horário de autorização de entrada

30 de Abril de 2020 às 19:59

Entrada autorizada por

Teste

Horário de saída

Horário de saída não registrado

Informações pessoais

Nome completo

Terceiro visitante

CPF

333.333.333-33

Data de nascimento

27 de Agosto de 1995

Placa do veículo

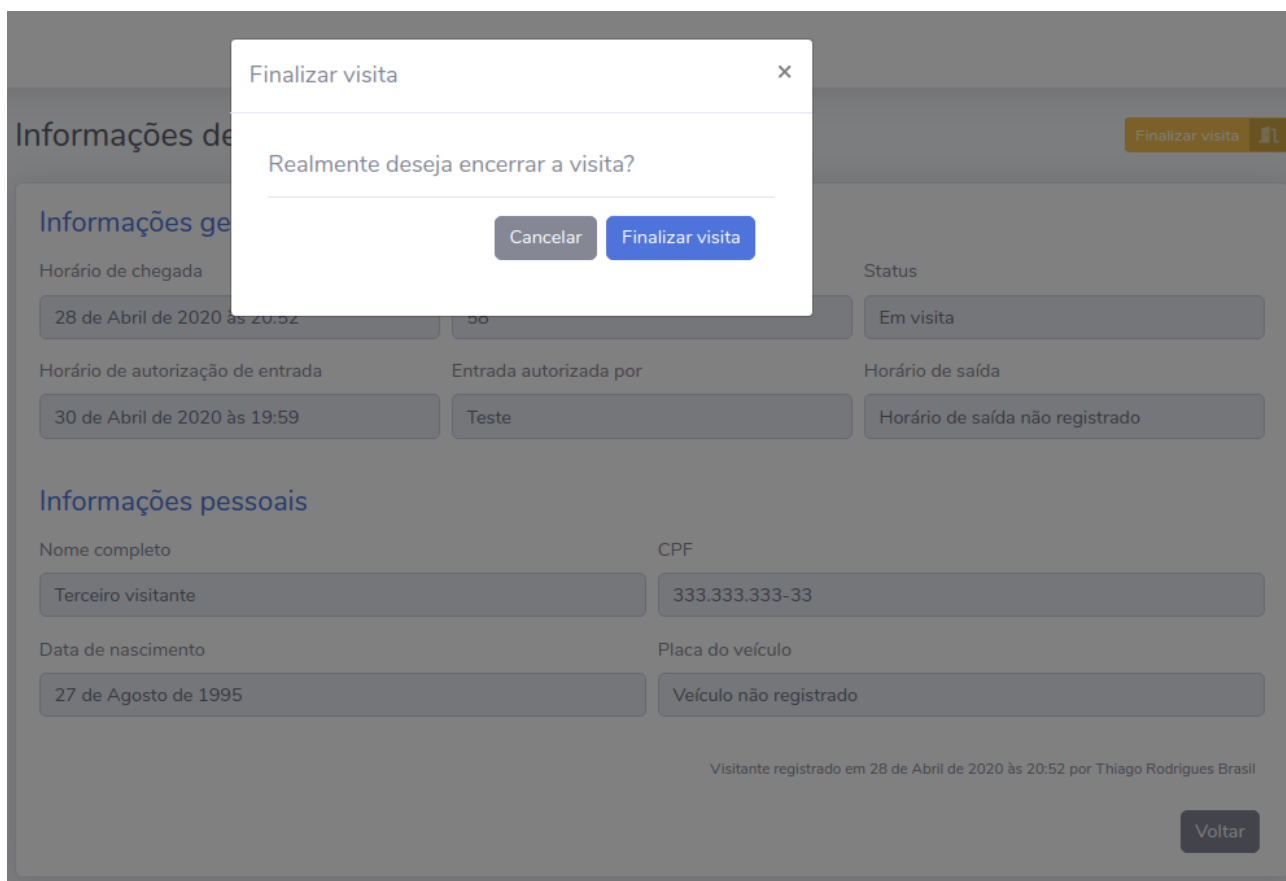
Veículo não registrado

Visitante registrado em 28 de Abril de 2020 às 20:52 por Thiago Rodrigues Brasil

Voltar

Tela de informações de visitante com botão para finalizar visita

Ao confirmar a ação clicando no botão "finalizar visita", o porteiro encerra o processo referente a este visitante e o horário em que a ação ocorreu é registrada.



Tela de informações de visitante ao fundo com alerta exibindo a confirmação de que o usuário realmente deseja encerrar a visita

Os botões "autorizar entrada" e "finalizar visita" são exibidos somente quando é possível executar a ação para o visitante em questão. Caso contrário, como quando o visitante já deixou as dependências do condomínio, nenhum botão é exibido.

Tecnologias utilizadas

O projeto [controle-visitantes](#) utiliza as seguintes tecnologias e recursos de código aberto:

- [Django framework](#)
 - [Django widget tweaks](#)
 - [Start Bootstrap - SB Admin 2](#)
-

Requisitos desejáveis

Para um bom aproveitamento de todo material e atividades aqui propostas, é desejável que você tenha os seguintes conhecimentos:

- Conhecimentos básicos da linguagem Python
 - Funções
 - Noções de Programação Orientada a Objetos
 - Classes
 - Instância de classe
 - Atributos
 - Métodos
- Conhecimentos básicos em HTML, CSS e JS
 - Noções de **Bootstrap**
- Sistema operacional baseado em Unix
- Conhecimentos básicos em terminal

Tópicos do livro

Capítulo 01

- Conhecendo o instrutor e sabendo mais sobre o curso
 - Sobre o instrutor
 - Sobre o curso
 - O que iremos construir?
 - A linguagem de programação Python
 - O que é Django Framework?
 - Instalando e configurando o ambiente para iniciar o projeto
 - Python 3.8
 - Virtualenv
 - Pip
 - Django framework
-

Capítulo 02

- Iniciando seu primeiro projeto Django
 - Um pouco mais sobre o django-admin
 - Entendendo a estrutura do projeto
 - Criando nosso primeiro aplicativo Django
 - Escrevendo nosso "Hello World"
-

Capítulo 03

- Escrevendo as models
 - Escrevendo a classe modelo
 - Escrevendo um manager personalizado
 - Alterando o modelo padrão nas configurações
 - Criando as tabelas do nosso banco de dados
- Criando um super usuário

- Conhecendo o Django Admin
 - Alterações necessárias nas configurações
-

Capítulo 04

- Criando aplicativo para gerenciar porteiros
 - Escrevendo as models do nosso aplicativo de porteiros
 - Conhecendo o campo DateField
 - Conhecendo o campo OneToOneField
 - Registrando nossa aplicação no Admin do Django
 - Aplicando as alterações em nosso banco de dados
 - Criando porteiro através do Admin do Django
-

Capítulo 05

- Configurando a aplicação para trabalhar com arquivos estáticos e templates HTML
 - Criando a pasta templates em nosso projeto
 - Criando a pasta static em nosso projeto
 - Criando views que renderizam templates
 - Conhecendo a função render
 - Entendendo as adaptações necessárias no template
 - Conhecendo a tag static
 - Alterando o caminho dos arquivos estáticos
 - Exibindo variáveis no template
 - Definindo nosso dicionário de contexto
 - Exibindo as informações nos templates
-

Capítulo 06

- Criando o aplicativo para gerenciar visitantes
- Escrevendo as models do nosso aplicativo de visitantes

- Conhecendo o campo DateTimeField
 - Conhecendo o campo ForeignKey
 - Registrando nossa aplicação no Admin do Django
 - Aplicando as alterações em nosso banco de dados
 - Adicionando visitante utilizando o Django Admin
 - Listando visitantes na página inicial da dashboard
 - Buscando registros de visitantes no banco de dados
 - Listando registros de visitantes no template HTML
-

Capítulo 07

- Criando tela para registro de novo visitante
 - Criando view para registrar visitante
 - Criando URL para mapear view
 - Adaptando nossos templates para trabalhar com a template engine do Django
 - Criando o template base
 - Adaptando template index
 - Adaptando template base
 - Adaptando template registrar_visitante
-

Capítulo 08

- Trabalhando com formulários no Django
 - Criando formulário para registro de visitante
 - Renderizando nosso formulário automaticamente
 - Melhorando a exibição do nosso formulário
 - Estilizando nosso formulário com django-widget-tweaks
 - Como instalar
 - Importando no template
 - Utilizando o render_field
-

Capítulo 09

- Preparando view para receber requisição do tipo POST
 - Conhecendo o objeto request
 - Conhecendo um pouco mais dos formulários
 - Tratando problema com atributo nulo
 - Exibindo uma mensagem para o usuário ao cadastrar novo visitante
 - Conhecendo o Django messages
 - Alterando o template para exigir as mensagens
 - Tratando possíveis erros em nosso formulário
 - Deixando nossas mensagens de erro mais claras
-

Capítulo 10

- Criando tela para exibir informações de visitante
 - Criando a view
 - Conhecendo o atalho get_model_or_404
 - Criando URL para acessar informações de visitante
 - Criando template para exibir informações de visitante
 - Criando métodos personalizados para exibir informações do Visitante
 - Criando método para exibir horário de saída
 - Criando métodos para exibir horário de autorização de entrada e morador responsável por autorizar a entrada
 - Criando método para exibir placa do veículo utilizado na visita
 - Utilizando métodos personalizados no template
 - Utilizando o Django para renderizar nossas URLs
 - Renderizando a URL para retornar à página inicial
-

Capítulo 11

- Criando funcionalidade para autorização de entrada de visitante
- Criando um status diferente para cada estágio da visita

- Criando o arquivo de migrações
 - Efetuando as alterações no banco de dados
 - Criando formulário para atualizar atributos específicos do visitante
 - Alterando view para autorizar entrada de visitante
 - Alterando template para exibir modal com formulário
 - Atualizando os campos `horario_autorizacao` e `status` diretamente
 - Atualizando o status
 - Conhecendo o `timezone` do Django
-

Capítulo 12

- Criando função para finalizar visita
 - Criando URL
 - Alterando template para exibir botão e modal para finalizar visita
 - Prevenindo erros e operações desnecessárias
 - Exibição condicional de botões para autorizar entrada e finalizar visita
 - Bloqueando o acesso à URL por métodos diferentes do POST
-

Capítulo 13

- Implementando melhorias em nossos templates
 - Exibindo botão com função de "voltar" e "cancelar" em páginas de informações e registro de visitante
 - Melhorando a exibição do CPF do visitante
 - Conhecendo o f-strings do Python
 - Utilizando método para exibir status do visitante
 - Implementando melhorias na estrutura do nosso projeto
-

Capítulo 14

- Criando aplicativos para administrar informações da dashboard

- Migrando view "index" para aplicativo dashboard
 - Conhecendo o método filter das querysets
 - Filtrando nossos visitantes por status
 - Contando os resultados de uma queryset
-

Capítulo 15

- Aprendendo a filtrar nossos visitantes por data
 - Conhecendo o field lookups da Queryset API
 - Filtrando apenas os registros do mês atual
 - Utilizando o timezone para descobrir o mês atual
 - Ordenando nossa lista de visitantes por horário de chegada
-

Capítulo 16

- Bloqueando o acesso para usuários não autenticados nas nossas views
 - Conhecendo o decorator login_required
 - Alterando a URL padrão para login e redirecionamento após login
 - Utilizando o sistema de autenticação do Django para nos fornecer a view de login
 - Criando o template de login
 - Renderizando formulário de login
 - Adicionando mensagem de erro em formulário de login
 - Criando URL para logout
 - Criando template de logout
 - Inserindo link para logout em dashboard
-

Capítulo final

- Encerramento e agradecimentos

Proposta de desenvolvimento

O condomínio Montanhas Azuis é um conceituado empreendimento que oferece conforto, qualidade e segurança para seus moradores. O condomínio nos procurou com o intuito de melhorar a experiência de trabalho de seus porteiros e tornar informatizado o processo de registro e administração de visitantes, de modo que os porteiros tenham acesso a uma página web e essas informações possam ser acessadas de qualquer lugar. Além do mais, desta forma, é possível garantir a segurança e a confiabilidade dos registros.

Entendendo o processo

Quando um visitante chega à portaria do condomínio, o mesmo deve informar alguns dados pessoais para que o porteiro anote estas informações no caderno de visitantes. Feito isso, o porteiro deve comunicar um morador que esteja na casa no momento e que possa autorizar a entrada deste visitante. Com a autorização concedida, o porteiro deve anotar o nome do morador e o horário em que a autorização ocorreu. Somente após esta etapa que o visitante pode adentrar ao condomínio e iniciar sua visita. Ao final da visita, o porteiro deve registrar o horário em que o visitante saiu das dependências do condomínio e ainda escrever seu nome para registrar que ele é o responsável pelo registro.

Todo esse processo, atualmente, ocorre de forma manual e através de um caderno que é destinado ao registro de visitantes. Devido a isso, o processo acaba levando mais tempo que deveria e as informações podem ter sua confiabilidade questionada e segurança ameaçada, pois problemas podem acontecer com o caderno ou as folhas deste caderno e até mesmo rasuras podem acontecer e prejudicar as informações ali contidas. Além dos pontos levantados, com um sistema web podemos automatizar algumas funções para poupar tempo e reduzir as informações que devem ser preenchidas pelo porteiro.

Ter um sistema web para registro e administração de visitantes é um projeto importante para o condomínio pois será um diferencial frente aos seus concorrentes e ainda vai melhorar bastante a experiência de trabalho dos porteiros que, com o sistema em funcionamento, terão menos informações para preencher e, conseqüentemente, menos tempo será perdido com as atividades relacionadas ao registro de visitantes.

Levantamento de requisitos

Com todo esse cenário em mente, precisamos desenvolver uma página web que seja acessível por meio das credenciais **e-mail** e **senha** e que possibilite o **registro dos visitantes**, conforme é feito atualmente no caderno. Para cadastrar um visitante é necessário informar seu **nome completo**, **CPF**, **data de nascimento**, o **número da casa a ser visitada**, a **placa do veículo do visitante**, se houver, e ainda o **horário em que o visitante se apresentou à portaria**.

Após o registro, o porteiro deve entrar em contato com um morador que esteja na casa no momento e informar à respeito da visita para que o morador autorize a entrada do visitante. Com a autorização concedida, o porteiro deve colocar no caderno o **nome do morador que autorizou a entrada do visitante** e o **horário em que essa autorização ocorreu**. Ao final da visita, o **horário em que o visitante deixou as dependências do condomínio** deve ser registrado junto do **nome do porteiro responsável pelo registro**. O fluxo e as etapas de todo o processo são as seguintes:



Imagem com etapas do processo de registro de visitantes, onde temos quatro etapas:

chegada na portaria, aguardando autorização, realizando visita e visita finalizada

A divisão do processo por etapas faz com que fiquem mais claras as atividades e a ordem em que elas devem ocorrer. Enxergando desta forma, podemos pensar em formas de dividir o preenchimento das informações e a automatização de determinadas informações.

Para poupar tempo do porteiro, ao receber as primeiras informações para o registro, seria interessante que o horário de chegada fosse preenchido automaticamente pois, desta forma, o porteiro fica preocupado somente em receber as informações pessoais do visitante, a casa a ser visitada e informar o placa do veículo, se houver.

Outro ponto interessante seria o registro automático do horário de autorização, assim que o porteiro informar o nome do morador responsável por autorizar a entrada do visitante. Essa informação pode ser preenchida num segundo momento, pois o visitante ficará na recepção aguardando o contato do porteiro com um morador que autorize a sua entrada. Preenchendo o nome do morador, vamos atualizar essa informação junto do horário de autorização e a visita se inicia. Ao final, claro, quando o visitante retorna à portaria para finalizar sua visita, devemos atualizar o horário de saída e inserir o nome do porteiro responsável pelo registro, tudo de forma automática.

Tudo isso deverá ocorrer em uma interface web onde seja possível visualizar os últimos visitantes registrados, ver informações detalhadas de cada visitante e ainda autorizar e finalizar suas visitas. Além destas informações, é desejável que o sistema web exiba um resumo das informações como número de visitantes aguardando autorização, número de visitantes no condomínio, número de visitas finalizadas e ainda o número de registros no mês.

Quais informações salvar

Nosso sistema deve estar preparado para armazenar uma série de informações, tanto dos porteiros, quanto dos visitantes. Essas informações são úteis para fins de controle e segurança de todos os envolvidos e também para o setor responsável pelos recursos humanos do condomínio.

Com relação aos porteiros, é necessário armazenar as seguintes informações para registro:

- E-mail (Utilizado na criação do usuário para acesso ao sistema)
- Nome completo
- CPF
- Telefone
- Data de nascimento

Com relação aos visitantes:

- Nome completo
- CPF
- Data de nascimento
- Casa a ser visitada
- Placa do veículo utilizado, se houver

Além das informações acima, ainda precisamos registrar as seguintes informações

- Horário de chegada na portaria
- Nome do morador responsável por autorizar a entrada do visitante
- Horário em que a autorização ocorreu
- Horário em que o visitante deixou as dependências do condomínio
- Porteiro responsável pelo registro

Capítulos

Capítulo 01

Conhecendo o instrutor e sabendo mais sobre o curso

Sobre o instrutor

Olá, meu nome é Thiago Brasil e na internet eu sou o @tchaguitos. Cursei Ciência da Computação e sou desenvolvedor web há aproximadamente 4 anos. Nesse tempo trabalhei principalmente com as linguagens Python e Javascript e me aventurei em vários projetos utilizando Django, NodeJS, ReactJS e mais algumas outras tecnologias da web. Além disso, trabalhei também com análise e visualização de dados geográficos.

Se você quiser ver alguns dos meus códigos, dá uma passadinha aqui no meu [Github](#)

Sobre o curso

Com o curso "Django Framework na prática" você vai poder iniciar sua carreira como desenvolvedor web utilizando a linguagem Python e aprender a desenvolver aplicações web seguras e escaláveis numa velocidade incrível. O objetivo do curso é explorar as principais funcionalidades do framework por meio da construção de um projeto real e aprender ao longo de seu desenvolvimento o que são as ferramentas que o framework nos dá e como utilizá-las.

Nesse curso a gente vai construir junto uma aplicação web do zero com Python que consiste num sistema de controle de visitantes para condomínio. A aplicação deverá ser capaz de possibilitar o registro e administração de visitantes e ainda contar com funcionalidades como visualização de informações de visitantes, autorização de entrada e finalização de visitas.

Depois desse curso você estará apto para desenvolver aplicações web robustas utilizando Django framework tendo visão ampla das boas práticas e principais funcionalidades que o framework nos permite utilizar.

O que iremos construir?

A partir da necessidade de informatizar o processo de registro de visitantes, um famoso condomínio entrou em contato conosco para desenvolver um sistema web que pudesse suprir essa demanda. Ao encontrar problemas na confiabilidade das informações e até mesmo para melhorar o fluxo de trabalho dos porteiros, o condomínio resolveu buscar ajuda.

Nosso sistema web deverá ser capaz de possibilitar o registro de visitantes, exibir os últimos visitantes registrados, exibir informações detalhadas de cada visitante e ainda autorizar e finalizar suas visitas, registrando informações específicas para cada etapa do processo. É interessante lembrar que existem algumas regras que os porteiros seguem à risca para manter a segurança dos moradores e nosso sistema deverá obedecer a essas regras.

Além destas informações, nosso sistema web deverá exibir um resumo das informações como o número de visitantes aguardando autorização, o número de visitantes realizando visitas no condomínio, o número de visitas finalizadas e ainda o número de registros realizados no mês.

A linguagem de programação Python

Python é uma linguagem de programação de código aberto poderosa, amigável e bastante popular. Com ela é possível desenvolver sistemas web, trabalhar com análise de dados, inteligência artificial e machine learning, nomes muito falados ultimamente, além de desenvolver aplicativos móveis e aplicações para desktop.

A linguagem foi criada por Guido van Rossum em 1991 com o objetivo de ser uma linguagem produtiva e legível, de modo que fosse fácil para humanos trabalharem em seus códigos, tornando menos custoso o processo de desenvolvimento e manutenção de sistemas.

Além da grande variedade de ferramentas que a própria linguagem disponibiliza, Python tem se tornado cada vez mais comum devido a sua utilização em áreas como Data Science, Machine Learning e Inteligência Artificial e, com isso, ganhando diversas bibliotecas e módulos que facilitam o dia a dia do desenvolvedor.

Por último e não menos importante: Python tem caminhado rumo ao status de linguagem de programação mais popular do mundo!

O que é Django Framework?

Django é um framework para construção de aplicações web escrito em Python gratuito e de código aberto. Um framework tem o objetivo de agrupar funcionalidades que são utilizadas com certa frequência na construção de aplicações exatamente com o intuito de economizar tempo dos desenvolvedores.

O Django, por sua vez, foi construído por desenvolvedores experientes e projetado pra resolver grande parte dos problemas existentes no desenvolvimento web. Sendo assim, a gente pode se concentrar em escrever a nossa aplicação sem a necessidade de ficar reinventando a roda.

Dentre as empresas que utilizam o Django no mundo, temos exemplos grandes como Instagram, Pinterest e a Mozilla. Aqui no Brasil existem diversas empresas que utilizam como a Olist, a Magalu e o pessoal da Globo.com.

Instalando e configurando o ambiente para iniciar o projeto

Antes de começar a colocar a mão na massa, a gente precisa preparar o nosso ambiente de desenvolvimento. Esse nosso ambiente é bem simples e tem como dependência apenas quatro itens. São eles:

- Python 3.8
- Pip
- Virtualenv
- Django framework



Todos os comandos aqui utilizados são compatíveis com distribuições Linux baseadas em Ubuntu (Debian) e as aulas foram gravadas utilizando o Ubuntu

Python 3.8

As versões mais recentes das distribuições linux baseadas no Ubuntu (Debian) já vêm com a versão 3.8.2 do Python instalada, mas se este não é o seu caso, basta seguir os comandos abaixo.

Antes de instalar o Python de fato, precisamos instalar alguns pacotes que vão nos ajudar com esta tarefa. Primeiro, vamos atualizar nossos pacotes e instalar algumas dependências:

```
1 $ sudo apt update
2 $ sudo apt install software-properties-common
```

Com isso feito, podemos agora adicionar o repositório PPA que será utilizado como fonte para instalarmos o Python3.8:

```
$ sudo add-apt-repository ppa:deadsnakes/ppa
```

O terminal vai pedir para que o você aperte `enter` para confirmar a ação. Faça isso para adicionar o repositório à sua lista e utilize os comandos abaixo para novamente atualizar os pacotes existentes e, finalmente, instalar a versão 3.8 do Python:

```
1 $ sudo apt update
2 $ sudo apt install python3.8
```

Vamos utilizar esta versão pois é a versão mais recente e estável que temos da linguagem. O Django encoraja a utilização de versões mais recentes da linguagem e ainda oferece

suporte oficial sempre para as últimas versões disponíveis. Como utilizaremos a versão 3.0 do Django, vamos já utilizar a versão 3.8.2 do Python lançada em 24 de fevereiro de 2020.

Virtualenv

Agora que temos a versão 3.8 do Python instalada, vamos instalar o Virtualenv para nos ajudar a organizar e isolar os ambientes de desenvolvimento dentro da nossa máquina. Com isso, a gente ganha a possibilidade de trabalhar em diversos projetos utilizando diferentes versões do Python e do Django na mesma máquina, sem correr o risco de que ocorram conflitos ou outros problemas relacionados a versão dos pacotes. Para instalar o Virtualenv vamos utilizar o seguinte comando:

```
$ sudo apt install virtualenv
```

Com o Virtualenv instalado, podemos criar um ambiente totalmente isolado para instalar as dependências relacionadas a cada projeto. Com o terminal aberto na pasta `home`, vamos criar um diretório chamado `dev` para mantermos nossos projetos. Para criar e acessar o diretório, vamos utilizar os seguintes comandos:


```
1 $ cd
2 $ mkdir dev
3 $ cd dev
```

Vamos utilizar o comando `cd` para garantir que estamos na pasta `home` do usuário logado em sua máquina e depois utilizamos o comando `mkdir dev` para criar a pasta `dev`. Com o diretório `dev` criado, vamos entrar nele utilizando o comando `cd dev` e criar o diretório que dará nome ao nosso projeto. Para isso, vamos utilizar os seguintes comandos:

```
1 $ mkdir controle-visitantes
2 $ cd controle-visitantes
```

Feito isso, vamos criar o nosso ambiente virtual utilizando o Virtualenv. Ele cria para nós um diretório onde ficam os pacotes instalados e utilizados dentro do nosso projeto. Vamos utilizar o comando:

```
$ virtualenv -p python3.8 env
```

 O comando `virtualenv` nos deixa passar alguns argumentos para deixar o ambiente mais próximo das nossas necessidades. Neste caso, estamos dizendo para ele qual a versão do Python queremos utilizar (`Python 3.8`) e o nome do nosso ambiente, que será `env`

Após criarmos o ambiente, temos que ativá-lo. Esse passo deve ser executado sempre que você for começar a trabalhar no projeto, pois nossa máquina contém pacotes e versões diferentes das que estão em nosso ambiente virtual e, por isso, precisamos avisar para nosso terminal de onde ele deverá executar os pacotes e até mesmo o Python. Para ativar nosso ambiente, utilizaremos o comando:

```
$ source env/bin/activate
```

Após execução do comando e ativação do ambiente, você deverá visualizar em seu terminal algo como:

```
(env)$
```

Pip

O pip é um gerenciador de pacotes bastante utilizado no mundo Python e ele vem instalado sempre que criamos um novo ambiente virtual utilizando o `virtualenv`. Sendo

assim, não vamos precisar instalá-lo, mas precisamos lembrar que só teremos a possibilidade de utilizar o comando `pip` com o ambiente virtual ativado.

Django framework

Com o ambiente virtual ativado, vamos começar a instalar as dependências do projeto. Por enquanto temos apenas o Django Framework, que é quem vai nos ajudar e muito nessa jornada. Para instalá-lo, vamos utilizar o Pip, nosso gerenciador de pacotes do universo Python utilizando o comando `pip`. Bem tranquilo, não?

Para que a gente não tenha problemas referentes à versão utilizada, vamos especificar ao Pip qual versão do Django queremos instalar:

```
(env)$ pip install Django==3.0.0
```

Ao fim de tudo e tendo sucesso em todas as instalações, temos o ambiente pronto para começar a desenvolver o projeto.

Sendo assim, bora pro próximo capítulo!

Capítulo 02

Iniciando seu primeiro projeto Django

Antes de partirmos para todas as atividades que envolvem o desenvolvimento da nossa dashboard, temos que iniciar nosso projeto, claro. Iniciar um projeto significa criar toda a estrutura básica necessária para garantir que o código que a gente vai escrever vai funcionar conforme esperado utilizando as tecnologias escolhidas.

Felizmente, o Django já nos dá alguns scripts que servem para executar tarefas específicas do nosso projeto e, dentre elas, criar o esqueleto de um novo projeto com essa estrutura básica necessária. Ao fim da execução do script, teremos algumas pastas e arquivos que poderão ser alterados conforme nossa necessidade.

Antes de executar os comandos necessários para iniciar projeto, vamos lembrar de ativar o ambiente virtual. Basta acessar a pasta do projeto (`cd controle-visitantes`) e utilizar o comando `source` passando caminho `env/bin/activate` :

```
1 $ cd controle-visitantes
2 $ source env/bin/activate
```

Após a ativação do ambiente, vamos criar um novo projeto utilizando o comando `startproject` do `django-admin` :

```
(env)$ django-admin startproject controle_visitantes .
```

O comando `startproject` pode receber, além do nome do projeto, o diretório em que o mesmo deverá ser iniciado (caso o diretório não seja informado, o Django criará um diretório de mesmo nome do projeto). Além disso, o Django permite apenas letras, números e o underline em nomes de projetos, por isso a pasta se chama `controle-visitantes` e o projeto `controle_visitantes` . Com o comando acima estamos iniciando nosso projeto

com nome `controle_visitantes` na pasta em que estamos trabalhando, isto é, na pasta `controle-visitantes` (note o ponto especificando o diretório atual).

Um pouco mais sobre o django-admin

O `django-admin` é um pacote de comandos úteis para realização de tarefas administrativas dentro da aplicação e dispõe de comandos para criar um novo projeto, iniciar o servidor de desenvolvimento, verificar erros e muito mais. Em complemento ao `django-admin`, o arquivo `manage.py` é criado.

O `manage.py` funciona exatamente como o `django-admin`. Na verdade, eles são a mesma coisa, com a diferença que o `manage.py`, por baixo dos panos, define um arquivo de configurações para ser utilizado pelo servidor de desenvolvimento. Em geral, o arquivo de configurações do nosso projeto.

Como o Django já possui um servidor de desenvolvimento integrado que nos possibilita rodar a aplicação localmente, vamos iniciá-lo e verificar se está tudo funcionando conforme esperado. Para isso, vamos utilizar o seguinte comando:

```
(env)$ python manage.py runserver
```



Por enquanto, vamos ignorar os avisos referentes às migrações de banco de dados. Em breve vamos entender melhor do que se trata.

Com o servidor de desenvolvimento rodando, vamos acessar o endereço <http://127.0.0.1:8000/>. Caso a tela abaixo apareça, significa que está tudo funcionando corretamente.



The install worked successfully! Congratulations!

You are seeing this page because `DEBUG=True` is in your settings file and you have not configured any URLs.



[Django Documentation](#)
Topics, references, & how-to's



[Tutorial: A Polling App](#)
Get started with Django



[Django Community](#)
Connect, get help, or contribute

Tela de boas vindas do Django framework onde é exibido um foguete em direção ao céu e uma mensagem informando que a aplicação está funcionando

Entendendo a estrutura do projeto

Como vimos anteriormente, o `django-admin` cria o esqueleto de um novo projeto e é isso que nós vamos entender agora: a estrutura que foi criada. Antes de tudo, vamos abrir a pasta do projeto em um editor de código. Eu vou utilizar o [VS Code](#), mas fique livre para escolher o seu favorito.

Com a pasta do projeto aberta no editor de código, vamos passar por cada um dos arquivos criados e entender o motivo pelo qual estão aí. A lista é a seguinte:

```
1 controle-visitantes/  
2     manage.py  
3     controle_visitantes/  
4         asgi.py  
5         __init__.py  
6         settings.py  
7         urls.py  
8         wsgi.py
```

- O primeiro diretório nomeado `controle-visitantes/` é apenas um container para armazenar os arquivos do nosso projeto. O nome desse diretório não é importante para o Django e pode ser alterado a qualquer momento.
- O arquivo `manage.py` é um utilitário de linha de comando que permite que a gente interaja com o projeto Django de diversas maneiras, conforme vimos anteriormente.
- O segundo diretório nomeado `controle_visitantes/` é o diretório que agrupa o pacote Python referente ao nosso projeto. O nome do diretório é o nome do nosso pacote e ele é importante pois nos auxilia no processo de importação de arquivos e funções. Sendo assim, alterar o nome deste diretório nos trará sérios problemas.
- O arquivo `controle_visitantes/__init__.py` é um arquivo vazio que diz ao Python que aquele diretório deve ser reconhecido e tratado como um pacote.
- O arquivo `controle_visitantes/settings.py` é o arquivo de configurações do nosso projeto Django. Nele nós podemos detalhar como o projeto funciona e quais definições estão disponíveis.
- O arquivo `controle_visitantes/urls.py` é o arquivo que declara as URLs do nosso projeto. Para que uma URL do nosso projeto seja acessível através do navegador, temos que declará-la neste arquivo.
- O arquivo `controle_visitantes/wsgi.py` é um ponto de integração para servidores web que implementam o WSGI, um padrão Python que descreve como os servidores devem se comunicar com as aplicações web.
- Já o arquivo `controle_visitantes/asgi.py` é o ponto de integração para servidores web utilizarem comunicação assíncrona



A partir da versão 3.0 o Django passou a dar suporte a alguns padrões de comunicação assíncrona

Criando nosso primeiro aplicativo Django

Agora que o ambiente de desenvolvimento está configurado, é hora da gente começar a colocar a mão na massa!

Conforme vimos, um projeto Django nada mais é que um pacote Python que deve seguir algumas convenções, como nomenclatura de arquivos e diretórios. O mesmo, obviamente, se estende para os aplicativos deste projeto. Sendo assim, devemos seguir uma estrutura básica dentro dos nossos aplicativos, que nada mais são que outros pacotes Python. A diferença básica entre eles é que um aplicativo deve executar uma tarefa em específico, como o gerenciamento de usuários do sistema, e um projeto é, na verdade, um conjunto de configurações e aplicativos que executam tarefas distintas. Um projeto pode ter vários aplicativos e um aplicativo pode estar presente em vários projetos.

Felizmente, podemos usar o nosso bom e velho amigo `manage.py` para nos auxiliar nessa tarefa de criar toda a estrutura necessária para um novo aplicativo. Para criar nossos aplicativos podemos utilizar o comando `startapp` passando o nome do aplicativo que deverá ser criado:

```
(env)$ python manage.py startapp usuarios
```

Após o comando ser executado, uma nova pasta, com o nome escolhido (usuarios), será criada dentro do projeto. Essa pasta terá a seguinte estrutura, que vamos entender melhor no decorrer das aulas:

```
1 usuarios/  
2     __init__.py  
3     admin.py  
4     apps.py  
5     migrations/  
6         __init__.py  
7     models.py  
8     tests.py  
9     views.py
```

Todos os aplicativos do seu projeto Django terão essa mesma estrutura devido às convenções e padrões que falamos anteriormente. Esses padrões são legais pois nos ajudam a manter uma certa organização, o que nos facilita bastante em casos de manutenções e deixa nosso código mais previsível.

Antes de começarmos a trabalhar no código de um novo aplicativo, temos que registrá-lo nas configurações do nosso projeto para que o mesmo seja reconhecido. Caso esse passo não seja executado, o projeto não saberá que a pasta `usuarios` é um aplicativo do projeto.

Vamos abrir o arquivo `settings.py` no diretório principal do projeto e procurar pela variável `INSTALLED_APPS`. Essa variável guarda o nome dos aplicativos que são utilizados no projeto. Por padrão, o Django já começa utilizando alguns aplicativos do próprio framework, como `admin`, `auth`, `sessions` e `messages`, cada um com uma finalidade específica.

```
1 INSTALLED_APPS = [  
2     "django.contrib.admin",  
3     "django.contrib.auth",  
4     "django.contrib.contenttypes",  
5     "django.contrib.sessions",  
6     "django.contrib.messages",  
7     "django.contrib.staticfiles",  
8 ]
```

Para registrar nosso aplicativo, basta colocarmos ele no final dessa lista e o Django fará todo o resto. Para manter uma melhor organização, vamos separar os aplicativos do Django dos nossos aplicativos nessa configuração. Para isso, basta inserir o código abaixo logo após a variável `INSTALLED_APPS`.

```
1 INSTALLED_APPS += [  
2     "usuarios",  
3 ]
```

Note que estamos utilizando um operador de atribuição diferente, sendo `+=` ao invés de `=`. Esse operador faz com que o valor existente na variável seja mantido e a gente acrescente o valor à direita do operador. Isto é, estamos mantendo os aplicativos do Django e adicionando os nossos. O código que teremos será o seguinte:

```
1  INSTALLED_APPS = [  
2      "django.contrib.admin",  
3      "django.contrib.auth",  
4      "django.contrib.contenttypes",  
5      "django.contrib.sessions",  
6      "django.contrib.messages",  
7      "django.contrib.staticfiles",  
8  ]  
9  
10 INSTALLED_APPS += [  
11     "usuarios",  
12 ]
```

Escrevendo nosso "Hello World"

O "Hello world" (ou "Olá mundo") é aquele famoso programa de computador que imprime na tela o texto "Hello world" e é sempre utilizado como exemplo minimalista de determinada linguagem ou framework. Não vamos deixar a tradição de lado e vamos implementar a nossa versão do programa em Django. A diferença é que, como estamos trabalhando na web, vamos exibir o texto no navegador, acessando a URL que vamos configurar. Mão na massa!

Vamos começar trabalhando no arquivo `views.py` do nosso aplicativo `usuarios`. A camada view é responsável por encapsular a lógica que recebe e responde as requisições dos nossos usuários, podendo ou não definir comportamentos específicos e buscar informações no banco de dados, por exemplo. Toda view no Django é uma função de retorno vinculada a uma URL específica. Sendo assim, não existe uma URL sem uma função de view.

```
1  from django.shortcuts import render
```

```
2 from django.http import HttpResponse
3
4 def index(request):
5     return HttpResponse("Hello world")
```

Essa é o exemplo mais básico de view que podemos escrever no Django. Ela apenas retorna um objeto do tipo `HttpResponse` que nada mais é, neste caso, que um texto simples.

Com a nossa função de view pronta, temos agora que mapear ela para ser chamada junto à uma URL. Para isso, vamos alterar o arquivo `urls.py` no diretório principal do nosso projeto. O arquivo terá um conteúdo parecido com o abaixo:

```
1 from django.urls import path
2 from django.contrib import admin
3
4 urlpatterns = [
5     path("admin/", admin.site.urls),
6 ]
```

O primeiro passo será importar o arquivo de views do nosso aplicativo `usuarios` e depois adicionar uma nova linha na lista `urlpatterns`. Vamos substituir o conteúdo do arquivo pelo seguinte código:

```
1 from django.urls import path
2 from django.contrib import admin
3
4 from usuarios.views import index
5
6 urlpatterns = [
7     path("admin/", admin.site.urls),
8
9     path(
10         "",
11         index,
12         name="index"
13     ),
14 ]
```

A função `path` recebe uma string que será a URL a ser acessada no navegador, uma função a ser executada e um nome para a URL ser identificada mais facilmente dentro do projeto. O nome da URL é bem útil para casos onde temos que renderizar o endereço completo da URL no template ou direcionar o usuário para uma página específica, por exemplo. Em breve vamos aprender mais sobre.

Feito isso, vamos utilizar o comando para iniciar nosso servidor de desenvolvimento e ver o que aparece ao acessarmos o endereço através do navegador.

```
(env)$ python manage.py runserver
```

Feito isso e não havendo erros no terminal, você deverá acessar o endereço `http://127.0.0.1:8000/` em seu navegador e visualizar nosso tão esperado "Hello world".

Capítulo 03

Escrevendo as models

Escrevendo a classe modelo

A camada *model* do nosso projeto, representada pelos arquivos `models.py`, nada mais é que uma representação exata do nosso banco de dados, sendo a classe uma tabela e seus atributos os campos dessa tabela. É nessa camada que guardamos as informações que serão disponibilizadas para outras camadas. Como o Django segue o princípio DRY - *don't repeat yourself* (algo tipo "não se repita"), o objetivo é definir o modelo de dados em um único lugar e automaticamente derivar informações e regras de negócio a partir dele.

O aplicativo `usuarios` será responsável por gerenciar os usuários do sistema. Ou seja, como vamos salvar as credenciais, quais credenciais vamos utilizar e tudo mais. Nesse ponto, é importante dizer que o Django já fornece um modelo padrão para usuários do sistema muito bom e útil para diversos cenários, mas que não dispõe de um suporte amigável para customizações. Se a gente precisar utilizar um e-mail ao invés de um nome de usuário para acessar o sistema, encontraríamos problemas com o modelo fornecido por padrão. Sendo assim, vamos criar nosso modelo personalizado para que seja possível disponibilizar o acesso à dashboard via e-mail.

O próprio Django, em sua documentação, cita exemplos em que o modelo padrão não é o mais apropriado e recomenda alternativas para contornar o problema. Uma das alternativas é sobrescrever o modelo padrão por outro que será definido por nós. Para isso, começaremos alterando o arquivo `models.py` dentro do diretório do nosso aplicativo de nome `usuarios`.


Ao abrir o arquivo, vamos apagar o conteúdo que foi colocado pelo Django e começar importando as classes `BaseUserManager`, `AbstractUser` e `PermissionsMixin`. Estas classes vão nos auxiliar na tarefa de criar um novo modelo para usuários em nosso projeto.

```
1 from django.db import models
2 from django.contrib.auth.models import (
```

```
3     BaseUserManager,  
4     AbstractBaseUser,  
5     PermissionsMixin,  
6 )
```

Vamos começar criando nossa classe `Usuario` como subclasse de `AbstractBaseUser` e `PermissionsMixin` e definir o atributo `e-mail`.

```
1 class Usuario(AbstractBaseUser, PermissionsMixin):  
2  
3     email = models.EmailField(  
4         verbose_name="E-mail do usuário",  
5         max_length=194,  
6         unique=True,  
7     )  
8
```

 Existem inúmeros tipos de campos que o Django traz por padrão, sendo eles classes contidas no pacote `models` do Django. O `EmailField` é apenas um deles e no decorrer do curso vamos conhecer mais deles.

Após definirmos o atributo `email`, vamos definir alguns outros que são obrigatórios para um modelo de usuário do Django, os campos `is_active`, `is_staff` e `is_superuser`. Vamos aproveitar também para criar a variável `USERNAME_FIELD` que é quem especifica para o Django qual campo deve ser utilizado como nome de usuário que, no nosso caso, é o campo `email`.

```
1 class Usuario(AbstractBaseUser, PermissionsMixin):  
2  
3     email = models.EmailField(  
4         verbose_name="E-mail do usuário",  
5         max_length=254,  
6         unique=True,  
7     )  
8
```

```

9      is_active = models.BooleanField(
10          verbose_name="Usuário ativo?",
11          default=True
12      )
13
14      is_staff = models.BooleanField(
15          verbose_name="Usuário é da equipe de desenvolvimento?",
16          default=False
17      )
18
19      is_superuser = models.BooleanField(
20          verbose_name="Usuário é um superusuário?"
21          default=False
22      )
23
24      USERNAME_FIELD = "email"
25
26      class Meta:
27          verbose_name = "Usuário"
28          verbose_name_plural = "Usuários"
29          db_table = "usuario"
30
31      def __str__(self):
32          return self.email

```

Após a definição destes campos e variáveis, vamos escrever uma classe interna à classe `Usuario` chamada `Meta`. Essa classe é comum a todos os modelos e serve para que a gente informe ao Django metadados. Existem diversas opções de metadados que podemos explicitar mas, por hora, vamos nos concentrar no `verbose_name`, `verbose_name_plural` e `db_table`. Estas opções deixam claro para o Django como ele deve apelidar o model, qual o apelido no plural e o nome da tabela no banco de dados referente ao model criado.

Vamos escrever também o método `__str__`. Esse método é obrigatório e chamado sempre que transformamos o objeto numa string para fins de exibição. Um dos casos em que isso ocorre é quando o Django precisa exibir a instância no *Django Admin* (não se preocupe com esse nome agora, vamos conhecê-lo melhor em breve). Sendo assim, seu método `__str__` deve sempre retornar um texto fácil e de rápida identificação para seres humanos. Feito isso, podemos partir para o segundo passo!

Escrevendo um manager personalizado

O segundo passo para substituímos o modelo de usuários padrão do Django é criarmos uma subclasse de `BaseUserManager` e sobrescrevermos os métodos `create_user` e `create_superuser`. Estes métodos são responsáveis por criar usuários e super usuários em nosso sistema e devem ser sobrescritos para se adequarem às nossas necessidades. Devido ao fato de estarmos estabelecendo uma relação de herança entre `UsuarioManager` e `BaseUserManager`, não precisamos implementar todos os atributos e métodos, pois estas informações são repassadas à classe filho.

Uma classe *manager* é uma interface que fornece informações sobre e como as queries devem ser executadas pela classe modelo quando houver interação com o banco de dados. Para cada classe modelo, existe pelo menos um *manager*.

Vamos escrever nossa classe *manager* em cima da classe `Usuario`:

```
1 class UsuarioManager(BaseUserManager):
2
3     def create_user(self, email, password=None):
4         usuario = self.model(
5             email=self.normalize_email(email),
6         )
7
8         usuario.is_active = True
9         usuario.is_staff = False
10        usuario.is_superuser = False
11
12        if password:
13            usuario.set_password(password)
14
15        usuario.save()
16
17        return usuario
18
19    def create_superuser(self, email, password):
20        usuario = self.create_user(
21            email=self.normalize_email(email),
22            password=password,
23        )
24
25        usuario.is_active = True
26        usuario.is_staff = True
27        usuario.is_superuser = True
28
29        usuario.set_password(password)
30        usuario.save()
```

```

31
32     return usuario
33
34 class Usuario(models.Model):
35     # código abaixo omitido...

```

Com a classe e métodos escritos, agora temos um *manager* com a função de criar usuários conforme a nossa necessidade. Com isso, nosso sistema está quase pronto para criar usuários, faltando apenas explicitar que a classe modelo deve utilizar este *manager* como padrão. Para isso, vamos adicionar um atributo na classe modelo com o nome do manager que queremos utilizar.

O Django utiliza o nome `objects` para o manager padrão da classe, sendo assim, vamos apenas sobrescrever o manager padrão pelo que nós criamos. Vamos criar o atributo `objects` na classe `Usuario` atribuindo a ele a classe `UsuarioManager`.

```

1  class Usuario(AbstractBaseUser, PermissionsMixin):
2      email = models.EmailField(
3          verbose_name="E-mail do usuário",
4          max_length=254,
5          unique=True,
6      )
7
8      is_active = models.BooleanField(
9          "usuario ativo?",
10         default=True
11     )
12
13     is_staff = models.BooleanField(
14         "usuario é da equipe de desenvolvimento?",
15         default=False
16     )
17
18     is_superuser = models.BooleanField(
19         "usuario é um superusuário?",
20         default=False
21     )
22
23     USERNAME_FIELD = "email"
24
25     objects = UsuarioManager()
26
27     class Meta:


```

```
28         verbose_name="Usuário"
29         verbose_name_plural = "Usuários"
30         db_table = "usuario"
31
32     def __str__(self):
33         return self.email
```

Alterando o modelo padrão nas configurações

Para dizermos ao Django que ele deve utilizar a nossa classe modelo ao invés do modelo padrão para usuários do sistema, temos que adicionar a variável `AUTH_USER_MODEL` ao arquivo `settings.py` e apontar para a classe a ser utilizada.

```
AUTH_USER_MODEL = "usuarios.Usuario"
```

 Ao escrevermos isso, estamos dizendo "hey, Django, use a classe Usuario do aplicativo usuarios como modelo de usuários do sistema".

Criando as tabelas do nosso banco de dados

Com o trecho de código que escrevemos o Django já é capaz de executar instruções para criação da tabela `usuario` no nosso banco de dados e disponibilizar uma API de acesso aos objetos do tipo `Usuario`, mas antes precisamos avisar ao Django a respeito destas alterações. Sempre que ocorrer alguma alteração nos modelos ou você criar um novo modelo, é necessário avisar ao Django para que ele cuide de toda a parte anterior à efetivação das mudanças no banco de dados.

Para isso, quando escrevemos uma classe modelo, temos que executar o comando `makemigrations`. Esse comando vai criar um arquivo de migração contendo todas as alterações que devem ser feitas no banco de dados, tais como criação de tabelas com determinados atributos, alteração nos atributos, dentre outros. Vamos executar o seguinte comando:

```
(env)$ python manage.py makemigrations usuarios
```

Ao executar o comando `makemigrations`, você está dizendo ao Django para armazenar as alterações realizadas em forma de *migração*. Uma *migração* nada mais é que um arquivo de texto contendo todos os passos que devem ser executados para efetivação das alterações no banco de dados. Aparecerá algo como isso na tela do terminal:

```
1 Migrations for 'usuarios':
2   usuarios/migrations/0001_initial.py
3     - Create model Usuario
```

Existe também um comando para rodar as migrações e gerenciar o *schema* do banco de dados de forma automática. O comando `migrate` é quem vai reunir todas as migrações que ainda não foram executadas e aplicar elas em seu banco de dados - isto é, vai sincronizar seu banco de dados com as informações que estão na classe modelo. Para efetuar as alterações em nosso banco de dados vamos executar o comando:

```
(env)$ python manage.py migrate
```

Migrações são um recurso poderoso pois nos permitem alterar as classes modelos ao longo do tempo sem a necessidade de manipular nosso banco de dados. O comando `migrate` é especialista em atualizar nosso banco de dados em tempo real sem perder dados.

```
1 Operations to perform:
2   Apply all migrations: admin, auth, contenttypes, sessions, usuarios
3
4 Running migrations:
5   Applying contenttypes.0001_initial... OK
6   Applying contenttypes.0002_remove_content_type_name... OK
7   Applying auth.0001_initial... OK
8   Applying auth.0002_alter_permission_name_max_length... OK
9   Applying auth.0003_alter_user_email_max_length... OK
```

```
10 Applying auth.0004_alter_user_username_opts... OK
11 Applying auth.0005_alter_user_last_login_null... OK
12 Applying auth.0006_require_contenttypes_0002... OK
13 Applying auth.0007_alter_validators_add_error_messages... OK
14 Applying auth.0008_alter_user_username_max_length... OK
15 Applying auth.0009_alter_user_last_name_max_length... OK
16 Applying auth.0010_alter_group_name_max_length... OK
17 Applying auth.0011_update_proxy_permissions... OK
18 Applying usuarios.0001_initial... OK
19 Applying admin.0001_initial... OK
20 Applying admin.0002_logentry_remove_auto_add... OK
21 Applying admin.0003_logentry_add_action_flag_choices... OK
22 Applying sessions.0001_initial... OK
```

Não se assuste com as tantas letras que vão aparecer no terminal. Elas nos informam quais aplicativos tiveram operações executadas e quais arquivos de migração foram utilizados para a migração em questão.

Agora temos nosso modelo personalizado de usuários criado e ativado em nosso banco de dados e estamos prontos para criar usuários.

Criando um super usuário

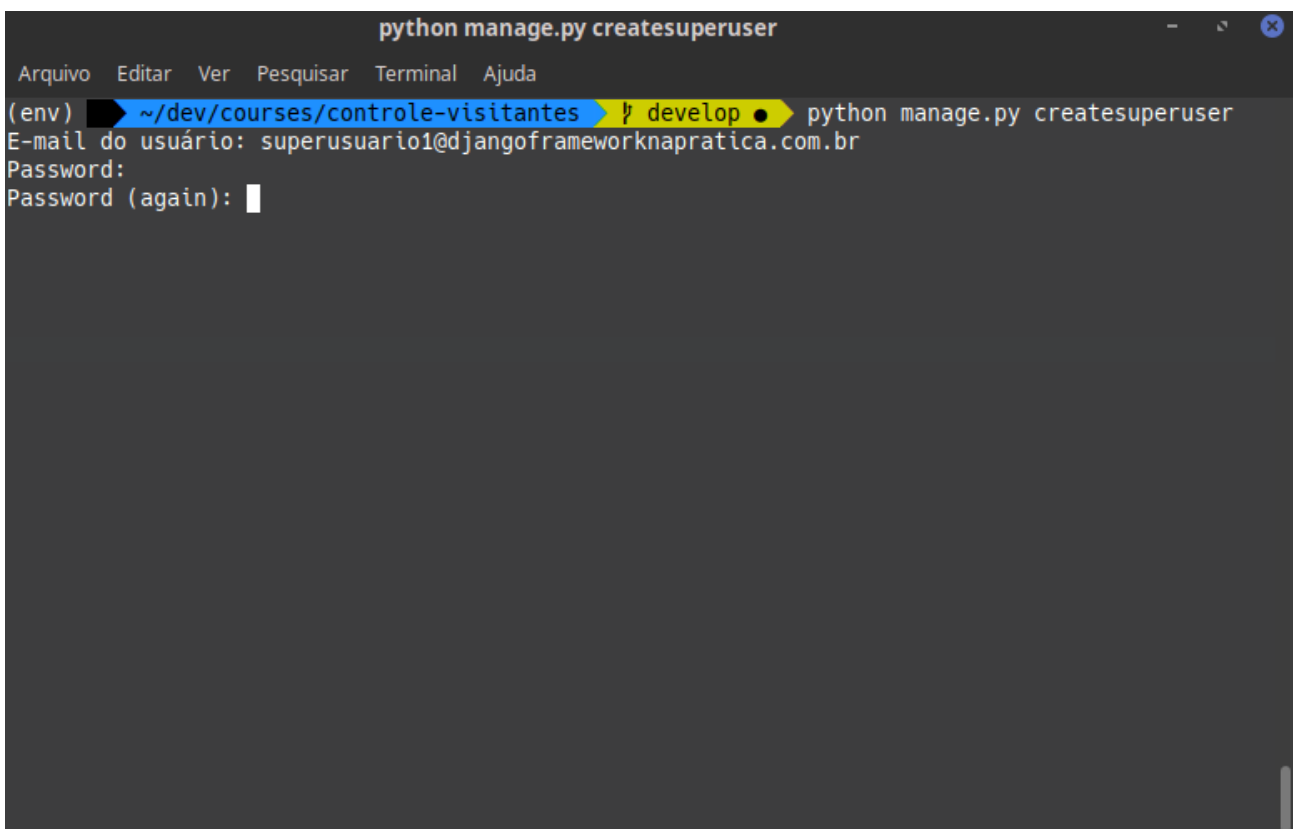
A gente viu que o Django segue uma filosofia que tenta aproveitar o máximo de coisas para evitar que a gente fique repetindo código. Pensando nisso, os desenvolvedores do framework disponibilizam um painel administrativo com funções para adicionar, alterar e deletar conteúdo com base nas classes modelo do nosso projeto.

O Django foi desenvolvido em um ambiente de redação, onde havia uma clara separação entre “produtores de conteúdo” e o site “público”. Gerentes utilizavam o sistema para adicionar notícias, eventos, resultados de esportes, por exemplo, e o conteúdo era exibido no site público.

Como a administração do Django não foi desenvolvida para ser utilizada pelos visitantes do site, mas sim pelos gerentes, temos que criar um tipo diferente de usuário, que seria nosso "super usuário". Para isso basta utilizarmos o comando `createsuperuser` do nosso bom e velho amigo `manage.py` :

```
(env)$ python manage.py createsuperuser
```

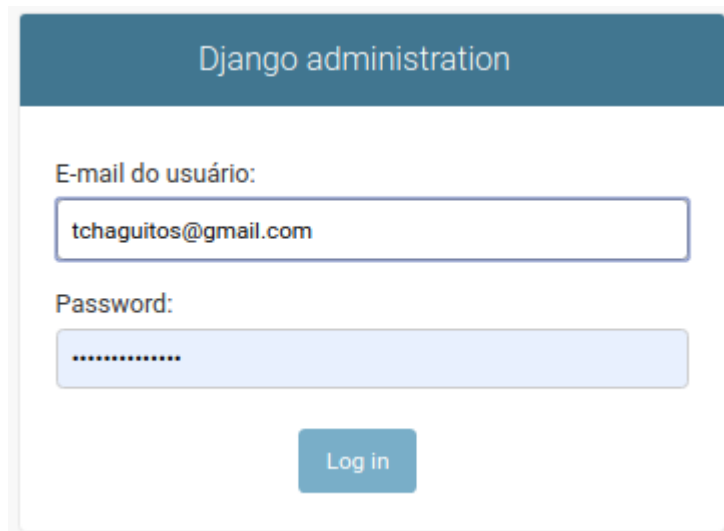
Ao executar o comando, o terminal ficará travado esperando que a gente informe o e-mail a ser utilizado pelo super usuário. Após o e-mail, temos que definir a senha e repetir essa senha. Não ocorrendo erros, temos o nosso super usuário criado e pronto para acessar o painel administrativo do Django.



```
python manage.py createsuperuser
Arquivo  Editar  Ver  Pesquisar  Terminal  Ajuda
(env) ~/dev/courses/controle-visitantes develop python manage.py createsuperuser
E-mail do usuário: superusuario1@djangoframeworknapratica.com.br
Password:
Password (again):
```

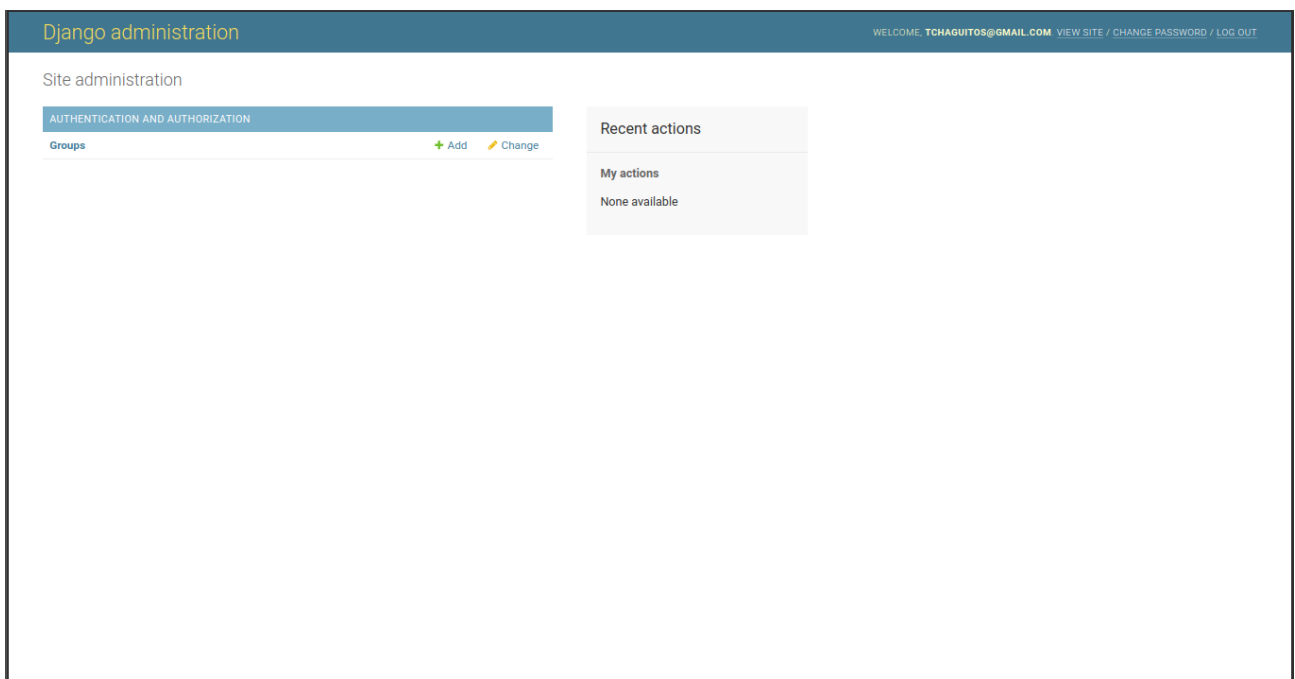
Tela de um terminal Linux após a execução do comando `createsuperuser` do Django. O terminal espera que o usuário insira E-mail e Senha para criar um super usuário

Vamos iniciar novamente nosso servidor de desenvolvimento e acessar o endereço <http://127.0.0.1:8000/admin/> em nosso navegador e vamos nos deparar com a tela de login do Django Admin.

The image shows the Django administration login interface. It has a dark blue header with the text "Django administration". Below the header, there are two input fields: "E-mail do usuário:" with the value "tchaguitos@gmail.com" and "Password:" with a masked password represented by dots. A blue "Log in" button is positioned below the password field.

Formulário de autenticação para acesso ao Django Admin. O formulário exibe os campos "E-mail do usuário" e "Senha", além de um botão para Log in

Utilize e-mail e senha informados na hora da criação do super usuário para acessar e, dando tudo certo, você será direcionado para a página inicial da administração do Django:

The image shows the Django administration dashboard. The header is dark blue with "Django administration" on the left and a welcome message "WELCOME TCHAGUITOS@GMAIL.COM" along with links "VIEW SITE / CHANGE PASSWORD / LOG OUT" on the right. The main content area is white. On the left, there's a "Site administration" section with a blue bar for "AUTHENTICATION AND AUTHORIZATION" and a link to "Groups" with "Add" and "Change" buttons. On the right, there's a "Recent actions" section with a "My actions" link and a message "None available".

Tela inicial do Django Admin, onde é exibida as ações recentes do usuário, além de uma mensagem de boas vindas e links para alterar senha e sair do Admin

Conhecendo o Django Admin

Ao acessarmos o painel administrativo, é possível notar que o aplicativo usuários não está sendo exibido. Isso porque para que nossas classes modelo sejam identificadas pelo painel administrativo é necessário alterar o arquivo `admin.py`. Esse arquivo sempre existirá dentro de um aplicativo criado utilizando o `manage.py` e é quem irá dizer ao Django que nossa classe deve ter uma interface de administração.

Vamos abrir o arquivo `usuarios/admin.py` e importar nossa classe `Usuario`:

```
1 from django.contrib import admin
2 from usuarios.models import Usuario
```

Após a importação, temos apenas que registrar a classe no admin. Para isso, basta utilizar o método `admin.site.register` passando a classe `Usuario` como argumento:

```
1 from django.contrib import admin
2 from usuarios.models import Usuario
3
4 admin.site.register(Usuario)
```

Feito isso, vamos voltar ao painel administrativo do Django e podemos notar que agora existe uma seção referente ao aplicativo usuarios. O mais legal é que os nomes que definimos em `verbose_name` e `verbose_name_plural` são utilizados aqui, além do valor que definimos como retorno do método `__str__`.

Alterações necessárias nas configurações

Nosso projeto já está rodando sem problemas e possui um modelo personalizado de usuários. Isso já é bem legal, mas ainda vamos realizar algumas alterações nas configurações para que ele se adeque ainda mais às nossas necessidades. Neste momento, vamos alterar configurações de fuso horário e idioma utilizado.

O Django possui integrado um módulo para internacionalização e localização. Esses módulos são interessantes pois permitem que a aplicação funcione em idiomas e formatos diferentes com base nas preferências do usuário. Um exemplo de funcionamento do módulo de internacionalização é o próprio painel administrativo do Django que funciona dessa maneira.

Para alterar o idioma padrão utilizado no projeto, vamos no arquivo `settings.py` e procurar pela variável `LANGUAGE_CODE`. Essa variável recebe uma string referente ao identificador do idioma e país de origem com base na especificação que define os formatos de idiomas para serem utilizados. Para nosso caso, utilizaremos a string `pt-BR` que faz com que o Django utilize português do Brasil como idioma principal do projeto. Vamos substituir o valor da variável pelo seguinte valor:

```
LANGUAGE_CODE = "pt-br"
```

Com isso, todas as mensagens e textos exibidos no painel administrativo já serão traduzidos automaticamente pelo módulo de internacionalização.



Tela inicial do Admin agora com o aplicativo Usuários sendo exibido

Nosso próximo passo é alterar o fuso horário padrão do projeto. Essa configuração é importante pois não queremos que datas erradas sejam exibidas para nossos usuários. Por padrão, o Django trabalha e exibe as datas no fuso horário `America/Chicago` (ou `UTC`) mas como esse não é o fuso horário para a nossa região, vamos inserir a configuração correta para nós. Existe também uma lista de fusos horários disponíveis e como podemos especificar eles através de uma string simples, como acontece no caso dos identificadores de idioma. Vamos utilizar o fuso horário `America/Sao_Paulo`. Ainda no arquivo `settings.py`, vamos encontrar a variável `TIME_ZONE` e alterar seu valor para o da nossa região:

```
TIME_ZONE = "America/Sao_Paulo"
```



Antes de realizar a modificação, observe os horários no painel administrativo do Django. O atributo "último login" e o histórico de modificações do usuário aparecem em um horário que não é o nosso.

Com isso, além do nosso projeto possuir um modelo personalizado de usuários, agora ele exibe mensagens e horários no idioma e fuso horário que é o mais apropriado para nossa região. A partir de agora é que as coisas vão ficar mais interessantes e o projeto começará a tomar forma! Aperte os cintos!

Capítulo 04

Criando aplicativo para gerenciar porteiros

Aprendemos como instalar nossas dependências, como iniciar um novo projeto e já até escrevemos um modelo personalizado de usuários. Com isso, podemos dizer que já temos uma base sólida para iniciar, de fato, a construção dos módulos que estão diretamente ligados aos requisitos que o sistema deverá atender.

A partir dos requisitos que temos, é possível identificar que existe a necessidade de que os porteiros do condomínio tenham acesso a uma página para registrar os visitantes. Com isso em mente, podemos concluir:

- Os porteiros terão um usuário para acessar o sistema
- Deverá existir um modelo para representar os porteiros do condomínio
- Os porteiros deverão ter acesso a uma dashboard com funcionalidades específicas
- Deverá existir uma página para um porteiro registrar um visitante no condomínio

Vamos nos concentrar em cada item separadamente e, por agora, vamos focar no desenvolvimento da classe modelo que vai representar nossos porteiros.

Já falamos que os aplicativos de um projeto Django devem executar tarefas específicas, como gerenciar os usuários do sistema - que é exatamente o que o nosso aplicativo `usuarios` faz. Sendo assim, vamos iniciar um novo aplicativo com nome de "porteiros" que será responsável por gerenciar tudo referente aos porteiros. Vamos utilizar nosso bom e velho amigo `manage.py` :

```
(env)$ python manage.py startapp porteiros
```

Conforme vimos, será criado um diretório com o nome informado para representar o módulo do nosso aplicativo e, além disso, temos que instalar nosso novo aplicativo nas configurações do projeto. Vamos abrir nosso arquivo de configurações e inserir o aplicativo "porteiros" na variável `INSTALLED_APPS` :

```
1 INSTALLED_APPS += [  
2     "usuarios",  
3     "porteiros",  
4 ]
```



Caso não se lembre, tudo bem... nosso arquivo de configurações é o `settings.py` .

Agora que criamos e registramos nosso aplicativo, vamos partir para a construção do modelo que vai representar os porteiros.

Escrevendo as models do nosso aplicativo de porteiros

Assim como escrevemos a classe `Usuario` para representar e descrever nossos usuários, vamos escrever a classe `Porteiro` que será a representação de nossos porteiros. Em nosso documento de requisitos é possível verificar que os porteiros devem possuir os seguintes atributos:

- Usuário para acesso ao sistema (e-mail)
- Nome completo
- CPF
- Telefone
- Data de nascimento

O primeiro passo será criar a classe `Porteiro` no arquivo `porteiros/models.py` . Como já sabemos os atributos que um porteiro deve ter e já usamos o campo do tipo `CharField` anteriormente, começaremos trabalhando nos atributos que são deste tipo (nome completo, CPF e telefone):

```
1 from django.db import models  
2
```

```

3 class Porteiro(models.Model):
4
5     nome_completo = models.CharField(
6         verbose_name="Nome completo",
7         max_length=194
8     )
9
10    cpf = models.CharField(
11        verbose_name="CPF",
12        max_length=11,
13    )
14
15    telefone = models.CharField(
16        verbose_name="Telefone de contato",
17        max_length=11,
18    )

```

Nossos três primeiros campos receberão apenas os argumentos `verbose_name` e `max_length`. O primeiro para dizer por qual nome devemos chamar o campo e o segundo para dizer o tamanho máximo permitido.



Apesar de CPF e telefone serem representados por números, possuem características e particularidades que fazem com que a gente trabalhe com eles como se fossem texto. Além disso, o telefone está com tamanho 11 pois vamos trabalhar com o DDD + 9 dígitos.

Conhecendo o campo `DateField`

Sabemos também que, por exigência do setor de RH, é necessário informar a data de nascimento do porteiro para realização do cadastro. Vamos adicionar mais um campo ao nosso modelo, agora com nome de `data_nascimento` e o tipo data (`DateField`).

Os campos do tipo data representam datas - obviamente, mas é interessante a gente prestar atenção ao fato de que esses campos são representados por instâncias do tipo `datetime.date` que é como as datas são tratadas no Python e podem receber dois argumentos que ainda não conhecemos: `auto_now` e `auto_now_add`.

O argumento `auto_now` diz para o Django que é necessário atualizar o valor sempre que nosso objeto for salvo. Se nós definirmos ele como `True` (verdadeiro), o valor de `data_nascimento` será atualizado para um valor atual sempre que atualizarmos as informações de um porteiro. No caso do argumento `auto_now_add`, ele diz para o Django que é necessário inserir a data atual como valor do atributo no momento da criação do registro no banco de dados e, feito isso, o valor não é atualizado novamente.

Como nosso objetivo é informar uma data que represente a data de nascimento do porteiro, vamos dizer ao Django que não é necessário preencher o campo automaticamente apenas setando os valores dos argumentos como `False`.

```
1  from django.db import models
2
3  class Porteiro(models.Model):
4
5      nome_completo = models.CharField(
6          verbose_name="Nome completo",
7          max_length=194
8      )
9
10     cpf = models.CharField(
11         verbose_name="CPF",
12         max_length=11,
13     )
14
15     telefone = models.CharField(
16         verbose_name="Telefone de contato",
17         max_length=11,
18     )
19
20     data_nascimento = models.DateField(
21         verbose_name="Data de nascimento",
22         auto_now_add=False,
23         auto_now=False
24     )
```

Conhecendo o campo OneToOneField

Até o momento utilizamos apenas campos do tipo texto e data, que são campos bastante úteis, mas que não são os ideais para todos os tipos de dados necessários para nosso

modelo. Sendo assim, temos que conhecer um pouco mais dos tipos de campos disponíveis nos modelos do Django.

Sabemos que um dos requisitos é que haja um usuário do sistema para cada porteiro. Sendo assim, podemos dizer que é necessário vincular um usuário ao modelo que irá representar os porteiros em nosso sistema.

Uma vez que já temos o nosso modelo de usuários definido, temos que apenas vinculá-lo ao modelo de porteiros. Para tal, vamos conhecer o campo `OneToOne` que é quem vai tornar explícito esse vínculo entre os modelos. Essencialmente, o campo `OneToOne` representa uma relação "um para um". Isto é, para cada porteiro existirá um único usuário.

Para utilizar esse tipo de campo é bem fácil e funciona de modo bem parecido com os outros que já conhecemos, mudando apenas o argumento obrigatório que deve ser passado para o campo funcionar corretamente. Acima do atributo `nome_completo`, vamos começar definindo o atributo `usuario` que será igual a `models.OneToOneField()`.

```
1  from django.db import models
2
3  class Porteiro(models.Model):
4
5      usuario = models.OneToOneField()
6
7      nome_completo = models.CharField(
8          verbose_name="Nome completo",
9          max_length=194
10     )
11
12     cpf = models.CharField(
13         verbose_name="CPF",
14         max_length=11,
15     )
16
17     telefone = models.CharField(
18         verbose_name="Telefone de contato",
19         max_length=11,
20     )
21
22     data_nascimento = models.DateField(
23         verbose_name="Data de nascimento",
24         auto_now_add=False,
25         auto_now=False
26     )
```

Conforme falamos, os argumentos que devem ser passados variam de campo para campo. Para o campo `OneToOne`, é necessário dizer qual modelo queremos que seja vinculado. Isto é, temos que dizer que o atributo `usuario` do modelo `Porteiro` deverá receber como valor uma instância da classe `Usuario`. Sempre que formos criar um `Porteiro` temos que criar também um `Usuario` e vinculá-lo ao `Porteiro` criado.

Vamos primeiro dizer qual é o modelo que queremos vincular ao atributo. Para o caso é o modelo `Usuario` do aplicativo `usuarios`. Podemos fazer isso passando apenas um texto contendo o caminho do modelo (`usuarios.Usuario`). Os outros dois argumentos são o `verbose_name`, que nós já conhecemos e o `on_delete`, que é um nome novo para nós. O `on_delete` diz para o Django o que deve ser feito com o registro do porteiro caso o usuário seja deletado. Nesse caso, se o usuario for removido da base de dados, o mesmo acontecerá com o porteiro.

```
1 from django.db import models
2
3 class Porteiro(models.Model):
4
5     usuario = models.OneToOneField(
6         "usuarios.Usuario",
7         verbose_name="Usuário",
8         on_delete=models.CASCADE
9     )
10
11     # código abaixo omitido...
```

Para finalizar, vamos escrever as classes e métodos que devem acompanhar todos os modelos do Django. Vamos começar com a classe `Meta` e depois escrever o método `__str__`. Como vimos, devemos escrever essa classe e método para definir informações de como o modelo pode ser chamado, o nome da tabela que irá armazenar as informações no banco de dados e como a instância é exibida ao ser transformada em string. Nossa classe `Porteiro` ficará assim:

```
1 from django.db import models
```



```

2
3 class Porteiro(models.Model):
4
5     usuario = models.OneToOneField(
6         "usuarios.Usuario",
7         verbose_name="Usuário",
8         on_delete=models.CASCADE
9     )
10
11     nome_completo = models.CharField(
12         verbose_name="Nome completo", max_length=194
13     )
14
15     cpf = models.CharField(
16         verbose_name="CPF",
17         max_length=11,
18     )
19
20     telefone = models.CharField(
21         verbose_name="Telefone de contato",
22         max_length=11,
23         blank=True
24     )
25
26     data_nascimento = models.DateField(
27         verbose_name="Data de nascimento",
28         auto_now=False
29     )
30
31     class Meta:
32         verbose_name = "Porteiro"
33         verbose_name_plural = "Porteiros"
34         db_table = "porteiro"
35
36     def __str__(self):
37         return self.nome_completo

```

Registrando nossa aplicação no Admin do Django

O próximo passo que vamos executar é tornar o nosso modelo de porteiros visível para o Admin do Django. Como já sabemos fazer isso, vai ser bem rápido!

Vamos abrir o arquivo `admin.py` do nosso aplicativo porteiros, importar a classe `Porteiro` e passá-la como argumento do método `admin.site.register()`.

```
1 from django.contrib import admin
2 from porteiros.models import Porteiro
3
4 admin.site.register(Porteiro)
```

Aplicando as alterações em nosso banco de dados

Feito isso, vamos agora validar o código escrito e criar as migrações do modelo criado utilizando o comando `makemigrations` .

```
(env)$ python manage.py makemigrations porteiros
```

Se ocorrer bem, vamos receber as seguintes informações em nosso terminal:

```
1 Migrations for 'porteiros':
2   porteiros/migrations/0001_initial.py
3   - Create model Porteiro
```

Com todas as informações necessárias para executar as alterações no banco de dados armazenadas em forma de migração, vamos pedir ao Django que efetue essas alterações em nosso banco. Para isso vamos executar o comando `migrate` .

```
(env)$ python manage.py migrate
```

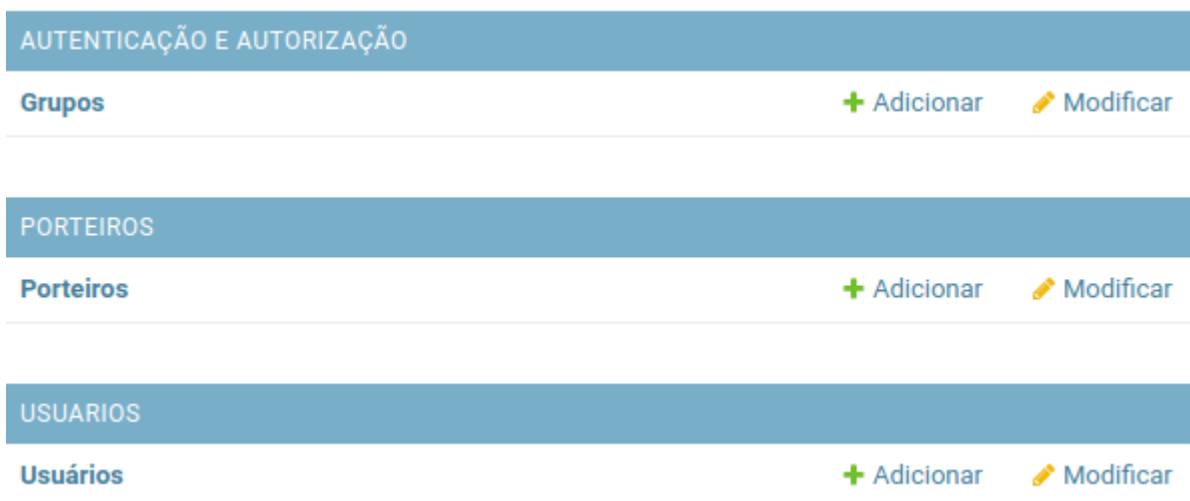
E, se tudo ocorrer bem, vamos receber em nosso terminal:

```
1 Operations to perform:
2   Apply all migrations: admin, auth, contenttypes, porteiros, sessions, us
```

```
3
4 Running migrations:
5   Applying porteiros.0001_initial.py... OK
```

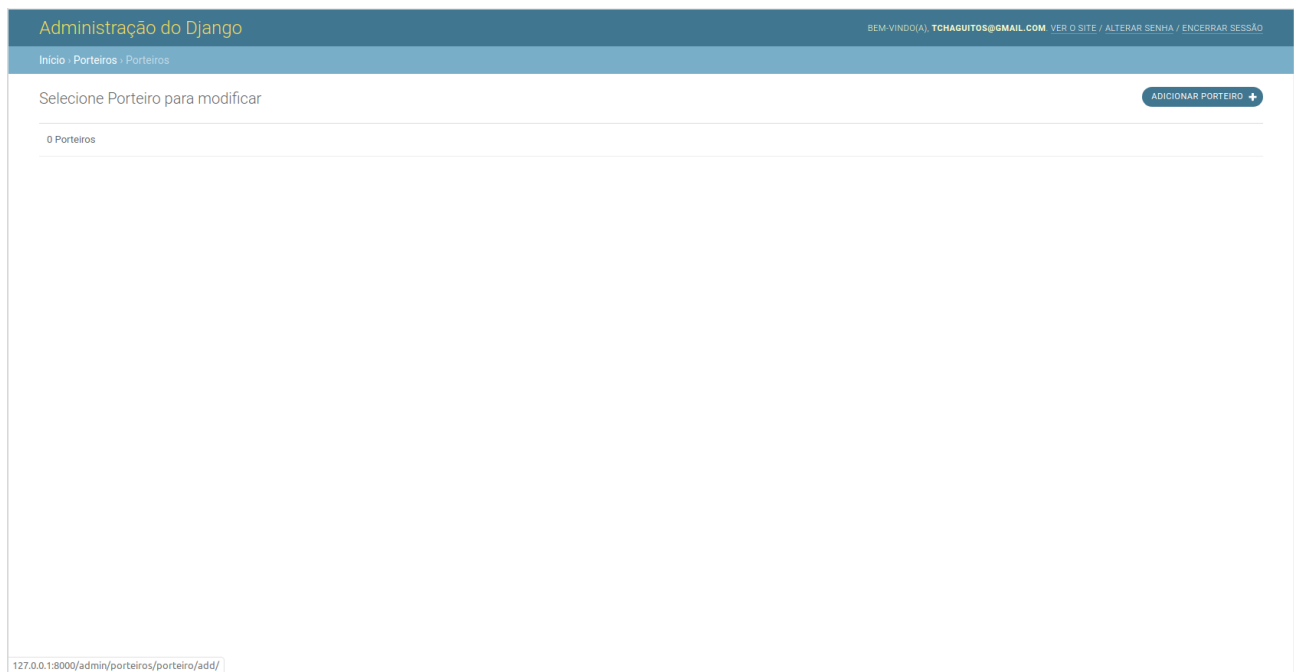
Criando porteiro através do Admin do Django

Como sabemos, o Django disponibiliza uma interface de administração para as classes de modelo criadas e registradas nos arquivos `admin.py`. Até agora, tudo que fizemos foi apenas visualizar e alterar informações de usuários, mas é possível fazer bem mais com o Admin do Django. Vamos acessar o admin através do navegador e clicar no item porteiros.




Captura tirada do Admin do Django focando nos links para a exibição de Grupos, Usuário e agora Porteiros também

A próxima tela deverá exibir a lista de porteiros registrados em nosso sistema. Como ainda não temos porteiros registrados, temos apenas a informação de que existem "0 Porteiros" e opção de adicionar um porteiro.




Tela listando os porteiros registrados em nosso banco de dados (no caso, ainda não temos porteiros registrados e, por isso, o texto "0 porteiros" é exibido)

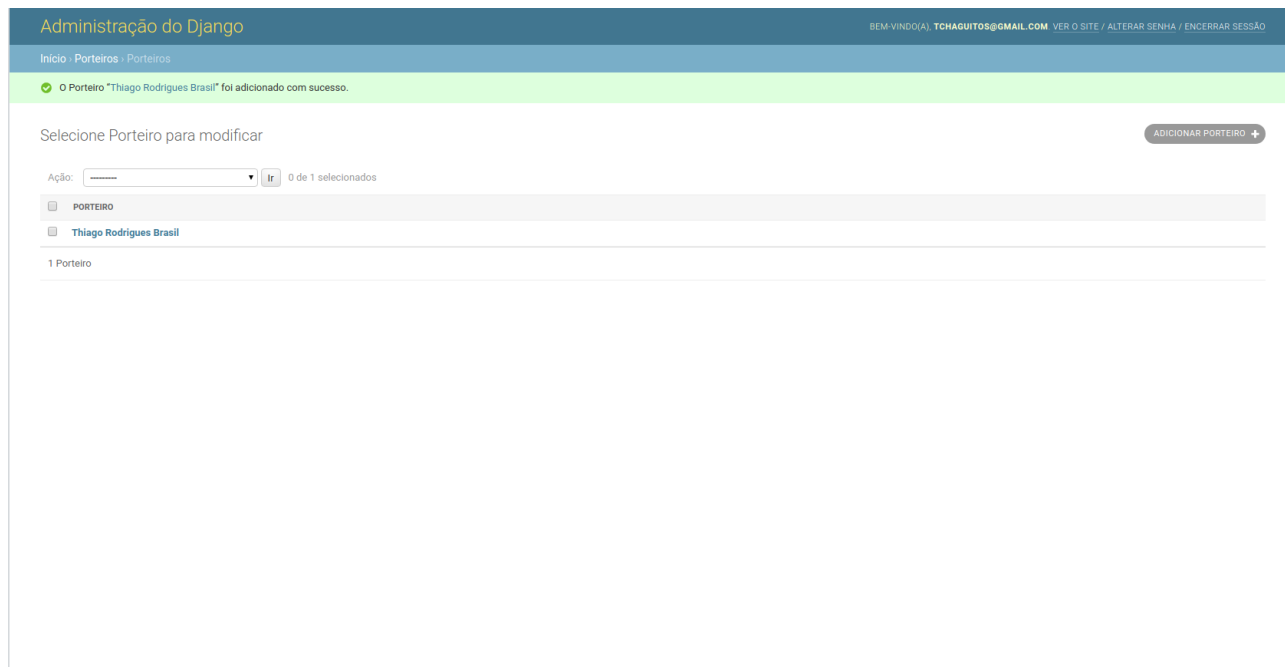
Vamos clicar no botão "adicionar porteiro" para ter acesso ao formulário de cadastro de porteiros. Para adicionar um porteiro, basta preencher as informações obrigatórias que são usuário, nome completo, CPF e data de nascimento. Note que os campos obrigatórios ficam destacados em negrito.

 O Admin do Django é tão interessante que já disponibiliza um elemento do tipo select para selecionarmos o usuário que será vinculado ao novo porteiro. Bacana, não?

Vamos selecionar o usuário criado através do terminal com o comando `createsuperuser` e preencher as informações necessárias. Esteja livre para preencher da sua maneira!

 Você deve utilizar o formato DD/MM/AAAA para a data de nascimento do porteiro

Se tudo estiver certo com os dados informados, o Django Admin vai nos redirecionar para a tela que lista os porteiros e mostrar uma mensagem de sucesso. Note que o Django já exibe a mensagem personalizada com o nome do porteiro criado e link para visualização das informações.



Tela listando os porteiros registrados em nosso banco de dados, agora com nosso primeiro porteiro registrado

Agora que temos um porteiro criado, podemos partir para a construção da dashboard que contará com as funções descritas em nosso documento de requisitos.

Capítulo 05

Configurando a aplicação para trabalhar com arquivos estáticos e templates HTML

Nos capítulos anteriores, iniciamos o projeto e criamos toda a estrutura necessária para administração de usuários do sistema e porteiros que serão os responsáveis por operar a dashboard proposta. Além disso, também preparamos o ambiente de desenvolvimento e aprendemos bastante sobre detalhes técnicos do funcionamento do Django.

Focamos nosso trabalho nos arquivos `models.py` e `admin.py` e também conhecemos o poder existente do Admin que o Django nos disponibiliza. Como nosso objetivo é desenvolver uma dashboard personalizada para exibir as informações dos visitantes do condomínio e implementar funcionalidades específicas, a partir de agora, trabalharemos para desenvolver os templates que irão apresentar as informações necessárias e executar as funcionalidades que vamos desenvolver. Sendo assim, posso dizer que a partir de agora as coisas começam a ficar mais interessantes!

Neste próximo módulo, aprenderemos a configurar o Django para trabalhar com arquivos estáticos (CSS e JS) e templates HTML.

⚠ O Django, por padrão, vem configurado para que já seja possível trabalhar com templates HTML, mas vamos alterar as configurações para que a haja uma maior organização dos arquivos e de modo que a gente agrupe todos os template numa só pasta

Criando a pasta templates em nosso projeto

Sendo um framework web, o Django precisa fornecer uma maneira de gerar os templates de forma dinâmica, de modo que seja possível exibir valores específicos e atender os diversos cenários. Essencialmente, um template é constituído por uma parte estática, que são os arquivos CSS e JS e partes que se repetem, e uma parte onde serão exibidas as informações desejadas, que variam de acordo com cada cenário.



Imagine em nosso caso em que os porteiros deverão registrar visitantes. Para cada visitante, teremos informações diferentes e, desta forma, o template deverá ser capaz de exibir essas informações de acordo com o contexto de cada visitante

Para resolver esse problema, o Django nos fornece uma *engine* rica e poderosa capaz de executar funções condicionais, loops, exibir valores e ainda possui diversas funcionalidades que podem ser utilizadas diretamente nos templates HTML através de tags.



Uma engine de template nada mais é que uma aplicação que visa facilitar o processo de criação de templates HTML dinâmicos e tornar o processo de envio e exibição de informações nos templates menos burocrático

Por padrão, o Django vem configurado para procurar os templates dentro de cada aplicativo. Isto é, em cada aplicativo deverá existir uma pasta **templates** para armazenar os templates referentes ao aplicativo em questão. Todavia, para uma melhor organização, utilizaremos uma pasta externa para armazenar todos os arquivos de templates do projeto.

Para isso, começaremos alterando o arquivo `settings.py` do nosso projeto. Nesse arquivo é possível encontrar a variável `TEMPLATES`, que é responsável por definir as configurações de template do projeto, como *engine* a ser utilizada, diretórios que armazenam os templates, dentre outras.

A variável `TEMPLATES` é uma lista que recebe um dicionário contendo valores específicos, tais como `BACKEND`, `DIRS`, `APP_DIRS` e `OPTIONS`, cada um com uma função específica. No nosso caso, vamos alterar o valor `DIRS` de uma lista vazia para uma lista contendo a string "templates", que é o nome da pasta que utilizaremos para armazenar os templates na raiz do projeto. A variável `TEMPLATES` ficará da seguinte forma:

```
1 TEMPLATES = [  
2     {
```

```
3     "BACKEND": "django.template.backends.django.DjangoTemplates",
4     "DIRS": ["templates"],
5     "APP_DIRS": True,
6     "OPTIONS": {
7         "context_processors": [
8             "django.template.context_processors.debug",
9             "django.template.context_processors.request",
10            "django.contrib.auth.context_processors.auth",
11            "django.contrib.messages.context_processors.messages",
12        ],
13    },
14 },
15 ]
```

Para facilitar as coisas e economizar um pouquinho de tempo, você pode fazer download da pasta **templates** zipada, extrair os arquivos e colocá-la na raiz do seu projeto:



Iniciar o download

templates.zip - 2KB

Criando a pasta static em nosso projeto

Feito isso, vamos agora definir as configurações para os arquivos estáticos do nosso projeto. Assim como para os templates, o Django também nos dá toda a estrutura necessária para trabalharmos com arquivos estáticos.

Por "arquivos estáticos", entenda arquivos do tipo CSS, JS (javascript) e imagens que serão utilizadas em nossos templates, tais como logotipo, imagem padrão para avatar de usuários, dentre outras.

Para realizarmos a configuração, vamos novamente alterar o arquivo `settings.py`. Ao final do arquivo, você vai encontrar a variável `STATIC_URL` que é onde nossos arquivos estáticos devem ficar, ou seja, na pasta `static` na raiz do projeto:

```
STATIC_URL = "/static/"
```


Abaixo da variável `STATIC_URL`, coloque também o seguinte trecho de código:

```
1 STATICFILES_DIRS = [  
2     os.path.join(BASE_DIR, "static")  
3 ]
```

Desta forma estamos dizendo para o Django que os arquivos estáticos devem ser procurados na pasta **static** na raiz do projeto. Para facilitar e economizar tempo novamente, faça download da pasta zipada clicando no link abaixo (agora da pasta **static**, claro) e a coloque na raiz do projeto:



Iniciar o download

static.zip - 2MB

Faça o download e coloque a pasta **static** na raiz do projeto, juntamente com a pasta **templates**. Feito isso, já podemos utilizar templates HTML e arquivos estáticos em nosso projeto.


Criando views que renderizam templates

O HTML (Linguagem de Marcação de HiperTexto) é o bloco de construção mais básico da pilha de tecnologias que compõem a web, mas é ela que dá significado e define a estrutura do conteúdo das páginas. Existem também tecnologias que descrevem aparência/apresentação (CSS) e funcionalidade/comportamento (Javascript) de uma página web (inclusive já falamos um pouquinho delas por aqui quando falamos sobre arquivos estáticos).

Basicamente, um arquivo de template é um arquivo de texto com extensão `.html`. Os navegadores interpretam esses arquivos de texto e cuidam de exibir exatamente da maneira que você enxerga pelo seu monitor.

Bacana não? Além disso, o HTML ajuda a dizer para os motores de busca o que é relevante, o que é texto, o que é imagem e tudo mais. Sendo assim, o HTML tem um papel

fundamental dentro da web!

 E ah, não se assuste com a palavra **HiperTexto** no nome, hipertexto são apenas os links entre as páginas que se conectam na web.

Como o Django já nos dá tudo que é necessário para criarmos aplicações web, ele também nos dá a possibilidade de criarmos views que renderizam templates. Isso significa que a partir de agora, ao invés de ser exibido um texto ao acessarmos uma URL através do navegador, como fizemos anteriormente, vamos dizer para o Django que é necessário exibir um template HTML, afim de exibir as informações de forma estruturada e de modo que fiquei fácil a compreensão para nossos usuários.

Conhecendo a função render

Para que o Django exiba um template ao invés de um texto em tela, precisaremos alterar o retorno da nossa view chamada `index`. Antes de seguir, vamos trabalhar um pouco a memória e lembrar como a nossa view está:

```
1 from django.http import HttpResponse
2
3 def index(request):
4     return HttpResponse("Olá, mundo!")
```

Até então, utilizamos o `HttpResponse` para retornar uma mensagem, mas a partir de agora utilizaremos a função `render` para exibir um template HTML no lugar dessa mensagem.

A função `render` é uma função de atalho do Django que nos possibilita combinar um template HTML e um dicionário de contexto. A função deve receber sempre a variável `request` e uma `string` representando o caminho do template a ser utilizado. Esses argumentos são obrigatórios e devem ser passados para a função `render` sempre que a mesma for utilizada.

Vamos alterar a nova view para que retorne a função `render` ao invés da classe `HttpResponse` passando a variável `request` e o caminho para o template `index.html` :

```
1 def index(request):  
2     return render(request, "index.html")
```

Como já fizemos o download das pastas **static** e **templates** e toda a configuração necessária para funcionamento de ambas, o Django já reconhece a pasta e busca pelo template `index.html` dentro dela.

Entendendo as adaptações necessárias no template

Como você deve ter percebido, o template não está sendo exibido como deveria. Isso porque os arquivos estáticos não foram carregados. Lembra que fizemos a configuração da variável `STATIC_URL` ? Pois bem, precisamos falar dela aqui pois para que os arquivos sejam carregados corretamente, o caminho relativo até eles deve estar correto e é aqui que a `STATIC_URL` entra em cena.

Conhecendo a tag static

Quando falamos anteriormente sobre a engine de templates do Django, falamos que ela é capaz de executar funções condicionais, loops, exibir valores e possui diversas outras funcionalidades que podem ser executadas diretamente nos templates através de tags. Vamos agora conhecer a primeira tag que vamos utilizar em nosso projeto, a tag **static**.

A tag static é a representação da variável `STATIC_URL` nos templates. O Django fornece essa tag no intuito de facilitar o trabalho com arquivos estáticos. Vamos aprender como utilizar a tag e resolver o problema de exibição do template.

O primeiro passo para utilizarmos a tag static é carregá-la no template. Para isso vamos inserir o seguinte trecho de código no topo do nosso HTML:

```
<!-- ... -->
```

```
1 <!DOCTYPE html>
2
3 {% load static %}
4
5 <html lang="pt-br">
```



Tags no Django são escritas dessa maneira: `{% %}` . Guarde isso, pois utilizaremos bastante no decorrer do curso.

Com isso já podemos utilizar a tag no template `index.html` .

Alterando o caminho dos arquivos estáticos

Como falamos, a tag `{% static %}` é a representação da pasta **static**. A tag nos dá um link para essa pasta para utilização no carregamento dos arquivos estáticos nos templates. Como é o Django que cuida de toda essa parte por nós, também vamos delegar a ele o carregamento dos nossos arquivos JS, CSS e imagens.

Alterando as importações dos arquivos CSS

Vamos alterar primeiro as importações dos arquivos CSS. As linhas que fazem o carregamento dos arquivos CSS no template são:

```
1 <link href="css/sb-admin-2.min.css" rel="stylesheet">
2 <link href="vendor/fontawesome-free/css/all.min.css" rel="stylesheet" type="text/css">
```

Vamos alterar os textos referentes a `href` para utilizarmos a tag `{% static %}` . As importações ficarão assim:

```
1 <link href="{% static 'css/sb-admin-2.min.css' %}" rel="stylesheet">
2 <link href="{% static 'vendor/fontawesome-free/css/all.min.css' %}" rel="stylesheet" type="text/css">
```

Alterando as importações dos arquivos JS

Para alterarmos as importações dos arquivos JS, vamos encontrar as linhas:

```
1 <script src="vendor/jquery/jquery.min.js"></script>
2 <script src="vendor/bootstrap/js/bootstrap.bundle.min.js"></script>
3 <script src="js/sb-admin-2.min.js"></script>
```

E alterá-las para que fiquem da seguinte forma:

```
1 <script src="{% static 'vendor/jquery/jquery.min.js' %}"></script>
2 <script src="{% static 'vendor/bootstrap/js/bootstrap.bundle.min.js' %}"></script>
3 <script src="{% static 'js/sb-admin-2.min.js' %}"></script>
```

Com isso, ao acessarmos <http://127.0.0.1:8000/> novamente no navegador, teremos o template exibido de forma estruturada com os arquivos CSS (exibição) e JS (comportamento) devidamente carregados. Nosso template será exibido desta forma no navegador:

CONTROLE DE VISITANTES

🏠 Início

Página Inicial

- VISITANTES AGUARDANDO AUTORIZAÇÃO: 0
- VISITANTES NO CONDOMÍNIO: 0
- VISITAS FINALIZADAS: 0
- VISITANTES REGISTRADOS NO MÊS ATUAL: 0

Visitantes recentes

Nome completo	CPF	Horário de chegada	Horário da autorização	Autorizado por	Mais informações
Don Corleone	123.123.123.12	22 de agosto de 2079 as 15:30	22 de agosto de 2079 as 15:38	Darth Vader	Ver informações

Copyright © Django framework na prática

Tela inicial da dashboard contento uma tabela que lista os visitantes recentes, nela é possível observar o visitante "Don Corleone"

Exibindo variáveis no template

Quando conhecemos a função `render` falamos que ela é responsável por combinar um template HTML e um dicionário de contexto, mas não fomos a fundo a respeito do que é um dicionário de contexto.

Um dicionário de contexto é a variável do tipo dicionário que pode ser passada como argumento para a função `render`. Quando passada, é possível acessarmos os valores contidos na variável diretamente no template através de tags específicas, diferentes das tags utilizadas anteriormente.

✔ Dicionário é uma estrutura de dados em Python de elementos (ou propriedades) não ordenados e que podemos acessar utilizando chaves. Os dicionários são estruturas poderosas e muito utilizadas. Existem linguagens que este tipo é conhecido como "matrizes associativas" ou apenas "objetos"

Definindo nosso dicionário de contexto

Para fazer isso, vamos no arquivo `views.py` e vamos criar a variável `context` acima do retorno da função. A função `index` ficará da seguinte forma:

```
1 def index(request):
2
3     context = {
4         "nome_pagina": "Início da dashboard",
5     }
6
7     return render(request, "index.html", context)
```

No código acima criamos a variável `context` já com o valor de `nome_pagina` definido como "Início da dashboard". Se a gente quisesse utilizar uma variável ao invés de um texto diretamente, poderíamos fazer desta forma:

```
1 def index(request):
2     nome_pagina = "Início da dashboard"
3
4     context = {
5         "nome_pagina": nome_pagina,
6     }
7
8     return render(request, "index.html", context)
```

Exibindo as informações nos templates

A partir de agora, vamos aprender um pouco mais sobre a linguagem de templates do Django. Ela foi projetada para ser poderosa e fácil de forma que seja confortável trabalhar com a linguagem HTML.

Essencialmente, templates são arquivos de texto, geralmente no formato HTML. Para o Django, um template pode conter variáveis que devem ser substituídas por valores quando o template for interpretado.

Agora que já definimos o nosso dicionário de contexto e passamos ele como argumento para a função `render`, vamos exibir essas informações no template `index.html`.

Para exibirmos os valores contidos no dicionário `context` basta utilizarmos a sintaxe para variáveis da linguagem de templates do Django: `{{ propriedade_do_dicionario }}`. Para nosso caso, vamos exibir o valor da propriedade `nome_pagina` que, dentro do dicionário `context`, corresponde ao texto **Início da dashboard**.

Vamos abrir o arquivo `index.html` e procurar pela seguinte linha:

```
<h1 class="h3 mb-0 text-gray-800">Página inicial</h1>
```

Vamos alterar o texto da tag `h1` (o texto **Página inicial**) para exibir também o valor da nossa variável `nome_pagina` passada na variável `context` da view. A linha deverá ficar assim:

```
<h1 class="h3 mb-0 text-gray-800">{{ nome_pagina }}</h1>
```

Volte para o navegador, atualize a página e veja a mágica acontecer: o valor

`{{ nome_pagina }}` será substituído pelo texto "Início da dashboard" que definimos no dicionário `context`. Se alterarmos o valor no arquivo `views.py` o mesmo acontece no `index.html`.

Capítulo 06

Criando o aplicativo para gerenciar visitantes

No último capítulo configuramos o Django para trabalhar com templates HTML e arquivos estáticos e criamos as pastas **templates** e **static** na raiz do projeto. Além disso, ainda aprendemos como passar informações de uma função *view* para o template através da variável **context**.

Como já definimos os usuários do sistema, os porteiros e fizemos as configurações dos templates que serão a base para construção da dashboard para controle de visitantes, podemos partir agora para a definição da classe modelo que irá representar os visitantes em nosso sistema.

Antes de mais nada, como já sabemos, devemos isolar as responsabilidades e, por isso, vamos criar um aplicativo para administrar toda a parte referente aos nossos visitantes. Vamos criar um novo aplicativo com nome de **visitantes** utilizando o `manage.py` :

```
(env)$ python manage.py startapp visitantes
```

Após criarmos o aplicativo utilizando o `manage.py` , vamos registrá-lo no arquivo de configurações, o `settings.py` , logo abaixo do aplicativo **porteiros**:

```
1  INSTALLED_APPS += [  
2      "usuarios",  
3      "porteiros",  
4      "visitantes",  
5  ]
```

Feito isso, vamos começar os trabalhos no arquivos `models.py` para definirmos as informações necessárias para o modelo de visitantes.

Escrevendo as models do nosso aplicativo de visitantes

Conforme falamos, a camada *model* (ou camada de modelo) é nossa fonte segura de dados e onde definimos o formato das informações que serão disponibilizadas para outras camadas da aplicação.

Precisamos guardar uma série de informações a respeito de quem deseja adentrar ao condomínio para realizar visitas a moradores, além da autorização de um morador que esteja na casa no momento da visita. O procedimento faz parte de normas do condomínio para fins de fiscalização, controle e segurança dos moradores. Segundo normas do condomínio, devemos guardar as seguintes informações referentes à visita:

1. Nome completo do visitante
2. CPF do visitante
3. Data de nascimento do visitante
4. Número da casa a ser visitada
5. Placa do veículo utilizado na visita, se houver
6. Horário de chegada na portaria
7. Horário de saída do condomínio
8. Horário de autorização de entrada
9. Nome do morador responsável por autorizar a entrada do visitante
10. Porteiro responsável por registrar visitante

Inicialmente, vamos nos concentrar nas informações de 1 a 5 para que possamos avaliar e escrever por partes o modelo de visitantes. Vamos escrever primeiro os atributos nome completo, CPF, data de nascimento, número da casa e placa do veículo, pois a gente já conhece a maioria desses tipos de dados. Nossa classe `Visitante` ficará assim:

```
1 from django.db import models
2
3 class Visitante(models.Model):
4     nome_completo = models.CharField(
5         verbose_name="Nome completo", max_length=194
6     )
7
8     cpf = models.CharField(
9         verbose_name="CPF",
10        max_length=11,
11    )
```

```

12
13     data_nascimento = models.DateField(
14         verbose_name="Data de nascimento",
15         auto_now=False,
16         auto_now_add=False,
17     )
18
19     numero_casa = models.PositiveSmallIntegerField(
20         verbose_name="Número da casa a ser visitada"
21     )
22
23     placa_veiculo = models.CharField(
24         verbose_name="Placa do veículo",
25         max_length=7,
26         blank=True,
27         null=True,
28     )

```

Conhecendo o campo DateTimeField

Antes de prosseguirmos, vamos conhecer o campo `DateTimeField`, um cara bem parecido com o `DateField` que já conhecemos com a diferença que, além da data, salva também o horário exato do registro. Assim como o `DateField`, o `DateTimeField` aceita `auto_now` e `auto_now_add` como argumentos, além das opções `blank` e `null`. Vamos utilizar o `DateTimeField` para definirmos os atributos que vão representar o horário de chegada e o horário de saída do visitante.

Primeiro vamos definir o atributo `horario_chegada`, que é quem representa o horário de chegada do visitante à portaria do condomínio. Como o atributo representa o horário de chegada do visitante à portaria, faz sentido que o mesmo seja preenchido no exato momento que registrarmos o visitante em nosso sistema. Para isso, utilizaremos a opção `auto_now_add` com o valor `True`, assim garantimos que o atributo receberá o valor da hora atual assim que o registro for adicionado ao banco de dados.

```

1  from django.db import models
2
3  class Visitante(models.Model):
4      # código acima omitido...
5
6      placa_veiculo = models.CharField(
7          verbose_name="Placa do veículo",

```

```

8         max_length=7,
9         blank=True,
10        null=True,
11    )
12
13    horario_chegada = models.DateTimeField(
14        verbose_name="Horário de chegada na portaria",
15        auto_now_add=True
16    )

```

Para o caso do horário de saída, utilizaremos uma configuração diferente para o atributo. Como precisamos setar o valor somente após a saída do visitante do condomínio, esse valor precisa ser registrado inicialmente como um valor em branco. Para isso, o Django nos dá a possibilidade de utilização do atributo `auto_now` como `False` e dizer que podemos aceitar valores em branco e nulos. Utilizando o `auto_now` como `False`, o campo receberá um valor em branco no momento da criação do registro no banco de dados.

```

1  from django.db import models
2
3  class Visitante(models.Model):
4      # código acima omitido...
5
6      placa_veiculo = models.CharField(
7          verbose_name="Placa do veículo",
8          max_length=7,
9          blank=True,
10         null=True,
11     )
12
13     horario_chegada = models.DateTimeField(
14         verbose_name="Horário de chegada na portaria",
15         auto_now_add=True
16     )
17
18     horario_saida = models.DateTimeField(
19         verbose_name="Horário de saída do condomínio",
20         auto_now=False,
21         blank=True,
22         null=True,
23     )

```

Conforme visto, além destas informações, precisamos guardar também o nome do morador que autorizou a entrada do visitante e o horário da autorização. Sendo assim, teremos mais dois atributos:

- Horário de autorização de entrada
- Nome do morador responsável por autorizar a entrada do visitante

Utilizaremos os campos já conhecidos `DateTimeField` e `CharField` para definir os atributos:

```
1  from django.db import models
2
3  class Visitante(models.Model):
4      # código acima omitido...
5
6      horario_saida = models.DateTimeField(
7          verbose_name="Horário de saída do condomínio",
8          auto_now=False,
9          blank=True,
10         null=True,
11     )
12
13     horario_autorizacao = models.DateTimeField(
14         verbose_name="Horário de autorização de entrada",
15         auto_now=False,
16         blank=True,
17         null=True,
18     )
19
20     morador_responsavel = models.CharField(
21         verbose_name="Nome do morador responsável por autorizar a entrada",
22         max_length=194,
23         blank=True,
24     )
```


Nada de novo por enquanto: utilizamos um campo do tipo `CharField` para armazenar o nome do morador responsável por autorizar a entrada e utilizamos um `DateTimeField` para armazenar o horário em que a autorização ocorreu. Conforme vimos, se não queremos que o campo `DateTimeField` seja preenchido na hora da criação ou atualização do registro, setamos o argumento `auto_now` como `False`. Isso garante também que o campo possa ser preenchido com um texto vazio.

Conhecendo o campo ForeignKey

Seguindo a lista de requisitos, o próximo atributo que devemos adicionar ao nosso modelo é a informação referente ao **porteiro responsável por registrar a entrada do visitante**.

Como criamos um modelo que representa nossos porteiros dentro do sistema, podemos associar esse modelo ao modelo de **visitante** por meio do campo `ForeignKey`. Esse campo cria um atributo que vincula um registro de modelo a outro registro de modelo. Neste caso, queremos vincular o modelo `Porteiro` ao modelo `Visitante` para representar o porteiro responsável pelo registro.

O campo `ForeignKey` é importante pois assim não precisamos replicar as informações do porteiro no registro de visitante, apenas referenciamos essas informações inserindo o `id` referente ao registro do **porteiro**. Como os modelos são representações das tabelas do nosso banco de dados, estamos dizendo algo como: *"hey, Django, procure essas informações na tabela de porteiros usando esse id!"* e o Django faz todo o trabalho de trazer essas informações por nós.

 É importante lembrar que tudo isso é possível devido à ORM do Django, que é quem abstrai todas as funcionalidades do banco de dados e manipula todas as interações necessárias

Abaixo do atributo `morador_responsavel`, vamos escrever o atributo `registrado_por` sendo do tipo `ForeignKey`, que representará a informação do **porteiro** responsável por registrar a entrada do visitante:

```
1  from django.db import models
2
3  class Visitante(models.Model):
4      # código acima omitido...
5
6      morador_responsavel = models.CharField(
7          verbose_name="Nome do morador responsável por autorizar a entrada
8          max_length=194,
9          blank=True,
10     )
```

```

11
12     registrado_por = models.ForeignKey(
13         "porteiros.Porteiro",
14         verbose_name="Porteiro responsável pelo registro",
15         on_delete=models.PROTECT
16     )

```

Os argumentos que o campo `ForeignKey` recebe são bem parecidos com os do `OneToOneField` que já conhecemos. Primeiro informamos a classe modelo que deverá ser relacionada: a classe `Porteiro` do aplicativo **porteiros**. Depois utilizamos o `verbose_name` para dar um nome descritivo para o campo e informamos o que deve ser feito com o registro do visitante caso o registro do **porteiro** da relação seja excluído. Nesse caso, utilizaremos para o `on_delete` a ação `models.PROTECT`, pois assim protegemos também o modelo de visitantes. Sempre que houver tentativa de exclusão de um registro de porteiro que esteja vinculado a um visitante, o Django mostrará um erro. O atributo está protegido contra exclusão.

Nosso próximo passo agora é apenas escrever a classe `Meta` e o método `__str__` da classe `Visitante`. Nosso modelo ficará assim:

```

1  from django.db import models
2
3  class Visitante(models.Model):
4      nome_completo = models.CharField(
5          verbose_name="Nome completo", max_length=194
6      )
7
8      cpf = models.CharField(
9          verbose_name="CPF",
10         max_length=11,
11     )
12
13     data_nascimento = models.DateField(
14         verbose_name="Data de nascimento",
15         auto_now=False
16     )
17
18     numero_casa = models.PositiveSmallIntegerField(
19         verbose_name="Número da casa a ser visitada",
20     )
21
22     placa_veiculo = models.CharField(

```

```

23         verbose_name="Placa do veículo",
24         max_length=7,
25         blank=True,
26         null=True,
27     )
28
29     horario_chegada = models.DateTimeField(
30         verbose_name="Horário de chegada na portaria",
31         auto_now_add=True
32     )
33
34     horario_saida = models.DateTimeField(
35         verbose_name="Horário de saída do condomínio",
36         auto_now=False,
37         blank=True,
38         null=True,
39     )
40
41     horario_autorizacao = models.DateTimeField(
42         verbose_name="Horário de autorização de entrada",
43         auto_now=False,
44         blank=True,
45         null=True,
46     )
47
48     morador_responsavel = models.CharField(
49         verbose_name="Nome do morador responsável por autorizar a entrada",
50         max_length=194,
51         blank=True,
52     )
53
54     registrado_por = models.ForeignKey(
55         "porteiros.Porteiro",
56         verbose_name="Porteiro responsável pelo registro",
57         on_delete=models.PROTECT
58     )
59
60     class Meta:
61         verbose_name = "Visitante"
62         verbose_name_plural = "Visitantes"
63         db_table = "visitante"
64
65     def __str__(self):
66         return self.nome_completo

```

Registrando nossa aplicação no Admin do Django

O próximo passo a ser executado, como já vimos, é tornar o nosso modelo visível para o Admin do Django. Então vamos lá!

Vamos abrir o arquivo `admin.py` do nosso aplicativo visitantes, importar a classe `Visitante` e passá-la como argumento da função `admin.site.register()`.

```
1 from django.contrib import admin
2 from visitantes.models import Visitante
3
4 admin.site.register(Visitante)
```

Aplicando as alterações em nosso banco de dados

Feito isso, mais uma vez vamos criar as migrações do modelo criado utilizando o comando `makemigrations`.

```
(env)$ python manage.py makemigrations visitantes
```

Se tudo ocorrer bem, vamos receber as seguintes informações em nosso terminal:

```
1 Migrations for 'visitantes':
2   porteiros/migrations/0001_initial.py
3   - Create model Visitante
```

Com todas as informações necessárias para executar as alterações no banco de dados armazenadas em forma de migração, vamos aplicar as alterações em nosso banco de dados com o comando `migrate`.

```
(env)$ python manage.py migrate
```

Devemos receber as seguintes informações em nosso terminal:

```
1 Operations to perform:
2   Apply all migrations: admin, auth, contenttypes, porteiros, sessions,
3   usuarios, visitantes
4
5 Running migrations:
6   Applying visitantes.0001_initial.py... OK
```

Adicionando visitante utilizando o Django Admin

Da mesma forma que cadastramos um porteiro utilizando o Admin do Django, vamos adicionar também um visitante. Vamos novamente acessar <http://127.0.0.1:8000/admin> e agora veremos o aplicativo **visitantes** disponível para nós.

AUTENTICAÇÃO E AUTORIZAÇÃO	
Grupos	+ Adicionar ✎ Modificar
PORTEIROS	
Porteiros	+ Adicionar ✎ Modificar
USUARIOS	
Usuários	+ Adicionar ✎ Modificar
VISITANTES	
Visitantes	+ Adicionar ✎ Modificar

Captura tirada do Admin do Django focando nos links para a exibição de Grupos, Usuários, Porteiros e agora Visitantes também

Dessa vez, vamos clicar diretamente no botão "adicionar" para que a gente vá direto para o formulário de cadastro de visitantes. O formulário a ser exibido deverá se parecer com isto:

Adicionar Visitante

Nome completo:	<input type="text"/>
CPF:	<input type="text"/>
Data de nascimento:	<input type="text"/> Hoje
Número da casa a ser visitada:	<input type="text"/>
Horário de autorização de entrada:	Data: <input type="text"/> Hoje Hora: <input type="text"/> Agora
Horário de saída do condomínio:	Data: <input type="text"/> Hoje Hora: <input type="text"/> Agora
Porteiro responsável pelo registro:	<input type="text"/>
Placa do veículo:	<input type="text"/>
Nome do morador responsável por autorizar a entrada do visitante:	<input type="text"/>

[Salvar e adicionar outro\(a\)](#) [Salvar e continuar editando](#) [SALVAR](#)

Formulário para registro de visitante. O formulário apresenta os campos da classe modelo

Por enquanto vamos preencher os campos obrigatórios **nome completo**, **CPF**, **data de nascimento** e **número da casa** a ser visitada e definir o **porteiro** responsável por registrar o visitante. Fique livre para preencher as informações à sua maneira. Após preencher os campos citados, clique em salvar e visualize a lista de visitantes agora com o novo visitante registrado.

Essa foi a primeira e última vez utilizamos o Django Admin para registrar um visitante, pois a partir de agora trabalharemos diretamente nos templates HTML da dashboard que vamos disponibilizar para os porteiros do condomínio.

Listando visitantes na página inicial da dashboard

Como já definimos nosso modelo e até registramos visitantes através do Admin, agora vamos aprender como buscar esses registros em nosso banco de dados.

Quando precisamos buscar registros em nosso banco de dados, devemos construir uma **Queryset** utilizando o **Manager** da classe modelo em questão. A classe Manager, conforme visto, define como as interações com o banco de dados devem acontecer e, por padrão, é

um atributo da classe chamado `objects` . Já uma queryset nada mais é que uma lista de objetos de um determinado tipo existentes em nosso banco de dados.

Sempre que definimos uma subclasse de `django.db.models.Model` , que é o que todos os nossos modelos são, o Django nos fornece de forma automática uma interface para realizar operações em nosso banco de dados, tais como buscar, atualizar, criar e deletar registros, mas por hora, vamos nos concentrar apenas em buscar os registros de visitantes.

Buscando registros de visitantes no banco de dados

Quando falamos da camada **view**, vimos que é ela quem deve encapsular toda a lógica necessária para apresentar os dados. Geralmente, as **views** devem buscar as informações no banco de dados, carregar o template e renderizar esse template com as informações buscadas. Uma view no Django tem a função de exatamente conectar a camada de modelo à camada de template.

O primeiro passo para buscarmos os registros é criar uma variável para armazenar os registros que serão retornados, para isto utilizaremos a variável `todos_visitantes` . A variável receberá uma **queryset** que será retornada pelo método `all()` do **Manager** `objects` do modelo **Visitante**.

Antes de tudo, vamos importar o modelo no arquivo `views.py` do aplicativo **usuarios**.

```
1 from django.shortcuts import render
2
3 from visitantes.models import Visitante
4
5 def index(request):
6     context = {
7         "nome_pagina": "Início da dashboard",
8     }
9
10    return render(request, "index.html", context)
```

Feito isso, vamos criar a variável `todos_visitantes` acima da variável `context` e definir seu valor como `Visitante.objects.all()` . Desta forma, estamos buscando todos os registros de visitantes existentes em nosso banco de dados.

Não podemos nos esquecer de colocar a variável `todos_visitantes` dentro do nosso dicionário `context` para que possamos acessá-la através do template. A função `index` ficará assim:

```
1 from visitantes.models import Visitante
2
3 def index(request):
4
5     todos_visitantes = Visitante.objects.all()
6
7     context = {
8         "nome_pagina": "Início da dashboard",
9         "todos_visitantes": todos_visitantes,
10    }
11
12    return render(request, "index.html", context)
```

Listando registros de visitantes no template HTML

Conhecendo a tag `for` e acessando atributos do visitante

Para exibir variáveis nos templates vimos que podemos utilizar a sintaxe de chaves (`{{ }}`), mas se tentarmos exibir uma queryset desta maneira, não será possível, pois uma queryset é uma lista que contém vários itens. É necessário percorrer os itens dessa lista e acessar item por item. Pra nossa sorte, o Django nos fornecer uma tag para que possamos executar loops em listas.

A tag `{% for %}` é quem vai nos ajudar agora. O que ela faz é justamente andar por todos os itens da lista e disponibilizar uma variável para que possamos acessar as informações da mesma. Parece um pouco confuso? Não se assuste, vamos entender melhor visualizando o código.

No nosso caso, buscamos todos os registros de visitantes que temos em nosso banco de dados e passamos essa lista como variável dentro do contexto da view. O que precisamos fazer agora é passar em cada item existente na lista de visitantes e exibir as informações como nome completo, CPF e data de nascimento.



Para acessarmos as informações dos visitantes nos templates, utilizaremos o nome dos atributos definidos da classe modelo.

Vamos abrir o arquivo de template `index.html` e buscar pelo elemento HTML `<tbody>` . É dentro dele, acima dos elementos `<tr>` que vamos definir o início da tag `{% for %}` . Para utilizar essa tag, precisamos também evidenciar onde o loop deve ser parado e fazemos isso utilizando a tag `{% endfor %}` .

Logo abaixo do elemento `<tbody>` insira o trecho

`{% for visitante in todos_visitantes %}` . Lembra que falamos da variável que a tag `{% for %}` disponibiliza para que possamos acessar as informações? Pois bem, podemos dar nome a ela e, neste caso, utilizaremos o nome `visitante` . O trecho de código ficará assim:

```
1 <tbody>
2     {% for visitante in todos_visitantes %}
3         <tr>
4             <td>Don Corleone</td>
5             <td>123.123.123.06</td>
6             <td>22 de agosto 15:30</td>
7             <td>22 de agosto 15:38</td>
8             <td>Darth Vader</td>
9             <td>
10                 <a href="#">
11                     Ver detalhes
12                 </a>
13             </td>
14         </tr>
15     {% endfor %}
16 </tbody>
```

Olha só que bacana: estamos dizendo para o Django "hey, cara, para cada `visitante` que existir na lista `todos_visitantes` , repita essa estrutura de elementos `<td>` . Com isso já estamos executando o loop na lista `todos_visitantes` , mas ainda não estamos exibindo os valores referentes a cada visitante existente no banco de dados. Para fazer isso, vamos

utilizar a sintaxe de chaves (`{{ }}`) em conjunto com variável `visitante` que criamos dentro da tag `{% for %}` .

Os atributos do visitante podem ser acessados utilizando a sintaxe

`visitante.nome_do_atributo` . Se queremos exibir o nome completo do visitante, vamos utilizar `visitante.nome_completo` . Faremos o mesmo com os atributos **CPF, horário de chegada, horário de autorização de entrada e morador responsável**. Realizando as alterações para exibirmos os atributos, o código ficará assim:

```
1 <tbody>
2   {% for visitante in todos_visitantes %}
3     <tr>
4       <td>{{ visitante.nome_completo }}</td>
5       <td>{{ visitante.cpf }}</td>
6       <td>{{ visitante.horario_chegada }}</td>
7       <td>{{ visitante.horario_autorizacao }}</td>
8       <td>{{ visitante.morador_responsavel }}</td>
9       <td>
10        <a href="#">
11          Ver detalhes
12        </a>
13      </td>
14    </tr>
15  {% endfor %}
16 </tbody>
```

Agora quando atualizarmos a página, vamos visualizar as informações dos visitantes registrados através do Admin. Caso queira testar, fique à vontade para registrar outros visitantes. Quando você acessar novamente <http://127.0.0.1:8000/> e atualizar a página, os novos visitantes serão adicionados à tabela de forma automática! Bem bacana, não?

Capítulo 07

Criando tela para registro de novo visitante

Assim como quando registramos um novo visitante através do Admin, precisaremos de um formulário para inserir as informações. Por isso, vamos trabalhar agora na tela que será responsável por exibir um formulário e registrar o visitante em nosso banco de dados.

Vimos que uma view é um tipo de função dentro da aplicação que conecta a camada de modelo à camada de template e, geralmente, renderiza um template específico com informações buscadas no banco de dados. Até agora escrevemos apenas views que buscam informações e renderizam o template utilizando essas informações, mas agora vamos trabalhar em views que também salvam informações no banco de dados.

A próxima view que vamos escrever, chamada de `registrar_visitante`, terá a responsabilidade de exibir um formulário, identificar e tratar uma requisição do tipo POST, validar o formulário com base das informações enviadas na requisição e salvar o novo visitante no banco de dados. Não se assuste, você vai ver como o Django nos ajuda abstraindo a maior parte desses requisitos.

Criando view para registrar visitante

Como você já deve ter percebido, existe um roteiro a ser seguido quando vamos criar novas funcionalidades num sistema web com Django, sendo o primeiro passo a criação da função de view no arquivo `views.py`. Como nossa função diz respeito ao registro de um novo visitante, vamos trabalhar dentro do aplicativo visitantes.

Vamos abrir o arquivo `views.py` do aplicativo visitantes e escrever a função de view `registrar_visitante`, que deverá renderizar o template `registrar_visitante.html`. Por hora, nossa view ficará assim:

```
1 from django.shortcuts import render
2
```



```
3 def registrar_visitante(request):
4
5     context = {}
6
7     return render(request, "registrar_visitante.html", context)
```

Assim como fizemos anteriormente, vamos baixar o arquivo HTML e agora colocá-lo na pasta **templates** localizada na raiz do nosso projeto:



Iniciar o download

registrar_visitante.zip - 2KB

Criando URL para mapear view

Quando criamos nossa primeira view, criamos também uma URL que é responsável por mapear a view para acessarmos ela através do navegador. Caso não se lembre do processo, não se preocupe, pois vamos repeti-lo agora.

Vamos abrir o arquivo `urls.py` do nosso projeto e, abaixo da URL de nome **index**, utilizando a função `path`, vamos criar a URL de nome **registrar_visitante** que deverá mapear a função de view `registrar_visitante`. Não podemos nos esquecer de importar as views do aplicativo visitantes!

O arquivo `urls.py` ficará assim:

```
1 from django.urls import path
2 from django.contrib import admin
3
4 import usuarios.views
5 import visitantes.views
6
7 urlpatterns = [
8     # codigo acima omitido...
9
10    path(
11        "",
12        usuarios.views.index,
13        name="index",
14    ),
```

```
15
16     path(
17         "registrar-visitante/",
18         visitantes.views.registrar_visitante,
19         name="registrar_visitante",
20     )
21 ]
```

Abra seu navegador e acesse <http://127.0.0.1:8000/registrar-visitante/> para verificar se está tudo funcionando corretamente. Se sim, o template baixado será exibido no navegador.

Adaptando nossos templates para trabalhar com a template engine do Django

Temos agora duas views que renderizam dois templates diferentes e expõem funcionalidades diferentes: uma delas, que é a página inicial da dashboard, exibe os visitantes registrados, e a segunda deverá possibilitar o registro de novos visitantes. Antes de seguir adiante, vamos realizar algumas alterações em nossos templates para que possamos aproveitar melhor as funcionalidades do framework que estamos utilizando.

Além das funcionalidades que falamos e exploramos, a engine de templates do Django também nos dá a possibilidade de reaproveitarmos trechos de código contidos em outros templates. No nosso caso, se você observar os templates `index.html` e `registrar_visitante.html`, vai notar que existem partes iguais nos dois templates e, para evitar isso, a engine de templates do Django nos fornece as tags `{% extends %}` e `{% block %}`.

Criando o template base

Antes de tudo, vamos criar um arquivo com nome de `base.html` na pasta **templates** e copiar o conteúdo do arquivo `index.html` para ele. O objetivo do nosso template `base.html` é armazenar a parte comum a todos os templates da dashboard. O que podemos chamar de parte central do nosso template, que é a parte que em um template

exibe uma tabela e no outro um formulário, será trocada de acordo com a view acessada e as barras lateral e superior e o rodapé serão mantidos no template `base.html`. Por hora, vamos apenas copiar o conteúdo do arquivo `index.html` para o arquivo `base.html` e deixá-lo de lado.

Adaptando template index

Com o template `base.html` criado, vamos fazer algumas adaptações em nosso template `index.html` para garantir que ele seja exibido corretamente fazendo uso da engine de templates do Django. Apague todo o conteúdo existente no arquivo `index.html` deixando apenas o conteúdo dentro do elemento HTML `<div class="container-fluid">`. O arquivo `index.html` ficará assim:

```
1 <div class="container-fluid">
2   <div class="d-sm-flex align-items-center justify-content-between mb-4">
3     <h1 class="h3 mb-0 text-gray-800">{{ nome_pagina }}</h1>
4   </div>
5
6   <div class="row">
7     <div class="col-xl-3 col-md-6 mb-4">
8       <div class="card border-left-warning shadow h-100 py-2">
9         <div class="card-body">
10          <div class="row no-gutters align-items-center">
11            <div class="col mr-2">
12              <div class="text-xs font-weight-bold text-warn">
13                <div class="h5 mb-0 font-weight-bold text-gray">
14              </div>
15            </div>
16            <div class="col-auto">
17              <i class="fas fa-user-lock fa-2x text-gray-300">
18            </div>
19          </div>
20        </div>
21      </div>
22    </div>
23
24    <div class="col-xl-3 col-md-6 mb-4">
25      <div class="card border-left-primary shadow h-100 py-2">
26        <div class="card-body">
27          <div class="row no-gutters align-items-center">
28            <div class="col mr-2">
29              <div class="text-xs font-weight-bold text-prim">
30              <div class="h5 mb-0 font-weight-bold text-gray">
```

```

31         </div>
32
33         <div class="col-auto">
34             <i class="fas fa-user-clock fa-2x text-gray-300"></i>
35         </div>
36     </div>
37 </div>
38 </div>
39 </div>
40
41 <div class="col-xl-3 col-md-6 mb-4">
42     <div class="card border-left-success shadow h-100 py-2">
43         <div class="card-body">
44             <div class="row no-gutters align-items-center">
45                 <div class="col mr-2">
46                     <div class="text-xs font-weight-bold text-success"></div>
47                     <div class="h5 mb-0 font-weight-bold text-gray-700"></div>
48                 </div>
49                 <div class="col-auto">
50                     <i class="fas fa-user-check fa-2x text-gray-300"></i>
51                 </div>
52             </div>
53         </div>
54     </div>
55 </div>
56
57 <div class="col-xl-3 col-md-6 mb-4">
58     <div class="card border-left-info shadow h-100 py-2">
59         <div class="card-body">
60             <div class="row no-gutters align-items-center">
61                 <div class="col mr-2">
62                     <div class="text-xs font-weight-bold text-info"></div>
63                     <div class="h5 mb-0 font-weight-bold text-gray-700"></div>
64                 </div>
65                 <div class="col-auto">
66                     <i class="fas fa-users fa-2x text-gray-300"></i>
67                 </div>
68             </div>
69         </div>
70     </div>
71 </div>
72 </div>
73
74 <div class="card shadow mb-4">
75     <div class="card-header py-3 d-sm-flex align-items-center justify-content-between">
76         <h6 class="m-0 font-weight-bold text-primary">Visitantes recientes</h6>
77     </div>
78
79     <div class="card-body">
80         <div class="table-responsive">
81             <table class="table table-bordered">

```

```

82         <thead>
83             <th>Nome</th>
84             <th>CPF</th>
85             <th>Horário de chegada</th>
86             <th>Horário da autorização</th>
87             <th>Autorizado por</th>
88             <th>Mais informações</th>
89         </thead>
90
91         <tbody>
92             {% for visitante in todos_visitantes %}
93                 <tr>
94                     <td>{{ visitante.nome_completo }}</td>
95                     <td>{{ visitante.cpf }}</td>
96                     <td>{{ visitante.horario_chegada }}</td>
97                     <td>{{ visitante.horario_autorizacao }}</td>
98                     <td>{{ visitante.morador_responsavel }}</td>
99                     <td>
100                         <a href="#">
101                             Ver informações
102                         </a>
103                     </td>
104                 </tr>
105             {% endfor %}
106         </tbody>
107     </table>
108 </div>
109 </div>
110 </div>
111 </div>

```

Com os templates devidamente separados, vamos trabalhar agora nas adaptações necessárias ao template `index.html`. O primeiro passo é inserirmos a tag `{% extends %}` no início do nosso arquivo, que é quem dirá ao Django que o template em questão é uma extensão de outro. A tag `{% extends %}` funciona de modo que precisamos identificar o template "pai" ou "mãe" que será utilizado na extensão. Isto é, neste caso, o template `index.html` será extensão do template `base.html`, sendo este o "seu template pai ou mãe". Na primeira linha do arquivo `index.html` insira o trecho `{% extends "base.html" %}`.

Além disto, precisamos também dizer ao Django qual trecho deverá ser utilizado para substituição. Faremos isso utilizando as tags `{% block %}` e `{% endblock %}` passando um nome a elas. Logo abaixo da tag `{% extends "base.html" %}` vamos inserir a tag

`{% block conteudo %}` e ao final do arquivo a tag `{% endblock conteudo %}` . Fazendo isso estamos deixando claro para o Django qual trecho deverá ser colocado no template `base.html` quando acessarmos a view que renderiza o template `index.html` . Nosso arquivo ficará assim após as adaptações:

```
1  {% extends "base.html" %}
2
3  {% block conteudo %}
4  <div class="container-fluid">
5      <div class="d-sm-flex align-items-center justify-content-between mb-4">
6          <h1 class="h3 mb-0 text-gray-800">{{ nome_pagina }}</h1>
7      </div>
8
9      <div class="row">
10         <div class="col-xl-3 col-md-6 mb-4">
11             <div class="card border-left-warning shadow h-100 py-2">
12                 <div class="card-body">
13                     <div class="row no-gutters align-items-center">
14                         <div class="col mr-2">
15                             <div class="text-xs font-weight-bold text-warn
16                             <div class="h5 mb-0 font-weight-bold text-gray
17                         </div>
18
19                         <div class="col-auto">
20                             <i class="fas fa-user-lock fa-2x text-gray-300
21                         </div>
22                     </div>
23                 </div>
24             </div>
25         </div>
26
27         <div class="col-xl-3 col-md-6 mb-4">
28             <div class="card border-left-primary shadow h-100 py-2">
29                 <div class="card-body">
30                     <div class="row no-gutters align-items-center">
31                         <div class="col mr-2">
32                             <div class="text-xs font-weight-bold text-prim
33                             <div class="h5 mb-0 font-weight-bold text-gray
34                         </div>
35
36                         <div class="col-auto">
37                             <i class="fas fa-user-clock fa-2x text-gray-30
38                         </div>
39                     </div>
40                 </div>
41             </div>
42         </div>
```

```

43
44     <div class="col-xl-3 col-md-6 mb-4">
45         <div class="card border-left-success shadow h-100 py-2">
46             <div class="card-body">
47                 <div class="row no-gutters align-items-center">
48                     <div class="col mr-2">
49                         <div class="text-xs font-weight-bold text-succ
50                         <div class="h5 mb-0 font-weight-bold text-gray
51                     </div>
52                     <div class="col-auto">
53                         <i class="fas fa-user-check fa-2x text-gray-30
54                     </div>
55                 </div>
56             </div>
57         </div>
58     </div>
59
60     <div class="col-xl-3 col-md-6 mb-4">
61         <div class="card border-left-info shadow h-100 py-2">
62             <div class="card-body">
63                 <div class="row no-gutters align-items-center">
64                     <div class="col mr-2">
65                         <div class="text-xs font-weight-bold text-info
66                         <div class="h5 mb-0 font-weight-bold text-gray
67                     </div>
68                     <div class="col-auto">
69                         <i class="fas fa-users fa-2x text-gray-300"></
70                     </div>
71                 </div>
72             </div>
73         </div>
74     </div>
75 </div>
76
77 <div class="card shadow mb-4">
78     <div class="card-header py-3 d-sm-flex align-items-center justify-
79         <h6 class="m-0 font-weight-bold text-primary">Visitantes recen
80     </div>
81
82     <div class="card-body">
83         <div class="table-responsive">
84             <table class="table table-bordered">
85                 <thead>
86                     <th>Nome</th>
87                     <th>CPF</th>
88                     <th>Horário de chegada</th>
89                     <th>Horário da autorização</th>
90                     <th>Autorizado por</th>
91                     <th>Mais informações</th>
92                 </thead>
93

```

```

94         <tbody>
95             {% for visitante in todos_visitantes %}
96                 <tr>
97                     <td>{{ visitante.nome_completo }}</td>
98                     <td>{{ visitante.cpf }}</td>
99                     <td>{{ visitante.horario_chegada }}</td>
100                    <td>{{ visitante.horario_autorizacao }}</td>
101                    <td>{{ visitante.morador_responsavel }}</td>
102                    <td>
103                        <a href="#">
104                            Ver informações
105                        </a>
106                    </td>
107                </tr>
108            {% endfor %}
109        </tbody>
110    </table>
111 </div>
112 </div>
113 </div>
114 </div>
115 {% endblock conteudo %}

```

Adaptando template base

Quando criamos o template `base.html`, copiamos o conteúdo de `index.html` para ele e o deixamos de lado, mas agora é hora de trabalharmos nele. O que temos que fazer é substituir o elemento HTML `<div class="container-fluid">` pelas tags `{% block conteudo %}` e `{% endblock conteudo %}`. O template `base.html` ficará assim:

```

1  <!DOCTYPE html>
2
3  {% load static %}
4
5  <html lang="pt-br">
6      <head>
7          <meta charset="utf-8">
8          <meta http-equiv="X-UA-Compatible" content="IE=edge">
9          <meta name="viewport" content="width=device-width, initial-scale=1
10
11         <title>Controle de Visitantes | Django Framework na prática</title>
12
13         <link href="https://fonts.googleapis.com/css?family=Nunito:200,200

```



```

14
15     <link href="{% static 'css/sb-admin-2.min.css' %}" rel="stylesheet"
16     <link href="{% static 'vendor/fontawesome-free/css/all.min.css' %}"
17 </head>
18
19 <body id="page-top">
20     <div id="wrapper">
21         <ul class="navbar-nav bg-gradient-primary sidebar sidebar-dark"
22             <a class="sidebar-brand d-flex align-items-center justify-
23                 <div class="sidebar-brand-icon rotate-n-15">
24                     <i class="fas fa-user-shield"></i>
25                 </div>
26
27                 <div class="sidebar-brand-text">Controle de Visitantes
28             </a>
29
30             <hr class="sidebar-divider my-0">
31
32             <li class="nav-item">
33                 <a class="nav-link" href="#">
34                     <i class="fas fa-home"></i>
35                     <span>Início</span>
36                 </a>
37             </li>
38
39             <hr class="sidebar-divider">
40         </ul>
41
42         <div id="content-wrapper" class="d-flex flex-column">
43             <div id="content">
44                 <nav class="navbar navbar-expand navbar-light bg-white"
45                     <button id="sidebarToggleTop" class="btn btn-link"
46                         <i class="fa fa-bars"></i>
47                     </button>
48
49                     <ul class="navbar-nav ml-auto">
50                         <li class="nav-item dropdown no-arrow">
51                             <a class="nav-link dropdown-toggle" href="#"
52                                 <span class="mr-2 d-none d-lg-inline t
53                                     <i class="fas fa-cog"></i>
54                                 </span>
55                             </a>
56
57                             <div class="dropdown-menu dropdown-menu-ri
58                                 <a class="dropdown-item" href="#" data
59                                     <i class="fas fa-sign-out-alt fa-s
60                                     Sair
61                                 </a>
62                             </div>
63                         </li>
64                     </ul>

```

```

65         </nav>
66
67         {% block conteudo %} {% endblock conteudo %}
68
69         <footer class="sticky-footer bg-white">
70             <div class="container my-auto">
71                 <div class="copyright text-center my-auto">
72                     <span>Copyright © Django framework na práti
73                 </div>
74             </div>
75         </footer>
76     </div>
77 </div>
78
79 <div class="modal fade" id="logoutModal" tabindex="-1" role="d
80     <div class="modal-dialog" role="document">
81         <div class="modal-content">
82             <div class="modal-header">
83                 <h5 class="modal-title" id="ModalLabel">Você r
84
85                 <button class="close" type="button" data-dismi
86                     <span aria-hidden="true">x</span>
87                 </button>
88             </div>
89
90             <div class="modal-body">Selecione "sair" se realme
91
92             <div class="modal-footer">
93                 <button class="btn btn-secondary" type="button
94                 <a class="btn btn-primary" href="#">Sair</a>
95             </div>
96         </div>
97     </div>
98 </div>
99 </div>
100
101 <script src="{% static 'vendor/jquery/jquery.min.js' %}"></script>
102 <script src="{% static 'vendor/bootstrap/js/bootstrap.bundle.min.j
103 <script src="{% static 'js/sb-admin-2.min.js' %}"></script>
104 </body>
105 </html>

```

Adaptando template registrar_visitante

Faremos o mesmo que fizemos com o template `index.html`, mas agora deixando apenas o conteúdo existente dentro do elemento HTML `<div class="container">` e inserindo as

tags `{% extends "base.html" %}` , `{% block conteudo %}` e `{% endblock conteudo %}`
. O template `registrar_visitante.html` ficará assim:

```
1  {% extends "base.html" %}
2
3  {% block conteudo %}
4  <div class="container">
5      <div class="d-sm-flex align-items-center justify-content-between mb-4">
6          <h1 class="h3 mb-0 text-gray-800">{{ nome_pagina }}</h1>
7      </div>
8
9      <div class="card shadow mb-4">
10         <div class="card-body">
11             <h4 class="mb-3 text-primary">
12                 Formulário para registro de novo visitante
13             </h4>
14
15             <form method="post">
16                 <div class="form-row">
17                     <p class="ml-2">Aqui deveria ter um formulário</p>
18                 </div>
19
20                 <div class="text-right">
21                     <button class="btn btn-primary" type="submit">
22                         <span class="text">Registrar visitante</span>
23                     </button>
24                 </div>
25             </form>
26         </div>
27     </div>
28 </div>
29 {% endblock %}
```

Caso prefira, você pode fazer download da pasta templates com as alterações realizadas até aqui clicando no link abaixo:



Iniciar o download

templates.zip - 3KB

Agora que adaptamos todos os nossos templates, vamos voltar ao desenvolvimento da view responsável por registrar nossos visitantes.

Capítulo 08

Trabalhando com formulários no Django

Manipular formulários não é uma tarefa tão fácil. Se observarmos o Admin do Django, podemos notar que existem diversos tipos de dados e maneiras diferentes de tratar e renderizar esses dados. Além disso, existe a estrutura HTML do formulário a ser renderizada no template, esse formulário deve validar as informações que são enviadas pelo usuário, salvar as informações ou exibir uma mensagem para o usuário caso os dados estejam inválidos, etc. Para simplificar nosso trabalho, o Django fornece ferramentas para automatizar e simplificar esse processo, garantindo também segurança para implementar as funcionalidades necessárias.

Um formulário pode ser definido como um conjunto de elementos dentro do elemento HTML `<form>` que permitem que o usuário insira textos, números, escolha opções e, ao fim, envie essas informações de volta para o servidor. No contexto da nossa aplicação web, um formulário pode significar também o formulário que a classe `Form` do Django nos disponibiliza, que é quem faz toda mágica por nós. Da mesma maneira que uma classe `Model` descreve toda estrutura lógica de um objeto, seu comportamento e a maneira como suas partes são representadas para nós, uma classe `Form` descreve um formulário e determina como ele funciona e se parece.

Criando formulário para registro de visitante

Assim como outras camadas importantes da arquitetura do nosso projeto, os formulários também devem ter um arquivo próprio para eles, mas que, neste caso, precisamos criar: o arquivo `forms.py`. Vamos abrir a pasta do nosso aplicativo **visitantes** e criar o arquivo `forms.py`. Após criarmos o arquivo, vamos abri-lo e importar o pacote forms do django. Ficará assim:

```
from django import forms
```

De forma semelhante aos campos de uma classe `Model` que são mapeados para os campos do banco de dados, os campos de uma classe `Form` são mapeados para elementos HTML. É exatamente assim que toda a mágica do Admin do Django funciona: mapeando os campos da sua classe `Model`, criando classes `Form` e renderizando esses campos como elementos HTML. Muito bacana, não?

Para quando queremos mapear automaticamente os campos de uma classe `Model`, o Django nos permite utilizar a classe `ModelForm`. Tudo que precisamos fazer é definir uma subclasse de `ModelForm` e depois identificarmos a classe `Model` a ser utilizada. Sendo assim, vamos também importar a classe **Visitante** e depois definir uma subclasse de `forms.ModelForm` de nome `VisitanteForm`. O arquivo `forms.py` ficará assim:

```
1 from django import forms
2 from visitantes.models import Visitante
3
4 class VisitanteForm(forms.ModelForm):
5     class Meta:
6         model = Visitante
7         fields = "__all__"
```

Criamos a subclasse de `forms.ModelForm` chamada `VisitanteForm`, que é quem representa nosso formulário e definimos a classe interna `Meta` para explicitarmos qual classe `Model` deve ser utilizada (`model = Visitante`) e quais campos devem ser renderizados (`fields = "__all__"`). Por hora, vamos utilizar a string `"__all__"` para renderizarmos todos os campos.

Renderizando nosso formulário automaticamente

Agora que definimos a classe que vai representar nosso formulário, podemos partir para a segunda etapa, que é renderizar esse formulário diretamente no HTML. Para isso, vamos trabalhar no arquivo `views.py` do aplicativo **visitantes**, começando pela importação do formulário. Após isso, vamos criar uma variável de nome `form` que será igual à uma

instância da classe `VisitanteForm` e passá-la dentro da variável `context` da view `registrar_visitante`. O arquivo ficará assim:

```
1 from django.shortcuts import render
2 from visitantes.forms import VisitanteForm
3
4 def registrar_visitante(request):
5
6     form = VisitanteForm()
7
8     context = {
9         "nome_pagina": "Registrar visitante",
10        "form": form,
11    }
12
13    return render(request, "registrar_visitante.html", context)
```

Apenas com as alterações realizadas, já podemos trabalhar no template `registrar_visitante.html` para que o formulário seja renderizado de forma automática. Vamos abrir o arquivo `registrar_visitante.html` dentro da pasta **templates** e procurar pelo elemento HTML `<form>`. Substitua todo o conteúdo existente dentro do elemento pela variável `{{ form }}` e acesse <http://127.0.0.1:8000/registrar-visitante/> em seu navegador. O arquivo `registrar_visitante.html` ficará assim:

```
1 <!-- codigo acima omitido -->
2 <div class="card-body">
3     <h4 class="mb-3 text-primary">
4         Formulário para registro de novo visitante
5     </h4>
6
7     <div class="container">
8         {{ form }}
9     </div>
10 </div>
11 <!-- codigo abaixo omitido -->
```



Definimos no contexto também a variável `nome_pagina`, mas desta vez, como "Registrar visitante". Note como o Django reconhece o valor da variável de acordo com cada view e altera o valor no template `base.html`

Melhorando a exibição do nosso formulário

Quando o assunto é criar formulários, o Django faz esse papel muito bem, além de prover uma funcionalidade segura e estável. Veja bem, com menos de 10 linhas conseguimos criar e renderizar um formulário que se adapta totalmente às necessidades do nosso modelo. Desta forma, é altamente recomendado utilizar os formulários do Django para automatizar nosso trabalho.

O Django faz muito bem o trabalho que se propõe a fazer: preparar e reestruturar os dados para renderização, criar o formulário para receber os dados e ainda processar e validar esses dados, mas quando precisamos renderizar essas informações no formato HTML de modo que fique mais atrativo para o usuário, faltam algumas opções. É aí que entra o **django-widget-tweaks**, um pacote Python muito interessante e útil disponibilizado pela comunidade para nos ajudar na renderização dos nossos formulários.

Estilizando nosso formulário com django-widget-tweaks

O **django-widget-tweaks** nos ajuda tornando mais fácil o processo de adicionar atributos e classes aos campos de um formulário Django, afim de aplicar classes personalizadas para alterar aparência ou comportamento dos elementos, quando necessário. Como estamos utilizando um tema que utiliza o **Bootstrap** como base, podemos também utilizar suas classes CSS para alterar a aparência dos elementos.

Isso resolve o problema do nosso formulário não estar sendo exibido de maneira atrativa para o usuário. Isso porque o Django renderiza um formulário HTML simples, sem adicionar classes para alterar o estilo dos elementos que compõem esse formulário. Sendo assim, utilizaremos o **django-widget-tweaks** para adicionar a classe `form-control` aos

campos do nosso formulário, e assim aplicar as características descritas no arquivo de estilização (CSS) do tema utilizado.


Como instalar

Para instalar o **django-widget-tweaks** utilizaremos nosso já conhecido gerenciador de pacotes: o `pip`. Para instalar vamos utilizar o seguinte comando:

```
(env)$ pip install django-widget-tweaks
```

Caso ocorra bem tudo, você terá instalado o **django-widget-tweaks** em seu ambiente virtual. Feito isso, também vamos adicionar o pacote à variável `INSTALLED_APPS` do nosso arquivo de configurações. Para uma melhor organização, vamos criar uma lista separada da lista dos nossos aplicativos.

```
1  # código acima omitido
2
3  INSTALLED_APPS += [
4      "widget_tweaks",
5  ]
6
7  INSTALLED_APPS += [
8      "usuarios",
9      "porteiros",
10     "visitantes",
11 ]
12
13 # código abaixo omitido
```

 Utilizamos três variáveis de mesmo nome e as incrementamos pois assim separamos os aplicativos do Django (primeira), os pacotes Python instalados (segunda) e os aplicativos criados por nós (terceira). Lembre-se que é necessário utilizar o operador `+=` quando queremos incrementar os valores existentes na variável

Importando no template

Agora que instalamos e registramos o pacote em nosso arquivo de configurações, temos que utilizar a tag `{% load widget_tweaks %}` sempre que precisarmos utilizar as funcionalidades do pacote num determinado template. Vamos adicionar a tag logo abaixo da primeira linha do template `registrar_visitante.html`. O arquivo ficará assim:

```
1 {% extends "base.html" %}
2
3 {% load widget_tweaks %}
4
5 <!-- código abaixo omitido -->
```

Utilizando o render_field

Existem duas maneiras que o **django-widget-tweaks** nos permite utilizar suas funcionalidades, mas utilizaremos a tag personalizada `{% render_field %}`, com ela conseguimos descrever nossos campos de forma bem parecida com o HTML5.

Vamos abrir o arquivo `registrar_visitante.html` e substituir a variável `{{ form }}` pelo elemento `<form method="post">` abaixo e seu conteúdo. O código ficará assim:


```
1 <!-- código acima omitido -->
2 <div class="card-body">
3     <h4 class="mb-3 text-primary">
4         Formulário para registro de novo visitante
5     </h4>
6
7     <p class="mb-5 ml-1">
8         <small>
9             O asterisco (*) indica que o campo é obrigatório
10        </small>
11    </p>
12
13    <form method="post">
14        <div class="form-row">
15            {% csrf_token %}
16
17            {% for field in form %}
```

```


18         <div class="form-group col-md-12">
19             <label>{{ field.label }} {% if field.field.required %}
20                 {% render_field field placeholder=field.label class="f
21         </div>
22     {% endfor %}
23 </div>
24
25     <div class="text-right">
26         <button class="btn btn-primary" type="submit">
27             <span class="text">Registrar visitante</span>
28         </button>
29     </div>
30 </form>
31 </div>
32 <!-- codigo abaixo omitido -->

```

Vamos adicionar também um trecho de código HTML com um aviso sinalizando que o asterisco (*) acima dos campos do formulário indica que o campo é obrigatório.

 A tag `{% csrf_token %}` fornece proteção para nossa aplicação, de modo a impedir que sites mal intencionados enviem requisições para ela. Caso a gente não coloque essa tag dentro dos nossos formulários, o Django não aceitará a requisição enviada e mostrará um erro pois não vai identificar a requisição como segura

Logo abaixo da tag `{% csrf_token %}`, estamos utilizando novamente a tag `{% for %}` para realizar um loop, mas desta vez na variável `form`. Quando realizamos um loop em nosso formulário, conseguimos acessar seus campos, e é exatamente o que precisamos fazer: executar um loop e acessar as informações de cada campo para que possamos passá-las para a tag `{% render_field %}` fazer o trabalho de renderização destes campos.

 Aqui temos uma novidade, a utilização da tag `{% if %}`. Uma estrutura condicional que pode ser utilizada em templates. O que estiver dentro dela só será exibido caso o resultado da expressão seja verdadeiro. Ou seja, quando

existem mensagens e a variável `messages` está definida, exibimos o trecho HTML

Para cada campo (*variável field*) em nosso formulário, criamos a estrutura padrão para campos de um formulário do nosso tema. Acessamos também a propriedade `label` para exibir o nome e o *placeholder* do `input` e passamos a variável que representa o campo para a tag `{% render_field %}`. Veja como ficará estrutura de cada campo:

```
1 <div class="form-group col-md-12">
2   <label>{{ field.label }} {% if field.field.required %} * {% endif %}</
3   {% render_field field placeholder=field.label class="form-control" %}
4 </div>
```

Note que definimos também os atributos `placeholder=field.label` e `class="form-control"`, além de verificarmos se o campo é obrigatório e, caso seja, colocamos um asterisco (*) ao lado do nome do campo. Acesse a página e veja na prática como o layout do nosso formulário melhorou e muito!

Capítulo 09

Preparando view para receber requisição do tipo POST

Agora que cuidamos da usabilidade do nosso formulário, podemos seguir com as outras partes da nossa view. Até o momento, apenas criamos a variável que representa o formulário e a passamos no contexto, o que faremos agora é preparar nossa view para receber os dados que serão enviados na requisição.

Conhecendo o objeto request

Você já deve ter notado que sempre que criamos uma view, precisamos que ela receba a variável `request` como argumento. Isso porque, quando falamos do protocolo que sustenta a web, o HTTP, requisições são o que movimentam toda a estrutura. Quando acessamos uma página web estamos enviando uma requisição do tipo `GET`. Isso tudo acontece em questão de segundos e por baixo dos panos, nos fios que conectam a web.

A variável `request` é uma representação da requisição que é enviada à view acessada e contém diversas informações como usuário logado, método HTTP utilizado, navegador e sistema operacional, dentre outras. No momento, nos interessa o método da requisição e o corpo que é enviado.

Para prepararmos a view para receber as informações da requisição, vamos adicionar um `if` abaixo da linha `form = VisitanteForm()` para verificar se o método da requisição é do tipo `POST`. Podemos fazer isso dessa forma:

```
1 from django.shortcuts import render
2 from visitantes.forms import VisitanteForm
3
4 def registrar_visitante(request):
5
6     form = VisitanteForm()
7
8     if request.method == "POST":
9         print("o método é post")
10
11     context = {
```

```
12     "nome_pagina": "Registrar visitante",
13     "form": form,
14 }
15
16 return render(request, "registrar_visitante.html", context)
```



O método `POST` é utilizado sempre que precisamos enviar informações para o servidor. No nosso caso, por exemplo, queremos enviar as informações de um novo visitante a ser registrado e fazemos isso através do formulário HTML

Agora que verificamos se o método da requisição enviada é do tipo `POST`, vamos reutilizar a variável `form`, agora atribuindo a ela outra instância do formulário `VisitanteForm()`, mas agora passando o corpo da requisição à classe, e utilizar o método `is_valid()` do formulário para validar as informações. O código ficará o seguinte:

```
1 from django.shortcuts import render
2 from visitantes.forms import VisitanteForm
3
4 def registrar_visitante(request):
5
6     form = VisitanteForm()
7
8     if request.method == "POST":
9         form = VisitanteForm(request.POST)
10
11         if form.is_valid():
12             form.save()
13
14     context = {
15         "nome_pagina": "Registrar visitante",
16         "form": form,
17     }
18
19     return render(request, "registrar_visitante.html", context)
```

Para passar o corpo da requisição para o formulário, basta utilizarmos a propriedade `POST` do objeto `request` e passá-lo como argumento ao criarmos a nova instância do

formulário, como feito na linha 9 (`form = VisitanteForm(request.POST)`). Feito isso, validamos as informações com o método `is_valid()` e salvamos o formulário utilizando o método `save()` .

Conhecendo um pouco mais dos formulários

Ao acessarmos a página, podemos notar que todos os campos do modelo estão sendo exibidos no formulário, e não é isso que queremos, pois algumas das informações devem ser preenchidas mediante autorização de moradores e dependem outros eventos.

Para especificar os campos que devem ser exibidos e utilizados no formulário, vamos voltar ao arquivo `forms.py` e alterar o atributo `fields` da classe `Meta` do nosso formulário. Abra o arquivo e substitua a string `"__all__"` por uma lista com os nomes dos campos que vamos exibir. O atributo `fields` ficará assim:

```
1  from django import forms
2  from visitantes.models import Visitante
3
4  class VisitanteForm(forms.ModelForm):
5      class Meta:
6          model = Visitante
7          fields = [
8              "nome_completo", "cpf", "data_nascimento",
9              "numero_casa", "placa_veiculo",
10             ]
```

Ao voltar para a página, vamos notar que agora apenas os campos que estão na lista `fields` da classe `Meta` estão sendo exibidos.

Tratando problema com atributo nulo

Quando criamos a classe modelo `Visitante` , falamos sobre o atributo `registrado_por` ser do tipo `ForeignKey` , um tipo de campo que cria um relacionamento entre as classes `Visitante` e `Porteiro` . Olhando a classe `VisitanteForm` , podemos notar que o

atributo não é colocado nos campos do formulário (`fields`), mesmo este sendo uma informação obrigatória em nosso modelo. Se tentarmos adicionar um visitante por meio do formulário, o Django apresentará um erro nos informando que o atributo `registrado_por` do modelo não pode ser nulo.

Para resolver o problema, o que vamos fazer é possibilitar que o campo seja preenchido de maneira automática. Isto é, o campo receberá o valor referente ao porteiro que está logado na dashboard no momento do cadastro.

Para fazer isso, antes de salvar o formulário vamos definir diretamente um valor para o atributo `registrado_por` na função de view. Vamos abrir o arquivo `views.py` e criar uma variável para receber o retorno do método `save` do formulário. Esse método aceita também um argumento opcional de nome `commit`, que quando definido como `False`, retorna uma instância do modelo utilizado no formulário que ainda não foi gravada no banco de dados. Isso é bem útil para quando queremos executar um processamento personalizado antes de salvar o objeto ou até mesmo utilizar outros métodos do modelo. O código vai ficar assim:

```
1  from django.shortcuts import render
2  from visitantes.forms import VisitanteForm
3
4  def registrar_visitante(request):
5
6      form = VisitanteForm()
7
8      if request.method == "POST":
9          form = VisitanteForm(request.POST)
10
11         if form.is_valid():
12             visitante = form.save(commit=False)
13
14             visitante.registrado_por = request.user.porteiro
15
16             visitante.save()
17
18         context = {
19             "nome_pagina": "Registrar visitante",
20             "form": form,
21         }
22
23     return render(request, "registrar_visitante.html", context)
```

Agora, ao invés de salvarmos o formulário diretamente, estamos guardando o resultado do método `save` com o argumento `commit=False`, definindo um valor para o atributo `registrado_por` diretamente e salvando o objeto através da variável `visitante`. Somente no momento em que chamamos o método `visitante.save()` que as alterações são registradas no banco de dados.

⚠ Lembra que falamos que a variável `request` guarda algumas informações da requisição, como usuário logado e método utilizado? Pois bem, conseguimos pegar informações do usuário logado acessando a propriedade `user` da variável `request` (`request.user`). No nosso caso, ainda estamos acessando uma outra propriedade do usuário, a propriedade `porteiro` (`request.user.porteiro`).

Isso acontece devido à funcionalidade de acesso entre os modelos que o Django disponibiliza. Assim como podemos acessar `porteiro.usuario`, definido diretamente como atributo do modelo, podemos fazer o mesmo para o inverso da relação.

Feito isso, vamos apenas importar mais um dos `shortcuts` do Django, além do `render`, que é o `redirect`. O que ele faz é exatamente redirecionar a view para uma URL que quisermos. Vamos utilizá-lo para evitar que os mesmos dados sejam enviados mais de uma vez ao nosso servidor. Sempre que um formulário for enviado e as informações forem salvas no banco de dados, vamos redirecionar a página. Para isso, basta importar o `redirect` ao lado do `render` e utilizá-lo passando o nome da URL para onde queremos mandar o usuário. O código ficará assim:

```
1 from django.shortcuts import render, redirect
2 from visitantes.forms import VisitanteForm
3
4 def registrar_visitante(request):
5
6     form = VisitanteForm()
7
8     if request.method == "POST":
9         form = VisitanteForm(request.POST)
```



```
10
11     if form.is_valid():
12         visitante = form.save(commit=False)
13
14         visitante.registrado_por = request.user.porteiro
15         visitante.save()
16
17         return redirect("index")
18
19     context = {
20         "nome_pagina": "Registrar visitante",
21         "form": form,
22     }
23
24     return render(request, "registrar_visitante.html", context)
```

Agora vamos voltar à página <http://127.0.0.1:8000/registrar-visitante/> e registrar um visitante. O visitante deverá ser registrado e a requisição redirecionada para a página inicial da dashboard.



O visitante registrado deverá estar listado na tabela de visitantes recentes

Exibindo uma mensagem para o usuário ao cadastrar novo visitante

Agora que o formulário está sendo exibido e funcionando corretamente, inclusive salvando os visitantes em nosso banco de dados, vamos melhorar um pouco a usabilidade da nossa dashboard. Sempre que o sistema finaliza uma ação solicitada pelo usuário, é interessante que seja dado um feedback visual para facilitar o entendimento a respeito do que aconteceu. Desta forma, o que faremos agora é trabalhar na view para que, quando o visitante for registrado, uma mensagem seja exibida dizendo algo como "hey, cara, o visitante foi registrado com sucesso!".

Conhecendo o Django messages

O Django já nos disponibiliza o módulo `messages` para resolver isso. Toda a configuração necessária para o funcionamento de suas funcionalidades já vêm por padrão quando criamos um novo projeto Django, então o que precisamos fazer é apenas inserir o código `from django.contrib import messages` para importar as funcionalidades e utilizá-las em nossas views. Vamos colocá-lo na primeira linha e o início do arquivo `views.py` do aplicativo **visitantes** ficará assim:

```
1 from django.contrib import messages
2 from django.shortcuts import render, redirect
3 from visitantes.forms import VisitanteForm
4
5 # código abaixo omitido
```

Com o módulo importado em nossa view, podemos utilizá-lo tranquilamente. Vamos adicionar uma mensagem de sucesso logo após a linha que salva a instância do visitante (`visitante.save()`) utilizando o método `success` do módulo `messages` e passando a ele a `request` e um texto para ser exibido. O arquivo `views.py` ficará assim:

```
1 from django.contrib import messages
2 from django.shortcuts import render, redirect
3 from visitantes.forms import VisitanteForm
4
5 def registrar_visitante(request):
6
7     form = VisitanteForm()
8
9     if request.method == "POST":
10         form = VisitanteForm(request.POST)
11
12         if form.is_valid():
13             visitante = form.save(commit=False)
14
15             visitante.registrado_por = request.user.porteiro
16
17             visitante.save()
18
19             messages.success(
20                 request,
21                 "Visitante registrado com sucesso"
22             )
23
```

```

24         return redirect("index")
25
26     context = {
27         "nome_pagina": "Registrar visitante",
28         "form": form,
29     }
30
31     return render(request, "registrar_visitante.html", context)

```

Alterando o template para exibir as mensagens

Nossa view para registro de visitantes está completa: estamos exibindo o formulário corretamente, verificando quando ocorre uma requisição do tipo POST, validando as informações enviadas, definindo automaticamente o porteiro que registrou o visitante, exibindo uma mensagem e ainda redirecionamos a requisição quando finalizamos todo o processo com sucesso. Ufa! É tanta coisa que ficou até difícil de listar.

Com tudo isso feito, temos agora que disponibilizar um lugar em nosso template para que a mensagem seja exibida, como um alerta mesmo. Como estamos direcionando nosso usuário para a página inicial da dashboard, faz sentido que a gente coloque a mensagem no template `index.html`, pelo menos por hora.

Vamos abrir o template `index.html` e, logo acima do primeiro elemento `<div class="row">` do arquivo, vamos inserir o seguinte trecho de código:

```

1  {% if messages %}
2      {% for message in messages %}
3          <div class="alert alert-success" role="alert">
4              {{ message }}
5          </div>
6      {% endfor %}
7  {% endif %}

```

O módulo de mensagens do Django também nos disponibiliza uma variável chamada `messages` que pode ser acessada nos templates. Com ela, conseguimos verificar se existem mensagens e, por meio de um loop, verificar as informações de cada mensagem. É o que estamos fazendo, primeiro verificamos se existem mensagem (`{% if messages %}`

), caso positivo, nós executamos um loop e acessamos a mensagem utilizando a variável criada no loop (`{{ message }}`).

Agora você pode cadastrar mais um visitante e ver a mensagem de sucesso sendo exibida!

Tratando possíveis erros em nosso formulário

Nossa mensagem de sucesso já está sendo exibida corretamente, mas o que acontece se ocorrer algum erro e os dados enviados não forem aceitos? Não podemos deixar que a aplicação pare e precisamos indicar para o usuário que os dados que ele inseriu estão incorretos.

Como nosso formulário de registro de visitante está no arquivo

`registrar_visitante.html` , trabalharemos nele. Logo acima do elemento `<form method="post">` , vamos inserir o seguinte trecho de código:

```
1 {% if form.errors %}
2     {% for field in form %}
3         {% if field.errors %}
4             {% for error in field.errors %}
5                 <div class="alert alert-warning" role="alert">
6                     {{ error }}
7                 </div>
8             {% endfor %}
9         {% endif %}
10    {% endfor %}
11 {% endif %}
```

A estrutura HTML é bem parecida com a utilizada para a mensagem de sucesso, mas com algumas pequenas diferenças. Mais uma vez, utilizaremos as tags `{% if %}` e `{% for %}` . Primeiro vamos verificar se existem erros no formulário (`{% if form.errors %}`) e, caso verdadeiro, realizar um loop nos seus campos (`{% for field in form %}`). Desta vez, verificamos também se existem erros em cada campo (`{% if field.errors %}`) e executamos um loop nesses erros (`{% for error in field.errors %}`), caso existam.

Deixando nossas mensagens de erro mais claras

Um último detalhe para melhorarmos ainda mais a experiência do usuário ao utilizar nossa dashboard é certamente melhorar os textos das mensagens que são exibidas para o usuário em caso de erro. Muito mais que informar que ocorreu um erro, essas mensagens devem direcionar o usuário para o correto preenchimento das informações.

Para fazer isso, vamos atuar diretamente no arquivo `forms.py` do aplicativo **visitantes**. O que faremos é adicionar o atributo `error_messages` à classe `Meta` da classe `VisitanteForm`, logo abaixo de `fields`.

O `error_messages` é um dicionário que deve conter chaves com os nomes dos campos do modelo. Desta forma, cada valor do dicionário `error_messages` representa um campo do formulário e é também um dicionário, mas que, desta vez, recebe como chave os tipos de erros que podem acontecer nos formulários do Django seguido da mensagem a ser exibida para cada erro. Por hora utilizaremos os tipos `required`, que funciona para quando o campo não é preenchido e `invalid`, para quando o formato da informação enviada é inválido.

Nosso formulário, a classe `VisitanteForm`, ficará assim:

```
1 class VisitanteForm(forms.ModelForm):
2     class Meta:
3         model = Visitante
4         fields = [
5             "nome_completo", "cpf", "data_nascimento",
6             "numero_casa", "placa_veiculo",
7         ]
8         error_messages = {
9             "nome_completo": {
10                 "required": "O nome completo do visitante é obrigatório pa
11             },
12             "cpf": {
13                 "required": "O CPF do visitante é obrigatório para o regis
14             },
15             "data_nascimento": {
16                 "required": "A data de nascimento do visitante é obrigatór
17                 "invalid": "Por favor, informe um formato válido para a da
18             },
19             "numero_casa": {
```

```
20         "required": "Por favor, informe o número da casa a ser vis  
21     }  
22 }
```

Capítulo 10

Criando tela para exibir informações de visitante

Seguindo as especificações que recebemos do cliente, sabemos que existe a necessidade de visualização das informações do visitante na dashboard. Sendo assim, é necessário que a gente trabalhe para que seja possível buscar as informações de cada visitante registrado e exibir essas informações em um template à parte, afim de mostrar mais detalhes de cada visitante.

Se você observar a tabela que lista os visitantes recentes na página inicial da dashboard, vai notar que existe um link para ver informações detalhadas de cada visitante. O que vamos fazer é desenvolver a view que vai buscar a informação de um visitante por vez e exibir essas informações de forma estruturada em um template HTML.

Criando a view

Como já sabemos, começaremos a trabalhar na funcionalidade pela função de view, no arquivo `views.py`. Abaixo da função `registrar_visitante()`, vamos criar a função `informacoes_visitante()`.

Essa função de view terá uma pequena diferença com relação à função `registrar_visitante()`, desta vez, além do argumento `request`, vamos também receber um argumento de nome `id`, que representará o `id` do visitante a ser buscado em nosso banco de dados. É assim que vamos identificar qual visitante devemos buscar.

Já vamos aproveitar para criar a view e deixar algumas coisas prontas, como a variável `context` e o retorno renderizando o template `informacoes_visitante.html`, que ainda vamos criar. Por hora, nossa função `informacoes_visitante()` ficará assim:

```
1 # código acima omitido
2
3 def informacoes_visitante(request, id):
```

```
4
5     context = {
6         "nome_pagina": "Informações de visitante",
7     }
8
9     return render(request, "informacoes_visitante.html", context)
```

Conhecendo o atalho `get_model_or_404`

Agora que nossa view está escrita e recebe um `id`, precisamos buscar o visitante em nosso banco de dados utilizando esse `id`. Existem várias formas de fazer isso, mas vamos utilizar o atalho `get_model_or_404()` do Django. Para utilizá-lo, temos que passar uma classe modelo a ser utilizada na busca e o parâmetro pelo qual queremos buscar (em nosso caso, utilizaremos o `id` e, por isso, vamos passá-lo como segundo argumento da função `get_model_or_404()`).

Antes de tudo, claro, vamos importar a função `get_model_or_404()` em nossa view juntamente com a classe modelo `Visitante`, pois também precisaremos dela. A função `get_model_or_404()` está localizada no mesmo pacote que as funções `render` e `redirect`, desta forma, vamos alterar as primeiras linhas do nosso código para o seguinte:

```
1 from django.contrib import messages
2 from django.shortcuts import (
3     render, redirect, get_object_or_404
4 )
5
6 from visitantes.models import Visitante
7 from visitantes.forms import VisitanteForm
8
9 # código abaixo omitido
```

Conforme vimos, para utilizar a função `get_object_or_404()`, precisamos passar a classe modelo e o atributo a ser utilizado para busca. Vamos passar a classe `Visitante` e dizer que vamos buscar o visitante pelo `id` e que o `id` é igual à variável que estamos recebendo como argumento da função `informacoes_visitante()`. Nosso código ficará assim:


```
1 def informacoes_visitante(request, id):
2
3     visitante = get_object_or_404(Visitante, id=id)
4
5     context = {
6         "nome_pagina": "Informações de visitante",
7         "visitante": visitante
8     }
9
10    return render(request, "informacoes_visitante.html", context)
```

Não vamos nos esquecer de passar a variável `visitante` no contexto, para que possamos acessá-la nos templates.

Criando URL para acessar informações de visitante

A essa altura você já deve ter percebido que precisamos mapear a nova view em uma URL para que a gente possa acessá-la através do navegador. Sendo assim, vamos trabalhar no arquivo `urls.py` do nosso projeto e criar a URL de nome `informacoes_visitante`.

Essa URL vai ser um pouco diferente das que criamos até agora. Conforme visto, precisamos passar um `id` como argumento para a função que será o identificador do visitante a ser buscado no banco de dados. Isso pode ser feito através da URL que será acessada, pois podemos passar o `id` diretamente no endereço da URL. Por exemplo, se quisermos buscar as informações do visitante de `id=1`, podemos acessar a URL <http://127.0.0.1:8000/visitantes/1/>.

O primeiro passo nós já fizemos, que é receber o argumento na função de view que será mapeada na URL. Agora temos que criar a URL no arquivos `urls.py` e utilizar a sintaxe `<int:id>` para indicar que precisamos receber uma variável do tipo `int` e de nome `id`. Nossa URL ficará assim:

```
1 from django.urls import path
2 from django.contrib import admin
3
```

```
4 from usuarios.views import index
5
6 from visitantes.views import (
7     registrar_visitante, informacoes_visitante
8 )
9
10 urlpatterns = [
11     # codigo acima omitido...
12
13     path(
14         "registrar-visitante/",
15         registrar_visitante,
16         name="registrar_visitante",
17     ),
18
19     path(
20         "visitantes/<int:id>/",
21         informacoes_visitante,
22         name="informacoes_visitante",
23     )
24 ]
```

Criando template para exibir informações de visitante

Com nossa view pronta, agora precisamos criar o arquivo `informacoes_visitante.html` na pasta **templates** do nosso projeto. Mais uma vez, você pode fazer download do template clicando no link abaixo:



Iniciar o download

informacoes_visitante.zip - 947B

Após o download, coloque o arquivo na pasta **templates** do projeto e abra em seu editor de texto, pois ainda vamos alterar algumas coisas nele.

Ao abrir o arquivo, você vai perceber que as informações estão definidas diretamente no template. Para tornar o template dinâmico e funcional, vamos alterar os valores do atributo `value` dos elementos `field` do nosso HTML. A estrutura é bem parecida com a que utilizamos nos formulários, com a diferença que vamos renderizar todos os campos manualmente para que possamos personalizar melhor a estrutura do nosso template.

Como passamos a variável `visitante` no contexto, podemos acessá-la diretamente utilizando a sintaxe `{{ visitante.nome_do_atributo }}`. Vamos alterar os valores estáticos para a sintaxe de template do Django e tornar nosso template dinâmico. Se você ficar na dúvida sobre qual atributo deve exibir, o elemento `<label>` pode te ajudar!

Antes que a gente esqueça, vamos alterar também o texto que exibe o porteiro responsável pelo registro e o horário que o visitante foi registrado. Para isso, vamos alterar o texto de "Visitante registrado em 15/05/2018 por Walter White" para "Visitante registrado em `{{ visitante.horario_chegada }}` por `{{ visitante.registrado_por }}`". O template ficará assim:

```
1  <div class="card-body">
2    <h4 class="mb-3 text-primary">
3      Informações gerais
4    </h4>
5
6    <form>
7      <div class="form-row">
8        <div class="form-group col-md-6">
9          <label>Horário de chegada</label>
10         <input type="text" class="form-control" value="{{ visitante.horario_chegada }}">
11       </div>
12
13       <div class="form-group col-md-6">
14         <label>Número da casa a ser visitada</label>
15         <input type="text" class="form-control" value="{{ visitante.numero_casa }}">
16       </div>
17     </div>
18
19     <div class="form-row">
20       <div class="form-group col-md-4">
21         <label>Horário de autorização de entrada</label>
22         <input type="text" class="form-control" value="{{ visitante.horario_entrada }}">
23       </div>
24
25       <div class="form-group col-md-4">
26         <label>Entrada autorizada por</label>
27         <input type="text" class="form-control" value="{{ visitante.registrado_por }}">
28       </div>
29
30       <div class="form-group col-md-4">
31         <label>Horário de saída</label>
32         <input type="text" class="form-control" value="{{ visitante.horario_saida }}">
33       </div>
34     </div>
35  </div>
```

```

35     </form>
36
37     <h4 class="mb-3 mt-4 text-primary">
38         Informações pessoais
39     </h4>
40
41     <form>
42         <div class="form-row">
43             <div class="form-group col-md-6">
44                 <label>Nome completo</label>
45                 <input type="text" class="form-control" value="{{ visitante.nome }}">
46             </div>
47
48             <div class="form-group col-md-6">
49                 <label>CPF</label>
50                 <input type="text" class="form-control" value="{{ visitante.cpf }}">
51             </div>
52         </div>
53
54         <div class="form-row">
55             <div class="form-group col-md-6">
56                 <label>Data de nascimento</label>
57                 <input type="text" class="form-control" value="{{ visitante.data_nascimento }}">
58             </div>
59
60             <div class="form-group col-md-6">
61                 <label>Placa do veículo</label>
62                 <input type="text" class="form-control" value="{{ visitante.placa }}">
63             </div>
64         </div>
65     </form>
66
67     <p class="mr-2 mt-3 mb-4 text-right">
68         <small>
69             Visitante registrado em {{ visitante.horario_chegada }} por {{ visitante.nome }}
70         </small>
71     </p>
72
73     <div class="mr-1 text-right">
74         <a href="#" class="btn btn-secondary text-white" type="button">
75             <span class="text">Voltar</span>
76         </a>
77     </div>
78 </div>

```

Feito isso, vamos abrir o navegador e acessar o endereço

<http://127.0.0.1:8000/visitantes/1/>. Você deverá visualizar as informações do primeiro visitante que registramos no banco de dados.

Criando métodos personalizados para exibir informações do Visitante

Conforme exibimos os campos, você deve ter observado que alguns deles ainda não estão preenchidos no banco de dados e, por isso, exibem um valor em branco ou uma informação pouco clara do que, de fato, representa (`None`). Para melhorar a exibição destes campos e, conseqüentemente, a melhorar a usabilidade da nossa dashboard, criaremos métodos personalizados nas classes modelo para que possamos exibir uma informação útil e clara, mesmo quando o campo não está preenchido.

Métodos são funções que existem dentro das classes e podem definir comportamentos para os objetos. Em nosso caso, criaremos métodos que alteram a forma com que as informações são exibidas para o usuário. Se, por exemplo, a entrada do morador ainda não tiver sido autorizada, podemos exibir algo como "Visitante aguardando autorização" nos campos `horario_autorizacao` e `morador_responsavel` . O mesmo vale para o campo `horario_saida` , que só será preenchido no momento que a visita for finalizada.

Criando método para exibir horário de saída

Vamos criar métodos que vão substituir a exibição de alguns atributos, começando pelo horário de saída. Antes de tudo, você precisa saber que para criar um método dentro de uma classe, tudo que precisamos fazer é criar uma função dentro dessa classe que receberá o argumento `self` . Esse argumento nos possibilita acessar as propriedades da própria classe.

Os métodos que buscam informações, em geral, recebem o nome de **getters** e mantemos sempre a chave "get" no início de seus nomes. Abaixo do atributo `registrado_por` , vamos escrever o método `get_horario_saida()` . Esse método, antes de tudo, precisa verificar se o atributo `horario_saida` está preenchido e, caso não esteja, retornar o texto "Horário de saída não registrado". Para fazer isso, vamos utilizar a estrutura condicional `if` . O método ficará assim:

```
1 # código acima omitido
2 def get_horario_saida(self):
3     if self.horario_saida:
```

```

4         return self.horario_saida
5
6         return "Horário de saída não registrado"
7
8     class Meta:
9         verbose_name = "Visitante"
10        verbose_name_plural = "Visitantes"
11        db_table = "visitante"
12
13    def __str__(self):
14        return self.nome_completo

```

Criando métodos para exibir horário de autorização de entrada e morador responsável por autorizar a entrada

Faremos o mesmo para os atributos `horario_autorizacao` e `morador_responsavel`, que serão exibidos somente se existir um valor a ser exibido. Caso contrário, vamos exibir um texto padrão. Vamos começar escrevendo o método `get_horario_autorizacao()`, que será bem parecido com o método `get_horario_saida()`. O método `get_horario_autorizacao()` ficará assim:

```

1 def get_horario_autorizacao(self):
2     if self.horario_autorizacao:
3         return self.horario_autorizacao
4
5     return "Visitante aguardando autorização"

```

Para o método `get_morador_responsavel()` vamos fazer bem parecido. Nosso método ficará assim:

```

1 def get_morador_responsavel(self):
2     if self.morador_responsavel:
3         return self.morador_responsavel
4
5     return "Visitante aguardando autorização"

```

Criando método para exibir placa do veículo utilizado na visita

O método `get_veiculo()` será parecido com os outros, mas também terá um outro texto padrão:

```
1 def get_placa_veiculo(self):
2     if self.placa_veiculo:
3         return self.placa_veiculo
4
5     return "Veículo não registrado"
```

Utilizando métodos personalizados nos templates

Com nossos métodos criados, temos que alterar o template

`informacoes_visitante.html` para que exiba os métodos ao invés dos atributos. A sintaxe para exibição de métodos nos templates é bem parecida com a que utilizamos para os atributos, inclusive.

Onde temos os atributos `horario_autorizacao`, `morador_responsavel`, `horario_saida` e `placa_veiculo`, vamos alterar para métodos criados. Ou seja, ao invés de `{{ visitante.horario_autorizacao }}`, utilizaremos `{{ visitante.get_horario_autorizacao }}`. O template ficará assim:

```
1 <div class="card-body">
2     <h4 class="mb-3 text-primary">
3         Informações gerais
4     </h4>
5
6     <form>
7         <div class="form-row">
8             <div class="form-group col-md-6">
9                 <label>Horário de chegada</label>
10                <input type="text" class="form-control" value="{{ visitante.get_horario_autorizacao }}">
11            </div>
12
13            <div class="form-group col-md-6">
```

```

14         <label>Número da casa a ser visitada</label>
15         <input type="text" class="form-control" value="{{ visitant
16     </div>
17 </div>
18
19     <div class="form-row">
20         <div class="form-group col-md-4">
21             <label>Horário de autorização de entrada</label>
22             <input type="text" class="form-control" value="{{ visitant
23         </div>
24
25         <div class="form-group col-md-4">
26             <label>Entrada autorizada por</label>
27             <input type="text" class="form-control" value="{{ visitant
28         </div>
29
30         <div class="form-group col-md-4">
31             <label>Horário de saída</label>
32             <input type="text" class="form-control" value="{{ visitant
33         </div>
34     </div>
35 </form>
36
37 <h4 class="mb-3 mt-4 text-primary">
38     Informações pessoais
39 </h4>
40
41 <form>
42     <div class="form-row">
43         <div class="form-group col-md-6">
44             <label>Nome completo</label>
45             <input type="text" class="form-control" value="{{ visitant
46         </div>
47
48         <div class="form-group col-md-6">
49             <label>CPF</label>
50             <input type="text" class="form-control" value="{{ visitant
51         </div>
52     </div>
53
54     <div class="form-row">
55         <div class="form-group col-md-6">
56             <label>Data de nascimento</label>
57             <input type="text" class="form-control" value="{{ visitant
58         </div>
59
60         <div class="form-group col-md-6">
61             <label>Placa do veículo</label>
62             <input type="text" class="form-control" value="{{ visitant
63         </div>
64     </div>

```



```

65     </form>
66
67     <p class="mr-2 mt-3 mb-4 text-right">
68         <small>
69             Visitante registrado em {{ visitante.horario_chegada }} por {{
70         </small>
71     </p>
72
73     <div class="mr-1 text-right">
74         <a href="#" class="btn btn-secondary text-white" type="button">
75             <span class="text">Voltar</span>
76         </a>
77     </div>
78 </div>

```

Não podemos esquecer do template `index.html`, onde também vamos utilizar os métodos `get_horario_autorizacao` e `get_morador_responsavel`. O trecho de código ficará assim:

```

1  <tbody>
2      {% for visitante in todos_visitantes %}
3          <tr>
4              <td>{{ visitante.nome_completo }}</td>
5              <td>{{ visitante.cpf }}</td>
6              <td>{{ visitante.horario_chegada }}</td>
7              <td>{{ visitante.get_horario_autorizacao }}</td>
8              <td>{{ visitante.get_morador_responsavel }}</td>
9              <td>
10                 <a href="{% url 'informacoes_visitante' id=visitante.id %}"
11                     Ver informações
12                 </a>
13             </td>
14          </tr>
15      {% endfor %}
16 </tbody>


```

Você pode criar os métodos que quiser e exibir as informações conforme precisar. Existem diversas possibilidades e aplicações, então sinta-se livre para explorar essas possibilidades!

Utilizando o Django para renderizar nossas URLs

Para acessar as informações de cada visitante, precisamos acessar a URL

`http://127.0.0.1/visitantes/{id}/`, onde o `{id}` será um valor diferente para cada visitante. Até agora fizemos isso manualmente, mas você deve estar se perguntando: como vamos fazer para renderizar uma URL diferente para cada de visitante de forma automática?

 Quando criamos nosso modelo de visitante, não criamos o atributo `id`, mas o Django faz isso por nós para que possamos utilizá-lo como `primary_key`. Além disso, o atributo `id` deverá ser único e, para cada novo visitante registrado, o valor será aumentado em um. Sendo assim, não existirá dois visitante de mesmo `id`.

Para nossa sorte, o pessoal responsável pelo Django já pensou em tudo por nós. Dentre as tags de template, existe a tag `{% url %}`. Ela tem a função de renderizar as URLs do nosso projeto de forma automática, bastando que a gente passe apenas o nome da URL da ser renderizada (sim, é exatamente o valor que definimos para o argumento `name` na definição da URL no arquivos `urls.py`).

Vamos abrir o arquivo `index.html` e utilizar a tag `{% url %}` para renderizar a URL que irá nos direcionar para o template de informações de cada visitante. Na tabela que exibe as informações dos visitantes recentes existe um link que exibe o texto "Ver detalhes". Vamos alterar o valor de `href` do elemento `<a>` de `#` para

`{% url 'informacoes_visitante' id=visitante.id %}`. Primeiro passamos nome da URL e depois podemos passar argumentos necessários que, para esse caso, é somente a `id`. Note também que, dentro do loop, o `id` de cada visitante é acessado da mesma forma que os outros atributos. O loop ficará assim:

```
1 {% for visitante in todos_visitantes %}
2     <td>{{ visitante.nome_completo }}</td>
3     <td>{{ visitante.cpf }}</td>
4     <td>{{ visitante.horario_chegada }}</td>
5     <td>{{ visitante.horario_autorizacao }}</td>
```

```

6      <td>{{ visitante.morador_responsavel }}</td>
7      <td>
8          <a href="{% url 'informacoes_visitante' id=visitante.id %}">
9              Ver detalhes
10         </a>
11     </td>
12 {% endfor %}

```

Para facilitar o acesso à URL de registro de visitantes, vamos inserir um botão ao lado do nome da página que deverá nos direcionar para a página <http://127.0.0.1:8000/registrar-visitante/>. Abaixo do elemento

`<h1 class="h3 mb-0 text-gray-800">{{ nome_pagina }}</h1>` vamos inserir o seguinte trecho de código:

```

1  <a href="{% url 'registrar_visitante' %}" class="btn btn-primary btn-icon-
2      <span class="text">Registrar visitante</span>
3
4      <span class="icon text-white-50">
5          <i class="fas fa-user-plus"></i>
6      </span>
7  </a>

```

Como nossa URL `registrar_visitante` não recebe argumentos, passamos para a tag apenas o nome da mesma.

Renderizando a URL para retornar à página inicial

Agora que já sabemos utilizar a tag `{% url %}`, vamos utilizá-la também para renderizar o endereço para nos direcionar à página inicial da dashboard. No menu lateral à esquerda existe um texto com um ícone e o escrito "Início", e é ele que vamos buscar em nosso template `base.html`. O trecho de código ficará assim:

```

1  <a class="nav-link" href="{% url 'index' %}">
2      <i class="fas fa-home"></i>
3      <span>Início</span>
4  </a>

```

Capítulo 11

Criando funcionalidade para autorização de entrada de visitante

Nos capítulos anteriores, nos dedicamos à criação das funcionalidades para registro e visualização de informações de visitantes. Aprendemos um pouco sobre como requisições funcionam, aprendemos a utilizar os formulários do Django, adaptamos os templates e aprendemos um pouco mais como eles funcionam, conhecemos o `django-widget-tweaks`, o Django `messages` e o `get_object_or_404` e ainda criamos métodos personalizados para nosso modelo de visitantes. Olha só quanta coisa conseguimos absorver somente nesses últimos capítulos!

Com essas funcionalidades criadas, precisamos seguir o roteiro que criamos com base nas necessidades do cliente, de modo que, agora, trabalharemos na funcionalidade que autoriza a entrada do visitante no condomínio. Quando um visitante chega na portaria e se identifica, o mesmo deve aguardar que o colaborador responsável, no caso o porteiro, entre em contato com um morador da casa a ser visitada e autorize que este visitante adentre ao condomínio.

Ao receber a informação de que o visitante pode adentrar ao condomínio, o porteiro deverá registrar na dashboard qual o nome do morador que autorizou a entrada deste visitante. Para fins de controle, também é necessário que o sistema registre o horário em que essa autorização ocorreu. Sendo assim, no momento da autorização do visitante, devemos registrar o horário e o nome do morador responsável pela autorização.


Criando um status diferente para cada estágio da visita

Antes de seguir em frente, precisamos analisar o cenário e extrair algumas informações com base nos fluxos e eventos que ocorrem em uma visita. Existem três cenários para o visitante: quando ele chega na portaria e está aguardando autorização (status 1), quando ele está dentro do condomínio realizando a visita (status 2) e quando ele vai embora e finaliza a visita (status 3).

Definir e tornar esses status explícitos é interessante pois assim podemos diferenciar os estágios em que cada visitante se encontra e ainda contabilizar visitantes estão aguardando autorização, em visita dentro do condomínio e quantos já foram embora. Assim temos mais clareza com relação às informações e ainda conseguimos contabilizar esses números para exibir na página inicial da nossa dashboard.

Vamos voltar ao nosso arquivo `models.py` do aplicativo **visitantes** e adicionar o atributo `status` ao modelo de Visitante. Ele será do tipo `CharField`, mas com a diferença que receberá uma lista pré determinada de opções disponíveis para escolha. Essa lista deverá guardar as opções disponíveis e devemos definir sempre o valor que será salvo em nosso banco de dados e o valor que será exibido para o usuário final. Antes de criar o atributo, vamos criar a variável `STATUS_VISITANTE`, que ficará da seguinte forma:

```
1 class Visitante(models.Model):
2
3     STATUS_VISITANTE = [
4         ("AGUARDANDO", "Aguardando autorização"),
5         ("EM_VISITA", "Em visita"),
6         ("FINALIZADO", "Visita finalizada"),
7     ]
8
9     # código abaixo omitido
```

 A lista `STATUS_VISITANTE` segue o que foi dito anteriormente e define os status `Aguardando autorização`, `Em visita` e `Finalizado`. O primeiro valor, em letras maiúsculas, será guardado no banco de dados e o segundo é o texto que será exibido para o usuário final. Sendo assim, quando o visitante receber o status `AGUARDANDO`, o texto **Aguardando autorização** será exibido.

Com isso, agora vamos adicionar o atributo `status` ao nosso modelo. Além dos argumentos `verbose_name` e `max_length` que já conhecemos, também vamos passar os argumentos `choices` e `default`. O primeiro é a lista que criamos com as opções disponíveis para escolha e o segundo é o valor padrão a ser definido quando uma instância do modelo for criada. Nosso código ficará assim:

```
1 class Visitante(models.Model):
2
3     STATUS_VISITANTE = [
4         ("AGUARDANDO", "Aguardando autorização"),
5         ("EM_VISITA", "Em visita"),
6         ("FINALIZADO", "Visita finalizada"),
7     ]
8
9     status = models.CharField(
10         verbose_name="Status",
11         max_length=10,
12         choices=STATUS_VISITANTE,
13         default="AGUARDANDO",
14     )
15
16     # código abaixo omitido
```

❗ O status default será AGUARDANDO pois sempre que um visitante chega à portaria, fica aguardando a autorização de um morador

Criando o arquivo de migrações

Como vimos anteriormente, sempre que mudamos a estrutura do nosso modelo, precisamos criar um arquivo que registra essas alterações em formato de migração para que seja possível efetuar as alterações no banco de dados posteriormente. Mais uma vez, utilizaremos o comando `makemigrations`:

```
(env)$ python manage.py makemigrations visitantes
```

O terminal deverá mostrar algo parecido com isto:

```
1 Migrations for 'visitantes':
2   visitantes/migrations/0005_visitante_status.py
3   - Add field status to visitante
```

Efetuando as alterações no banco de dados

Nada diferente do que já vimos, vamos agora utiliza o comando `migrate` para efetuar as mudanças em nosso banco de dados:

```
(env)$ python manage.py migrate
```

Criando formulário para atualizar atributos específicos do visitante

Sabemos que precisamos registrar o nome do morador responsável por autorizar a entrada do visitante, além de salvar data e hora e, agora que temos um status, alterar esse status. Por padrão e uma questão lógica, como falamos no tópico anterior, sempre que criamos um visitante o mesmo recebe o status `AGUARDANDO` e, quando sua entrada for autorizada, vamos alterar esse status para `EM_VISITA`.

Criaremos a funcionalidade na tela que exibe as informações do visitante, de forma que, quando o visitante estiver aguardando autorização, vamos exibir um botão para executar a funcionalidade que autorizará sua entrada. Utilizaremos um formulário para receber o nome do morador responsável e as informações referentes ao horário de autorização e status serão definidas diretamente na view.

Para começar, vamos abrir o arquivo `forms.py` do aplicativos **visitantes** e criar o formulário `AutorizaVisitanteForm`, uma subclasse de `ModelForm` bem parecida com a que já criamos, com a diferença que terá apenas o campo `morador_responsavel` na lista `fields`:

```
1 class AutorizaVisitanteForm(forms.ModelForm):
2     morador_responsavel = forms.CharField(required=True)
3
4     class Meta:
5         model = Visitante
6         fields = [
```

```
7         "morador_responsavel",
8     ]
9     error_messages = {
10         "morador_responsavel": {
11             "required": "Por favor, informe o nome do morador responsável"
12         }
13     }
```

Ao criarmos um formulário, podemos também sobrescrever os campos definidos automaticamente por causa da classe modelo, caso a gente queira complementar alguma informação ou personalizar algum comportamento. No nosso caso, como o atributo `morador_responsavel` não é obrigatório no modelo, também não será no formulário, mesmo que a gente esteja recebendo apenas ele. Por isso vamos sobrescrever o campo no formulário, afim de explicitar que ele deve ser obrigatório. Além disso, também vamos definir uma mensagem de erro para caso o campo não seja preenchido. Feito isso, já podemos utilizar o formulário em nossa view.

Alterando view para autorizar entrada de visitante

Agora que criamos um formulário para receber o nome do morador responsável, temos que realizar algumas alterações em nossa view e em nosso template para que o formulário seja exibido e funcione corretamente.

Já escrevemos um código bem parecido com o que vamos escrever agora, o da view `registrar_visitante`. Desta vez, vamos importar o formulário `AutorizaVisitanteForm` no arquivo `views.py` do aplicativo **visitantes**. O trecho onde as importações são feitas ficará assim:

```
1 from django.contrib import messages
2 from django.shortcuts import (
3     render, redirect, get_object_or_404
4 )
5
6 from visitantes.models import Visitante
7 from visitantes.forms import (
8     VisitanteForm, AutorizaVisitanteForm
9 )
```



```
10
11 # código abaixo omitido
```

Com o formulário importado no arquivo `views.py`, vamos trabalhar na função `informacoes_visitante()`, exatamente da forma que fizemos na função `registrar_visitante()`: criar a variável `form`, verificar se o método POST está sendo utilizado na requisição, passar o corpo da requisição para o formulário, verificar se as informações são válidas, salvar o formulário, exibir a mensagem de sucesso e redirecionar o usuário. Não podemos nos esquecer, claro, de passar o formulário no contexto da view. A função ficará assim:

```
1  def informacoes_visitante(request, id):
2
3      visitante = get_object_or_404(Visitante, id=id)
4
5      form = AutorizaVisitanteForm()
6
7      if request.method == "POST":
8          form = AutorizaVisitanteForm(
9              request.POST, instance=visitante
10         )
11
12         if form.is_valid():
13             form.save()
14
15             messages.success(
16                 request,
17                 "Entrada de visitante autorizada com sucesso"
18             )
19
20             return redirect("index")
21
22     context = {
23         "nome_pagina": "Informações de visitante",
24         "visitante": visitante,
25         "form": form,
26     }
27
28     return render(request, "informacoes_visitante.html", context)
```

Note que, desta vez, passamos também o argumento `instance` para o nosso formulário. Quando fazemos isso, o Django entende que queremos atualizar o objeto em questão (o visitante com `id` igual à passada para a função, no caso). Estamos dizendo para o Django atualizar o visitante em questão utilizando a informação do corpo da requisição.

Alterando template para exibir modal com formulário

Como passamos a variável `form` no contexto, podemos agora alterar nosso template para que exiba esse formulário. Como as informações do visitante já ocupam grande parte do template, utilizaremos um elemento HTML conhecido como modal para exibir o formulário. Na verdade, ele é feito unindo as tecnologias HTML, CSS e Javascript e é um elemento que se sobrepõe aos outros, quase como um pop-up (aquelas janelas chatas que piscam na tela).

Antes de tudo, vamos adicionar o botão que será responsável por exibir o modal com o formulário. Abra o arquivo `informacoes_visitante.html` e abaixo do elemento

`<h1 class="h3 mb-0 text-gray-800">` insira o seguinte trecho de código:

```
1 <!-- código acima omitido -->
2 <h1 class="h3 mb-0 text-gray-800">{{ nome_pagina }}</h1>
3
4 <!-- trecho de código a ser inserido -->
5 <div>
6     <a href="#" class="btn btn-success btn-icon-split btn-sm" data-toggle=
7         <span class="text">Autorizar entrada</span>
8
9         <span class="icon text-white-50">
10             <i class="fas fa-user-check"></i>
11         </span>
12     </a>
13 </div>
14 <!-- código abaixo omitido -->
```

O que estamos fazendo é adicionar um elemento `<div>` ao lado do título da página que possui um link (elemento `<a>`) para um elemento modal chamado de `#modal1`, que ainda vamos inserir na página. O template ficará parecido com isso:



Cabeçalho com título da página e botão verde escrito "Registrar visitante"

Feito isso, adicione também o código HTML do modal antes do fechamento da tag do elemento `<div class="container">` :

```
1 <div class="modal fade" id="modal1" tabindex="-1" role="dialog" aria-label="
2   <div class="modal-dialog" role="document">
3     <div class="modal-content">
4       <div class="modal-header">
5         <h5 class="modal-title" id="exampleModalLabel">Autorizar e
6
7         <button type="button" class="close" data-dismiss="modal" a
8           <span aria-hidden="true">&times;</span>
9         </button>
10      </div>
11
12      <div class="modal-body">
13        <form method="post">
14          {% csrf_token %}
15
16          <div class="form-group">
17            <label for="id_morador_responsavel" class="col-for
18              {% render_field form.morador_responsavel placehold
19            </div>
20
21            <div class="modal-footer">
22              <button type="button" class="btn btn-secondary" da
23              <button type="submit" class="btn btn-primary">Auto
24            </div>
25          </form>
26        </div>
27      </div>
28    </div>
29  </div>
```

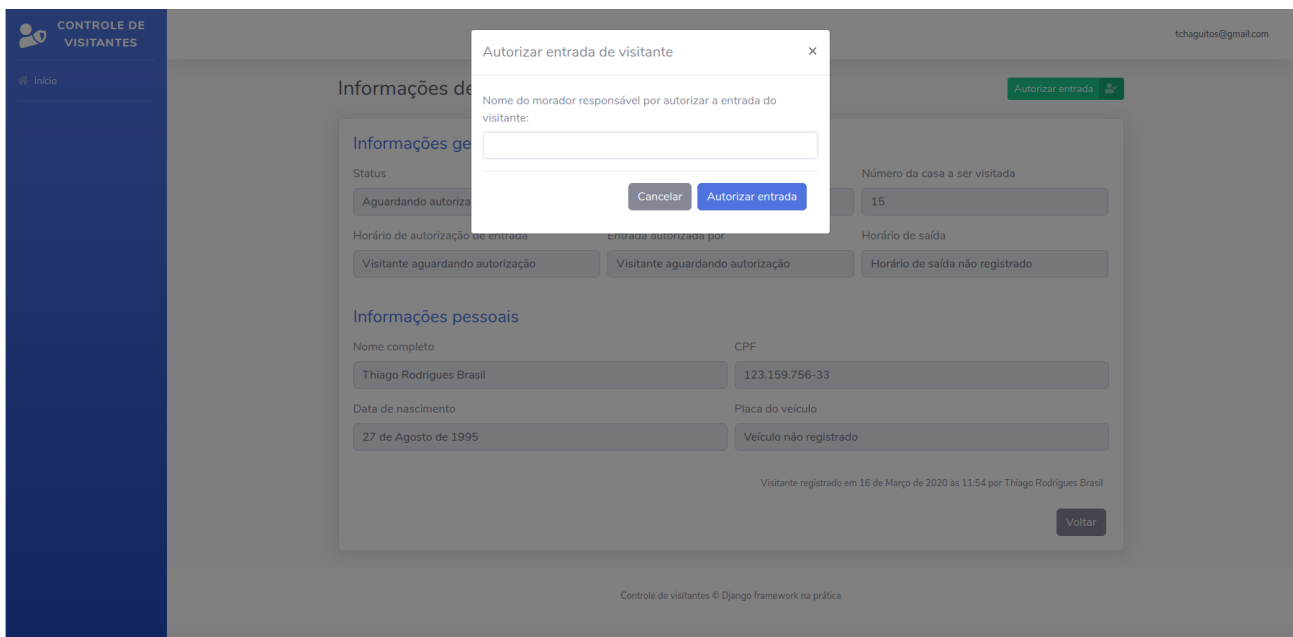
Conforme falado, esse modal deve exibir nosso formulário de cadastro de morador responsável, e é isso que estamos fazendo. Criamos a estrutura HTML para o formulário dentro do elemento `<div class="modal-body">` de forma bem parecida com que foi feito anteriormente para renderizar o campo, com a diferença que agora estamos acessando o

campo `morador_responsavel` do formulário diretamente para passá-lo para a tag `{% render_field %}`.

Já utilizamos a tag `{% render_field %}` anteriormente, quando renderizamos nosso formulário para registro de novos visitantes. A dinâmica utilizada aqui será a mesma, com a diferença que vamos acessar o campo do formulário diretamente (`form.morador_responsavel`).

i Note que o template baixado no capítulo anterior já possui a tag de importação do `django-widget-tweaks` (`{% load widget_tweaks %}`).

Feito isso, vamos agora visualizar as informações de um visitante qualquer e tentar autorizar sua entrada por meio do formulário que criamos. Note que, quando clicamos no botão, o modal com o formulário aparece na tela:



Template de informações de visitante ao fundo e alerta bom formulário para registro do nome do morador responsável

Atualizando os campos `horario_autorizacao` e `status`

diretamente

Ao testar o formulário e constatar que tudo ocorreu bem, você deve ter notado que, apesar do nome do morador responsável ter sido atualizado, os valores de `horario_autorizacao` e `status` continuaram os mesmos. Isso porque nosso formulário está atualizando apenas o campo `morador_responsavel`, e era isso que estávamos esperando, visto que colocamos apenas este campo na propriedade `fields` do formulário em questão. Mas como vamos atualizar estes outros campos?

Atualizando o status

Quando criamos o formulário para registro de visitantes, definimos o valor do atributo `registrado_por` diretamente e é o que faremos neste caso também. Para isso, vamos alterar a view para que seja possível setar esses valores diretamente.

```
1  # código acima omitido
2
3  if form.is_valid():
4      visitante = form.save(commit=False)
5
6      visitante.status = "EM_VISITA"
7
8      visitante.save()
9
10     messages.success(
11         request,
12         "Entrada de visitante autorizada com sucesso"
13     )
14
15     return redirect("index")
16
17 # código abaixo omitido
```

Dessa forma, já estamos definindo o valor que o status receberá caso o formulário seja válido e salvando o novo visitante, mas ainda precisamos registrar o horário em que essa autorização ocorreu, ou seja, o horário em que o formulário atualizou o atributo `morador_responsavel` e alterou o status para `EM_VISITA`.

Conhecendo o timezone do Django

O Python, por padrão, possui um módulo para trabalhar com datas e horas que é o `datetime`, mas por algumas questões referentes aos horários diferentes que são suportados, o Django nos recomenda a utilização do módulo `timezone`. Esse módulo nos fornece inúmeras ferramentas para trabalharmos com datas de forma bem facilitada já considerando a `timezone` em que nossa aplicação está contextualizada. O primeiro passo é importarmos o módulo na view.

```
1 from django.contrib import messages
2 from django.shortcuts import (
3     render, redirect, get_object_or_404
4 )
5
6 from visitantes.models import Visitante
7 from visitantes.forms import (
8     VisitanteForm, AutorizaVisitanteForm
9 )
10
11 from django.utils import timezone
12
13 # código abaixo omitido
```

O `timezone` possui um método chamado `now()` que nos retorna data e hora do momento em que a chamada ao método ocorreu. Sendo assim, caso o registro do visitante ocorra no dia 21 de agosto de 2020 às 15:00:00, o método `timezone.now()` retornaria exatamente essa data e hora. Dessa forma, tudo que precisamos fazer é igualar o atributo `horario_autorizacao` à chamada do método `timezone.now()`. Nossa view ficará assim:

```
1 # código acima omitido
2
3 if form.is_valid():
4     visitante = form.save(commit=False)
5
6     visitante.status = "EM_VISITA"
7     visitante.horario_autorizacao = timezone.now()
8
9     visitante.save()
10
```

```
11     messages.success(  
12         request,  
13         "Entrada de visitante autorizada com sucesso"  
14     )  
15  
16     return redirect("index")  
17  
18 # código abaixo omitido
```

Dessa forma, atualizamos o nome do morador responsável através do formulário e, caso a informação seja válida, atualizamos os atributos `status` e `horario_autorizacao` diretamente.

Capítulo 12

Criando função para finalizar visita

Agora que criamos a função para autorizar a entrada do visitante, precisamos também criar a função que finaliza a visita. Para a primeira, utilizamos o formulário

`AutorizaVisitanteForm` para atualiza o nome do morador responsável e definimos manualmente o status e o horário de autorização.

Desta vez, precisamos atualizar apenas o valor do atributo `horario_saida` e alterar o status para `FINALIZADO`, que é o status para quando o visitante deixa o condomínio. Sendo assim, vamos criar uma outra view que será responsável por receber um `id`, buscar um visitante com este `id` e atualizar essas informações. A view será um pouco parecida com a view `informacoes_visitante`.

Abaixo da função `informacoes_visitante` crie e função `finalizar_visita`:

```
1  # código acima omitido
2
3  def finalizar_visita(request, id):
4
5      if request.method == "POST":
6          visitante = get_object_or_404(Visitante, id=id)
7
8          visitante.status = "FINALIZADO"
9          visitante.horario_saida = timezone.now()
10
11         visitante.save()
12
13         messages.success(
14             request,
15             "Visita finalizada com sucesso"
16         )
17
18         return redirect("index")
```

A função `finalizar_visita` deverá receber um `id` como argumento e utilizar a função `get_object_or_404` para buscar o visitante do `id` que foi passado. Após isso vamos

atualizar os atributos `status` e `horario_saida` diretamente e salvar o visitante. A diferença aqui é que não utilizaremos um formulário e nossa view será acessada somente através do método `POST`. Todo o resto continuará bem parecido com as funções que já escrevemos antes.

Para garantir que as operações serão realizadas somente quando o método `POST` for utilizado, vamos escrever um `if` para certificar essa informação (`if request.method == "POST":`) e, caso seja verdadeira, vamos executar as operações necessárias. Note que, mais uma vez, estamos utilizando o método `timezone.now()` mas, desta vez, para o atributo `horario_saida`, e setando diretamente o `status` que agora deve receber o status `FINALIZADO`.

Criando URL

Assim como todas as outras funções de view que escrevemos, essa também será mapeada por meio de uma URL para que a gente possa acessá-la pelo navegador. Vamos para o nosso arquivos `urls.py` e criar essa nova URL.

A URL que irá mapear a função `finalizar_visita` será bem parecida com a URL `informacoes_visitante`, mas a diferença é que adicionaremos, após o `id`, o trecho `finalizar-visita/`. Com isso, conseguimos diferenciar qual função de view chamar para quando o usuário desejar apenas visualizar as informações de um visitante e para quando desejar finalizar uma visita. Nosso arquivo `urls.py` ficará assim:

```
1 from django.contrib import admin
2 from django.urls import path
3
4 from usuarios.views import index
5
6 from visitantes.views import (
7     registrar_visitante, informacoes_visitante,
8     finalizar_visita
9 )
10
11 urlpatterns = [
12     path("admin/", admin.site.urls),
```

```

13
14     path(
15         "",
16         index,
17         name="index",
18     ),
19
20     path(
21         "registrar-visitante/",
22         registrar_visitante,
23         name="registrar_visitante",
24     ),
25
26     path(
27         "visitantes/<int:id>/",
28         informacoes_visitante,
29         name="informacoes_visitante",
30     ),
31
32     path(
33         "visitantes/<int:id>/finalizar-visita/",
34         finalizar_visita,
35         name="finalizar_visita"
36     )
37 ]

```

Alterando template para exibir botão e modal para finalizar visita

Agora que temos a URL para onde devemos enviar uma requisição para sinalizar que queremos finalizar uma visita, vamos alterar as partes do template que vão possibilitar a interação do usuário com essa funcionalidade.

Assim como inserimos um botão para quando queremos autorizar a entrada de um visitante, vamos inserir um botão para quando quisermos finalizar a visita. Abra o template `informacoes_visitante.html` e insira o seguinte trecho de código abaixo do botão responsável por autorizar a entrada do visitante:

```

1 <a href="#" class="btn btn-warning btn-icon-split btn-sm" data-toggle="modal"
2   <span class="text">Finalizar visita</span>

```

```

3
4     <span class="icon text-white-50">
5         <i class="fas fa-door-open"></i>
6     </span>
7 </a>

```

O template ficará assim:

Informações de visitante

Autorizar entrada
Finalizar visita

Informações gerais

Status

Visita finalizada

Horário de chegada

16 de Março de 2020 às 11:54

Número da casa a ser visitada

15

Horário de autorização de entrada

30 de Março de 2020 às 21:35

Entrada autorizada por

Thiago

Horário de saída

6 de Abril de 2020 às 17:51

Informações pessoais

Nome completo

Thiago Rodrigues Brasil

CPF

123.159.756-33

Data de nascimento

27 de Agosto de 1995

Placa do veículo

Veículo não registrado

Visitante registrado em 16 de Março de 2020 às 11:54 por Thiago Rodrigues Brasil

Template de informações de visitante agora também com botão amarelo escrito "Finalizar visita"

Note que a estrutura é bem parecida com a que utilizamos no outro botão, mas quando observamos o atributo `data-target` podemos notar que agora ele é igual a `#modal2`. Isso porque vamos também criar um outro modal para ser exibido quando o usuário clicar no botão para finalizar uma visita. A função desse modal é obter a confirmação se é isso mesmo que o usuário deseja fazer.

Ainda no mesmo arquivo, mas agora ao final do arquivo, vamos colocar o seguinte trecho de código logo abaixo da estrutura HTML do primeiro modal:

```

1 <div class="modal fade" id="modal2" tabindex="-1" role="dialog" aria-label
2     <div class="modal-dialog" role="document">

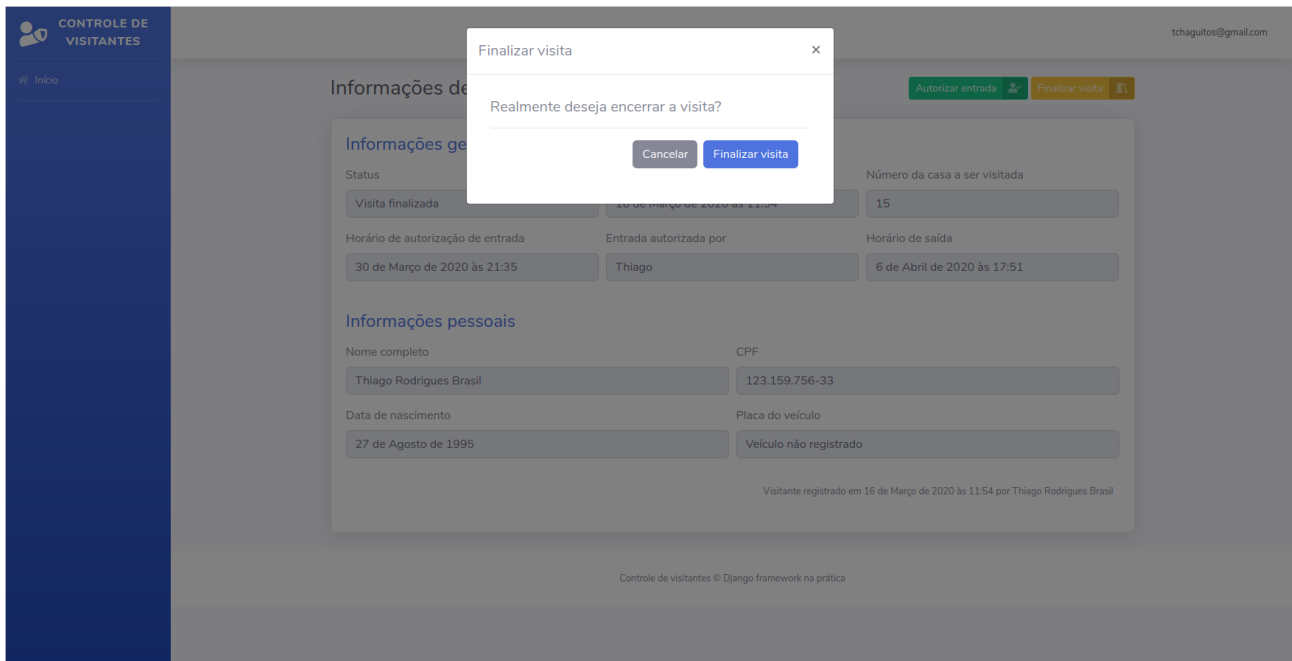
```

```

3      <div class="modal-content">
4          <div class="modal-header">
5              <h5 class="modal-title" id="exampleModalLabel">Finalizar v
6
7              <button type="button" class="close" data-dismiss="modal" a
8                  <span aria-hidden="true">&times;</span>
9              </button>
10         </div>
11
12         <div class="modal-body">
13             <div class="modal-body">
14                 <h5 class="mb-3">
15                     Realmente deseja encerrar a visita?
16                 </h5>
17
18                 <form method="post" action="{% url 'finalizar_visita'
19                     {% csrf_token %}
20
21                     <input hidden>
22
23                     <div class="modal-footer">
24                         <button type="button" class="btn btn-secondary
25                         <button type="submit" class="btn btn-primary">
26                     </div>
27                 </form>
28             </div>
29         </div>
30     </div>
31 </div>
32 </div>

```

Nosso segundo modal será exibido da seguinte maneira:



Template de informações de visitante ao fundo com alerta solicitando confirmação do usuário de que realmente deseja finalizar a visita

Esse segundo modal deverá exibir a mensagem "Realmente deseja encerrar a visita?" e conter um formulário que enviará uma requisição do tipo `POST` para a URL que criamos anteriormente. Esse formulário precisa ter apenas o campo renderizado pela tag `{% csrf_token %}` para identificar que as requisições podem ser aceitas pelo nosso servidor.

i Estamos enviando uma requisição do tipo `POST` para URL pois é recomendada a utilização deste método sempre que precisamos alterar informações em nosso banco de dados

O que muda tudo aqui é o atributo `action` do formulário HTML. Graças a ele podemos direcionar um formulário para uma URL diferente da que estamos, diferentemente de como fizemos com os outros formulários. Dessa forma, conseguimos enviar uma requisição do tipo `POST` para a URL `{% url 'finalizar_visita' id=visitante.id %}` com toda informação que precisamos para identificar o visitante a ser atualizado assim que o usuário clicar no botão "Finalizar visita" dom modal confirmando a ação.

Vai em frente e teste a nova funcionalidade implementada!

Prevenindo erros e operações desnecessárias

Nos passos anteriores, implementamos a funcionalidade que finaliza as visitas dentro da nossa dashboard. Você deve ter notado que mesmo quando a visita já foi finalizada, os botões são exibidos. Isso não é bom pois o usuário pode se confundir e clicar em um dos botões, alterando as informações existentes no nosso banco de dados.

Para prevenir que isso aconteça, vamos verificar o status do visitante e exibir os botões com base no valor desse status. Funcionará assim:

- Se o visitante estiver com status `AGUARDANDO`, vamos exibir o botão para **autorizar a entrada**
- Se o visitante estiver com status `EM_VISITA`, vamos exibir o botão para **finalizar a visita**
- E, finalmente, se o visitante estiver com status `FINALIZADO`, não vamos exibir botões

Exibição condicional de botões para autorizar entrada e finalizar visita

Para fazer isso, vamos utilizar a tag `{% if %}` para verificar o status do visitante e renderizar um botão por vez. Primeiro, vamos criar a instrução `if` para verificar se o status é `AGUARDANDO` e renderizar o botão para autorizar a entrada do visitante.

Utilizando a tag `{% if %}` vamos definir a condição `visitante.status == "AGUARDANDO"` para que o botão apareça. Isto é, o HTML referente ao botão só será renderizado no template caso o status do visitante seja `AGUARDANDO`. Nosso código ficará assim:

```
1 {% if visitante.status == "AGUARDANDO" %}
2     <a href="#" class="btn btn-success btn-icon-split btn-sm" data-toggle=
3         <span class="text">Autorizar entrada</span>
4
5         <span class="icon text-white-50">
6             <i class="fas fa-user-check"></i>
7         </span>
8     </a>
9 {% endif %}
```

Abaixo do código que escrevemos, vamos criar uma outra condição com a tag `{% if %}` . Dessa vez, queremos verificar se o status é o `EM_VISITA` . O código completo ficará assim:

```
1 <div class="d-sm-flex align-items-center justify-content-between mb-4">
2   <h1 class="h3 mb-0 text-gray-800">{{ nome_pagina }}</h1>
3
4   <div class="">
5     {% if visitante.status == "AGUARDANDO" %}
6       <a href="#" class="btn btn-success btn-icon-split btn-sm" data-
7         <span class="text">Autorizar entrada</span>
8
9         <span class="icon text-white-50">
10           <i class="fas fa-user-check"></i>
11         </span>
12       </a>
13     {% endif %}
14
15     {% if visitante.status == "EM_VISITA" %}
16       <a href="#" class="btn btn-warning btn-icon-split btn-sm" data-
17         <span class="text">Finalizar visita</span>
18
19         <span class="icon text-white-50">
20           <i class="fas fa-door-open"></i>
21         </span>
22       </a>
23     {% endif %}
24   </div>
25 </div>
```

Se você acessar a página de informação de algum visitante, notará que os botões não estão aparecendo juntos mais. Além disso, se você for na página de informação de um visitante que já deixou as dependências do condomínio, isto é, finalizou sua visita, notará que nenhum botão é exibido.

Dessa forma conseguimos evitar que operações desnecessárias sejam realizadas e que as informações do nosso banco de dados sejam alteradas de forma indevida.

Bloqueando o acesso à URL por métodos diferentes do POST

Ao contrário das outras funções que escrevemos, a função `finalizar_visita` não poderá ser acessada através do método `GET`. O método `GET` é utilizado por uma requisição sempre que precisamos buscar informações em um servidor, como é o caso nas outras funções (estamos buscando o template e todo o contexto relacionado a ele antes de enviar informações para o usuário). Se você notar as funções `registrar_visitante` e `informacoes_visitante`, vai perceber que definimos algumas variáveis fora da instrução `if` que verifica se o método utilizado é o `POST`. Isso porque precisamos dessas variáveis quando o usuário acessa a página, como é o caso do formulário que deverá ser exibido mesmo que uma requisição `POST` não seja enviada.

Para garantir que nossa view possa ser acessada somente pelo método `POST`, vamos utilizar a classe `HttpResponseNotAllowed` para nos ajudar. Ela é quem vai cuidar de toda parte de bloquear o acesso via método `GET` e retornar uma mensagem para o usuário quando isso ocorrer. Antes de tudo, precisamos importá-la em nosso arquivo `views.py` do aplicativo visitantes:

```
1 from django.contrib import messages
2 from django.shortcuts import (
3     render, redirect, get_object_or_404
4 )
5
6 from django.http import HttpResponseNotAllowed
7
8 from visitantes.models import Visitante
9 from visitantes.forms import (
10     VisitanteForm, AutorizaVisitanteForm
11 )
12
13 from django.utils import timezone
14
15 # código abaixo omitido
```

Feito isso, tudo que precisamos fazer é utilizar a instrução `else` e retornar a classe `HttpResponseNotAllowed` passando uma lista com os métodos permitidos e uma mensagem a ser exibida caso o método utilizado pela requisição seja diferente. Nosso código ficará assim:


```
1 def finalizar_visita(request, id):
2
3     if request.method == "POST":
4         visitante = get_object_or_404(Visitante, id=id)
5
6         visitante.status = "FINALIZADO"
7         visitante.horario_saida = timezone.now()
8
9         visitante.save()
10
11         messages.success(
12             request,
13             "Visita finalizada com sucesso"
14         )
15
16         return redirect("index")
17
18     else:
19         return HttpResponseNotAllowed(
20             ["POST"],
21             "Método não permitido"
22         )
```

Com isso, permitimos que a view seja acessada somente pelo método `POST` e que, quando outro método for utilizado, a view vai retornar o código `HTTP 405` e exibir a mensagem "Método não permitido".

Se você quiser, teste em seu navegador: <http://127.0.0.1:8000/visitantes/4/finalizar-visita/>.

Capítulo 13

Implementando melhorias em nossos templates

Olha só que legal: finalizamos as funcionalidades de maior valor da nossa dashboard. Os porteiros já podem registrar visitantes, autorizar a entrada deles após contato com um morador e ainda sinalizar que a visita foi encerrada. Bem bacana, não?

Agora que temos as principais funcionalidades prontas, vamos nos concentrar em melhorar a experiência de utilização da dashboard e ainda melhorar a estrutura do nosso projeto, de modo que seja mais fácil realizar futuras manutenções.

Começaremos com algumas alterações nos templates que visam melhorar a experiência de utilização da dashboard.

Exibindo botão com função de "voltar" e "cancelar" em páginas de informações e registro de visitante

Nossa primeira melhoria será inserir os botões com as ações **cancelar** e **voltar** nas páginas de registro de visitante e informações de visitante. Vamos primeiro abrir o arquivo

`registrar_visitante.html` e procurar pelo seguinte trecho de código:

```
1 <div class="text-right">
2   <button class="btn btn-primary" type="submit">
3     <span class="text">Registrar visitante</span>
4   </button>
5 </div>
```


Acima do elemento `<button class="btn btn-primary" type="submit">` vamos inserir um link para a página inicial da nossa dashboard com o texto "Cancelar", aproveitando algumas classes do Bootstrap para que o link tenha a aparência de um botão. O código ficará assim:

```
1 <div class="text-right">
```

```

2     <a href="{% url 'index' %}" class="btn btn-secondary text-white" type=
3         <span class="text">Cancelar</span>
4     </a>
5
6     <button class="btn btn-primary" type="submit">
7         <span class="text">Registrar visitante</span>
8     </button>
9 </div>

```

 Como a única maneira que podemos chegar até a página de registro de um novo visitante é pelo início da dashboard, faz sentido utilizarmos apenas um link fixo para a home da dashboard

O template `informacoes_visitante.html` já possui o botão com a ação **voltar** visível, mas não temos um link para onde o botão deve enviar o usuário. Vamos procurar pelo seguinte trecho de código e inserir o link para a URL `index` em seu atributo `href` :

```

1 <div class="mr-1 text-right">
2     <a href="#" class="btn btn-secondary text-white" type="button">
3         <span class="text">Voltar</span>
4     </a>
5 </div>

```

O código ficará assim:

```

1 <div class="mr-1 text-right">
2     <a href="{% url 'index' %}" class="btn btn-secondary text-white" type=
3         <span class="text">Voltar</span>
4     </a>
5 </div>

```

Melhorando a exibição do CPF do visitante

Uma outra melhoria interessante seria na exibição do CPF do visitante. Estamos exibindo os números todos sem nenhuma separação, como geralmente o número de CPF é apresentado. Aprendemos que é possível criar métodos nas classes modelo para alterar comportamentos e até já criamos métodos que utilizamos para melhorar a exibição de alguns atributos. Agora vamos criar o método `get_cpf` que deverá retornar o CPF do visitante já formatado com pontos e traço.

Vamos abrir o arquivo `models.py` e abaixo do método `get_placa_veiculo` vamos criar o método `get_cpf` que, por enquanto, irá retornar o CPF caso o mesmo exista. O código ficará assim:

```
1  # código acima omitido
2
3  def get_placa_veiculo(self):
4      if self.placa_veiculo:
5          return self.placa_veiculo
6
7      return "Veículo não registrado"
8
9  def get_cpf(self):
10     if self.cpf:
11         return self.cpf
12
13     return "CPF não registrado"
14
15  # código abaixo omitido
```

Conhecendo o f-strings do Python

Precisamos recortar a string algumas vezes, separar esses recortes e depois montar uma outra string com cada parte obedecendo aos pontos e ao traço do formato padrão para exibição de CPF (`xxx.xxx.xxx-xx`). Esse processo pode parecer um pouco complicado mas não é.

Antes de tudo, vamos recortar as partes que compõem o CPF. Vamos utilizar os índices da variável `cpf` (que será igual ao CPF do visitante) para recortar cada parte, os intervalos (`[0:3]` , `[3:6]` , `[6:9]` e `[9:]`). Além disso, vamos também criar variáveis para guardar as partes do CPF:

```

1  def get_cpf(self):
2      if self.cpf:
3          cpf = str(self.cpf)
4
5          cpf_parte_um = cpf[0:3]
6          cpf_parte_dois = cpf[3:6]
7          cpf_parte_tres = cpf[6:9]
8          cpf_parte_quatro = cpf[9:]
9
10         # código abaixo omitido

```

Com as quatro partes do CPF recortadas e guardadas em variáveis, temos agora que colocá-las em ordem no formato padrão do CPF. Para nos ajudar com isso, vamos utilizar um recurso do Python chamado strings literais ou `f-strings`.

Uma f-string (ou string literal) é toda cadeia de caracteres prefixada por `f` ou `F`, onde pode conter também campos para substituição de variáveis ou expressões, delimitadas por chaves `{}`. Utilizando uma string literal, podemos criar a string já formatada e inserir os intervalos recortados do CPF na ordem por meio dos campos de substituição. Abaixo temos a variável que vamos retornar no método, onde cada par de chaves `{}` será substituído por um intervalo da string que representa o CPF do visitante:

```

cpf_formatado = f"{}.{}.{}-{}"

```

Agora vamos inserir as variáveis que representam as partes do CPF do visitante na ordem e dentro das chaves `{}` da variável `cpf_formatado`, nossa string literal. E já que nosso objetivo é retornar o CPF já formatado, vamos retornar essa variável. O método ficará assim:

```

1  def get_cpf(self):
2      if self.cpf:
3          cpf = self.cpf
4
5          cpf_parte_um = cpf[0:3]
6          cpf_parte_dois = cpf[3:6]
7          cpf_parte_tres = cpf[6:9]

```

```

8         cpf_parte_quatro = cpf[9:]
9
10        cpf_formatado = f"{cpf_parte_um}.{cpf_parte_dois}.{cpf_parte_tres}"
11
12        return cpf_formatado
13
14    return "CPF não registrado"

```

Feito isso, temos agora que substituir os acessos ao atributo `cpf` do modelo pela chamada ao método `get_cpf`. Primeiro no template `index.html` e depois no `informacoes_visitante.html`.

```

1  <!-- código acima omitido -->
2
3  <tbody>
4      {% for visitante in pagina_obj %}
5          <tr>
6              <td>{{ visitante.nome_completo }}</td>
7              <td>{{ visitante.get_cpf }}</td>
8
9              <!-- código abaixo omitido -->

```

E agora no `informacoes_visitante.html`:

```

1  <!-- código acima omitido -->
2  <div class="form-group col-md-6">
3      <label>CPF</label>
4      <input type="text" class="form-control" value="{{ visitante.get_cpf }}"
5  </div>
6  <!-- código abaixo omitido -->

```

Utilizando método para exibir o status do visitante

Ao contrário de alguns atributos que tivemos que criar métodos para exibi-los de maneira personalizada, para o atributo `status` isso não é necessário.

Quando definimos as opções de escolha para o `status`, definimos uma string para ser salva no banco de dados e uma para funcionar como `label` da string salva, como se fosse um nome descritivo mesmo. Por baixo dos panos o Django cria um método para exibir o label que nós definimos, bastando apenas que a gente utilize exatamente como fizemos com os outros método. Por padrão, o nome do método é `get_nomeatributo_display` que, para o nosso caso, é o `get_status_display`.

Agora que sabemos como utilizar o método, vamos alterar alguns templates para que a gente exiba o status do visitante juntamente das informações do mesmo. Primeiro vamos abrir o arquivo `informacoes_visitante.html` e procurar pelo seguinte trecho de código:

```
1 <div class="form-row">
2   <div class="form-group col-md-6">
3     <label>Horário de chegada</label>
4     <input type="text" class="form-control" value="{{ visitante.horari
5   </div>
6
7   <div class="form-group col-md-6">
8     <label>Número da casa a ser visitada</label>
9     <input type="text" class="form-control" value="{{ visitante.numero
10  </div>
11 </div>
```

O trecho de código acima é o responsável por renderizar a primeira linha das informações gerais a respeito da visita no template em questão. Vamos alterá-lo para exibir ao lado do número da casa, o status em que o visitante se encontra. Para fazer isso, primeiro vamos alterar a classe `col-md-6` presente nos elementos `<div class="form-group col-md-6">` para `col-md-4`. Essa é uma classe de estilo do Bootstrap e nos ajuda a organizar as colunas de um template de modo que se dividam na tela. Caso você queira saber mais sobre o sistema de grid do Bootstrap, pode acessar [esse link](#).

Feito isso, o que vamos fazer é inserir mais um elemento

`<div class="form-group col-md-4">` abaixo do que exiba o número da casa, desta vez para exibir o status do visitante. O código ficará assim:

```
1 <div class="form-row">
2   <div class="form-group col-md-4">
```

```

3         <label>Horário de chegada</label>
4         <input type="text" class="form-control" value="{{ visitante.horari
5     </div>
6
7     <div class="form-group col-md-4">
8         <label>Número da casa a ser visitada</label>
9         <input type="text" class="form-control" value="{{ visitante.numero
10    </div>
11
12    <div class="form-group col-md-4">
13        <label>Status</label>
14        <input type="text" class="form-control" value="{{ visitante.get_st
15    </div>
16 </div>

```

Agora só precisamos inserir mais uma coluna na tabela do template `index.html` para exibir o status do usuário logo ali na página inicial. Primeiro vamos procurar pelo elemento `<thead>` e, abaixo do elemento `<th>` com o texto "Horário de chegada", vamos inserir um elemento `<th>` com texto "Status". O código ficará assim:

```

1 <thead>
2     <th>Nome</th>
3     <th>CPF</th>
4     <th>Horário de chegada</th>
5     <th>Status</th>
6     <th>Horário da autorização</th>
7     <th>Autorizado por</th>
8     <th>Mais informações</th>
9 </thead>

```

Agora, claro, vamos adicionar também uma linha que será responsável por exibir o status utilizando o método `get_status_display`. Ficará assim:

```

1 {% for visitante in pagina_obj %}
2     <tr>
3         <td>{{ visitante.nome_completo }}</td>
4         <td>{{ visitante.get_cpf }}</td>
5         <td>{{ visitante.horario_chegada }}</td>
6         <td>{{ visitante.get_status_display }}</td>
7         <td>{{ visitante.get_horario_autorizacao }}</td>

```



```
8         <td>{{ visitante.get_morador_responsavel }}</td>
9         <td>
10             <a href="{% url 'informacoes_visitante' id=visitante.id %}">
11                 Ver informações
12             </a>
13         </td>
14     </tr>
15 {% endfor %}
```

Implementando melhorias na estrutura do nosso projeto

Uma alteração interessante que vamos implementar é criar uma pasta de nome `apps` na raiz do nosso projeto para agrupar todos os nossos aplicativos. Vamos começar alterando o arquivo `settings.py`. Vamos importar o módulo `sys` e depois adicionar a pasta `apps` ao projeto.

```
1 import os
2 import sys
3
4 # código abaixo omitido
```

Feito isso tudo que precisamos fazer é adicionar a seguinte linha de código abaixo da variável `ALLOWED_HOSTS`:

```
1 sys.path.append(
2     os.path.join(BASE_DIR, "apps")
3 )
```

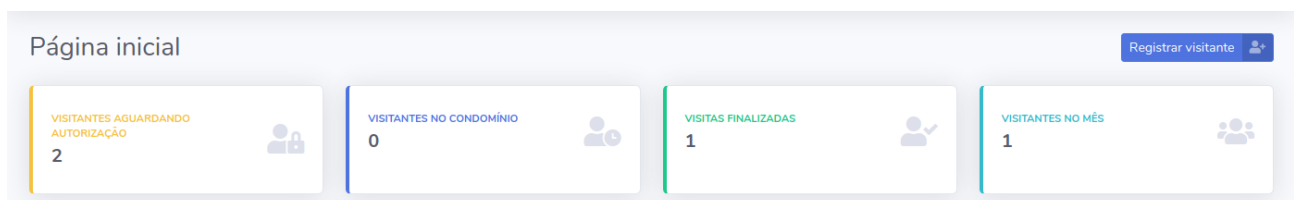
Agora, vamos criar a pasta **apps** e mover as pastas dos nossos aplicativos para ela.

Capítulo 14


Criando aplicativo para administrar informações da dashboard

No capítulo anterior nós começamos a implementar algumas melhorias visando uma melhor experiência de utilização da nossa dashboard e ainda atuamos de forma a melhorar a estrutura do projeto para facilitar futuras manutenções. Seguindo nesse mesmo caminho, vamos criar um aplicativo chamado **dashboard** para administrar melhor as informações relacionadas principalmente à página inicial da dashboard. É importante também a gente lembrar que os aplicativos de um projeto Django devem dividir responsabilidades e cada um deles ter um único objetivo.

Apesar da gente ter finalizado as principais funcionalidades, ainda precisamos buscar alguns números para que sejam mostrados na página inicial. Se você observar o template, vai perceber que existem elementos que nos sugerem que devemos exibir o número de visitantes de cada status e quantos visitantes foram registrados no mês atual. Queremos fazer algo desse tipo, mas com dados dinâmicos:



Captura de tela realizada a partir do template inicial da dashboard onde foca apenas nos blocos que exibem o número de visitantes aguardando autorização, em visita, visita finalizada e o total de visitantes registrados no mês


 Todas essas informações podem ser tiradas a partir da queryset que busca todos os visitantes no banco de dados. Em breve vamos aprender como podemos fazer isso e tornar os dados na nossa dashboard dinâmicos.

Para começar vamos criar o aplicativo utilizando o `manage.py` :

```
(env)$ python manage.py startapp dashboard
```

E depois adicionar o novo aplicativo ao arquivo de configurações, o `settings.py` :

```
1  INSTALLED_APPS += [  
2      "usuarios",  
3      "porteiros",  
4      "visitantes",  
5      "dashboard",  
6  ]
```

 Não vamos executar as operações `makemigrations` e `migrate` pois não fizemos nenhuma alteração relacionada com o banco de dados. Estamos apenas acessando e buscando os dados já existentes.

Migrando view "index" para aplicativo dashboard

Agora que nós temos um aplicativo para gerenciar as informações da nossa dashboard, vamos migrar a função de view `index` para o aplicativo **dashboard**. Para isso, vamos copiar o código do arquivo `usuarios/views.py` para `dashboard/views.py`. O arquivo `views.py` do aplicativo usuários (`usuarios/views.py`) ficará vazio e o arquivo `views.py` do aplicativo dashboard (`dashboard/views.py`) ficará assim:

```
1  from django.shortcuts import render  
2  from visitantes.models import Visitante  
3  
4  def index(request):  
5  
6      todos_visitantes = Visitante.objects.all()  
7  
8      context = {
```

```
9         "nome_pagina": "Início da dashboard",
10        "todos_visitantes": todos_visitantes,
11    }
12
13    return render(request, "index.html", context)
```

Para finalizar a migração da nossa view, vamos também alterar o arquivo `urls.py`. Ao invés de `usuarios.views` vamos importar a nossa função de `dashboard.views`. O arquivo ficará assim:

```
1  from django.contrib import admin
2  from django.urls import path
3
4  from dashboard.views import index
5
6  from visitantes.views import (
7      registrar_visitante,
8      informacoes_visitante,
9      finalizar_visita,
10 )
11
12 urlpatterns = [
13     path("admin/", admin.site.urls),
14
15     path(
16         "",
17         index,
18         name="index"
19     ),
20
21     # código abaixo omitido
22 ]
```

Conhecendo o método filter das querysets

Agora que migramos a view para o aplicativo dashboard, vamos conhecer métodos para filtrar os visitantes de modo que a gente consiga buscar e exibir os dados que precisamos: o número de visitantes em cada status e o número total de visitantes registrados no mês atual.

O primeiro método das querysets que vamos conhecer é o método `filter()`. Ele nos ajuda a filtrar os resultados de uma queryset. Nos capítulos anteriores aprendemos que toda busca no banco de dados retorna uma queryset, um tipo específico do Django, e que podemos manipular esses resultados.

Na view que estamos trabalhando já existe uma queryset, que é a variável `todos_visitantes`. Ela guarda a lista de todos os visitantes existentes em nosso banco de dados, o que precisamos fazer é filtrar essa lista de modo que seja possível classificar os visitantes por status. Ou seja, precisamos ter uma lista de visitantes com status aguardando, outra de visitantes com status em visita e outra com as visitas finalizadas.

Filtrando nossos visitantes por status

O primeiro passo será criar uma variável para receber os resultados. Vamos utilizar o nome `visitantes_aguardando` pois por agora queremos apenas os visitantes que estão com o status `AGUARDANDO`. Para fazer isso vamos utilizar o método `filter()` na variável `todos_visitantes` passando a condição `status="AGUARDANDO"` como argumento para o método. Isto é, vamos filtrar da queryset `todos_visitantes` apenas os visitantes que estão com `status` igual a `AGUARDANDO`.

```
1 def index(request):
2
3     todos_visitantes = Visitante.objects.all()
4
5     visitantes_aguardando = todos_visitantes.filter(
6         status="AGUARDANDO"
7     )
8
9     context = {
10         "nome_pagina": "Início da dashboard",
11         "todos_visitantes": todos_visitantes,
12     }
13
14     return render(request, "index.html", context)
```

Vamos fazer isso com todos os outros status para que possamos ter uma lista de visitantes para cada status e passar as variáveis criadas para o contexto da função.

```

1  def index(request):
2
3      todos_visitantes = Visitante.objects.all()
4
5      # filtrando os visitantes por status
6      visitantes_aguardando = todos_visitantes.filter(
7          status="AGUARDANDO"
8      )
9
10     visitantes_em_visita = todos_visitantes.filter(
11         status="EM_VISITA"
12     )
13
14     visitantes_finalizado = todos_visitantes.filter(
15         status="FINALIZADO"
16     )
17
18     context = {
19         "nome_pagina": "Início da dashboard",
20         "todos_visitantes": todos_visitantes,
21         "visitantes_aguardando": visitantes_aguardando,
22         "visitantes_em_visita": visitantes_em_visita,
23         "visitantes_finalizado": visitantes_finalizado,
24     }
25
26     return render(request, "index.html", context)

```

Contando os resultados de uma queryset

Agora que nós já filtramos os visitantes por status, precisamos contar quantos registros existem em cada queryset, certo? É isso que o método `count()` faz por nós. Tudo que precisamos fazer é utilizá-lo nas querysets `visitantes_aguardando`, `visitantes_em_visita` e `visitantes_finalizado`. Podemos fazer isso no contexto mesmo:

```

1  context = {
2      "nome_pagina": "Início da dashboard",
3      "todos_visitantes": todos_visitantes,
4      "visitantes_aguardando": visitantes_aguardando.count(),
5      "visitantes_em_visita": visitantes_em_visita.count(),
6      "visitantes_finalizado": visitantes_finalizado.count(),
7  }

```

Feito isso, agora nós vamos exibir essas variáveis no template. Vamos abrir o template `index.html` e utilizar a sintaxe para exibição de variáveis. O trecho de código ficará assim:

```
1  <div class="row">
2    <div class="col-xl-3 col-md-6 mb-4">
3      <div class="card border-left-warning shadow h-100 py-2">
4        <div class="card-body">
5          <div class="row no-gutters align-items-center">
6            <div class="col mr-2">
7              <div class="text-xs font-weight-bold text-warning">
8                <div class="h5 mb-0 font-weight-bold text-gray-800">
9              </div>
10           </div>
11           <div class="col-auto">
12             <i class="fas fa-user-lock fa-2x text-gray-300"></i>
13           </div>
14         </div>
15       </div>
16     </div>
17   </div>
18
19   <div class="col-xl-3 col-md-6 mb-4">
20     <div class="card border-left-primary shadow h-100 py-2">
21       <div class="card-body">
22         <div class="row no-gutters align-items-center">
23           <div class="col mr-2">
24             <div class="text-xs font-weight-bold text-primary">
25               <div class="h5 mb-0 font-weight-bold text-gray-800">
26             </div>
27           </div>
28           <div class="col-auto">
29             <i class="fas fa-user-clock fa-2x text-gray-300"></i>
30           </div>
31         </div>
32       </div>
33     </div>
34   </div>
35
36   <div class="col-xl-3 col-md-6 mb-4">
37     <div class="card border-left-success shadow h-100 py-2">
38       <div class="card-body">
39         <div class="row no-gutters align-items-center">
40           <div class="col mr-2">
41             <div class="text-xs font-weight-bold text-success">
42               <div class="h5 mb-0 font-weight-bold text-gray-800">
```

```

43         </div>
44         <div class="col-auto">
45             <i class="fas fa-user-check fa-2x text-gray-300"><
46         </div>
47     </div>
48 </div>
49 </div>
50 </div>
51
52 <div class="col-xl-3 col-md-6 mb-4">
53     <div class="card border-left-info shadow h-100 py-2">
54         <div class="card-body">
55             <div class="row no-gutters align-items-center">
56                 <div class="col mr-2">
57                     <div class="text-xs font-weight-bold text-info tex
58                     <div class="h5 mb-0 font-weight-bold text-gray-800
59                 </div>
60                 <div class="col-auto">
61                     <i class="fas fa-users fa-2x text-gray-300"></i>
62                 </div>
63             </div>
64         </div>
65     </div>
66 </div>
67 </div>

```

Se você atualizar a página inicial da dashboard, vai observar que agora os números estão dinâmicos e aparecendo conforme os registros do nosso banco de dados. Foi bem tranquilo resolver essa, certo? Então vamos para o próximo desafio!

Capítulo 15

Aprendendo a filtrar nossos visitantes por data

No capítulo anterior nós aprendemos um pouco sobre os métodos `filter()` e `count()` e até já filtramos os visitantes por status. Agora o que precisamos fazer é filtrar os visitantes com base na data em que o mesmo foi cadastrado para que depois seja possível filtramos esses registros por um mês em específico, que é o nosso objetivo.

Para fazer isso, vamos novamente utilizar o método `filter()`, mas dessa vez passando uma condição diferente, pois agora o que queremos é filtrar pela data de chegada. Por hora, vamos somente passar uma data para que o Django filtre somente os visitantes registrados nessa data em específico

Conhecendo o field lookups da Queryset API

Antes de filtrar o horário de chegada pela data, precisamos conhecer os `field lookups`. Eles são filtros que nos possibilitam especificar algumas condições para nossos campos ou acessar propriedades. Por exemplo, nosso campo `horario_chegada` também recebe uma data, pois é do tipo `DateTimeField`. Sendo assim, podemos acessar somente a data existente no valor do campo.

Podemos utilizá-los com alguns métodos das querysets e entre eles o método `filter()`. Nós vamos utilizar `__` após o nome do atributo que passamos como condição para o método `filter()`. Se a gente quiser filtrar os visitantes pela data contida no atributo `horario_chegada`, podemos fazer isso:

```
1 visitantes_mes = todos_visitantes.filter(  
2     horario_chegada__date="2020-04-04"  
3 )
```

Nesse caso, estamos buscando apenas os visitantes que foram registrados no dia 04 de maio de 2020. Existem vários outros *lookups* e cada tipo de campo possui os seus.

Filtrando apenas os registros do mês atual

Agora que sabemos que é possível filtrar por propriedades contidas em alguns atributos e até filtramos os visitantes de um dia específico, vamos conhecer um outro `field lookup`, o `month`. Assim como nós podemos buscar somente pela data, que está contida no `DateTimeField`, podemos também buscar pelo mês, que é o que esse *lookup* faz. Dessa vez vamos buscar todos os visitantes registrados no mês de maio (mês 05).

```
1 visitantes_mes = todos_visitantes.filter(  
2     horario_chegada__month="05"  
3 )
```

Utilizando o timezone para descobrir o mês atual

Descobrimos como filtrar nossos visitantes pelo mês em que foram registrados, mas ainda precisamos fazer com que o método filtre os visitantes do mês atual de forma automática, isto é, que o mês atual seja reconhecido e passado para o método `filter()` como uma variável.

Para fazer isso vamos utilizar um velho conhecido, o `timezone.now()`. Esse método retorna o valor referente a data e hora em que foi invocado e, a partir dessa data e hora, podemos buscar o mês. Para importar o módulo `timezone` basta utilizar o seguinte código:

```
from django.utils import timezone
```

Feito isso, vamos criar a variável `hora_atual` sendo igual ao método `timezone.now()` e depois acessar a propriedade `month` da variável `hora_atual` para passá-la à variável

que vamos criar de nome `mes_atual` .

```
1 # filtrando visitantes por data (mês atual)
2 hora_atual = timezone.now()
3 mes_atual = hora_atual.month
4
5 visitantes_mes = todos_visitantes.filter(
6     horario_chegada__month=mes_atual
7 )
```

Agora temos o valor referente ao mês atual numa variável e podemos passá-la como valor para a condição do método `filter()` . Com isso nós já temos uma lista com os visitantes que foram registrados no mês atual. Vamos agora passar essa variável no contexto da função e depois exibi-la no template.

```
1 context = {
2     "nome_pagina": "Página inicial",
3     "visitantes": visitantes,
4     "visitantes_aguando": visitantes_aguando.count(),
5     "visitantes_em_visita": visitantes_em_visita.count(),
6     "visitantes_finalizado": visitantes_finalizado.count(),
7     "visitantes_mes": visitantes_mes.count(),
8 }
```

Agora vamos voltar para o arquivo `index.html` e alterar o último trecho, onde vamos exibir a quantidade de visitantes registrados no mês atual:

```
1 <div class="col-xl-3 col-md-6 mb-4">
2     <div class="card border-left-info shadow h-100 py-2">
3         <div class="card-body">
4             <div class="row no-gutters align-items-center">
5                 <div class="col mr-2">
6                     <div class="text-xs font-weight-bold text-info text-up">
7                         <div class="h5 mb-0 font-weight-bold text-gray-800">{{
8                     </div>
9
10                    <div class="col-auto">
11                        <i class="fas fa-users fa-2x text-gray-300"></i>
```

```
12         </div>
13     </div>
14 </div>
15 </div>
16 </div>
```

Ao voltar para a página inicial da dashboard, o número de visitantes registrados no mês já estará sendo exibido.

Ordenando nossa lista de visitantes por horário de chegada

O último passo antes da gente finalizar mais um capítulo é ordenar a lista de visitantes recentes por horário de chegada. Isto é, queremos que os registros de visitantes sigam uma certa ordem e essa ordem seja baseada no horário de chegada, de forma que os registros mais recentes fiquem sempre no topo.

Para isso vamos utilizar o método `order_by()` que funciona de forma parecida com o `filter()`, com a diferença que precisamos passar somente o nome do atributo que queremos utilizar como parâmetro. Vamos utilizá-lo em substituição ao método `all()`:

```
1 todos_visitantes = Visitante.objects.order_by(
2     "-horario_chegada"
3 )
```

Note que estamos utilizando o operador (`-`) antes do nome do atributo `horario_chegada`. O operador utilizado dessa forma faz com que os registros sejam listados em ordem descendente que, para data, tem o comportamento de ordenar do registro mais recente para o mais antigo.

Capítulo 16

Bloqueando o acesso para usuários não autenticados nas nossas views

Estamos chegando na reta final do nosso projeto. Todos os requisitos descritos estão implementados e agora nós vamos cuidar de alguns requisitos que são chamados não funcionais. Esse tipo de requisito raramente é descrito mas está presente em praticamente todas as aplicações web existentes: telas de login, telas de logout e o bloqueio do acesso não autenticado a páginas com informações delicadas, por exemplo.

Daqui pra frente cuidaremos dos requisitos mencionados e vamos conhecer alguns recursos bem interessantes do Django que vão nos ajudar a poupar bastante tempo. O Django possui embutido em suas funcionalidades, alguns módulos voltados para a criação, administração e autenticação de usuários, conhecido por sistema de autenticação do Django. O comando `createsuperuser`, por exemplo, faz parte de suas funcionalidades.

Pense na dashboard que estamos criando: as informações que estamos exibindo ali são confidenciais e não devem ficar expostas para que qualquer um possa encontrá-las na web. Desta forma, nós precisamos bloquear as URLs para que somente usuários autenticados com e-mail e senha possam ter acesso e visualizar essas informações.

Conhecendo o decorator `login_required`

Para bloquear o acesso não autenticado às views, o Django nos fornece um caminho rápido para que a gente consiga implementar essa funcionalidade. Nós vamos utilizar o decorator `login_required` para tornar nossas views acessíveis somente após autenticação. Caso o usuário não esteja autenticado, não poderá acessar a view e será direcionado para uma tela de login.

Um decorator nada mais é que um método que envolve e modifica comportamentos de uma função. É isso que estamos fazendo: pedindo que o decorator `login_required` faça a função ser acessível somente após autenticação e, com isso, estamos alterando o comportamento da função.

Para utilizar esse decorator vamos primeiro importá-lo no topo do arquivo `views.py` do aplicativo dashboard:

```
1 from django.shortcuts import render
2 from django.contrib.auth.decorators import login_required
3
4 # código abaixo omitido
```

Agora tudo que precisamos fazer é colocar um `@` antes do nome do decorator em cima da função `index`. O código ficará assim:

```
1 @login_required
2 def index(request):
3
4     todos_visitantes = Visitante.objects.order_by(
5         "-horario_chegada"
6     )
7
8     # filtrando os visitantes por status
9     visitantes_aguardando = todos_visitantes.filter(
10         status="AGUARDANDO"
11     )
12
13     visitantes_em_visita = todos_visitantes.filter(
14         status="EM_VISITA"
15     )
16
17     visitantes_finalizado = todos_visitantes.filter(
18         status="FINALIZADO"
19     )
20
21     # filtrando visitantes por data (mês atual)
22     hora_atual = timezone.now()
23     mes_atual = hora_atual.month
24
25     visitantes_mes = todos_visitantes.filter(
26         horario_chegada__month=mes_atual
27     )
28
29     context = {
30         "nome_pagina": "Início da dashboard",
31         "todos_visitantes": todos_visitantes,
32         "visitantes_aguardando": visitantes_aguardando.count(),
```

```

33         "visitantes_em_visita": visitantes_em_visita.count(),
34         "visitantes_finalizado": visitantes_finalizado.count(),
35         "visitantes_mes": visitantes_mes.count(),
36     }
37
38     return render(request, "index.html", context)

```

Agora, se você tentar acessar a URL <http://127.0.0.1:8000/> sem estar autenticado, o Django irá exibir uma mensagem de erro. Esse erro ocorre porque quando utilizamos o decorator `login_required`, também é necessário configurar a URL de login e a URL para qual o usuário deverá ser direcionado após se autenticar.

✓ Caso esteja autenticado, vá até o admin e clique em "encerrar sessão" ou cliquei nesse link: <http://127.0.0.1:8000/admin/logout/>

Vamos fazer isso em todas as view criadas que fazem parte da dashboard. São elas: `registrar_visitante`, `informacoes_visitante` e `finalizar_visita`.

Alterando a URL padrão para login e redirecionamento após login

Agora que bloqueamos o acesso às views utilizando o decorator, precisamos definir as informações citadas anteriormente no arquivo `settings.py`. Vamos configurar as variáveis `LOGIN_URL` e `LOGIN_REDIRECT_URL` e definir seus valores como `"login"` e `"index"`. O arquivo ficará assim:

```

1  # código acima omitido
2  STATICFILES_DIRS = [
3      os.path.join(BASE_DIR, "static")
4  ]
5
6  LOGIN_URL = "login"
7  LOGIN_REDIRECT_URL = "index"

```

```
8 # código abaixo omitido
```

Não se preocupe com a URL login. Vamos criá-la no próximo passo.

Utilizando o sistema de autenticação do Django para nos fornecer a view de login

Conforme dito, vamos utilizar o sistema de autenticação do Django para nos ajudar com todo código necessário para autenticar e deslogar os usuários da nossa dashboard. Para começar vamos abrir o arquivo `urls.py` e importar os seguinte módulo:

```
1 from django.contrib import admin
2 from django.urls import path
3
4 from django.contrib.auth import views as auth_views
5
6 import dashboard.views
7 import visitantes.views
8
9 # código abaixo omitido
```

O que estamos fazendo é importar o arquivo de views do módulo `django.contrib.auth`. Fazendo isso, nós podemos utilizar as funções e classes disponíveis no módulo `django.contrib.auth.views`. Abaixo da função `path()` que define a URL para o admin, vamos definir a nossa URL de login:

```
1 urlpatterns = [
2     path("admin/", admin.site.urls),
3
4     path(
5         "login/",
6         auth_views.LoginView.as_view(
7             template_name="login.html"
8         ),
9         name="login"
```



```
10     ),
11
12     # código abaixo omitido
13 ]
```

Como importamos o arquivo inteiro com o nome de `auth_views`, vamos acessar suas funções e classes por meio desse nome. Note que no lugar da view que deveríamos passar para a função `path()`, estamos passando uma classe existente no módulo `auth_views` e utilizando seu método `as_view()`. Esse método nos permite utilizar as views padrões do Django para autenticação que ainda nos permitem criar um template personalizado. O argumento `template_name` serve para que a gente diga para o Django qual template deve ser utilizado na view.

Com essa configuração, já temos uma URL de login. Agora precisamos de um template, claro.

Criando o template de login

Vamos criar o arquivo `login.html` com o seguinte código:

```
1  <!DOCTYPE html>
2
3  {% load static %}
4  {% load widget_tweaks %}
5
6  <html lang="pt-br">
7      <head>
8          <meta charset="utf-8">
9          <meta http-equiv="X-UA-Compatible" content="IE=edge">
10         <meta name="viewport" content="width=device-width, initial-scale=1">
11
12         <title>Controle de Visitantes | Autenticação</title>
13
14         <link href="https://fonts.googleapis.com/css?family=Nunito:200,200i,400,400i,600,600i,700,700i,800,800i,900,900i" rel="stylesheet">
15
16         <link href="{% static 'css/sb-admin-2.min.css' %}" rel="stylesheet">
17         <link href="{% static 'vendor/fontawesome-free/css/all.min.css' %}" rel="stylesheet">
18     </head>
```

```

19
20 <body class="bg-gradient-primary">
21   <div class="container">
22     <div class="row justify-content-center">
23       <div class="col-xl-10 col-lg-12 col-md-9">
24         <div class="card o-hidden border-0 shadow-lg my-5">
25           <div class="card-body p-0">
26             <div class="row">
27               <div class="col-lg-6 d-none d-lg-block bg-
28
29               <div class="col-lg-6">
30                 <div class="p-5">
31                   <div class="text-left mb-5">
32                     <h1 class="h4 text-gray-900 mb-
33
34                     <p>Faça login para continuar</
35                   </div>
36
37                   <form method="post" class="user">
38                     <p>ué cadê o formulário?</p>
39
40                     <button class="btn btn-primary
41                       <span class="text">Acessar
42                     </button>
43                   </form>
44                 </div>
45               </div>
46             </div>
47           </div>
48         </div>
49       </div>
50     </div>
51   </div>
52 </div>
53
54   <script src="{% static 'vendor/jquery/jquery.min.js' %}"></script>
55   <script src="{% static 'vendor/bootstrap/js/bootstrap.bundle.min.j
56   <script src="{% static 'js/sb-admin-2.min.js' %}"></script>
57 </body>
58 </html>

```

Se você preferir, pode fazer download do arquivo zipado e colocar na pasta **templates** do seu projeto clicando no link abaixo:

[Iniciar o download](#)

login.zip - 1KB

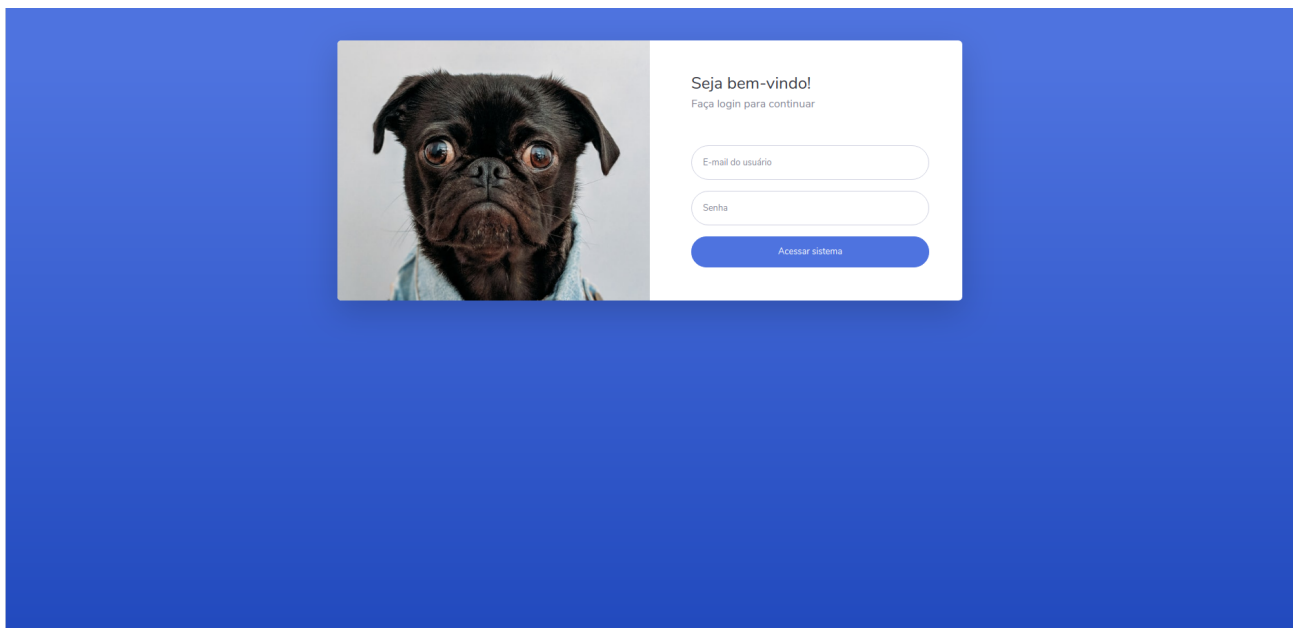
Renderizando formulário de login

Quando utilizamos a classe `LoginView` o Django nos dá tudo que precisamos para exibir o template, tratar a requisição, os possíveis erros do formulário e ainda autenticar o usuário. Junto disso tudo, temos a possibilidade de utilização da variável `form` que representa nosso formulário de autenticação.

Quando criamos nosso modelo personalizado de usuário, o Django preparou um formulário baseado nesse modelo, de forma bem parecida com que fizemos nos formulários que criamos. Vamos utilizar a mesma estratégia que utilizamos para renderizar o formulário de registro de visitante. O código do formulário ficará assim e deverá substituir o elemento `<p>` dentro do `<form>` :

```
1 <div class="form-row">
2     {% csrf_token %}
3
4     {% for field in form %}
5         <div class="form-group col-md-12">
6             {% render_field field placeholder=field.label class="form-cont
7         </div>
8     {% endfor %}
9 </div>
```

Vamos acessar novamente a URL <http://127.0.0.1:8000/> e agora podemos notar que fomos direcionados para a tela de login que acabamos de criar. Essa é a nossa tela de login:



Página de login com escritos de boas vindas e um formulário contendo E-mail e Senha, além do botão azul para acesso à dashboard escrito "Acessar dashboard". O fundo da tela é azul e há também a foto de uma cachorrinho ao lado do formulário, como se estivesse encarando o usuário

Adicionando mensagem de erro em formulário de login

Agora que estamos exibindo nosso formulário, vamos adicionar só mais uma coisa no template: um alerta para caso o formulário retorne algum erro. Dessa vez vamos apenas informar que o e-mail ou a senha estão incorretos. O template completo ficará assim:

```
1  <!DOCTYPE html>
2
3  {% load static %}
4  {% load widget_tweaks %}
5
6  <html lang="pt-br">
7      <head>
8          <meta charset="utf-8">
9          <meta http-equiv="X-UA-Compatible" content="IE=edge">
10         <meta name="viewport" content="width=device-width, initial-scale=1">
11
12         <title>Controle de Visitantes | Autenticação</title>
13
14         <link href="https://fonts.googleapis.com/css?family=Nunito:200,200"
```

```

15
16     <link href="{% static 'css/sb-admin-2.min.css' %}" rel="stylesheet"
17     <link href="{% static 'vendor/fontawesome-free/css/all.min.css' %}"
18 </head>
19
20 <body class="bg-gradient-primary">
21     <div class="container">
22         <div class="row justify-content-center">
23             <div class="col-xl-10 col-lg-12 col-md-9">
24                 <div class="card o-hidden border-0 shadow-lg my-5">
25                     <div class="card-body p-0">
26                         <div class="row">
27                             <div class="col-lg-6 d-none d-lg-block bg-
28
29                             <div class="col-lg-6">
30                                 <div class="p-5">
31                                     <div class="text-left mb-5">
32                                         <h1 class="h4 text-gray-900 mb
33
34                                         <p>Faça login para continuar</
35                                     </div>
36
37                                     {% if form.errors %}
38                                         <div class="alert alert-warnin
39                                             E-mail ou senha incorretos
40                                         </div>
41                                     {% endif %}
42
43                                     <form method="post" class="user">
44                                         <div class="form-row">
45                                             {% csrf_token %}
46
47                                             {% for field in form %}
48                                                 <div class="form-group
49                                                     {% render_field fi
50                                                 </div>
51                                             {% endfor %}
52                                         </div>
53
54                                         <button class="btn btn-primary
55                                             <span class="text">Acessar
56                                         </button>
57                                     </form>
58                                 </div>
59                             </div>
60                         </div>
61                     </div>
62                 </div>
63             </div>
64         </div>
65     </div>

```

```
66
67     <script src="{% static 'vendor/jquery/jquery.min.js' %}"></script>
68     <script src="{% static 'vendor/bootstrap/js/bootstrap.bundle.min.js' %}"></script>
69     <script src="{% static 'js/sb-admin-2.min.js' %}"></script>
70 </body>
71 </html>
```

Com o template finalizado, vamos novamente tentar acessar a URL <http://127.0.0.1:8000/>. Dessa vez, quando formos direcionados para a tela de login, vamos utilizar as credenciais que criamos (e-mail e senha) e acessar o sistema.

Criando URL para logout

Com a tela de login criada, vamos trabalhar para também criar a tela de logout, que será útil para quando o usuário quiser sair da dashboard.

Antes de tudo, vamos criar a URL de `logout` quase da mesma forma com que criamos a URL de login. Vamos abrir o arquivo `urls.py` e utilizar a função `path` para criar a URL `logout`. O Código ficará assim:

```
1  # código acima omitido
2
3  urlpatterns = [
4      path("admin/", admin.site.urls),
5
6      path(
7          "login/",
8          auth_views.LoginView.as_view(
9              template_name="login.html"
10             ),
11          name="login"
12      ),
13
14      path(
15          "logout/",
16          auth_views.LogoutView.as_view(
17              template_name="logout.html"
18             ),
19          name="logout"
```

```
20     ),
21
22     # código abaixo omitido
23 ]
```

Note que dessa vez estamos utilizando a classe `LogoutView` .

Criando template de logout

O processo será bem parecido com o executado para criação da URL da login. Como já criamos a URL, vamos agora partir para a criação do template que deverá ser exibido quando a URL `logout` for acessada. Vamos criar o arquivo `logout.html` na pasta `templates` com o seguinte código:

```
1  <!DOCTYPE html>
2
3  {% load static %}
4  {% load widget_tweaks %}
5
6  <html lang="pt-br">
7      <head>
8          <meta charset="utf-8">
9          <meta http-equiv="X-UA-Compatible" content="IE=edge">
10         <meta name="viewport" content="width=device-width, initial-scale=1">
11
12         <title>Controle de Visitantes | Logout</title>
13
14         <link href="https://fonts.googleapis.com/css?family=Nunito:200,200i,400,400i,600,600i,700,700i,800,800i,900,900i" rel="stylesheet">
15
16         <link href="{% static 'css/sb-admin-2.min.css' %}" rel="stylesheet">
17         <link href="{% static 'vendor/fontawesome-free/css/all.min.css' %}" rel="stylesheet">
18     </head>
19
20     <body class="bg-gradient-primary">
21         <div class="container">
22             <div class="row justify-content-center">
23                 <div class="col-xl-10 col-lg-12 col-md-9">
24                     <div class="card o-hidden border-0 shadow-lg my-5">
25                         <div class="card-body p-0">
26                             <div class="row">
27                                 <div class="col-lg-6 d-none d-lg-block bg-
```

```

28
29         <div class="col-lg-6">
30             <div class="p-5">
31                 <div class="text-left mb-5">
32                     <h1 class="h4 text-gray-900 mb-5">
33
34                         <p>Você escolheu sair da dashb
35                     </div>
36
37                         <a href="{% url 'login' %}" class=
38                             <span class="text">Voltar para
39                         </a>
40                     </div>
41                 </div>
42             </div>
43         </div>
44     </div>
45 </div>
46 </div>
47 </div>
48
49     <script src="{% static 'vendor/jquery/jquery.min.js' %}"></script>
50     <script src="{% static 'vendor/bootstrap/js/bootstrap.bundle.min.j
51     <script src="{% static 'js/sb-admin-2.min.js' %}"></script>
52 </body>
53 </html>

```

Nosso template de logout exibe apenas uma mensagem dizendo que o usuário saiu do sistema e mostra um botão para ele voltar para a página de login, caso queira.

Mais uma vez, se você preferir, você pode fazer o download do arquivo zipado:

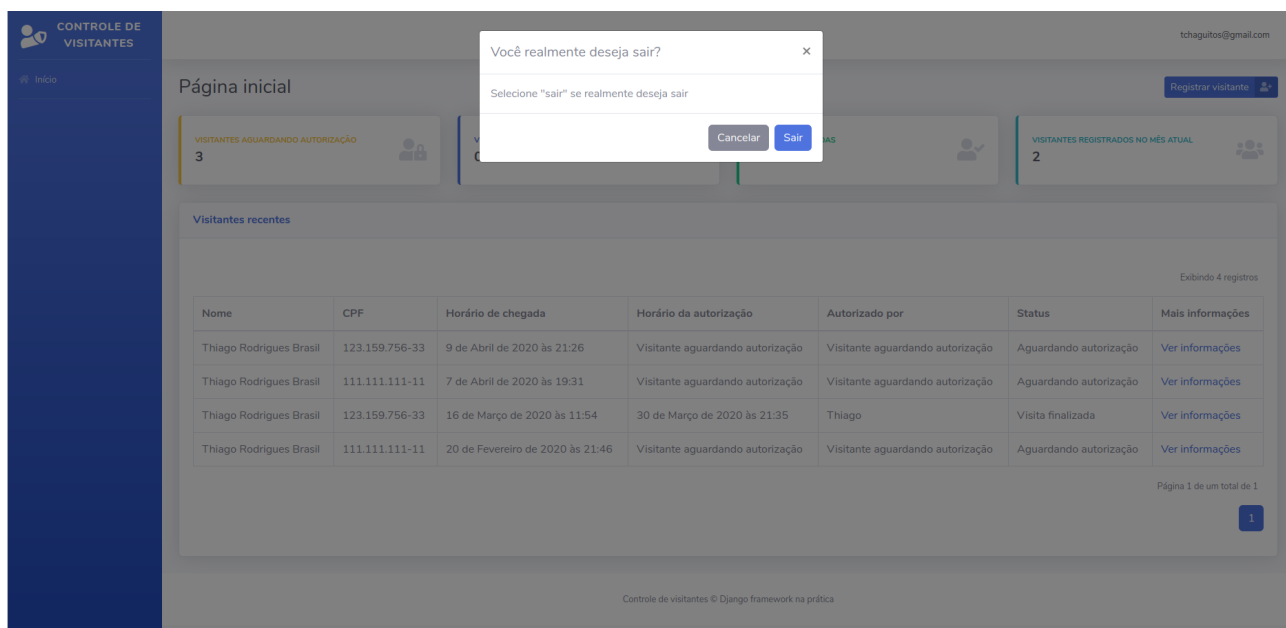


Iniciar o download

logout.zip - 1KB

Inserindo link para logout em dashboard

Se em algum momento anterior você clicou no ícone da engrenagem, no canto superior direito, descobriu o botão `sair`. Ele abre um modal pedindo que o usuário confirme se realmente deseja sair:

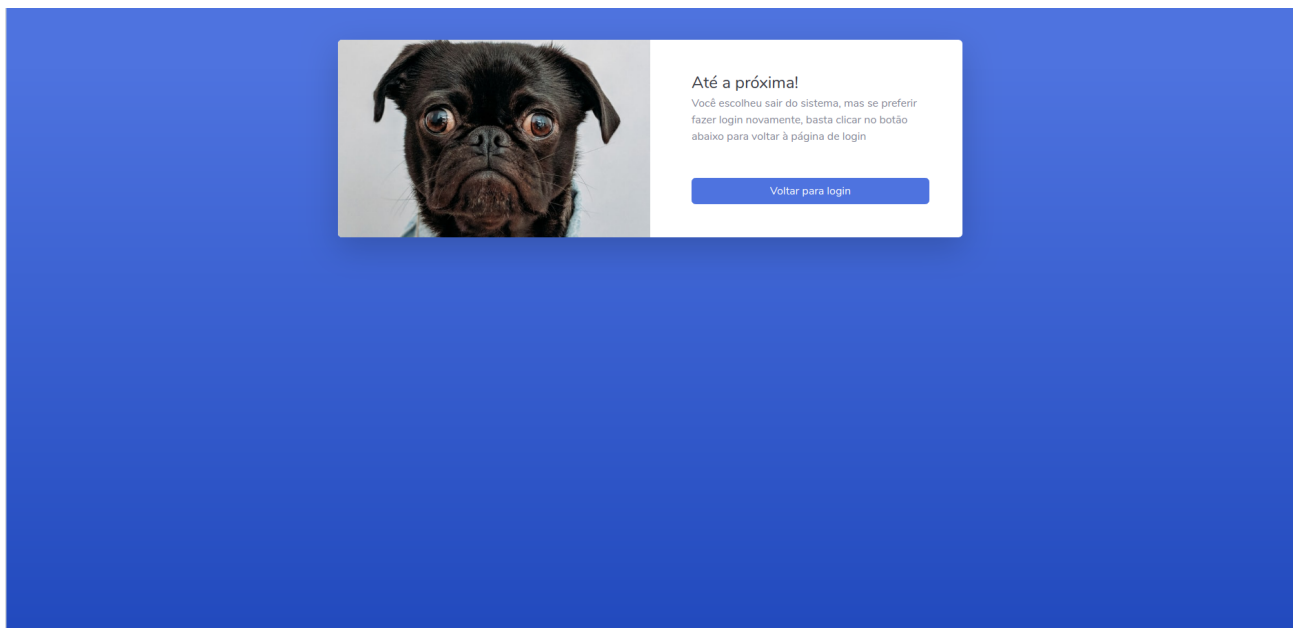


Template da página inicial da dashboard ao fundo com um alerta pedindo a confirmação de que o usuário realmente deseja sair do sistema

Vamos inserir o link para a página logout no botão que confirma a ação que o usuário deseja sair. No template `base.html`, vamos procurar pelo modal de id `logoutModal` e depois o link "sair" dentro do elemento `<div class="modal-footer">`. O HTML do botão ficará assim:

```
<a class="btn btn-primary" href="{% url 'logout' %}">Sair</a>
```

Com isso, quando o usuário clicar no botão "sair" confirmando que deseja sair da dashboard, será direcionado para a página de logout.



Tela de logout contendo uma mensagem de despedida e dizendo que o usuário pode clicar no botão abaixo para retornar à página de login. Abaixo um botão azul escrito "Voltar para login" e à esquerda a foto do cachorro como na página de login

Capítulo final

Chegamos ao capítulo final da nossa apostila e agora temos um projeto web completo pronto para ser implantado em ambiente de produção. Tudo isso feito com Python e o Django framework.

Nossa aplicação web administra todo o fluxo de entrada e saída de visitantes do condomínio, possibilitando desde o registro dos visitantes e exibição de informações, até as funcionalidades de autorização de entrada e finalização de visita.

No decorrer do material utilizamos os vários recursos que o Django nos fornece e aprendemos a utilizar as principais ferramentas na prática com o projeto que desenvolvemos. Daqui para frente, ainda temos muito chão para trilhar, é claro, mas com certeza iniciamos com uma base sólida que será aproveitada para os próximos projetos e estudos, por isso, não pare por aqui.

Eu espero que você tenha aproveitado e gostado de ter passado essas horas aqui comigo aprendendo sobre Django framework e, se você quiser aprender mais, continue acompanhando as redes sociais e o youtube:

- [Twitter](#)
- [Github](#)

Em caso de dúvidas, o fórum está à disposição para que possamos aproveitar ao máximo o espaço e resolver as dúvidas de todos.

Um forte abraço e nos vemos por aí!