

Flask de A a Z

Crie aplicações web mais completas e robustas em Python



Sumário

- [ISBN](#)
- [Agradecimentos](#)
- [Sobre o autor](#)
- [Prefácio](#)
- [Sobre o livro](#)
- [Introdução](#)
 - [1 Configuração do Python](#)
 - [2 Primeiros passos com Flask](#)
- [Estrutura do projeto - Padrão MVC](#)
 - [3 Trabalhando com Models](#)
 - [4 Trabalhando com Routes](#)
 - [5 Trabalhando com Controllers](#)
 - [6 Área administrativa no Flask](#)
 - [7 Trabalhando com o SQLAlchemy](#)
 - [8 Trabalhando com views](#)
- [Autenticação e requisição segura](#)
 - [9 API Rest no Flask](#)
 - [10 Autenticação e segurança no Flask](#)
 - [11 Trabalhando com serviços de e-mail](#)

ISBN

Impresso e PDF: 978-85-7254-033-9

EPUB: 978-85-7254-034-6

MOBI: 978-85-7254-035-3

Caso você deseje submeter alguma errata ou sugestão, acesse
<http://erratas.casadocodigo.com.br>.

Agradecimentos

Dedico este livro primeiramente a Deus, pois foi Ele quem me concedeu chegar até aqui e realizou sonhos muito além daqueles que um dia pensei que poderia sonhar, meu melhor Amigo e Pai, que me entende, me ouve e me ajuda, em todos os momentos de minha vida, sem Ele não sou e não desejo ser nada.

À minha mãe Andréa que me criou e lutou para que eu me tornasse o homem que hoje sou, que passou por momentos difíceis para que eu pudesse sorrir e acreditou durante todo o tempo que eu chegaria até aqui e a meus pais França e Vilma, que me acompanharam e me acolheram em sua família aos 13 anos, família esta que posso chamar de minha e que também foi fundamental para que eu alcançasse meus objetivos e ao meu pai Hamilton que é um exemplo de perseverança viva e que sempre me gerou inspiração para seguir acreditando em meus sonhos.

À minha companheira, amiga, musa, namorada, noiva e esposa, meu eterno amor, minha inspiração e um dos motivos pelo qual permaneço buscando ser uma pessoa melhor a cada dia, para permanecer sendo capaz de fazê-la se apaixonar por mim todos os dias, como me apaixono por ela em todos os instantes, obrigado por estar ao meu lado cada segundo, acreditando em mim.

Aos meus irmãos, Diogo e Victor, que são exemplos que incansavelmente sempre segui na certeza de que me tornaria um grande e honrado homem, e a minha pequena e linda irmã Maria Luiza, que com seu sorriso repleto e sincero sempre me traz uma luz e alegria, até mesmo em momentos complicados e conturbados que todos temos em nosso dia a dia.

À minha tia Janete e tio Eduardo, que investiram em meus estudos e assim me ajudaram a entrar para o ramo da tecnologia e ao meu melhor amigo e irmão Marcos Abraão, que nesses mais de 15 anos de amizade sempre apostou grande em mim e investiu seu tempo e esforço.

Ao meu amigo Vinícius Albino, por ser essa pessoa que acrescenta em minha vida com seus conselhos, me ajudando a ser uma pessoa melhor e mais madura.

Ao meu padrasto Saulo, que junto de minha mãe Andréa auxiliou em minha criação, sempre me tratando como um filho, me aconselhando e ajudando em tudo que pôde.

A todos os meus alunos, que são o real motivo e inspiração para que eu continue todos os dias buscando conhecimento para ser compartilhado entre todos de forma a ajudar no crescimento profissional e pessoal de cada um.

A toda a equipe da Casa do Código que foram fundamentais para que este livro viesse a ter o nível de qualidade que o leitor merece, em especial à Vivian Matsui que desde o primeiro contato acreditou em meu sonho e fez parte de todo o processo como minha editora-chefe.

A todos vocês o meu muito obrigado, por fazerem parte da minha vida e acreditarem que eu me tornaria alguém cujo coração teria como missão compartilhar o aprendizado e sabedoria que graças a Deus vindo de vocês me tornou o homem que hoje sou.

Sobre o autor

Tiago Luiz é graduado em Análise e Desenvolvimento de Sistemas pelas Faculdades Integradas Simonsen, professor na área de Tecnologia há mais de 6 anos, atualmente trabalha como cientista de dados onde desenvolve sistemas e scripts em Python focados em processamento de um alto volume de dados. Especialista em Python, Google Maps e Adobe Muse é fundador do Canal Digital Cursos, onde existem cursos inteiramente online focados em ajudar pessoas a entrarem para o mercado de trabalho.

Prefácio

Este livro foi criado com o foco de partilhar com você, leitor ou leitora, mais sobre meus conhecimentos e experiências com a ferramenta Flask, uma das mais utilizadas pelos programadores Python no mundo.

O conteúdo é bem direto, sem rodeios, mas ao mesmo tempo com explicações proveitosas e práticas. A didática foi construída para que seu entendimento ocorra de um modo descontraído e fácil, focando diretamente no que precisa ser dito a você. Há também imagens que expressem os resultados desejados no sistema que será desenvolvido no livro, para você acompanhar sua evolução ao decorrer dos capítulos.

O que mais me alegra é saber que este livro é capaz de auxiliar um profissional de desenvolvimento na criação de seus projetos de forma segura, efetiva e rápida, com uma ferramenta inteiramente poderosa e completa. Além dos capítulos essenciais que estão no livro, dispomos também de alguns conteúdos extras que acreditamos ser de grande valor para sua vida como programador Python, como autenticação segura com JWT e serviço de envio de e-mail. Considere como um presente de quem deseja ver seu crescimento profissional.

Faça bom proveito deste conteúdo, foi um imenso prazer compartilhar com você o conhecimento que recebi de muitas pessoas que, como eu, tiveram o intuito de ajudar outras a crescerem profissionalmente e alcançarem seus sonhos.

Tiago Lui

Sobre o livro

Neste livro, você aprenderá a trabalhar com uma das linguagens mais utilizadas no mundo, o Python.

O tema principal a ser abordado será o framework Flask. Por ser uma ferramenta do Python robusta e completa, ela permite que a aplicação Web seja construída de forma a atender às necessidades de seu usuário final, ao mesmo tempo em que flexibiliza o fluxo de desenvolvimento de uma equipe, ou até mesmo de um desenvolvedor que tenha projetos pessoais. O Flask é um dos melhores frameworks para se trabalhar no Python atualmente.

O Flask está em sua versão 1.0.2, e seu primeiro release foi feito em abril de 2010, então podemos ver que ele é bem maduro, tendo em vista que são mais de 9 anos de atualizações pelas quais ele vem se solidificando.

Outra tecnologia muito interessante que trabalharemos no livro e que será fundamental para nosso crescimento é o SQLAlchemy, uma biblioteca muito interessante e completa, que nos permite trabalhar com diversos bancos de dados relacionais dentro do Python. A interação do SQLAlchemy com o Flask é excelente e isso trará muitas vantagens para nossos estudos.

Um breve resumo do mercado de Python

De acordo com o blog da GeekHunter, o Python está entre as 10 linguagens de programação mais bem pagas atualmente, mas não é só isso que traz vantagens em utilizar o Python. Sua sintaxe de fácil escrita e sua flexibilidade, quando bem utilizadas, permitem que tenhamos excelentes projetos com estruturas totalmente reutilizáveis e nada burocráticas.

Link da postagem: <https://blog.geekhunter.com.br/salario-de-programador-cargos-em-alta-2019/>

Público e pré-requisitos

Este livro é voltado para desenvolvedores que possuem conhecimento da linguagem Python e desejam utilizá-la em sistemas Web e APIs.

Recomendamos que você tenha conhecimento básico em Python e criação de ambientes virtuais (*Virtualenvs*). Não é necessário conhecimento nas ferramentas citadas na introdução, mas o conteúdo abordado será de nível avançado.

O que aprenderei neste livro?

Você aprenderá a criar aplicações Web e APIs Rest totalmente robustas utilizando Flask, SQLAlchemy e outras ferramentas que o Python possui.

Criaremos um sistema de gerenciamento de produtos, onde gerenciaremos não somente produtos, mas suas categorias, usuários e suas funções no painel de acesso, podendo limitar um usuário para que ele acesse apenas a API do sistema ou o administrador também. Ao término teremos uma aplicação que possuirá uma API e uma área administrativa completa e personalizada para atender nossas regras de negócio.

Com o Flask temos a possibilidade de criar APIs para nossos aplicativos móveis, com muita qualidade e de forma bem robusta. Também conseguimos criar uma área administrativa bem completa através dos recursos que o Flask e o SQLAlchemy proporcionam em conjunto, de um modo bem seguro.

Como estudar com este livro?

O livro foi escrito para ser estudado com a mão na massa, trazendo explicações bem sólidas sobre o assunto junto da execução prática de etapas de um sistema de gerenciamento de estoque, que será construído no decorrer dos capítulos, além de uma API que usuários

autorizados poderão utilizar para consumir os dados do sistema externamente através de um `app` OU `website` .

Além das explicações contendo práticas bem elaboradas e de fáceis entendimentos, contamos também com algumas observações e dicas em cada tema, conforme a experiência do autor. São questões ou conselhos que deixarei para evitar que você passe por algum problema que já enfrentei utilizando o framework.

Introdução

Nessa primeira parte temos dois capítulos que servirão como introdução ao livro. O primeiro abordará a maneira correta de configurar seu computador para trabalhar com Python e o segundo falará um pouco sobre o que é o Flask, seus pontos fortes e fracos, as vantagens de utilizá-lo, e já iniciaremos a criação do sistema de gerenciamento utilizando o Flask.

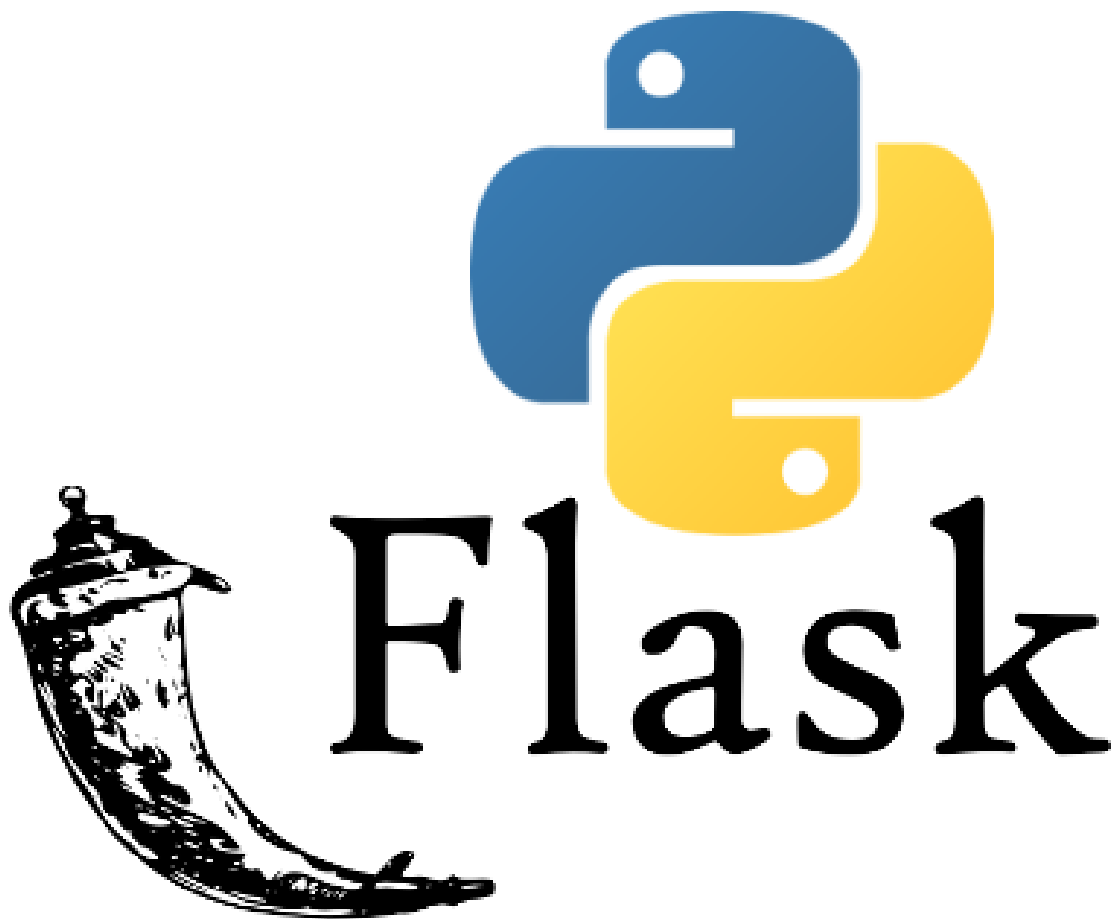


Figura 1: Logo da linguagem e da ferramenta

CAPÍTULO 1 **Configuração do Python**

Neste capítulo, veremos como fazer as instalações básicas do Python em nossa máquina e rodaremos nosso primeiro script para testar se tudo está dentro do esperado. Caso já tenha o Python em sua máquina, sintá-se à vontade para ir ao capítulo 2.

Vamos demonstrar utilizando os Sistemas Operacionais Windows e Linux (mais precisamente Ubuntu), mas tenha em mente que se seu sistema operacional for qualquer outro que trabalhe baseado em Linux, ou se for MacOS, os comandos de Linux provavelmente vão funcionar ou estarão bem próximos dele.

1.1 Instalando o Python e suas dependências

Windows

Vá até o link <https://www.python.org/downloads/> e faça o download da versão mais atual que estiver disponível. Clicando em `download`, isso provavelmente já ocorrerá direto. Uma dica é deixar o caminho de instalação exatamente onde o instalador deixa por padrão, pois colocá-lo em outro caminho pode impactar em questões de permissão de usuário etc.

No momento da escrita do livro a versão do Python era a 3.7.x.

Pronto, ao fazer isso seu Python já está instalado. Agora vamos configurar as variáveis de ambiente e instalar o gerenciador de pacotes do Python chamado `pip`.

Variáveis de ambiente do Python no Windows

Fique atento, pois o instalador do Python possui a opção de adicionar o `path` de instalação diretamente na variável de ambiente. Existem casos em que essa opção não existe, ou mesmo

marcando-a pode ocorrer de o `path` não estar adicionado nas variáveis de ambiente. Caso isso ocorra siga os passos a seguir:

1. Abra o painel de controle e navegue até as configurações de sistema.
2. Selecione as configurações avançadas do sistema.
3. Clique em variáveis de ambiente.
4. Procure nas variáveis do sistema pela variável `path`.
5. Clique em `Editar`.
6. Veja se os valores `C:\Python37` e `C:\Python37\Scripts` já estão no campo de valor da variável de ambiente; se não existirem, adicione-os ao final da linha separados por ponto e vírgula (;). O `python37`, neste exemplo, é referente à pasta onde o Python foi instalado no seu sistema, então este valor pode ser diferente caso esteja instalando outra versão. Por exemplo, se for a versão `2.7.15` do Python, o valor será `Python27`; se for `3.7.0`, o valor será `Python37` e assim por diante. Veja o caminho exato da sua instalação e substitua o valor acima.
7. Após isso, clique em `OK`.

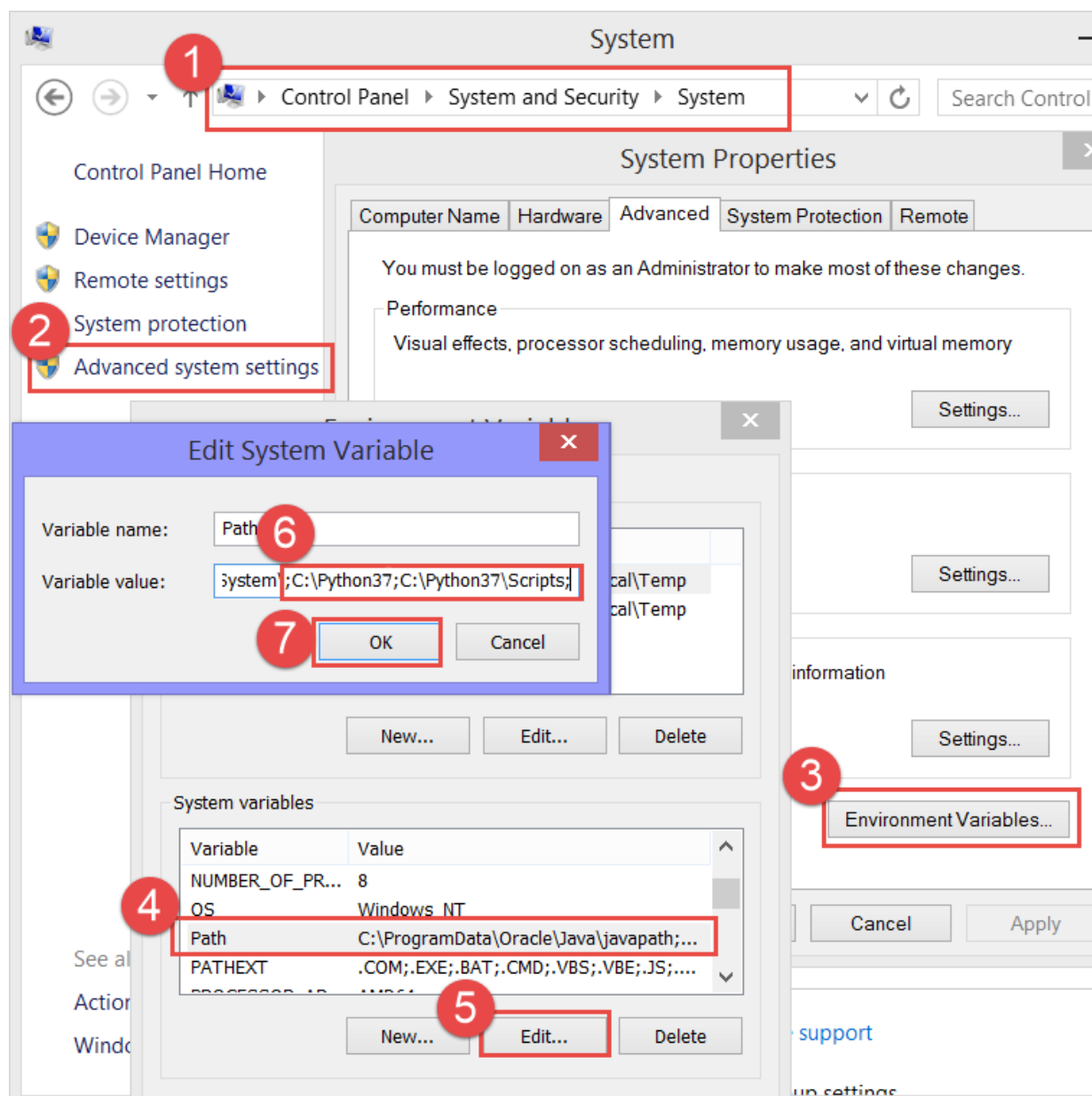


Figura 1.1: Painel de Controle do Windows

Instalando o PIP no Windows

O `pip` é um gerenciador de pacote que o Python possui. Com ele, conseguimos baixar novos pacotes/bibliotecas, e também gerenciá-los, com recursos que permitem listar tudo que temos já instalado

em nosso projeto e fazer a instalação desses pacotes com mais facilidade em outros ambientes.

A instalação do `pip` só funciona a partir da versão 2.7.9 do Python.

No menu do Windows, vá em `Executar`, abra seu terminal `cmd` e rode o comando a seguir para instalar o pacote do `pip`:

```
c:\Users\Tiago>python -m ensurepip
```

Ao término, ele retornará que todos os requisitos estão satisfeitos. Então, rode o comando a seguir:

```
c:\Users\Tiago>python -m ensurepip --upgrade:
```

Esse comando serve para realizar possíveis atualizações que não vieram no momento da instalação do `pip`.

Agora você possui o `pip` instalado e atualizado em seu computador.

Criando um virtualenv no Windows

O `virtualenv` é o ambiente virtual que o Python permite que você crie para poder trabalhar em diferentes versões do Python dentro de uma mesma máquina. Cada projeto pode lidar com uma versão diferente do Python, graças a esse recurso. Dentro dele fazemos as instalações de todas as bibliotecas que estamos utilizando dentro do projeto, assim podemos controlá-las de forma isolada dentro de um caso, sem interferir em outro. Isso permite que tenhamos projetos mais seguros em questão de desenvolvimento.

Para instalar um `virtualenv` no Windows, rode o comando a seguir dentro do terminal:

```
c:\Users\Tiago>c:\Python37\Scripts\pip.exe install virtualenv
```

Não se esqueça de que, se sua versão for diferente, o caminho `python37` poderá ser diferente do mostrado aqui.

No local que preferir, crie uma pasta onde ficará o nosso primeiro projeto e chame-a de `hello_world`.

Dentro dessa pasta, rode o comando a seguir, para criar seu `virtualenv`. Assim poderemos realizar futuramente as instalações de pacotes necessários para o projeto sem afetar outros possíveis projetos que existirão em seu computador:

```
c:\Users\Tiago\Documents\hello_world>c:\Python37\Scripts\virtualenv.exe  
venv
```

Com seu `virtualenv` criado, é necessário que ele seja ativado. Pense nele da seguinte forma: o terminal que possuir o `virtualenv` ativado terá todas as bibliotecas daquele projeto ativadas nele. Para ativar seu `virtualenv` rode o seguinte comando:

```
c:\Users\Tiago\Documents\hello_world>venv\Scripts\activate
```

Pronto, agora temos tudo pronto para iniciar nosso projeto. Seu terminal ficará parecido com o seguinte:

```
(venv) C:\Users\Documents\hello_world>
```

Fique atento: se você fechar o terminal, precisará abrir outro e rodar esse comando novamente. Isso porque o ambiente, como dito, é virtual e precisa ser iniciado dentro do terminal que for utilizado toda vez que esse terminal for aberto pela primeira vez.

Ubuntu

Verifique primeiramente se você já possui o Python instalado por padrão em seu Ubuntu, rodando o comando:

```
tiago_luiz@ubuntu:~$ which python
```


Provavelmente, ele retornará algo como `/usr/bin/python` , o que verifica que o Python já está instalado. Mas se ele não retornar isso, e sim algo como `which: no python in (/usr/local/sbin:/usr/local/bin:/usr/bin:/usr...)` , precisaremos instalá-lo, então siga os próximos passos.

Por meio do gerenciador de pacotes `apt-get` , rode o comando a seguir:

```
tiago_luiz@ubuntu:~$ sudo apt-get install python3.7
```

No caso de ser outra versão, substitua o `python3.7` pela versão desejada.

O `pip` no Ubuntu é mais simples do que no Windows. Para rodá-lo, digite o comando a seguir, cuja função é a mesma que foi explicada no tópico sobre Windows:

```
tiago_luiz@ubuntu:~$ sudo apt-get install python-pip
```

O Python não exige variáveis de ambiente em sistemas Linux, então fique tranquilo pois podemos pular essa parte.

Criando um virtualenv no Ubuntu

Como já explicamos no tópico de Windows o que é `virtualenv` , vamos direto para sua instalação e inicialização no projeto.

Para instalar um `virtualenv` no Ubuntu, rode o comando:

```
tiago_luiz@ubuntu:~$ sudo pip3 install virtualenv
```

Crie uma pasta onde preferir para ficar nosso primeiro projeto, e chame-a de `hello_world` .

Com a pasta criada, entre neste diretório pelo terminal e crie seu `virtualenv` através do comando a seguir:

```
tiago_luiz@ubuntu:~/hello_world$ virtualenv -p python3 venv
```

Agora, para que seu projeto rode com base neste `virtualenv`, você precisa executar o seguinte comando:

```
tiago_luiz@ubuntu:~/hello_world$ source ./venv/bin/activate
```

Pronto, agora temos tudo pronto para iniciar nosso projeto. Seu terminal ficará da seguinte forma, ou algo próximo a isto, dependendo do caminho do projeto.

```
(venv) tiago_luiz@ubuntu:~/hello_world$
```

1.2 Escolhendo uma IDE

A escolha da IDE não é algo obrigatório, pois podemos utilizar até mesmo o bloco de notas para rodar Python, mas ter um ambiente para desenvolver a aplicação facilitará bastante nosso desenvolvimento.

Algumas das IDEs a seguir são muito recomendadas e eu mesmo tenho vivência com todas as listadas:

- PyCharm (<https://www.jetbrains.com/pycharm/>)
- Visual Studio Code (<https://code.visualstudio.com/>)
- Atom (<https://atom.io/>)
- Sublime Text (<https://www.sublimetext.com/>)

Todas são gratuitas, fique atento apenas para o fato de o PyCharm tem uma versão *community* gratuita e a versão *professional*, que é paga.

Neste livro, usaremos o Visual Studio Code.

1.3 Testando o ambiente para começar

Agora que temos tudo pronto, todo o processo está correto e sua máquina já está pronta para rodar o Python, vamos fazer nosso primeiro script antes de iniciar nossa jornada no mundo das aplicações Web com Flask.

Com o `virtualenv` iniciado, dentro da pasta `hello_world` crie um arquivo chamado `run.py`. Dentro do arquivo, escreva o código a seguir, e em seguida vamos entender o que ele faz:

```
# -*- coding: utf-8 -*-  
print('Olá Mundo')
```

A primeira linha de código é utilizada para dizer ao Python que ele deverá interpretar nossas Strings (textos) na codificação `utf-8`. Não entraremos nesse assunto de codificação neste livro, isso é apenas um teste para ver se o Python está instalado.

A segunda linha possui o método `print`, que é utilizado para exibir valores no terminal. Em nosso caso, o `print` exibirá o valor `Olá Mundo` apenas para testarmos se o Python está executando corretamente.

Agora, rode o comando a seguir para executar seu código:

```
(venv) tiago_luiz@ubuntu:~/hello_world$ python run.py
```

Você verá o seguinte resultado:

```
(venv) tiago_luiz@ubuntu:~/hello_world$ python run.py  
Olá Mundo  
(venv) tiago_luiz@ubuntu:~/hello_world$
```

Conclusão

Ao longo do livro, veremos mais a fundo sobre o Python e o framework Flask, que é o verdadeiro foco do livro. Este primeiro capítulo foi apenas um manual para auxiliar na configuração do ambiente de desenvolvimento.

Todos os códigos deste livro estão disponíveis no GitHub do autor: https://github.com/tiagoluizrs/livro_flask/.

CAPÍTULO 2

Primeiros passos com Flask

O Flask é uma ferramenta que foi desenvolvida em 01 de abril de 2010. Ela é muito utilizada para criar aplicações Web, que é o que veremos neste livro. As aplicações criadas com Flask têm diversas vantagens, devido à flexibilidade que a ferramenta proporciona para criação de APIs com áreas administrativas, para gerenciar os dados enviados e recebidos. Com Flask, faremos tudo isso de forma segura, eficaz e muito simples.

Neste capítulo, começaremos a estruturar nosso sistema de gerenciamento de estoques, criando as configurações do projeto e rodando nosso primeiro *run* do projeto. Esta primeira etapa ainda é mais voltada para o sistema, então por enquanto não criaremos nada diretamente ligado ao gerenciador de estoque.

2.1 Instalando o Flask

O `flask` é uma `lib` do Python e como qualquer outra precisa ser instalado para funcionar. Rode o comando a seguir para instalar o Flask em seu `virtualenv`.

```
(venv) tiago_luiz@ubuntu:~/livro_flask$ pip install flask
```

2.2 Estrutura do projeto

Uma boa estrutura de projeto torna-o legível e de fácil manutenção. A seguir, você verá uma estrutura de projeto simples de se entender e de utilizar. Utilizando essa referência, crie uma pasta chamada

livro_flask e, dentro dela, os arquivos do seu projeto, seguindo exatamente esta estrutura:

```
livro_flask
|   app.py
|   config.py
|   migrate.py
|   run.py
|
+---admin
|   Admin.py
|   Views.py
|
+---model
|   Category.py
|   Product.py
|   User.py
|   Role.py
|
+---controller
|   Email.py
|   Product.py
|   User.py
|
+---static
|   +---css
|   |       login.css
|   |       home.css
|   |
|
+---templates
|   login.html
|   home_admin.html
|   new_password.html
|   recovery.html
```

Fique tranquilo, ao longo do capítulo essa estrutura fará mais sentido para você. Se ainda assim você tiver alguma dúvida e quiser ampliar a visão sobre a estrutura do projeto, acesse o link dele no GitHub para compará-lo com o da estrutura desse exemplo anterior https://github.com/tiagoluizrs/livro_flask/.

2.3 Regra de negócios do sistema

Para termos uma visão mais ampla do contexto do projeto que vamos desenvolver, é importante conhecermos as regras de negócio dele. Elas serão fundamentais para que tenhamos um padrão correto durante o desenvolvimento do projeto e possamos realizar uma entrega de qualidade do nosso sistema. A seguir, deixamos listadas todas as regras de negócio do sistema de gerenciamento de estoque que criaremos.

Usuários e permissões

As funções administrativas têm como finalidade permitir o que cada usuário pode fazer no sistema. Afinal as permissões do administrador, como deletar ou editar dados de outro usuário, não devem valer para todo mundo, por exemplo. Em uma aplicação, é preciso dar diferentes permissões para cada tipo de usuário. Em nosso caso, temos as funções administrativas:

- Admin: possui permissão em todas as áreas do sistema;
- Gerente: pode listar, criar, editar, deletar os produtos e categorias do sistema;
- Logista: pode listar, criar e editar os produtos do sistema, mas não poderá deletar um produto;
- Cliente/Usuário: pode utilizar a API para visualizar os produtos do sistema e seu próprio perfil de usuário.

Telas

É por meio das telas que ocorre a interação do usuário com os dados do sistema. A seguir, elas estão divididas em três focos: login, usuários e produtos.

- Login:
 - Tela de login do usuário;
 - Tela de "Esqueci minha senha";
- Usuários:
 - Tela de lista de usuários com botão de editar, deletar e criar novo usuário;
 - Tela de formulário para criação e edição do usuário;
- Funções:
 - Tela de lista de funções que o sistema possui.
 - Tela de formulário para criação e edição da função;
- Categorias:
 - Tela que listará as categorias dos produtos;
- Produtos:
 - Tela de lista de produtos com botão de editar, deletar e criar novo produto;
 - Tela de formulário para criação, edição e visualização do produto.

Estrutura do JSON da API

A estrutura do JSON será bem simples, apenas para descrever como fazemos para enviar nossos dados via JSON em uma API RESTful.

```
{  
  "id": "Id do produto no banco de dados",  
  "name": "Nome do produto",  
  "description": "Descrição do produto",  
  "qtd": "Quantidade em estoque do produto",  
  "image": "Imagem do produto",  
  "price": "Preço do produto",  
}
```



```
"date_created": "Data da criação do produto"  
}
```

Com todas as regras de negócios, telas e estruturas definidas podemos começar a criar os primeiros arquivos do nosso projeto. Então vamos lá!

2.4 Arquivos de configuração e execução

Agora que toda a estrutura necessária já foi criada, chegou a hora de partirmos para os códigos. Os primeiros serão os arquivos de configuração e *run* do projeto. Ainda não vamos executar nenhuma ação que será vista no navegador, mas esses arquivos são fundamentais para que o projeto tenha um excelente funcionamento.

config.py

Este arquivo será o responsável por dizer ao Python em que ambiente nosso projeto estará rodando, se ele estará em produção, teste ou desenvolvimento. Isso é necessário porque cada ambiente tem uma configuração diferente. Afinal, quando estamos em nosso ambiente local (desenvolvimento) geralmente utilizamos um banco de dados local, algo que já é diferente em teste e produção, onde temos bancos de dados diferentes para eles. Logo, o arquivo de configuração é muito importante para separarmos o que deverá ter em cada ambiente diferente.

Veja o código a seguir, que deve ir em seu `config.py`, e adiante falaremos sobre ele.

config.py

```
import os  
import random, string  
  
class Config(object):
```

```

    CSRF_ENABLED = True
    SECRET = 'ysb_92=qe#djf8%ng+a*#4rt#5%3*4k5%i2bck*gn@w3@f&-&'
    TEMPLATE_FOLDER =
os.path.join(os.path.dirname(os.path.abspath(__file__)), 'templates')
    ROOT_DIR = os.path.dirname(os.path.abspath(__file__))
    APP = None

class DevelopmentConfig(Config):
    TESTING = True
    DEBUG = True
    IP_HOST = 'localhost'
    PORT_HOST = 8000
    URL_MAIN = 'http://%s:%s/' % (IP_HOST, PORT_HOST)

class TestingConfig(Config):
    TESTING = True
    DEBUG = True
    IP_HOST = 'localhost' # Aqui geralmente é um IP de um servidor na
nuvem e não o endereço da máquina local
    PORT_HOST = 5000
    URL_MAIN = 'http://%s:%s/' % (IP_HOST, PORT_HOST)

class ProductionConfig(Config):
    DEBUG = False
    TESTING = False
    IP_HOST = 'localhost' # Aqui geralmente é um IP de um servidor na
nuvem e não o endereço da máquina local
    PORT_HOST = 8080
    URL_MAIN = 'http://%s:%s/' % (IP_HOST, PORT_HOST)

app_config = {
    'development': DevelopmentConfig(),
    'testing': TestingConfig(),
    'production': ProductionConfig()
}

app_active = os.getenv('FLASK_ENV')

```

Inicialmente teremos 3 subclasses, pois precisaremos rodar nosso projeto em 3 ambientes diferentes. Começaremos no ambiente de desenvolvimento, depois iremos ao ambiente de teste e, por fim, ao

de produção. Cada subclasse corresponde a um ambiente, como podemos observar em seus nomes: `DevelopmentConfig`, `TestingConfig`, `ProductionConfig`.

Temos também uma superclasse chamada `Config`, cujas constantes são herdadas pelas subclasses. Como vemos, estas constantes serão sempre as mesmas, independente de qual seja o ambiente em que estamos rodando nosso projeto; já as subclasses possuem constantes que, apesar de terem nomes idênticos, possuem valores diferentes. Tenha em mente que somente uma delas será usada por vez e, no momento do uso, ela herdará as constantes de `Config` como citamos anteriormente.

Imagine que estamos rodando o projeto em teste. Então, a subclasse `TestingConfig` possui a constante `DEBUG` com seu valor setado como `True`. Agora, se formos trabalhar no ambiente de produção, a subclasse `ProductionConfig` também possui essa constante, porém seu valor estará setado como `False`. Isso significa que, no ambiente de teste, teremos o debug da aplicação ativado, mas em produção não. Perceba que existem outras constantes que recebem valores diferentes de acordo com o ambiente em que estivermos trabalhando, como o `IP_HOST` e a `PORT_HOST`. Você pode criar suas próprias constantes e adicioná-las dentro das subclasses ou da superclasse.

Entendendo o que cada um dos valores faz

Vamos entender qual a verdadeira finalidade de cada constante que está setada no arquivo de configurações:

- `CSRF_ENABLED` : habilita o uso de criptografia em sessões do Flask;
- `SECRET` : será usada em alguns momentos para criar chaves e valores criptografados. Com ela, vamos misturar valores e usá-los na criptografia de senhas.
- `TEMPLATE_FOLDER` : caminho do local em que os arquivos de template do projeto ficarão.

- `ROOT_DIR` : caminho do local em que a raiz do projeto se encontra. Usaremos muito isso quando quisermos salvar algum arquivo em pasta.
- `APP` : constante que receberá a propriedade do app. Inicia com valor nulo.
- `TESTING` : constante que habilita o ambiente de teste no Flask. Com ela, alguns recursos de *warning* e erros ficam habilitados para exibição.
- `DEBUG` : do mesmo jeito que `TESTING` , esta constante habilita e desabilita os debugs que o Python exibe no console de execução. Ele não desabilita a criação de um arquivo de log, apenas desabilita o log na tela do terminal.
- `IP_HOST` : usaremos esta constante para dizer qual é o `IP` da máquina em que estamos rodando o projeto. Quando o projeto está como local, o `IP` geralmente é `localhost` OU `127.0.0.1` . Quando enviamos o projeto para uma máquina na nuvem, geralmente essa máquina recebe outro `IP` , por isso colocamos valores diferentes em cada ambiente.
- `PORT_HOST` : a porta da aplicação é algo fundamental para que a aplicação rode. Afinal, em um servidor podem existir diversas aplicações rodando, porém cada uma rodará em uma porta. Por padrão utilizamos a porta `8080` ; no caso de desenvolvimento usaremos a porta `8000` .
- `URL_MAIN` : esta constante une o endereço de `IP` com a porta para gerar o endereço principal da sua aplicação, por exemplo: <http://localhost:8000>.
- `app_config` : possui as três subclasses dentro de si e será usado para dizer ao projeto qual das três vamos usar, ou seja, qual a configuração de ambiente escolhida.
- `app_active` : ela receberá um dos três valores: `development` , `testing` e `production` . Esse valor será atribuído através de uma variável de ambiente, ou seja, poderemos trocá-lo dinamicamente. Usamos a função `os.getenv('NOME_DA_CHAVE')` para setar um valor a ela através da criação de uma variável de ambiente que tenha o nome da chave que escolhermos. Mais à frente faremos exatamente isso. O projeto usará as

configurações escolhidas no momento em que setarmos o valor da variável de ambiente.

Agora já sabemos como trabalhar com o arquivo de configurações. Depois executaremos a aplicação para ver como ela se comporta nos três ambientes diferentes. Por enquanto vamos voltar ao arquivo `app.py`.

app.py

O arquivo `app.py` será visto de modo aprofundado no capítulo 8, em que falaremos sobre *views*, mas ele é fundamental para que o projeto funcione corretamente. Ele é responsável por conter a inicialização das propriedades da ferramenta Flask, banco de dados e, além disso, as rotas do projeto (veremos também mais à frente sobre rotas). O arquivo `app.py` é o segundo arquivo que será chamado no momento do *run*. Ele distribuirá as tarefas que cada parte do sistema deverá realizar, inicializando cada parte do projeto.

app.py

```
# -*- coding: utf-8 -*-
from flask import Flask

# config import
from config import app_config, app_active

config = app_config[app_active]
def create_app(config_name):
    app = Flask(__name__, template_folder='templates')

    app.secret_key = config.SECRET
    app.config.from_object(app_config[config_name])
    app.config.from_pyfile('config.py')

@app.route('/')
def index():
    return 'Hello World!'
```

```
return app
```

A primeira linha diz para o Python que usaremos o tipo de codificação `utf-8` em nossa aplicação, assim nosso sistema estará pronto para trabalhar com caracteres latinos que possuem acentos, traços e outros símbolos conhecidos em nosso idioma.

Após isso, é feita a importação das variáveis `app_config` e `app_active` que possuem respectivamente as subclasses de configuração e o tipo de ambiente que vamos utilizar. A `app_config`, como vimos, é um dicionário que possui as configurações das três subclasses do arquivo `config.py`, cada uma dentro de uma chave. É aí que a `app_active` entra, ela recebe um valor específico que será exatamente uma destas três chaves (`development`, `testing`, `production`). Assim, ela será usada para selecionar dinamicamente qual das subclasses armazenadas em `app_config` usaremos como configuração.

Imagine que `app_active` tenha o valor `production` armazenado dentro de si. Ao rodar o projeto, chamaremos `config` da seguinte maneira:

```
config = app_config[app_active], que seria a mesma coisa que config = app_config['production']. Assim estaremos atribuindo a config os valores da subclasse ProductionConfig().
```

Já a `def create_app` será utilizada mais à frente no arquivo `run.py` para iniciar a aplicação com todas as configurações que estão dentro dela.

A linha com a variável `app` receberá a instância do objeto Flask que faz com que todo o sistema funcione baseado nas configurações que o Flask possui. Nela também podemos fazer algumas configurações, como o nome da pasta de templates.

As três linhas seguintes possuem as configurações da aplicação. Na primeira, nós passamos o valor da `SECRET` que criamos no arquivo `config.py`; na segunda, o objeto de subclasse de configuração (`DevelopmentConfig`, `TestingConfig` OU `ProductionConfig`) com todas as

constantes de configuração que estão na classe e as herdadas da superclasse `Config`; na terceira linha dizemos ao Flask que `config.py` é nosso arquivo de configuração, para que ele possa acessá-lo sempre que necessário.

Para testarmos se nosso projeto está rodando corretamente, temos a `def index` que rodará na `url` principal uma mensagem que mostrará que o Flask está funcionando corretamente.

Por fim, retornamos nessa `def` a variável `app` contendo o objeto da classe `Flask` instanciado lá em cima com todas as configurações realizadas e prontas para serem executadas no navegador. Esse retorno será usado no próximo arquivo `run.py`, onde ele receberá o valor retornado de `app` para rodar a aplicação com todos os valores já inicializados.

run.py

O arquivo `run` é bem mais simples. Nele, apenas vamos criar as configurações de execução da aplicação baseado nos arquivos `app` e `config`. Escreva o código a seguir em seu arquivo `run.py` e adiante vamos entendê-lo.

run.py

```
import sys
from app import create_app
from config import app_config, app_active

config = app_config[app_active]
config.APP = create_app(app_active)

if __name__ == '__main__':
    config.APP.run(host=config.IP_HOST, port=config.PORT_HOST)
    reload(sys)
```

No código do exemplo, temos a importação da `def create_app`, que criamos no arquivo `app.py`. No final do tópico sobre o arquivo `app.py` falamos que ele retorna a variável `app`. Aqui no arquivo `run.py`,

esse valor é atribuído ao objeto `config` em sua constante `APP`. Após essa atribuição, poderemos utilizá-lo em qualquer parte do sistema.

A `def __main__` é responsável por rodar a `def run` que será responsável por chamar todas as outras `defs` que serão executadas dentro do projeto.

O `reload` é utilizado para atualizar dados em tempo de execução no programa, assim o sistema sempre estará atualizado com as modificações.

Com todos esses arquivos criados e configurados, nossa aplicação está pronta para seu primeiro `run`. Vamos realizar alguns passos fundamentais para que nossa aplicação rode corretamente.

2.5 Nosso primeiro run

Arquivo de configuração

Todas as vezes que você abrir um novo terminal para rodar a aplicação será necessário dizer ao Python em qual ambiente você deseja trabalhar. Nosso `config.py` tem três ambientes, vamos ver um trecho do código a seguir para lembrar.

```
...
app_config = {
    'development': DevelopmentConfig(),
    'testing': TestingConfig(),
    'production': ProductionConfig()
}

app_active = os.getenv('FLASK_ENV')
```

Dependendo do que escolhermos, ele vai chamar uma das subclasses, isto é, ou a de `development`, OU `testing`, OU `production`.

No final do arquivo `config.py` nós setamos a variável `app_active` com o valor `os.getenv('FLASK_ENV')`. Isso significa que o `app_active` receberá o valor da variável de ambiente chamada `FLASK_ENV`. Rode o comando a seguir para dizer ao Python qual subclasse de configuração usar.

```
tiago_luiz@ubuntu:~/livro_flask$ export FLASK_ENV=development
```

Rodando o comando run

Agora que tudo está instalado e configurado, nosso projeto está pronto para rodar. Rode o comando a seguir, que diz ao Python para que ele rode o arquivo `run.py`.

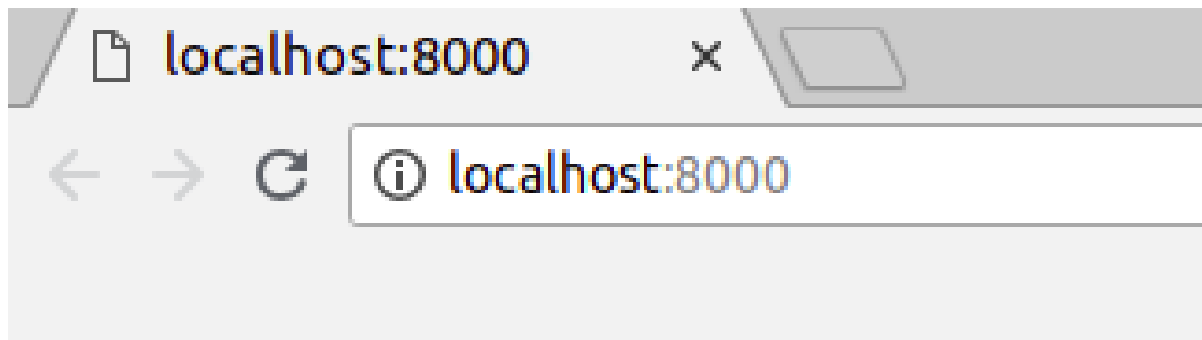
```
(venv) tiago_luiz@ubuntu:~/livro_flask$ python run.py
```

Você verá em seu terminal algo semelhante ao que está adiante:

```
(venv) tiago_luiz@ubuntu:~/livro_flask$ python run.py
* Serving Flask app "app" (lazy loading)
* Environment: development
* Debug mode: on
* Running on http://localhost:8000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 302-072-687
```

Essa é a saída que aparecerá no terminal da aplicação. Nela, teremos todos os dados principais sobre nosso App, como o ambiente em que ela está rodando (`development`, `testing` ou `production`), se está com o modo debug ativado (`Debugger is active!`), a URL onde a aplicação está rodando (`Running on http://localhost:8000/`) entre outras informações que podem ser importantes no decorrer do desenvolvimento do projeto.

Abra no browser a URL que está sendo exibida na saída do terminal da aplicação e veja se aparecerá a seguinte página.



Meu primeiro run

Figura 2.1: Browser rodando o primeiro run

Conclusão

Na próxima parte começaremos a estruturar nosso projeto de modo mais avançado. Guarde esses passos, pois eles são fundamentais e serão usados muitas vezes durante o desenvolvimento.

Estrutura do projeto - Padrão MVC

Nesta etapa, nosso projeto já estará configurado e pronto para ser desenvolvido.

O Python não exige que uma arquitetura específica seja seguida para funcionar, mas costumamos dizer que a boa prática é sempre bem-vinda. Para nosso projeto, escolhi trabalhar com os conceitos de `Model`, `View` e `Controller`, mais conhecido como padrão MVC, um dos mais utilizados no mundo inteiro.

Aqui, começaremos a criar os itens principais, que são os `Models`, `Views`, `Controllers` e `Routes`. Você pode ter estranhado que mais um elemento foi citado, as `routes`, mas em breve você entenderá o motivo.

Além desse rico e poderoso padrão, vamos aprender também sobre o `SQLAlchemy` e o `flask migrate`, ferramentas que auxiliarão bastante nosso processo de desenvolvimento.

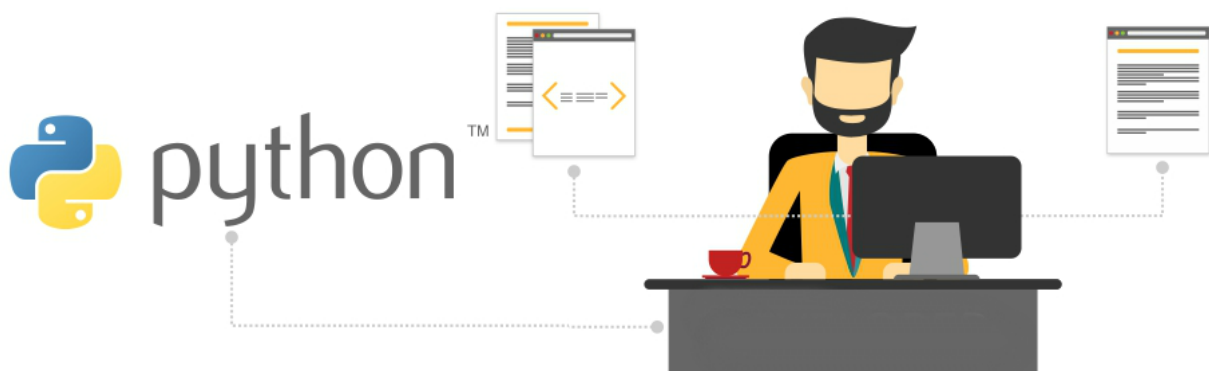


Figura 1: Estrutura principal do projeto

CAPÍTULO 3

Trabalhando com Models

As `models` são os arquivos de um sistema que refletem a estrutura exata do banco de dados. Com elas podemos criar no arquivo exatamente o que desejamos que exista em nosso banco, e todas as vezes em que modificarmos esta estrutura no arquivo, fazemos o que é chamado de `migrate` em nossa base para realizar as atualizações.

Migrate é um conceito que se tornou ferramenta em várias linguagens, fique atento ao fato de que ele é muito além de um comando. No Flask, por exemplo, temos vários comandos no `migrate` para realizar tarefas na base de dados.

3.1 Banco de dados

Primeiramente, precisamos criar nosso banco de dados e colocar suas configurações no arquivo `config.py`. Como o foco do livro não é abordar SQL aqui, é importante que você use o código a seguir para criar o banco de dados que usaremos no projeto.

```
CREATE DATABASE livro_flask CHARACTER SET UTF8 collate utf8_general_ci
```

Agora vá até o arquivo `config.py` e adicione a constante `SQLALCHEMY_DATABASE_URI` abaixo da constante `APP`, como no exemplo a seguir:

`config.py`

```
import os
import random, string

class Config(object):
    CSRF_ENABLED = True
    SECRET = 'ysb_92=qe#djf8%ng+a*#4rt#5%3*4k5%i2bck*gn@w3@f&-&'
    TEMPLATE_FOLDER =
os.path.join(os.path.dirname(os.path.abspath(__file__)), 'templates')
```

```

ROOT_DIR = os.path.dirname(os.path.abspath(__file__))
APP = None
SQLALCHEMY_DATABASE_URI =
'mysql+mysqldb://user:passwd@host:port/database'
#Preencha com os dados do seu banco de dados
# User - Usuário do banco
# Passwd - Senha do usuário
# Host - Geralmente no local fica localhost
# Port - Geralmente 3306 no mysql
# Database - Nome do banco de dados
...

```

Observe que você terá uma configuração de banco para cada ambiente, `production`, `testing` e `development`. No caso do livro deixaremos essa constante na classe `Config`, mas em um projeto é correto adicioná-la nas subclasses `production`, `testing` e `development`, para que haja uma configuração para cada ambiente.

A linha que adicionamos possui os dados de conexão ao seu banco de dados. Mais à frente ela conectará o banco à aplicação Flask.

No arquivo `app.py` também precisaremos adicionar algumas linhas a mais. Veja a seguir o que mudou e altere em seu arquivo:

```

# -*- coding: utf-8 -*-
from flask import Flask

# config import
from config import app_config, app_active
config = app_config[app_active]

# Adicionar essa linha
from flask_sqlalchemy import SQLAlchemy

## Até aqui
def create_app(config_name):
...

```

```

app.config.from_pyfile('config.py')
# Adicionar essa linha
app.config['SQLALCHEMY_DATABASE_URI'] = config.SQLALCHEMY_DATABASE_URI
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
db = SQLAlchemy(config.APP)
db.init_app(app)
# Até aqui
...

```

Não se assuste se a aplicação disser que o módulo não foi encontrado, já vamos instalá-lo.

As primeiras linhas adicionadas foram as da importação da `lib` do `SQLAlchemy` e da instância `db = SQLAlchemy(config.APP)`. Elas ficam fora da `def create_app` e trabalham aqui da mesma maneira que nas `models`.

Dentro da `def create_app`, atribuiremos ao `app.config['SQLALCHEMY_DATABASE_URI']` a URL de conexão do banco de dados que configuramos no arquivo `config.py`, bem como as configurações de *track* de modificações do banco de dados, para que o `migrate` e a aplicação detectem quando houver mudanças no banco que vierem de outros lugares que não sejam da própria aplicação.

Agora precisamos instalar as `libs` do Python que realizarão a conexão com o nosso banco de dados MySQL. Rode os seguintes comandos para instalar o `python3-dev`, `libmysqlclient-dev`, `mysqlclient` e `MySQLdb` para a nossa aplicação Flask funcionar corretamente.

1. Esse comando instalará os componentes do Python necessários para abrimos uma conexão com o banco de dados MySQL :

```

(venv) tiago_luiz@ubuntu:~/livro_flask$ sudo apt-get install
python3-dev libmysqlclient-dev

```

2. Esse comando instalará a lib `mysqlclient` que existe para o Python:

```
(venv) tiago_luiz@ubuntu:~/livro_flask$ pip install mysqlclient
```

3. E por fim essa lib é responsável por permitir que executemos os comandos `MySQL` em uma aplicação Python:

```
(venv) tiago_luiz@ubuntu:~/livro_flask$ pip install flask-mysqldb
```

Fique atento, pois o `MySQLdb` foi depreciado a partir da versão 3 do Python e a única maneira de instalá-lo no Python 3 é pela lib `flask-mysqldb`. Para outras aplicações é utilizado o `pymysql`.

3.2 Criando a estrutura da Model

Vamos escrever a estrutura que desejamos para o banco de dados em nossos arquivos `User.py`, `Role.py`, `Category.py` e `Product.py` da pasta `model`. Faremos isso de modo bem simples. Vamos começar criando o arquivo `User.py` e depois falaremos sobre os elementos que estão escritos nele.

User.py

```
# -*- coding: utf-8 -*-
from flask_sqlalchemy import SQLAlchemy
from config import app_config, app_active
from model.Role import Role

config = app_config[app_active]

db = SQLAlchemy(config.APP)

class User(db.Model):
    id=db.Column(db.Integer,primary_key=True)
    username=db.Column(db.String(40),unique=True,nullable=False)
```

```
email=db.Column(db.String(120),unique=True,nullable=False)
password=db.Column(db.String(80),nullable=False)

date_created=db.Column(db.DateTime(6),default=db.func.current_timestamp(),
nullable=False)

last_update=db.Column(db.DateTime(6),onupdate=db.func.current_timestamp(),
nullable=True)
recovery_code=db.Column(db.String(200),nullable=True)
active=db.Column(db.Boolean(),default=1,nullable=True)
role=db.Column(db.Integer,db.ForeignKey(Role.id),nullable=False)
```

Agora vamos entender o que criamos no exemplo anterior:

imports

A primeira linha do arquivo importa o módulo `SQLAlchemy` de dentro da biblioteca `flask_sqlalchemy`. Talvez você esteja estranhando o fato de não termos instalado a biblioteca, mas ela já foi instalada junto ao Flask quando executamos seu comando de instalação.

Após isso, importamos a `app_config` com as configurações necessárias, de acordo com o ambiente que escolhemos (como visto no capítulo 2).

app_config e app_active

Como já vimos, essas variáveis serão responsáveis por trazer a subclasse específica para o ambiente que selecionarmos entre os três: `development`, `testing` e `production`. Ela será chamada aqui, pois precisamos utilizar a constante `APP` para conectar nossa `model` com o banco de dados.

SQLAlchemy()

A variável `db` recebe o objeto `SQLAlchemy` passando como parâmetro as configurações do projeto que estão dentro da constante `APP`. Agora a ferramenta `SQLAlchemy` está vinculada ao nosso projeto.

db.Model

A classe da nossa `model`, aqui no caso `User`, herda do objeto `SQLAlchemy` a classe `Model`, como podemos ver. Isso existe para dizermos ao Flask que nossa classe se representará como uma `model`, e servirá para podermos realizar o que chamamos de CRUD entre o banco e a aplicação, ou seja, com isso podemos realizar ações entre nosso sistema e o banco de dados.

db.Column()

Dentro da nossa classe, temos as variáveis que são representadas de forma pública, cada uma delas representa um campo do banco de dados. Como podemos perceber, todas elas recebem o objeto `Column` que recebe como parâmetro uma série de atributos e valores. Vamos ver uma tabela com os atributos e valores que o objeto `Column` pode receber:

Nome	Valor	Significado
Campo	Integer , String , Text , DateTime , Float , Boolean , PickleType , LargeBinary	Tipo da coluna
primary_key	True ou False	Diz que é uma <code>primary key</code> . Não é necessário passá-lo quando for <code>False</code>
unique	True ou False	Diz que é uma <code>unique key</code> . Não é necessário passá-lo quando for <code>False</code>
nullable	True ou False	Diz se o campo pode ser nulo ou não
default	Valor padrão quando nada for inserido	

Nome	Valor	Significado
<code>onupdate</code>	Um valor que deseja inserir ao atualizar o campo	Diz que valor passará a ter quando o campo for atualizado
<code>db.ForeignKey</code>		Cria relacionamento entre tabelas. Entre parênteses, deve ser passado o nome da tabela e o campo <code>db.ForeignKey(Classe.atributo)</code>

Tipos de campos da Model

Vamos ver os campos padrões que existem na classe `Model` e que são muito úteis para trabalharmos com o banco de dados.

Nome	Valor	Significado
<code>Integer</code>		Inteiro
<code>String</code>	<code>(Size)</code>	String com tamanho máximo (opcional em alguns bancos de dados)
<code>Text</code>		Texto unicode mais longo (Longtext)
<code>DateTime</code>	<code>(Size)</code>	data e hora.
<code>Float</code>		Ponto flutuante
<code>Boolean</code>		Valores booleanos
<code>PickleType</code>		Armazena um objeto Python em <i>pickled</i> (Objeto serializado)
<code>LargeBinary</code>		Armazena grandes dados binários arbitrários

Agora que sabemos tudo o que é necessário por enquanto sobre `models`, vamos reproduzir os códigos a seguir em cada arquivo

`model` correspondente. Acima do código está o nome do arquivo, que se encontra dentro da pasta `model`. Esses códigos não possuem nada a mais que não tenha sido explicado no arquivo `User.py`, por isso não precisaremos falar sobre eles.

Role.py

```
# -*- coding: utf-8 -*-
from flask_sqlalchemy import SQLAlchemy
from config import app_active, app_config

config = app_config[app_active]
db = SQLAlchemy(config.APP)

class Role(db.Model):
    id=db.Column(db.Integer,primary_key=True)
    name=db.Column(db.String(40),unique=True,nullable=False)
```

Category.py

```
# -*- coding: utf-8 -*-
from flask_sqlalchemy import SQLAlchemy
from config import app_config, app_active

config = app_config[app_active]

db = SQLAlchemy(config.APP)

class Category(db.Model):
    id=db.Column(db.Integer,primary_key=True)
    name=db.Column(db.String(20),unique=True,nullable=False)
    description=db.Column(db.Text(),nullable=False)
```

Agora repita esse processo no arquivo `Product.py` da pasta `model` com o código a seguir:

Product.py

```
# -*- coding: utf-8 -*-
from flask_sqlalchemy import SQLAlchemy
from config import app_config, app_active
```

```

from model.User import User
from model.Category import Category

config = app_config[app_active]

db = SQLAlchemy(config.APP)

class Product(db.Model):
    id=db.Column(db.Integer,primary_key=True)
    name=db.Column(db.String(20),unique=True,nullable=False)
    description=db.Column(db.Text(),nullable=False)
    qtd=db.Column(db.Integer,nullable=True,default=0)
    image=db.Column(db.Text(),nullable=True)
    price=db.Column(db.Numeric(10,2),nullable=False)

    date_created=db.Column(db.DateTime(6),default=db.func.current_timestamp(),
        nullable=False)

    last_update=db.Column(db.DateTime(6),onupdate=db.func.current_timestamp(),
        nullable=False)
    status=db.Column(db.Boolean(),default=1,nullable=True)
    user_created=db.Column(db.Integer,db.ForeignKey(User.id),nullable=False)
    category=db.Column(db.Integer,db.ForeignKey(Category.id),nullable=False)

```

Em nossa próxima etapa, começaremos a realizar a migração da estrutura do banco de dados que queremos para o MySQL .

3.3 Instalando o Flask Migrate

Agora é o momento em que faremos a migração da estrutura do banco de dados criado no Flask para a nossa base de dados.

Rode o comando a seguir para instalar o Flask-Migrate em seu projeto (não se esqueça de estar com o virtualenv ativado).

```
(venv) tiago_luiz@ubuntu:~/livro_flask$ pip install Flask-Migrate
```

Agora rode o próximo comando para instalar o componente `Flask-Script`, que auxiliará a criação das migrações do banco.

```
(venv) tiago_luiz@ubuntu:~/livro_flask$ pip install Flask-Script
```

Configurando o `migrate.py`

Com tudo isso feito, vamos agora editar nosso arquivo `migrate.py`, que será responsável por replicar no banco de dados MySQL exatamente a mesma estrutura que criamos em nossas `models`. Abra o arquivo `migrate.py` e escreva o código a seguir.

`migrate.py`

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_script import Manager
from flask_migrate import Migrate, MigrateCommand
from config import app_active, app_config
config = app_config[app_active]

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = config.SQLALCHEMY_DATABASE_URI
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

db = SQLAlchemy(app)
migrate = Migrate(app, db)

manager = Manager(app)
manager.add_command('db', MigrateCommand)

class Role(db.Model):
    id=db.Column(db.Integer,primary_key=True)
    name=db.Column(db.String(40),unique=True, nullable=False)

class User(db.Model):
    id=db.Column(db.Integer,primary_key=True)
    username=db.Column(db.String(40),unique=True,nullable=False)
    email=db.Column(db.String(120),unique=True,nullable=False)
    password=db.Column(db.String(80),nullable=False)
```

```
date_created=db.Column(db.DateTime(6),default=db.func.current_timestamp(),
nullable=False)
```

```
last_update=db.Column(db.DateTime(6),onupdate=db.func.current_timestamp(),
nullable=True)
```

```
recovery_code=db.Column(db.String(200),nullable=True)
```

```
active=db.Column(db.Boolean(),default=1,nullable=True)
```

```
role=db.Column(db.Integer,db.ForeignKey(Role.id),nullable=False)
```

```
class Category(db.Model):
```

```
    id=db.Column(db.Integer,primary_key=True)
```

```
    name=db.Column(db.String(20),unique=True,nullable=False)
```

```
    description=db.Column(db.Text(),nullable=True)
```

```
class Product(db.Model):
```

```
    id=db.Column(db.Integer,primary_key=True)
```

```
    name=db.Column(db.String(20),unique=True,nullable=False)
```

```
    description=db.Column(db.Text(),nullable=False)
```

```
    qtd=db.Column(db.Integer,nullable=True,default=0)
```

```
    image=db.Column(db.Text(),nullable=True)
```

```
    price=db.Column(db.Numeric(10,2),nullable=False)
```

```
date_created=db.Column(db.DateTime(6),default=db.func.current_timestamp(),
nullable=False)
```

```
last_update=db.Column(db.DateTime(6),onupdate=db.func.current_timestamp(),
nullable=True)
```

```
    status=db.Column(db.Integer,default=1,nullable=True)
```

```
    user_created=db.Column(db.Integer,db.ForeignKey(User.id),nullable=False)
```

```
    category=db.Column(db.Integer,db.ForeignKey(Category.id),nullable=False)
```

```
if __name__ == '__main__':
```

```
    manager.run()
```

Perceba que esse arquivo recebe junto ao SQLAlchemy o objeto Migrate , que realiza a migração das tabelas que colocarmos dentro do arquivo. Cada classe representa uma tabela no banco MySQL.

Perceba que o `migrate` é um arquivo separado. Ele não será executado com o projeto, mas apenas para realizar modificações na estrutura do banco de dados. Todas as vezes em que uma nova `model` for criada, a estrutura dela deverá ser replicada abaixo das tabelas já adicionadas ao `migrate` e, se uma tabela for modificada, a classe que representa essa tabela também deverá ser modificada e um novo `migrate` deverá ser rodado.

Rodando o migrate

Rode em seu terminal o comando a seguir para iniciar a configuração do `migrate` em seu projeto.

```
(venv) tiago_luiz@ubuntu:~/livro_flask$ python migrate.py db init
```

Perceba que uma nova pasta foi criada em seu projeto. Ela possui uma série de configurações que o `migrate` cria, mas não se preocupe com elas, pois a ferramenta cuidará de tudo para você. Essas configurações vão gerenciar seu `migrate`, criando, editando e deletando mudanças na base de dados, conforme você for modificando seu arquivo `migrate.py`.

Veja uma estrutura resumida após o comando de `db init` do `migrate`:

```
livro_flask
├── app.py
├── config.py
├── migrate.py
├── run.py
├── +---migrations # Essa é a nova pasta.
│   └── ...
├── +---model
│   ├── Category.py
│   ├── Product.py
│   └── User.py
```

```
|         Role.py  
|  
| ...
```

Com a configuração de migração estabelecida, rode o comando a seguir para realizar o `migrate`.

```
(venv) tiago_luiz@ubuntu:~/livro_flask$ python migrate.py db migrate
```

Esse comando criará o arquivo de migração que será usado para atualizar o banco. Agora que o `migrate` preparou o arquivo, o próximo comando vai executar as modificações no banco de dados.

```
(venv) tiago_luiz@ubuntu:~/livro_flask$ python migrate.py db upgrade
```

Todas as vezes em que você desejar realizar uma mudança no banco, será necessário atualizar o arquivo `migrate.py` e rodar os comandos `db migrate` e `db upgrade`, exatamente nessa ordem. O comando `db init` não é necessário, pois ele serve apenas para iniciar a configuração do `migrate` no projeto e só será rodado na primeira vez em que o `migrate` for realizado.

Conferindo o banco de dados

Se entrarmos agora, via linha de comando em nosso banco de dados, poderemos ver a estrutura dele listada, conforme nós criamos. Então entre no banco de dados com suas credenciais, como na linha de comando a seguir (não esqueça que seu usuário e senha podem ser outros, o meu usuário é `root`).

```
(venv) tiago_luiz@ubuntu:~/livro_flask$ mysql -u root -p
```

Conectado ao MySQL, execute o próximo comando, que serve para você poder selecionar o banco de dados que deseja utilizar nesse console:

```
mysql> USE livro_flask;
```

E agora que falamos ao console que desejamos trabalhar com o banco `livro_flask`, rode o comando a seguir para listar as tabelas desse banco de dados. Se as tabelas aparecerem com os mesmos

nomes que as `models` , significa que nosso `migrate` rodou com sucesso:

```
mysql> SHOW TABLES;
```

O resultado deverá ser algo assim, refletindo exatamente a estrutura das nossas `models` :

```
mysql> SHOW TABLES;
+-----+
| Tables_in_livro_flask |
+-----+
| alembic_version       |
| category              |
| product               |
| user                  |
| role                  |
+-----+
3 rows in set (0.00 sec)
```

Conclusão

Agora temos toda a estrutura do banco de dados criada e toda a estrutura do `migrate` pronta!

Uma curiosidade é a tabela `alembic_version` , criada pelo `migrate` para controlar as versões de migração que são realizadas na base de dados. Não exclua essa tabela e não escreva nela, seus valores são modificados pelo `flask_migrate` .

CAPÍTULO 4

Trabalhando com Routes

Nosso sistema está tomando forma. Em breve ele já terá seus principais itens funcionando. Neste capítulo veremos como criar as rotas da aplicação. Elas são fundamentais para que nosso sistema funcione corretamente. Então, vamos lá.

4.1 Entendendo as rotas

O conceito de rotas em sistemas Web é bem similar ao de rotas de um mapa. Elas existem como uma maneira de se chegar a um determinado lugar. Se quisermos que os usuários cheguem até a página de login, faremos uma rota para a página de login. Se quisermos chegar até a página de "Esqueci minha senha", teremos uma rota para ela também. Vejamos os dois exemplos de rotas.

Rota para a tela de login

`https://seusistema.com/login`

Rota para a tela de esqueci minha senha

`https://seusistema.com/recovery-password`

Como acabamos de ver, as rotas são o conjunto da URL padrão do sistema mais um nome. O caminho que desejamos acessar será atrelado àquele nome, ou seja, se adicionarmos ao final de nossa URL padrão (<https://seusistema.com/>) o nome `login`, acessaremos um caminho de nosso sistema. Esse caminho na maioria das vezes é uma tela, mas também pode ser um retorno `json` ou `xml`.

4.2 Nossa primeira rota

No capítulo em que criamos o arquivo `app.py` acabamos por criar uma rota básica chamada `index`, mas não falamos a fundo sobre ela, nem aprofundamos sobre as demais possibilidades existentes nesse recurso poderoso que são as rotas.

Inicialmente, tenha em mente que nossas rotas ficarão no arquivo `app.py`, e isso é uma boa prática que costumamos cumprir nas aplicações feitas em Flask. Conhecer bem sobre rotas é fundamental para que criemos uma aplicação consistente e completa. A seguir, vamos ver a criação de nossas primeiras rotas. Veja o próximo código, que deverá ser inserido no arquivo `app.py`:

```
...
db.init_app(app)

@app.route('/')
def index():
    return 'Meu primeiro run'

# O código acima já está no arquivo. Abaixo temos as novas rotas que
devem ser inseridas.

@app.route('/login/')
def login():
    return 'Aqui entrará a tela de login'

@app.route('/recovery-password/')
def recovery_password():
    return 'Aqui entrará a tela de recuperar senha'
```

Se rodarmos nossa aplicação agora conseguiremos acessar as duas rotas pelo navegador. Vamos testar rodando o comando `python run.py`.

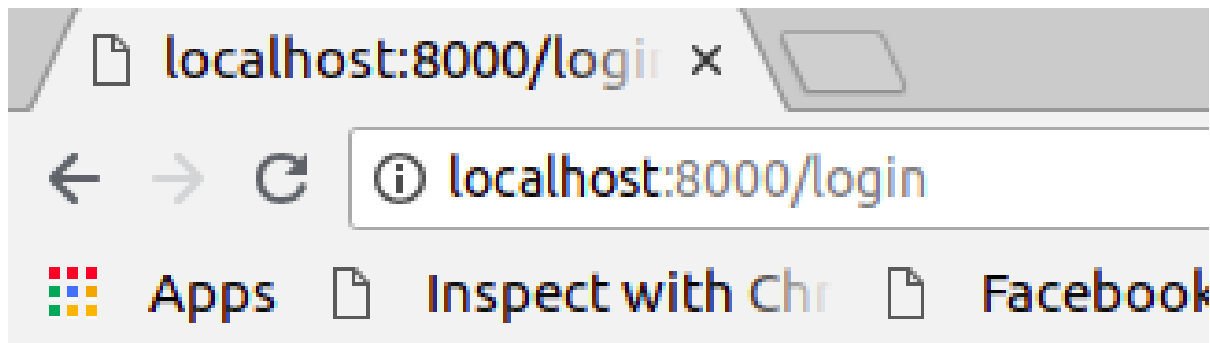
```
(venv) tiago_luiz@ubuntu:~/livro_flask$ python run.py
* Serving Flask app "app" (lazy loading)
* Environment: development
* Debug mode: on
```

- * Running on `http://localhost:8000/` (Press CTRL+C to quit)
- * Restarting with stat
- * Debugger is active!
- * Debugger PIN: 302-072-687

Não esqueça de estar com a variável de ambiente `export FLASK_ENV=development` setada e seu `virtualenv` ativado.

Agora acesse as rotas e veja o resultado:

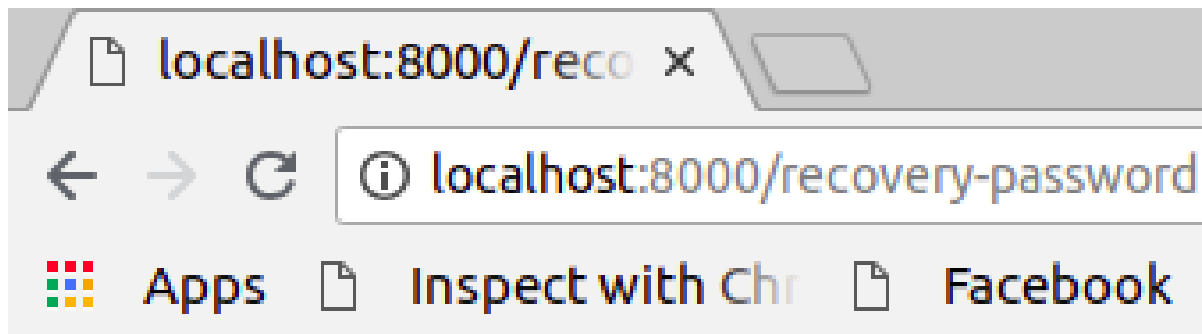
Rota <http://localhost:8000/login>



Aqui entrará a tela de login

Figura 4.1: Browser rodando a rota login

Rota <http://localhost:8000/recovery-password>



Aqui entrará a tela de recuperar senha

Figura 4.2: Browser rodando a rota recovery password

Como podemos perceber, ainda não há uma mudança efetiva nas rotas, somente um pequeno texto diferente para cada, apenas para mostrar que alternamos entre elas. Vamos falar um pouco sobre cada linha que escrevemos no código anterior.

- `@app.route` : essa é a notação que utilizamos para dizer que estamos criando uma rota. Dentro dos parênteses dessa notação precisamos colocar o nome que daremos para ela. No código anterior, temos `/login` e `/recovery_password`, desse modo, agora teremos duas novas rotas criadas. Ao rodarmos o projeto, conseguiremos acessá-las e obteremos como resultado os valores de `return` que cada rota está retornando.
- `login()` **OU** `recovery_password` : cada rota, ao ser acionada, precisa ter uma `def` que executará suas devidas ações.

Esses são basicamente os elementos de que precisamos para criar uma rota. Será dentro da `def` de uma rota que acionaremos a `controller` para realizar as ações necessárias dessa rota, ou seja, a rota de login acionará a `controller` que verificará os dados do usuário, se ele tem ou não acesso ao sistema, e retornará para a tela de login a informação do usuário. Mais à frente faremos mais ações dentro de uma rota.

4.3 Rotas dinâmicas

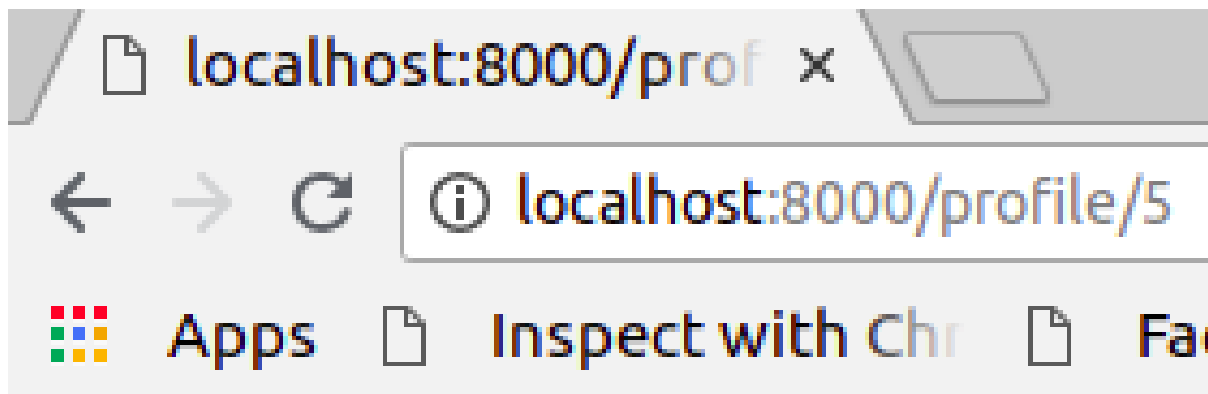
Uma rota pode também receber valores dinâmicos. Em uma tela de perfil do usuário, podemos acessar os dados daquele usuário através da rota de perfil e o id correspondente a ele. Vamos ver um exemplo; por ora veremos apenas como recuperar os dados da rota, e mais à frente faremos com que ela acesse os dados do banco.

```
...
@app.route('/recovery-password/')
def recovery_password():
    return 'Aqui entrará a tela de recuperar senha'

@app.route('/profile/<int:id>/')
def profile(id):
    return 'O ID desse usuário é %d' % id
```

Agora acesse a rota e veja o resultado:

Rota <http://localhost:8000/profile/5>



O ID desse usuário é 5

Figura 4.3: Browser rodando a rota login

Como podemos ver, na rota `profile` temos o que chamamos de `<variable_name>`, uma propriedade que permite enviarmos valores dentro de uma rota. A `variable_name` precisa ter um nome e ele

precisa ser único dentro daquela rota. Outra observação é que ela precisa ser passada como parâmetro dentro da `def` que está atrelada à rota. No caso anterior, a `def profile` terá como parâmetro `id`.

Podemos criar rotas dinâmicas maiores, por exemplo:

```
...
@app.route('/recovery-password/')
def recovery_password():
    return 'Aqui entrará a tela de recuperar senha'

@app.route('/profile/<int:id>/action/<action>/')
def profile(id, action):
    return 'Ação %s usuário de ID %d' % (action, id)
```

Agora acesse a rota e veja o resultado:

Rota <http://localhost:8000/profile/5/action/action1>

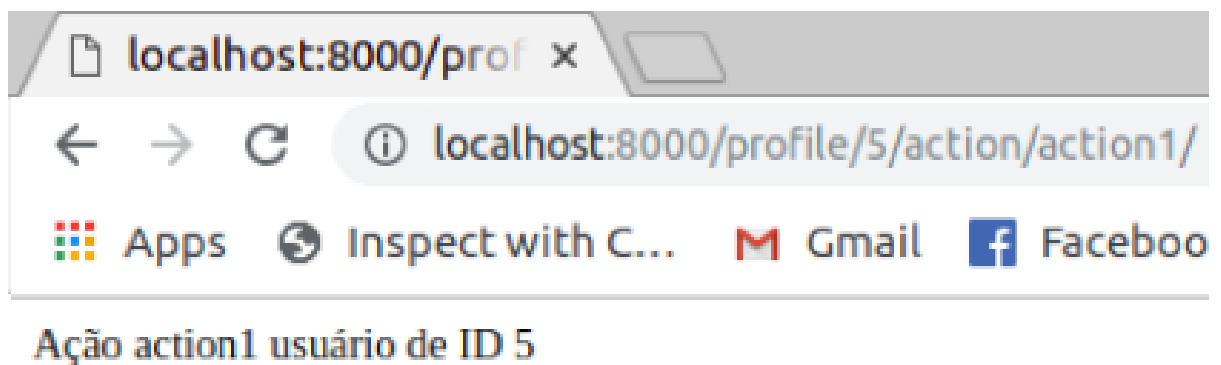


Figura 4.4: Browser rodando a rota profile com action1

Se analisarmos, a saída do dado sempre será como `string`. Se quisermos que um dado retorne como inteiro podemos adicionar um detalhe, que seria `<int:id>`, no início da `variable_name`.

Como podemos ver, é possível que uma rota tenha diversas `variable_names`, e com isso é possível trazer algumas facilidades ao

sistema. Por exemplo, em uma mesma rota podemos ter várias ações. Vamos ver o exemplo a seguir:

...

```
@app.route('/profile/<int:id>/action/<action>/')
def profile(id, action):
    if action == 'action1':
        return 'Ação action1 usuário de ID %d' % id
    elif action == 'action2':
        return 'Ação action2 usuário de ID %d' % id
    elif action == 'action3':
        return 'Ação action3 usuário de ID %d' % id
```

Agora acesse as rotas e veja o resultado:

Rota <http://localhost:8000/profile/5/action/action1>

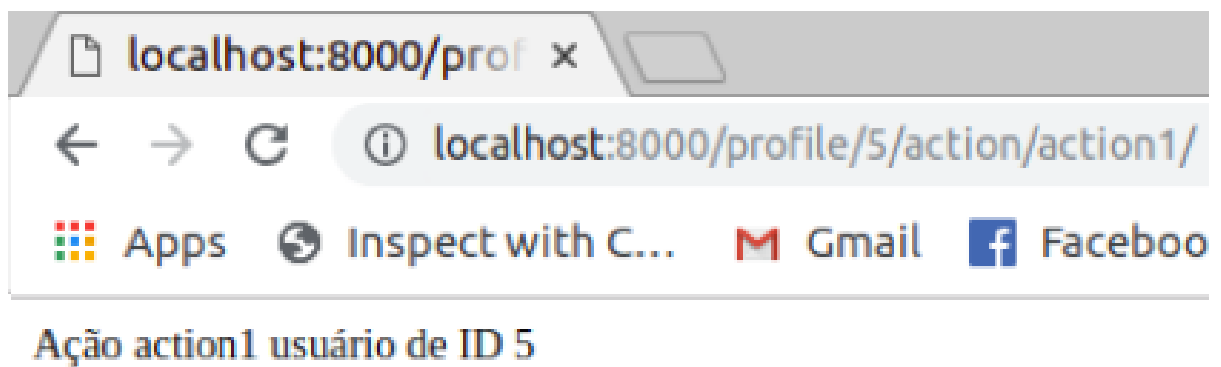


Figura 4.5: Browser rodando a rota profile com action1

Rota <http://localhost:8000/profile/5/action/action2>

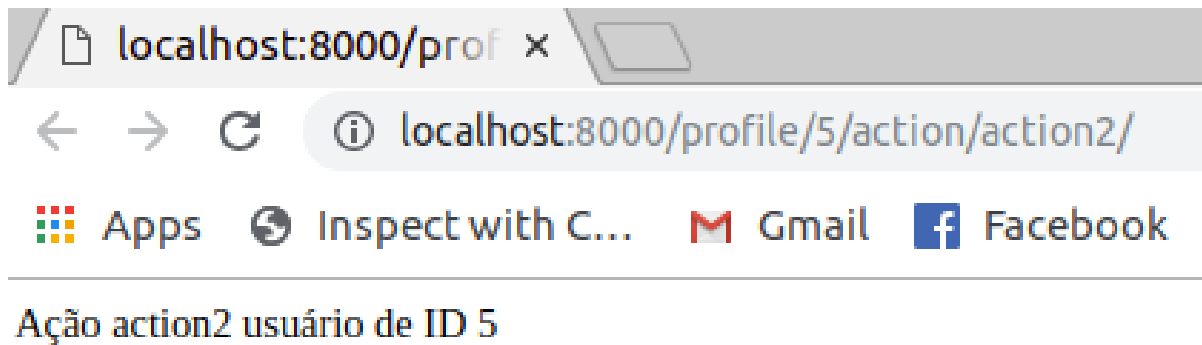


Figura 4.6: Browser rodando a rota profile com action2

Rota <http://localhost:8000/profile/5/action/action3>

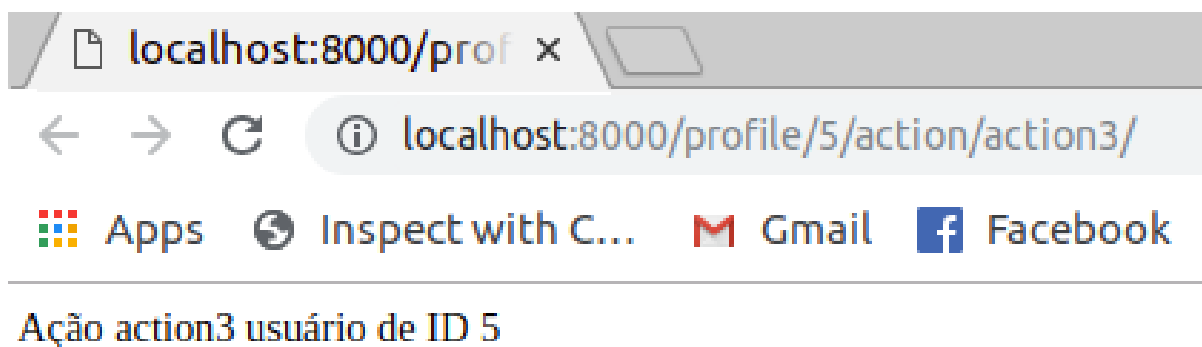


Figura 4.7: Browser rodando a rota profile com action3

Em uma mesma rota conseguimos realizar tarefas de forma dinâmica. Mais à frente veremos como criar as ações de profile em nosso sistema.

Essa última rota pode ser removida, pois apenas a utilizamos para demonstrar como podemos criar diversas `variable_names` em uma mesma rota.

Protocolo HTTP

Não podemos criar um sistema Web sem falar do protocolo HTTP, que é fundamental para que nossa aplicação faça a comunicação entre o que chamamos de `client` e `server` - que, em resumo, são o

navegador e o servidor. O navegador envia para o servidor informações que serão processadas nele e, claro, após processadas um resultado será devolvido para o navegador.

O protocolo HTTP define um conjunto de métodos de requisição que são responsáveis por indicar uma ação que será executada no servidor para uma ação específica. Podemos dar um simples exemplo, onde o navegador envia para o servidor um valor de email e password que o usuário inseriu em um formulário. O servidor processará esses valores, verificando no banco de dados se eles correspondem a algum usuário cadastrado. Caso isso ocorra, retornará um resultado ao navegador, permitindo a entrada do usuário no sistema; caso contrário, retornará um resultado dizendo que os dados estão incorretos, ou que o usuário não existe.

O protocolo HTTP possui 9 verbos, chamados assim pois servem para informar ao servidor que ação desejamos realizar. Podemos querer editar parcialmente ou por completo uma informação do sistema, listar dados, deletar, entre outras ações. O servidor precisa saber qual ação queremos realizar para tomar a decisão correta.

Vejamos a seguir os 6 principais verbos do protocolo HTTP e a descrição de cada um deles:

- **GET:** solicita a representação de um recurso específico. Estas requisições são muito utilizadas em pesquisas. A pesquisa do Google, por exemplo, retorna resultados que são requisitados por um método GET.
- **HEAD:** solicita uma resposta de forma idêntica ao método GET, porém sem conter o corpo da resposta. O GET envia na URL do navegador os dados para o servidor, e isso pode ser inseguro em algumas situações, por isso existe o método HEAD.
- **POST:** é utilizado para enviar dados ao servidor que causarão alteração no recurso requisitado. Podemos usar como exemplo a criação de um usuário, pois modificará informações no banco de dados, adicionando-o.

- **PUT**: é utilizado para atualizarmos um dado no servidor. Se quisermos modificar os dados de um usuário no banco de dados, enviaremos essas informações através de um método `PUT`.
- **DELETE**: remove um dado. Se quisermos deletar um usuário do banco de dados, enviaremos o `id` dele por meio de um método `DELETE`.
- **PATCH**: assim como o `PUT`, é utilizado para aplicar modificações de dados no servidor; porém essas alterações são parciais em um recurso, ou seja, o `PUT` é usado para modificar tudo, e o `PATCH`, apenas partes.

Olhando agora os métodos do protocolo HTTP, podemos perceber que é importante utilizá-los quando precisarmos solicitar dados do servidor ou quando precisarmos alterar dados dele. Não precisaremos usá-los sempre, somente quando necessário. No decorrer do livro veremos em quais circunstâncias convém usá-los.

Vamos ver agora alguns exemplos do uso dos métodos do protocolo HTTP. Por ora, não faremos mudanças no banco de dados, apenas executaremos as requisições HTTP para vermos como elas são recebidas em nosso `servidor`. Para fazer isso sem precisarmos ter muito trabalho, usaremos uma ferramenta incrível chamada **postman**. Ela pode ser encontrada no link a seguir, de acordo com seu sistema operacional: <https://www.getpostman.com/apps/>. Você também pode achá-lo como uma extensão do Google Chrome. O download e instalação são bem simples, por isso pularemos essa etapa.

A seguir veremos 2 exemplos da aplicação dos métodos HTTP, tendo em vista que somente os verbos mudam, mas o comportamento deles é idêntico. Confira e replique o código em seu projeto no arquivo `app.py`. Ele será de suma importância para entendermos sobre métodos HTTP.

```
# -*- coding: utf-8 -*-
from flask import Flask, request
```

```

# Importe a lib request no topo do arquivo
...

# Código já existente, que não precisa ser modificado

...
@app.route('/profile/<int:id>/action/<action>/')
def profile(id, action):
    if action == 'action1':
        return 'Ação action1 usuário de ID %d' % id
    elif action == 'action2':
        return 'Ação action2 usuário de ID %d' % id
    elif action == 'action3':
        return 'Ação action3 usuário de ID %d' % id
    # O código da def profile já existe, continue a partir dele.

@app.route('/profile', methods=['POST'])
def create_profile():
    username = request.form['username']
    password = request.form['password']

    return 'Essa rota possui um método POST e criará um usuário com os
dados de usuário %s e senha %s' % (username, password)

@app.route('/profile/<int:id>', methods=['PUT'])
def edit_total_profile(id):
    username = request.form['username']
    password = request.form['password']

    return 'Essa rota possui um método PUT e editará o nome do usuário
para %s e a senha para %s' % (username, password)

return app

```

Agora que escrevemos podemos perceber que é possível recuperarmos os valores de um formulário em nossas rotas. Para isso, precisamos apenas utilizar o método `form` da lib `request` assim `request.form['nome_do_input']`.

A `lib request` será responsável por receber todos os valores de uma requisição. Com ela podemos receber os valores de uma requisição `post` , `get` , valores de um formulário etc.

Vamos agora utilizar o **postman** para realizar os testes dessas requisições, tendo em vista que não é possível passarmos as URLs do exemplo diretamente no navegador , pois elas retornaram que o método `POST` ou `PUT` não são permitidos nele.

Método POST:

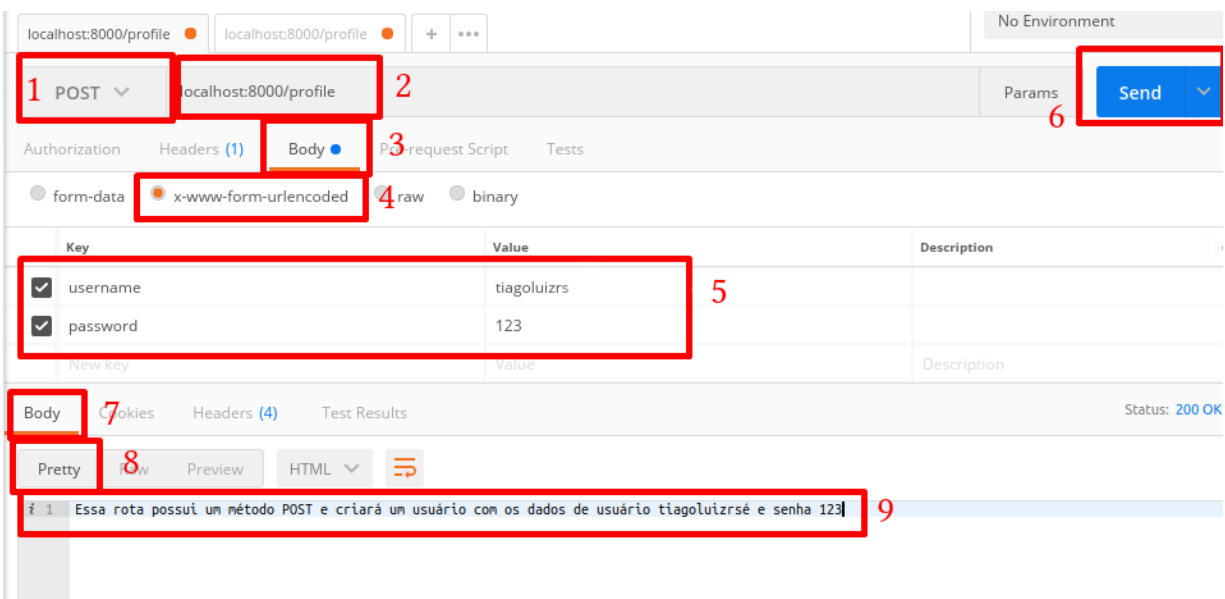


Figura 4.8: Postman executando o método POST na rota profile

Passo a passo:

1. Selecione o método que será requisitado, que nesse caso deverá ser o `POST` ;
2. Coloque a `url` correspondente à rota que você requisitará, nesse caso <http://localhost:8000/profile>;
3. Selecione a aba `body` , pois nela colocaremos os valores que simularão um formulário `HTML` ;
4. Selecione a opção `x-www-form-urlencoded` , pois ela permite que nossos dados sejam enviados no `encoding` da aplicação que no caso é `utf-8` ;

5. Adicione os dois campos que representam os campos do formulário HTML `username` e `password`, eles devem conter os mesmos nomes que estão dentro do método `request.form` que está no arquivo `app.py`;
6. Clique em `Send` para enviar a requisição ao servidor;
7. Clique na aba `Body` pois ela exibirá o retorno do servidor e o status da resposta;
8. Clique em `Pretty` para que sua resposta fique formatada com todos os caracteres corretamente e visualmente como uma página da Web;
9. Agora veja o retorno que foi enviado pelo servidor e se ele está igual ao que criamos no código do exemplo anterior.

A seguir temos uma imagem do método `PUT` que segue o mesmo modelo, tendo apenas como diferente os dois primeiros passos:

Método PUT:



Figura 4.9: Postman executando o método PUT na rota profile

Etapas diferentes do processo anterior:

1. Selecione o método `PUT`;

2. Coloque a `url` correspondente à rota que você requisitará, nesse caso <http://localhost:8000/profile/3/>

Os demais passos deverão ser seguidos da mesma maneira que o anterior.

Com isso, agora sabemos como enviar dados do navegador (`client`) para o servidor (`server`) através de formulários HTML. Em breve criaremos nossos templates com formulários HTML para realizar as modificações que nosso sistema precisa fazer no banco de dados.

Uma coisa interessante é que podemos utilizar mais de um método HTTP em uma mesma rota. Veja um exemplo a seguir:

```
"""Mesmo método do exemplo do post,
não crie um novo, apenas sobrescreva-o."""
@app.route('/profile', methods=['POST', 'GET'])
def create_profile():
    if request.method == 'POST':
        return 'Método POST sendo requisitado'
    elif request.method == 'GET':
        return 'Método GET sendo requisitado'
```

Não precisa adicionar essa rota no projeto, é apenas uma demonstração.

Veja que ao utilizar dois métodos na mesma rota precisamos verificar qual está sendo chamado no momento da requisição ao servidor e, para isso, usamos o `request.method` da `lib request` - ela será muito utilizada em nosso projeto.

Conclusão

Agora que sabemos trabalhar com as rotas, é possível fazermos a comunicação entre o navegador e o servidor da nossa aplicação. Como sabemos, as telas do sistema representarão a comunicação por parte do navegador e com os dados coletados nelas

conseguiremos enviar ações de criação, modificação e leitura através das requisições `HTTP` para nosso `servidor` .

No próximo capítulo veremos como trabalhar com as `controllers` que são responsáveis por receber os dados das rotas e salvar na base de dados através das `models` .

CAPÍTULO 5

Trabalhando com Controllers

Nosso sistema agora já possui rotas, `models` e agora precisa que criemos nossas *controllers*. Elas são fundamentais para a boa organização e solidez. Veja na imagem a seguir um diagrama que esclarece melhor a função das `controllers` :

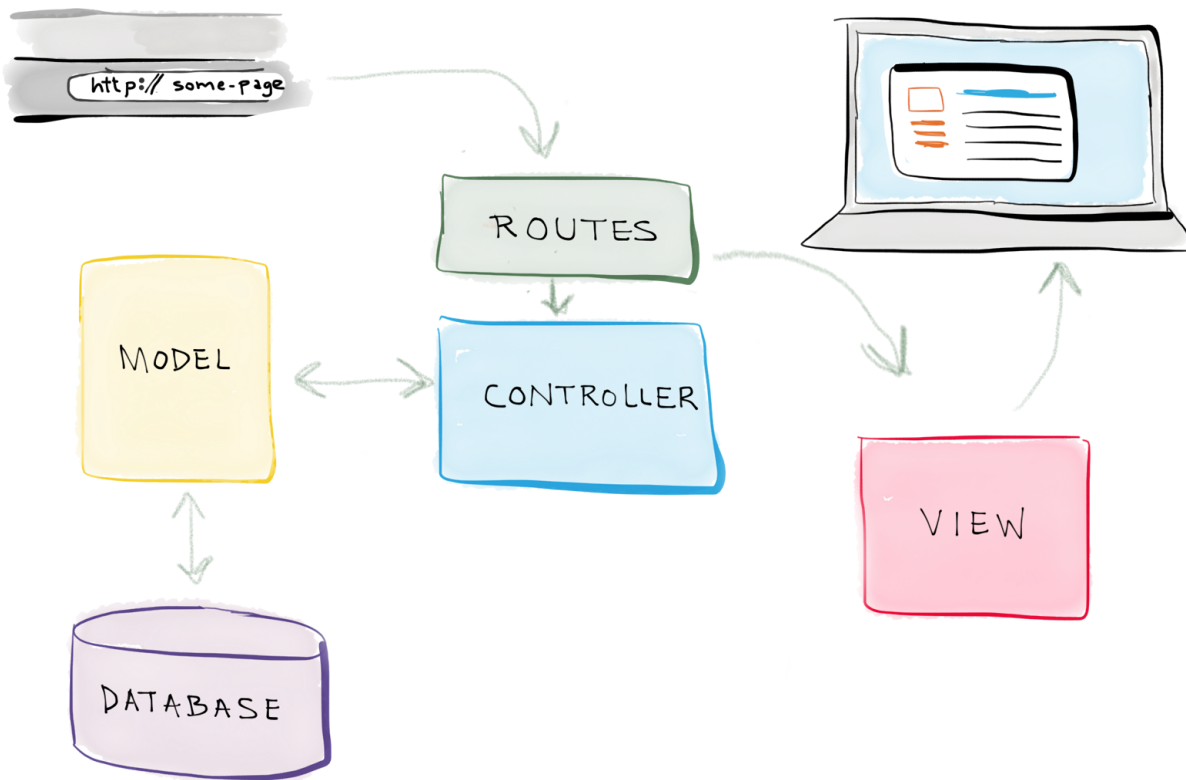


Figura 5.1: Diagrama do padrão MVC

É possível percebermos que as **rotas** recebem as requisições e através das **controllers** se comunicam com as **models**.

Neste capítulo criaremos as `controllers` que permitirão fazermos o login e a recuperação de senha do usuário.

5.1 Configurando nossa controller

Antes de começar, vamos instalar a `lib` que usaremos mais à frente para criptografia da senha de nossos usuários:

```
(venv) tiago_luiz@ubuntu:~/livro_flask$ pip install passlib[bcrypt]
```

Abra o arquivo `User.py` que está dentro da pasta `model` e incremente-o com o código a seguir:

model/User.py

```
from flask_sqlalchemy import SQLAlchemy
from config import app_active, app_config
from model.Role import Role
# Adicione a linha à seguir
from passlib.hash import pbkdf2_sha256

config = app_config[app_active]
db = SQLAlchemy(config.APP)

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(40), unique=True, nullable=False)
    email = db.Column(db.String(120), unique=True, nullable=False)
    password = db.Column(db.String(80), nullable=False)
    date_created = db.Column(db.DateTime(6),
default=db.func.current_timestamp(), nullable=False)
    last_update = db.Column(db.DateTime(6),
onupdate=db.func.current_timestamp(), nullable=False)
    recovery_code = db.Column(db.String(100), nullable=True)
    status = db.Column(db.Boolean(), default=1, nullable=False)
    role = db.Column(db.Integer, db.ForeignKey(Role.id), nullable=False)

# Adicione estes métodos
def get_user_by_email(self):
    """
    Construiremos essa função capítulos depois
    """
    return ''
```

```

def get_user_by_id(self):
    """
        Construiremos essa função capítulos depois
    """
    return ''

def update(self, obj):
    """
        Construiremos essa função capítulos depois
    """
    return ''

def hash_password(self, password):
    try:
        return pbkdf2_sha256.hash(password)
    except Exception as e:
        print("Erro ao criptografar senha %s" % e)

def set_password(self, password):
    self.password = pbkdf2_sha256.hash(password)

def verify_password(self, password_no_hash, password_database):
    try:
        return pbkdf2_sha256.verify(password_no_hash, password_database)
    except ValueError:
        return False

```

Vamos entender melhor as novas linhas adicionadas:

- `from passlib.hash import pbkdf2_sha256` : importação da lib que usaremos para criptografar as senhas dos usuários;
- `get_user_by_email()` : modelo do método que criaremos para fazer `select` no banco de dados com a cláusula `where` buscando na coluna `email` e verificar se o usuário existe ou não;
- `get_user_by_id()` : modelo do método que criaremos para fazer `select` no banco de dados com a cláusula `where` buscando na

coluna `id` para listar os dados desse usuário na tela de editar usuário;

- `update()` : modelo do método que criaremos para editar um usuário no banco de dados;
- `hash_password()` : método que usamos para pegar uma senha `string` e convertê-la para uma `string` criptografada e assim aumentarmos a segurança de nosso sistema; o método `hash_password` adicionará o método `self.pbkdf2_sha256.hash` que pertence à `lib pbkdf2_sha256` ;
- `verify_password()` : método que verificará se o `password` enviado pelo usuário é compatível com o que está no banco de dados. Isso acontecerá através do método `pbkdf2_sha256.verify` , que pertence à `lib pbkdf2_sha256` .

Agora adicionaremos o seguinte código ao arquivo `User.py` que está dentro da pasta `controller` :

controller/User.py

```
from model.User import User

class UserController():
    def __init__(self):
        self.user_model = User()

    def login(self, email, password):
        """
        Pega os dados de e-mail e salva no
        atributo da model de usuário.
        """
        self.user_model.email = email

        """
        Verifica se o usuário existe no banco de
        dados
        """
        result = self.user_model.get_user_by_email()

        """
```

```

    Caso o usuário exista o result não será
    None
    """
    if result is not None:
        """
        Verifica se o password que o usuário
        enviou, agora convertido em hash, é
        igual ao password que foi pego no
        banco de dados para esse usuário.
        """
        res = self.user_model.verify_password(password, result.password)

        # Se for o mesmo retornará True
        if res:
            return result
        else:
            return {}
    return {}

def recovery(email):
    """
    A recuperação de e-mail será criada no
    capítulo 11. Trabalhando com serviços de
    e-mail.
    """
    return ''

```

Vamos entender o código:

- `__init__` : o método construtor que receberá a instância da `model` , através do atributo `user_model` , para ser usada em qualquer método que for criado na `controller` e precise interagir com a tabela `user` do banco de dados;
- `login()` : método que será utilizado para ligar a rota de login com a `model` de usuário. Ela conterá a lógica que acionará o método de `select` da `model` e retornará ou não os dados do usuário para o arquivo `app.py` ;
- `recovery_password()` : método que será usado mais à frente para a recuperação de senha; por ora não trabalharemos com ele,

pois teremos um capítulo voltado ao serviço de envio de e-mail no Python.

Agora que temos a `controller` basicamente construída, vamos editar nosso arquivo `app.py` e chamar os métodos `login_post()` e `send_recovery_password()` dentro de cada rota correspondente. Veja o código a seguir e edite seu arquivo:

app.py

```
# -*- coding: utf-8 -*-
"""Adicione os componentes
redirect e render_template abaixo"""
from flask import Flask, request, redirect, render_template

# config import
from config import app_config, app_active

# Adicione a controller UserController
# controllers
from controller.User import UserController

config = app_config[app_active]

from flask_sqlalchemy import SQLAlchemy

def create_app(config_name):
    app = Flask(__name__, template_folder='templates')

    # Existe mais código aqui
    ...
    # Existe mais código aqui

    @app.route('/login/')
    def login():
        return 'Aqui entrará a tela de login'

    @app.route('/login/', methods=['POST'])
    def login_post():
        user = UserController()
```

```

email = request.form['email']
password = request.form['password']

result = user.login(email, password)

if result:
    return redirect('/admin')
else:
    return render_template('login.html', data={'status': 401, 'msg':
'Dados de usuário incorretos', 'type': None})

@app.route('/recovery-password/')
def recovery_password():
    return 'Aqui entrará a tela de recuperar senha'

@app.route('/recovery-password/', methods=['POST'])
def send_recovery_password():
    user = UserController()

    result = user.recovery(request.form['email'])

    if result:
        return render_template('recovery.html', data={'status': 200, 'msg':
'E-mail de recuperação enviado com sucesso'})
    else:
        return render_template('recovery.html', data={'status': 401, 'msg':
'Erro ao enviar e-mail de recuperação'})

return app

```

Não se preocupe com os arquivos `login.html` e `recovery.html`, em breve vamos editá-los.

Agora que editamos o arquivo `app.py`, vamos entender melhor as linhas de código novas que foram adicionadas:

- `from controller.User import UserController` : essa é a linha em que importamos o arquivo de `controller` que usaremos para a interação entre as rotas e as `models` do usuário;
- `user.login` : método que chamamos da `controller user` para verificar se os dados do usuário estão corretos ou não;
- `user.recovery` : método que chamamos da `controller user` para enviar um e-mail de recuperação de senha para o usuário;
- `redirect` : método utilizado para redirecionar o usuário para outra rota, nesse caso, para a rota de admin que ainda não foi criada (veremos isso no próximo capítulo);
- `render_template` : método utilizado para renderizar um template `html` (veremos mais a fundo no capítulo 8, sobre views).

Como você pode ver, eu removi as rotas de `profile` da `app.py`, pois eram apenas exemplos para entendermos melhor sobre rotas, e não precisaremos delas daqui em diante.

Conclusão

Trabalhar com `controllers` é relativamente simples como pudemos ver. Ela não é uma ferramenta ou algo específico do Flask que demande a instalação de uma `lib`, por exemplo, ela é apenas um conceito que mantém a organização do nosso projeto. Se pararmos para analisar, é possível pegar os métodos que estão na `controller` e criá-los diretamente na rota, mas isso não é algo recomendável, pois com as `controllers` podemos reaproveitar os métodos em outras rotas.

Este foi um capítulo mais simples, porém as `controllers` serão utilizadas constantemente nos próximos capítulos, principalmente na criação de API e na área de administração do sistema.

CAPÍTULO 6

Área administrativa no Flask

Até aqui, já temos o conhecimento necessário em `models`, rotas e `controllers` para avançar o nível do nosso sistema de forma bem mais profissional.

A criação de uma área administrativa no Flask é simples e completa. Ele nos permite criar algo seguro e pronto para usarmos alinhado com nossas regras de negócio. Então vamos começar.

6.1 Configurando nosso admin

Primeiro, é necessário instalarmos a `lib` do `flask_admin`. Rode o comando a seguir:

```
(venv) tiago_luiz@ubuntu:~/livro_flask$ pip install flask-admin
```

Agora que temos a `lib` instalada, vamos modificar o arquivo `app.py` para ativamos a área administrativa do Flask.

app.py

```
# -*- coding: utf-8 -*-
from flask import Flask, request, redirect, render_template

# config import
from config import app_config, app_active

"""Adicione a seguir o import do arquivo Admin.py"""
# admin
from admin.Admin import start_views

# controllers
from controller.User import UserController
```

```

config = app_config[app_active]

from flask_sqlalchemy import SQLAlchemy

def create_app(config_name):
    app = Flask(__name__, template_folder='templates')

    app.secret_key = config.SECRET
    app.config.from_object(app_config[config_name])
    app.config.from_pyfile('config.py')
    app.config['SQLALCHEMY_DATABASE_URI'] = config.SQLALCHEMY_DATABASE_URI
    app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
    db = SQLAlchemy(config.APP)

    """
    Adicione a linha a seguir em seu arquivo
    """

    start_views(app,db)
    """Até aqui"""

    db.init_app(app)
    ...

```

Vamos entender o que cada linha significa:

- `from admin.admin import start_views` : aqui, nós estamos importando o método `start_views` de dentro do arquivo `admin.py` , que está na pasta `admin` ;
- `start_views(app,db)` : este é o método responsável por ativar a área administrativa do Flask, ele ainda não existe, mas a seguir vamos criá-lo dentro do arquivo `Admin.py` .

Agora, no arquivo `Admin.py` , adicione o seguinte código:

Admin.py

```

# -*- coding: utf-8 -*-
from flask_admin import Admin
from flask_admin.contrib.sqla import ModelView

```

```

from model.Role import Role
from model.User import User
from model.Category import Category
from model.Product import Product

def start_views(app, db):
    admin = Admin(app, name='Meu Estoque', template_mode='bootstrap3')

    admin.add_view(ModelView(Role, db.session, "Funções",
category="Usuários"))
    admin.add_view(ModelView(User, db.session, "Usuários",
category="Usuários"))
    admin.add_view(ModelView(Category, db.session, 'Categorias',
category="Produtos"))
    admin.add_view(ModelView(Product, db.session, "Produtos",
category="Produtos"))

```

Vamos entender o que cada linha faz:

- `from flask_admin import Admin`: é a lib do Flask que utilizamos para criar a área administrativa do nosso sistema. Ela possui vários recursos para criação de `views` customizadas para nosso sistema;
- `from flask_admin.contrib.sqla import ModelView`: é um componente que existe dentro da lib do `flask_admin`, que utilizaremos para ativar as `views` em nosso sistema, baseadas em nossas `models`;
- `admin = Admin(app, name='Meu Estoque', template_mode='bootstrap3')`: este é o momento do código em que chamamos o método construtor do `Admin`, que será responsável pelas configurações básicas do nosso sistema, como nome, template, página inicial e diversas outras configurações que veremos mais à frente;
- `admin.add_view`: é o método que usamos para criar uma `view` em nosso sistema admin;
- `ModelView`: é um recurso do `flask_admin` que permite que criemos em nosso admin as telas do administrador baseadas na estrutura de nossa `model` - você verá um exemplo adiante;

- `category` : utilizamos esse atributo para que mais de um item fique junto no menu do admin. O que fizemos no código anterior fará com que haja um menu chamado `Produtos` e dentro dele haverá um link para a página de `Categorias` e outro para a de `Produtos`.

Agora que tudo está configurado, rode seu projeto para vermos como funciona a área administrativa.

```
(venv) tiago_luiz@ubuntu:~/livro_flask$ python run.py
* Serving Flask app "app" (lazy loading)
* Environment: development
* Debug mode: on
* Running on http://localhost:8000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 302-072-687
```

Entre na URL <http://localhost:8000/admin> e você verá um painel parecido com o da imagem a seguir:

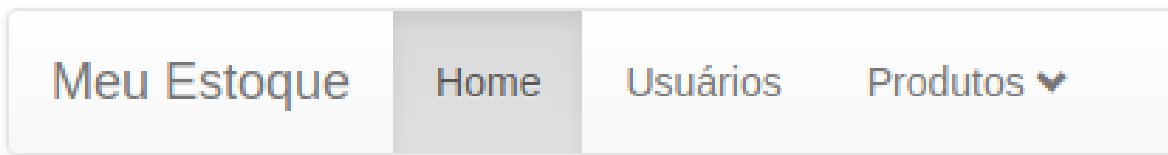


Figura 6.1: Primeiro run do admin

Relacionamento no painel admin

Se entrarmos na tela de criação de categoria, poderemos criar um novo item. Isso é fundamental, pois sem ele não conseguiríamos criar nenhum produto, tendo em vista que cada produto precisa ter uma categoria atrelada a si, afinal criamos uma `foreign key` de categoria na tabela de produtos. Isso também se aplica ao usuário, que só poderá ser criado se houver uma `Role` atrelada a si, então,

antes de criar um usuário, precisamos criar uma função lá no painel de `roles` do nosso admin. Vamos fazer isso. Entre no admin e crie uma categoria, similar à do exemplo a seguir.

1. Clique no menu `Produtos > Categorias` :

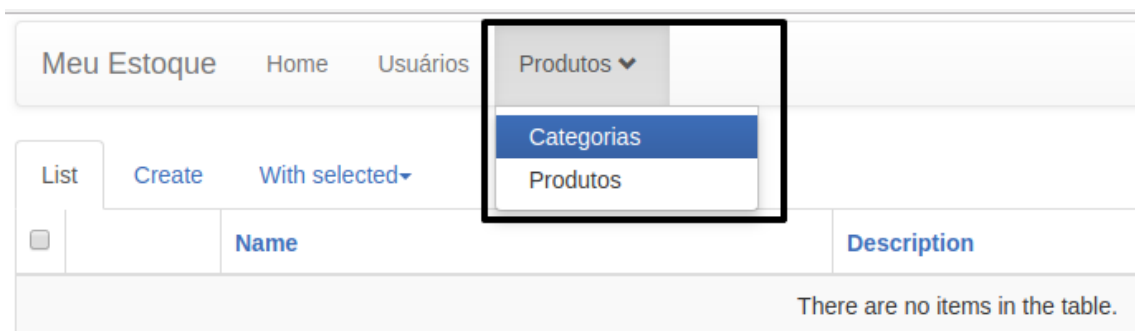


Figura 6.2: Botão categorias

2. Clique no botão `create` :

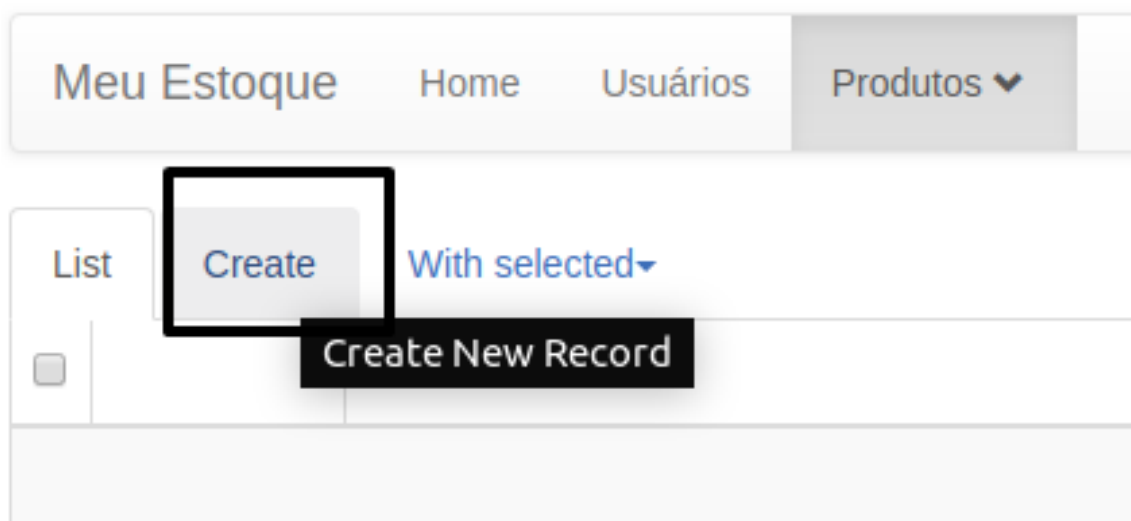
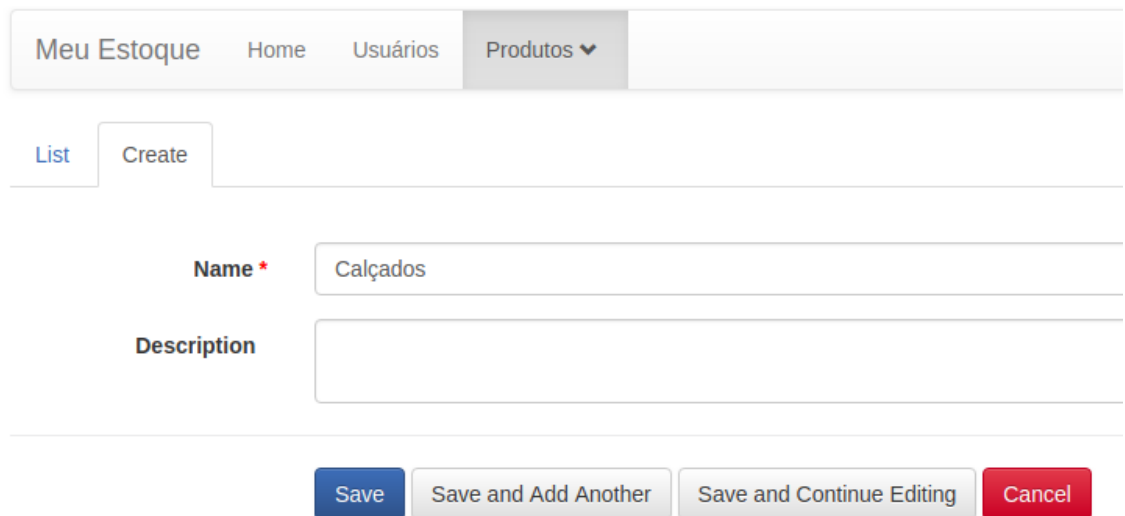


Figura 6.3: Botão criar

3. Preencha como preferir e clique em `save` :



Meu Estoque Home Usuários **Produtos ▼**

List Create

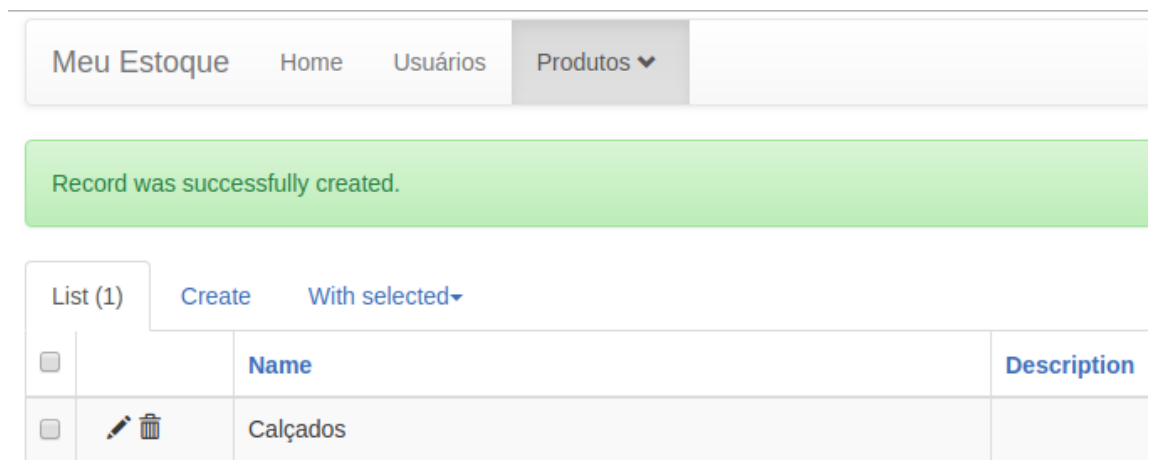
Name * Calçados

Description

Save Save and Add Another Save and Continue Editing Cancel

Figura 6.4: Formulário preenchido

4. Produto criado com sucesso!



Meu Estoque Home Usuários **Produtos ▼**

Record was successfully created.

List (1) Create With selected▼



		Name	Description
<input type="checkbox"/>	 	Calçados	

Figura 6.5: Categoria criada

Agora temos uma categoria criada, mas se entrarmos na tela de produtos e tentarmos criar um produto, veremos que o campo de categoria ainda não aparece para nós e nem o usuário que criará o produto. Veja:

Meu Estoque

Home

Usuários

Produtos ▾

List

Create

Name *

Description *

Qty

0

Image

Price *

Date Created

Last Update *

Status

1

Save

Save and Add Another

Save and Continue Editing

Cancel

Figura 6.6: Tela de criação de produto

Isso acontece porque precisamos realizar uma pequena configuração em nossos arquivos `Product.py` , `Category.py` e `User.py` que estão na pasta `model` . Abra os arquivos e adicione as modificações a seguir:

Product.py

```
# -*- coding: utf-8 -*-
from flask_sqlalchemy import SQLAlchemy
"""Adicione o item a seguir em seu código"""
from sqlalchemy.orm import relationship
from config import app_active, app_config
```

```

from model.User import User
from model.Category import Category

config = app_config[app_active]
db = SQLAlchemy(config.APP)

class Product(db.Model):
    id=db.Column(db.Integer,primary_key=True)
    name=db.Column(db.String(20),unique=True,nullable=False)
    description=db.Column(db.Text(),nullable=False)
    qtd=db.Column(db.Integer,nullable=True, default=0)
    image=db.Column(db.Text(),nullable=True)
    price=db.Column(db.Numeric(10,2),nullable=False)

    date_created=db.Column(db.DateTime(6),default=db.func.current_timestamp(),
        nullable=False)

    last_update=db.Column(db.DateTime(6),onupdate=db.func.current_timestamp(),
        nullable=True)
    status=db.Column(db.Integer,default=1,nullable=False)
    user_created=db.Column(db.Integer,db.ForeignKey(User.id),nullable=True)
    category=db.Column(db.Integer,db.ForeignKey(Category.id),nullable=True)
    """Adicione o item a seguir em seu código"""
    usuario=relationship(User)
    categoria=relationship(Category)

```

Category.py

```

# -*- coding: utf-8 -*-
from flask_sqlalchemy import SQLAlchemy
from config import app_active, app_config
from model.User import User

config = app_config[app_active]
db = SQLAlchemy(config.APP)

class Category(db.Model):
    id=db.Column(db.Integer,primary_key=True)
    name=db.Column(db.String(20),unique=True,nullable=False)
    description=db.Column(db.Text(),nullable=True)

```



```

"""Adicione os itens a seguir em seu código"""
def __repr__(self):
    return self.name

```

User.py

```

# -*- coding: utf-8 -*-
from flask_sqlalchemy import SQLAlchemy
from config import app_active, app_config
from passlib.hash import pbkdf2_sha256

config = app_config[app_active]
db = SQLAlchemy(config.APP)

class User(db.Model):
    id=db.Column(db.Integer,primary_key=True)
    username=db.Column(db.String(40),unique=True,nullable=False)
    email=db.Column(db.String(120),unique=True,nullable=False)
    password=db.Column(db.String(200),nullable=False)

    date_created=db.Column(db.DateTime(6),default=db.func.current_timestamp(),
        nullable=False)

    last_update=db.Column(db.DateTime(6),onupdate=db.func.current_timestamp(),
        nullable=False)
    recovery_code=db.Column(db.String(100),nullable=True)
    active=db.Column(db.Boolean(),default=1,nullable=False)

    """Adicione os itens a seguir em seu código"""
    def __repr__(self):
        return '%s - %s' % (self.id, self.username)
    ...

```

Vamos entender o que foi adicionado:

- `from sqlalchemy.orm import relationship`: é a lib de relacionamento que o `sqlalchemy` possui, será ela que permitirá

- que haja relacionamento em nosso painel de admin;
- `relationship(Category)` : é o método que importamos da `lib` que citamos no item anterior. Ele criará um campo no formulário de criação e edição, que representará o elemento que faz relacionamento com aquela tabela. Em nosso caso, esse método vai criar um campo de `select` , que listará todas as categorias, pois elas fazem relacionamento com a tabela de produtos;
 - `__repr__` : é o método que retorna o identificador que será usado nos `selects` que serão exibidos no admin se referindo à `Foreign key` . Quando não criamos o método, cada item criado será exibido nesse modelo `<Category 1>` , `<User 1>` , mas com o método, ao termos um `select` de usuários por exemplo, aparecerá o nome do usuário que é bem mais legível.

Vamos ver a seguir um exemplo do `select` com e sem o método `__repr__` :

Exemplo da model `Category` sem o método `__repr__` :

Categoria

`<Category 2>`

Figura 6.7: Tela de criação de produto.

Exemplo da model `Category` com o método `__repr__` :

Categoria

Calçados

Figura 6.8: Tela de criação de produto.

Uma vez que entendemos como funciona, vamos modificar alguns pontos no arquivo `model` de usuário e de `role`, pois precisamos que o campo `role` seja exibido no painel de criação de usuários, sem uma `role` criada, não temos como criar um usuário. O processo é basicamente o mesmo. Vamos entrar no arquivo `User.py` e `Role.py` da pasta `model` e fazer as seguintes mudanças.

Role.py

```
# -*- coding: utf-8 -*-
from flask_sqlalchemy import SQLAlchemy
from config import app_active, app_config

config = app_config[app_active]
db = SQLAlchemy(config.APP)

class Role(db.Model):
    id=db.Column(db.Integer,primary_key=True)
    name=db.Column(db.String(40),unique=True,nullable=False)

    """Adicione os itens a seguir em seu código"""
    def __repr__(self):
        return self.name
```

User.py

```
# -*- coding: utf-8 -*-
from flask_sqlalchemy import SQLAlchemy
"""Adicione o item a seguir em seu código"""
from sqlalchemy.orm import relationship
"""Até aqui"""
from config import app_active, app_config
from passlib.hash import pbkdf2_sha256
from model.Role import Role

config = app_config[app_active]
db = SQLAlchemy(config.APP)

class User(db.Model):
    id=db.Column(db.Integer,primary_key=True)
    username=db.Column(db.String(40),unique=True,nullable=False)
```

```
email=db.Column(db.String(120),unique=True,nullable=False)
password=db.Column(db.String(200),nullable=False)

date_created=db.Column(db.DateTime(6),default=db.func.current_timestamp(),
nullable=False)

last_update=db.Column(db.DateTime(6),onupdate=db.func.current_timestamp(),
nullable=False)
recovery_code=db.Column(db.String(100),nullable=True)
active=db.Column(db.Boolean(),default=1,nullable=False)
role=db.Column(db.Integer,db.ForeignKey(Role.id),nullable=False)
"""Adicione o item a seguir em seu código"""
funcao=relationship(Role)
...
```

Sem existir uma função criada, não conseguiremos criar um usuário. Crie no painel de `Roles` as 4 funções que estão descritas em nossa regra de negócios.

- Admin;
- Gerente;
- Logista;
- Cliente.

Após criar teremos as funções no painel.

Meu Estoque

Home

Usuários ▼

Produtos ▼

List (4)

Create

WITH SELECTED ▼









<input type="checkbox"/>		Name
<input type="checkbox"/>	 	Admin
<input type="checkbox"/>	 	Cliente
<input type="checkbox"/>	 	Gerente
<input type="checkbox"/>	 	Logista

Figura 6.9: Funções de usuário.

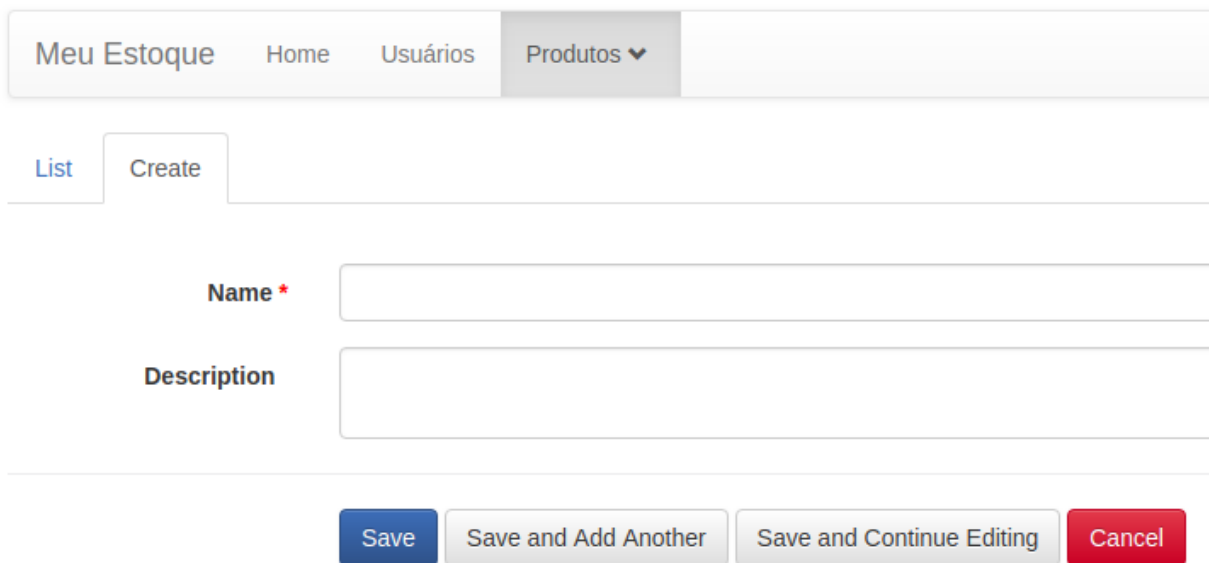
Agora podemos criar nosso primeiro usuário, sem nenhum problema. Faremos isso nas próximas seções. E após criar o usuário voltaremos à criação do produto.

6.2 Personalizando o admin

Como falamos, no Flask podemos personalizar o painel admin e deixá-lo conforme nossa regra de negócios. Vamos ver como fazer isso, desde o tema que usaremos para deixar o admin mais atrativo, até a edição das `ModelViews` do admin que permitirão organizar

nosso painel para que cada tipo de usuário acesse apenas aquilo que lhe é permitido.

Como acabamos de falar, `ModelViews` são recursos que permitem expressarmos as estruturas de nossas `models` como formulários em nosso admin, desse modo, conseguimos realizar a persistência do dado com mais facilidade. Vejamos a seguir uma imagem com o exemplo da criação de uma categoria.



The image shows a web interface for creating a category. At the top, there is a navigation bar with links: 'Meu Estoque', 'Home', 'Usuários', and 'Produtos' (which is highlighted with a dropdown arrow). Below the navigation bar, there are two tabs: 'List' and 'Create'. The 'Create' tab is active, showing a form with two input fields: 'Name' (with a red asterisk indicating it is required) and 'Description'. At the bottom of the form, there are four buttons: 'Save' (blue), 'Save and Add Another' (light gray), 'Save and Continue Editing' (light gray), and 'Cancel' (red).

Figura 6.10: Exemplo de formulário da ModelView - Criação de categoria

Como vemos, as colunas `name` e `description` que criamos como atributos na `model`, e que são colunas da tabela categoria de nosso banco de dados, se refletem em nosso sistema admin. Isso ocorre através do uso da `ModelView`.

Vamos começar falando sobre esse assunto.

Customizando a ModelView

Para a customização da `ModelView`, vamos editar nossos arquivos `Views.py` e `Admin.py`.

Agora vamos adicionar o código a seguir no arquivo `Views.py`, esse código criará uma `view` customizada para a tela de registro de usuário:

Views.py

```
# -*- coding: utf-8 -*-
from flask_admin.contrib.sqla import ModelView

from config import app_config, app_active
config = app_config[app_active]

class UserView(ModelView):
    column_exclude_list = ['password', 'recovery_code']
    form_excluded_columns = ['last_update', 'recovery_code']

    form_widget_args = {
        'password': {
            'type': 'password'
        }
    }

    def on_model_change(self, form, User, is_created):
        if 'password' in form:
            if form.password.data is not None:
                User.set_password(form.password.data)
            else:
                del form.password
```

Como podemos perceber, a `ModelView` é composta por uma tela que lista os dados criados, uma tela que cria novos dados e uma que edita os dados criados.

Vamos entender o que o código anterior faz:

- `from flask_admin.contrib.sqla import ModelView`: aqui nós importamos um componente da `lib flask_admin` que já utilizamos no arquivo `Admin.py`, porém, aqui ele será usado como superclasse da classe `UserView`, ou seja, a classe `UserView` herdará elementos da classe `ModelView`. Isso nos

permite ter tudo que a `ModelView` tem dentro dessa nossa `view` customizada e incrementar mais o que quisermos nela;

- `column_exclude_list` : é o atributo que nos permite excluir elementos da tela que lista os dados criados no administrador. No caso anterior, optamos por excluir a coluna de senha, para que não seja exibida quando os usuários forem listados no sistema;
- `form_excluded_columns` : esse recurso remove campos do formulário de criação e edição de item. Fique atento, pois se você remover um campo que é de tipo `NOT NULL` em seu banco de dados, receberá um erro ao tentar criar ou editar um item;
- `form_widget_args` : esse atributo nos permite modificar as propriedades de um elemento do formulário da `view`. No caso que utilizamos, o campo de senha era de tipo texto, permitindo que fosse exibida a senha enquanto digitada. Com esse atributo, através de um dicionário, nós enviamos à `ModelView` a informação de que queremos que ele seja de tipo `password`. Podemos aplicar qualquer modificação com esse recurso. Por exemplo, tornar um campo apenas para leitura, com o `readonly: True` ;
- `on_model_change` : é um método que é executado todas as vezes que criamos ou salvamos uma informação desse `ModelView`. Se criarmos um novo usuário ou editarmos seus dados, no momento em que clicarmos no botão `save` o método `on_model_change` será executado;
- `form.password.data` : o atributo `form` contém todos os valores que foram inseridos no formulário da `view` atual, ou seja, se existe um campo chamado `username`, podemos recuperar o valor adicionado nela através de `form.username.data` ;
- `User.set_password()` : este método nós criamos dentro da `model` no capítulo 5, em que falamos sobre `controllers`. Quando o utilizarmos, ele vai pegar a string que escrevemos no formulário do usuário e convertê-la para a hash que estamos utilizando, assim nossa senha será salva de modo criptografado. Sem usar esse método nossa senha seria salva da forma como foi escrita no formulário da `UIView`.

Agora que criamos a `view` de usuário, vamos implementá-la em nosso arquivo `Admin.py`.

Admin.py

```
# -*- coding: utf-8 -*-
from flask_admin import Admin
from flask_admin.contrib.sqla import ModelView

from model.User import User
from model.Category import Category
from model.Product import Product

"""
Adicione essa linha ao código
"""
from admin.Views import UserView

def start_views(app, db):
    admin = Admin(app, name='Meu Estoque', template_mode='bootstrap3')

    admin.add_view(ModelView(Role, db.session, "Funções",
category="Usuários"))
    """
    Altere a linha a seguir da ModelView do User para UserView
    """
    admin.add_view(UserView(User, db.session, "Usuários"))
    """Até aqui"""
    admin.add_view(ModelView(Category, db.session, 'Categorias',
category="Produtos"))
    admin.add_view(ModelView(Product, db.session, "Produtos",
category="Produtos"))
```

Vamos entender melhor:

- `from admin.Views import UserView`: aqui estamos importando o arquivo `Views.py` da pasta `admin` pois é nele que criaremos a `ModelView` customizada;
- `UserView`: é o método que criaremos no arquivo `Views.py` que customizará nossa `view` de usuários.

Agora podemos criar um novo usuário e ele funcionará corretamente. Vamos ver um exemplo de painel de criação de usuário.

Painel de criação de usuário:

Meu Estoque Home **Usuários** Produtos

List Create

Funcao * Admin

Username * tiagoluzrs

Email * tiagoluzribeirodasilva@gmail.com

Password *

Date Created 2019-03-02 11:23:00

Status ☒

Save Save and Add Another Save and Continue Editing Cancel

Figura 6.11: Exemplo de ModelView customizada

Painel de listagem de usuários:

Meu Estoque Home **Usuários** Produtos

List (1) Create With selected

	Funcao	Username	Email	Date Created	Last Update	Status
<input type="checkbox"/>	Admin	tiagoluzrs	tiagoluzribeirodasilva@gmail.com	2019-03-02 11:23:00		<input checked="" type="checkbox"/>

Figura 6.12: Exemplo de ModelView customizada

O campo `Last Update` só receberá um valor quando você realizar alguma atualização no usuário.

A `ModelView` possui vários atributos interessantes que podem ser muito úteis para deixar seu sistema mais profissional para atender suas regras de negócio. Como podemos ver na imagem, existe a opção de criar, editar e deletar nossos itens. Alguns dos atributos que veremos possibilitam que desativemos esses recursos. Isso será útil quando estivermos criando nosso recurso de login, para podermos separar de acordo com o que cada tipo de usuário poderá realizar aqui.

No tópico a seguir veremos como personalizar mais ainda nossa `ModelView`.

Atributos da `ModelView` customizada

Vejamos alguns atributos que podemos adicionar a uma `view` customizada em nosso sistema. Eles são excelentes, pois permitem habilitar ou desabilitar recursos muito úteis do sistema.

- `can_create = False` : permite habilitar ou desabilitar o recurso de criar novos itens na `ModelView` ;
- `can_edit = False` : permite habilitar ou desabilitar o recurso de editar itens da `ModelView` ;
- `can_delete = False` : permite habilitar ou desabilitar o recurso de deletar itens da `ModelView` ;
- `can_view_details = True` : permite visualizarmos os dados daquele item da `model` , sem precisar entrar nele, através da abertura de um `pop-up` .
- `column_exclude_list = ['password', 'recovery_code']` : permite remover da visualização da listagem dos dados, os campos que

forem passados dentro da lista. No exemplo, excluímos o `password` , mas poderíamos tirar qualquer outro campo existente.

- `column_sortable_list = ['username']` : permite que selecionemos quais colunas estão habilitadas para serem usadas como filtros de ordenação. No exemplo, a coluna `username` ficará habilitada e ao clicarmos nela, ela ordenará a lista em ordem alfabética de modo crescente; se clicarmos novamente, ordenará de modo decrescente.

Se você adicionar `column_sortable_list` à sua `view` , todas as colunas poderão ser usadas como ordenadores, exceto as colunas como `funcao` , que são colunas de Foreign key - para que elas sejam colunas filtráveis, você terá que adicioná-las manualmente pela propriedade `column_sortable_list` . Para remover todas é só passar `column_sortable_list = []` .

- `column_default_sort = ('username', True)` : diferente da anterior, essa propriedade faz com que a página já carregue baseada na coluna selecionada. Se o segundo atributo for de valor `True` , ele ordenará em ordem crescente; se for `False` , ela ordenará de forma decrescente.

Para adicionar mais de um campo para ordenação da lista você precisará passar várias tuplas dentro de uma lista:

```
column_default_sort = [('username', True), ('date_created', True)]
```

- `column_searchable_list = ['username', 'email']` : permite que seja adicionada à tela de listagem dos dados da `view` uma barra de pesquisa, que pesquisará com base nos itens que forem passados dentro da lista. No caso aqui, foram passados `username` e `email` .
- `column_filters = ['username', 'email']` : de modo similar, criará um menu que permitirá ao usuário realizar um filtro, baseado nos

itens que forem passados na lista.

- `column_editable_list = ['username', 'email']` : esse é um ótimo recurso para otimizar tempo. Ao habilitá-lo, todos os itens que forem passados na lista poderão ser editados na tela de listagem dos dados da `view`, sem que você precise entrar na edição completa do item.
- `page_size = 50` : quantidade de itens listados por página.
- `can_set_page_size = True` : permite que um `select` seja exibido, onde o usuário poderá dizer quantos itens quer que sejam carregados na lista por página. `True` habilita, `False` desabilita esse `select`.
- `column_details_exclude_list = ['password']` : permite remover colunas para que não sejam exibidas quando o usuário quiser listar os detalhes daquele item criado. No exemplo, estamos ocultando a senha, para que não seja exibida.
- `column_details_list = ['username']` : permite que digamos exatamente quais colunas queremos que sejam exibidas ao listarmos os detalhes de item da `ModelView`.
- `column_export_exclude_list = ['password']` : permite remover colunas para que não sejam importadas no arquivo `csv`.
- `column_export_list = ['username', 'email']` : permite que digamos exatamente quais colunas queremos que sejam exportadas em nosso arquivo `csv`.
- `export_max_rows` : máximo de linhas que serão exportadas no `csv`.
- `export_types = ['json', 'yaml', 'csv', 'xls', 'df']` : formatos de exportação do relatório. Podem ser passados todos dentro da lista ou apenas os desejados.

- `create_modal = True` : habilitará um recurso do sistema, com o qual, ao usuário clicar no botão de criar novo item, será aberto um `pop-up` de criação, em vez de haver redirecionamento para uma tela de criação de item.
- `edit_modal = True` : da mesma maneira que o anterior, mas em vez de criar, será aberto um `pop-up` de edição do item.
- `form_choices` : esse atributo permite que peguemos um campo e adicionemos um `select` com escolhas preestabelecidas por nós, de modo que o usuário só poderá optar por um desses itens.

```
form_choices = {
    'sexo': [
        ('Masculino', 'M'),
        ('Feminino', 'F')
    ]
}
```

O campo precisa existir na `view` para ser modificado.

- `form_excluded_columns = ['last_update', 'recovery_code']` : esse recurso remove campos do formulário de criação e edição de item. Fique atento, pois, se você remover um campo que é de tipo `NOT NULL` em seu banco de dados, receberá um erro ao tentar criar ou editar um item.
- `form_widget_args` : esse recurso permite alterarmos as propriedades de um campo do formulário de criação/edição. Por ser muito grande, o exemplo a seguir o descreve melhor.

```
form_widget_args = {
    'description': {
        'rows': 10,
        'style': 'color: black'
    }
}
```

- `can_export = True` : esse atributo permite que exportemos os dados que aparecem na tela de listagem da `view` em formato `csv` . Isso é muito útil para algumas situações, em que precisamos exportar relatórios.

Fique atento: ao utilizar o termo listagem, estou me referindo às telas da `view` que listam os dados que criamos no admin. Ao utilizar o termo lista, estou me referenciando à estrutura de dados lista, que é representada por 2 colchetes, `[]` .

Agora vamos alterar nosso arquivo `Views.py` e adicionar alguns desses atributos:

Views.py

```
# -*- coding: utf-8 -*-
from flask_admin.contrib.sqla import ModelView

from config import app_config, app_active
config = app_config[app_active]

class UserView(ModelView):
    column_exclude_list = ['password', 'recovery_code']
    form_excluded_columns = ['last_update', 'recovery_code']

    form_widget_args = {
        'password': {
            'type': 'password'
        }
    }

    """
    Adicione as linhas a seguir na UserView
    """

    can_set_page_size = True
    can_view_details = True
    column_searchable_list = ['username', 'email']
    column_filters = ['username', 'email', 'funcao']
```

```

column_editable_list = ['username', 'email', 'funcao', 'active']
create_modal = True
edit_modal = True
can_export = True
column_sortable_list = ['username']
column_default_sort = ('username', True)
column_details_exclude_list = ['password', 'recovery_code']
column_export_exclude_list = ['password', 'recovery_code']
export_types = ['json', 'yaml', 'csv', 'xls', 'df']

"""
Até aqui

def on_model_change(self, form, User, is_created):
    if 'password' in form:
        if form.password.data is not None:
            User.set_password(form.password.data)
        else:
            del form.password

```

Veja algumas imagens dos resultados desses atributos.

Meu Estoque

Home

Usuários

Produtos

List (1)

Create

Export

Add Filter

With selected

Search: username, email





<input type="checkbox"/>		Funcao	Username	Email	Date Created	Last Update	Status
<input type="checkbox"/>	  	Admin	tiagoluzrs	tiagoluzribeirosilva@gmail.com	2019-03-02 11:23:00	2019-03-02 11:23:00	

Figura 6.13: Listagem de usuários customizada.

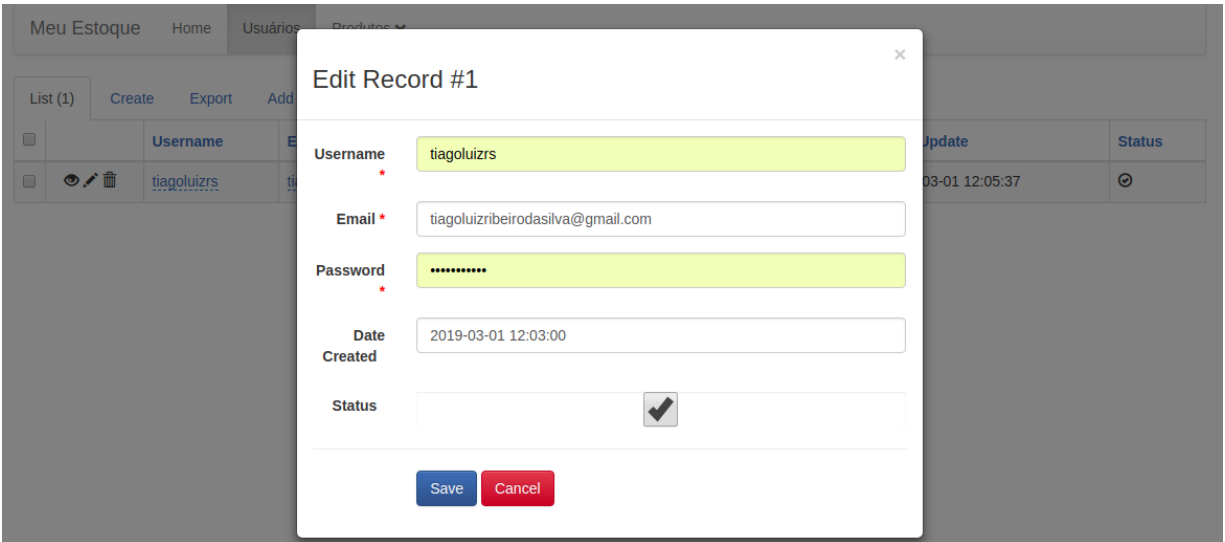


Figura 6.14: Edição de usuário com Modal.

Meu Estoque	Home	Usuários	Produtos ▾
List	Create	Edit	Details
Filter			
Username	tiagoluzrs		
Email	tiagoluzribeirodasilva@gmail.com		
Password	\$pbkdf2-sha256\$29000\$KSUEYGxtzbn3fg9hrFXKOQ\$vkqWQ\$SsbSaEEjrLNhAxAJQs/fG1eTZR0JEIMNmNPVJ8		
Date Created	2019-03-01 12:03:00		
Last Update	2019-03-01 12:05:37		
Status	True		

Figura 6.15: Exibição de detalhes do usuário.

Meu Estoque	Home	Usuários	Produtos ▾
-------------	------	----------	------------



List (1)	Create	Export	Add Filter ▾	With selected ▾	Search: username, email
tiagoluzrs	✓	✕			
			Date Created	Last Update	Status
 	tiagoluzrs	tiagoluzribeirodasilva@gmail.com	2019-03-01 12:03:00	2019-03-01 12:05:37	🔒

Figura 6.16: Edição de um campo do usuário sem entrar na tela de edição.

Fique atento: os elementos de relacionamento, como função, por exemplo, devem ser colocados na lista com o nome do relacionamento e não o nome original. Ou seja, nós adicionamos `funcao` e não `role` na lista da `ModelView`. Veja: `column_filters = ['username', 'email', 'funcao']`.

Outro ponto interessante é que esse tipo de campo não funciona em todos os lugares. A `column_searchable_list` não aceita criar um campo de pesquisa com elementos que possuem Foreign key.

Customizando as labels

No admin, também é possível customizar as `labels` dos campos. Podemos alterar seus nomes e adicionar descrições. Vamos fazer isso na classe `UserView` do arquivo `Views.py` que está na pasta `admin` e ver o resultado.

Primeiro vamos configurar o texto das `labels`.

Views.py:

```
# -*- coding: utf-8 -*-
from flask_admin.contrib.sqla import ModelView

from config import app_config, app_active
config = app_config[app_active]

class UserView(ModelView):
    """
    Adicione as linhas a seguir na UserView
```

```

"""
column_labels = {
    'funcao': 'Função',
    'username': 'Nome de usuário',
    'email': 'E-mail',
    'date_created': 'Data de criação',
    'last_update': 'Última atualização',
    'active': 'Ativo',
    'password': 'Senha',
}
"""

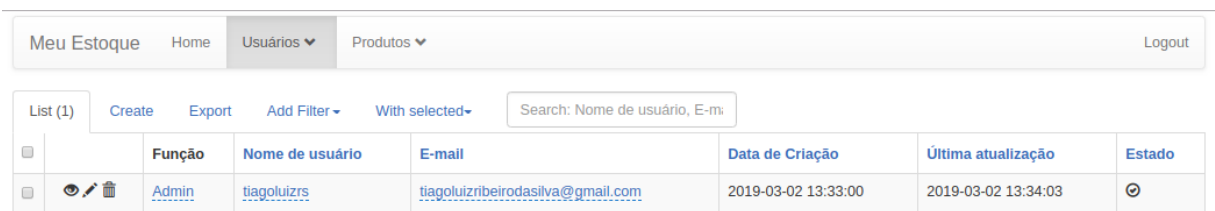
Até aqui
"""

column_exclude_list = ['password', 'recovery_code']
form_excluded_columns = ['last_update', 'recovery_code']
...

```

A propriedade `column_labels` permite que renomeemos as `labels` da nossa `ModelView`. Isso é ótimo também para trabalharmos com sistemas multilinguagem.

Veja na imagem a seguir como fica o painel com a configuração das `labels`.



The screenshot shows a web application interface with a top navigation bar and a main content area. The top bar has links for 'Meu Estoque', 'Home', 'Usuários' (selected), and 'Produtos', along with a 'Logout' button. Below the navigation bar is a table with columns for user management. The table has a search bar and several action buttons (Create, Export, Add Filter, With selected). The table contains one row of user data.

	Função	Nome de usuário	E-mail	Data de Criação	Última atualização	Estado
	Admin	tiagoluihrs	tiagoluizribeirodasilva@gmail.com	2019-03-02 13:33:00	2019-03-02 13:34:03	

Figura 6.17: Labels traduzidas para português.

Agora vamos adicionar as descrições de cada `label`. Para isso, precisaremos adicionar algumas linhas no arquivo `views.py`, dentro da classe `UserView`. É bem similar ao que fizemos no código anterior.

Views.py

```

# -*- coding: utf-8 -*-
from flask_admin.contrib.sqla import ModelView

from config import app_config, app_active
config = app_config[app_active]

class UserView(ModelView):
    column_labels = {
        'funcao': 'Função',
        'username': 'Nome de usuário',
        'email': 'E-mail',
        'date_created': 'Data de criação',
        'last_update': 'Última atualização',
        'active': 'Ativo',
        'password': 'Senha',
    }

    """
    Adicione as linhas a seguir na UserView
    """

    column_descriptions = {
        'funcao': 'Função no painel administrativo',
        'username': 'Nome de usuário no sistema',
        'email': 'E-mail do usuário no sistema',
        'date_created': 'Data de criação do usuário no sistema',
        'last_update': 'Última atualização desse usuário no sistema',
        'active': 'Estado ativo ou inativo no sistema',
        'password': 'Senha do usuário no sistema',
    }

    """
    Até aqui

    """"Existe mais código abaixo, que é irrelevante agora""""
    ...

```

A propriedade `column_descriptions` permite que adicionemos uma descrição na `label` desejada, o que é ótimo porque com isso

podemos deixar o sistema mais simples e compreensível para o usuário utilizar.

Veja na imagem a seguir como fica o painel com a configuração de descrição de label .

Meu Estoque

Home

Usuários

Produtos

Logout

List (1)

Create

Export

Add Filter

With selected

Search: Nome de usuário, E-m




<input type="checkbox"/>		Função	Nome de usuário	E-mail	Data de Criação	Última atualização	Estado
<input type="checkbox"/>	 	Admin	tiagoluzrs	tiagoluzrbeirodasilva@gmail.com	2019-03-02 13:33:00	2019-03-02 13:34:03	

Figura 6.18: Labels com ícone de interrogação.

Meu Estoque

Home

Usuários ▾

Produtos




List (1)

Create

Export

Add Filter ▾

With s

		Função ?	Nome de usuário ?
<input type="checkbox"/>			
<input type="checkbox"/>	  	Admin	tiagoluzrs

Função no painel administrativo

Figura 6.19: Label com descrição ativada ao clicar no ícone.

Esses elementos são ótimos para melhorarmos a experiência de uso do nosso painel admin.

Menus agrupados

Como vimos em alguns exemplos anteriores, nossos menus ficaram agrupados, sem que precisássemos fazer esforço para tal. Isso

ocorreu porque existe um parâmetro que passamos no método `add_view`, chamado `category`. Com ele, quando mais de um `ModelView` possuir o mesmo nome, será criado um menu com esse nome e os `ModelViews` serão agrupados dentro desse menu. Esse atributo já está implementado em nosso código. Veja o exemplo:

```
admin.add_view(UserView(User, db.session, "Usuários"))
admin.add_view(ModelView(Category, db.session, 'Categorias',
category="Produtos"))
admin.add_view(ModelView(Product, db.session, "Produtos",
category="Produtos"))
```

As `views` `Category` e `Product` ficaram agrupadas em um menu chamado `Produtos`. Se você notar, antes do parâmetro `category` temos o nome da `ModelView`, que é o nome a ser exibido como link do menu, para ir até a página. Vamos ver uma imagem de exemplo.

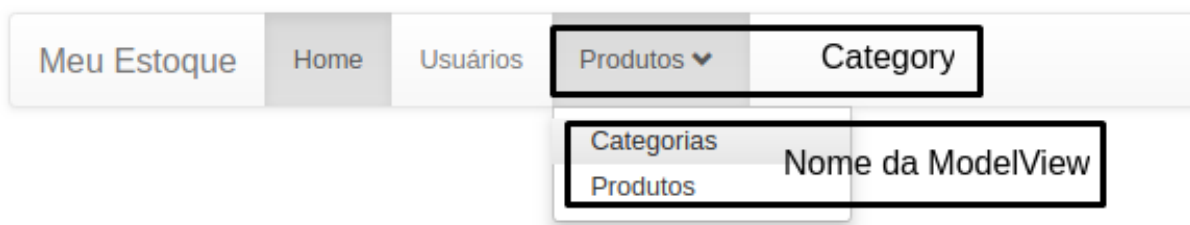


Figura 6.20: Exemplo de menu.

Fique atento: na `ModelView Product` temos a `category` com o nome `Produtos` e antes disso temos também o nome da `ModelView`, que por sua vez também é chamada `Produtos`, elas não são a mesma coisa.

Podemos adicionar menus personalizados em nosso painel. Isso é útil quando queremos criar um botão de logout, por exemplo. Vamos modificar o código do arquivo `Admin.py` para que possamos adicionar o botão de logout em nosso painel.

Admin.py

```
# -*- coding: utf-8 -*-
from flask_admin import Admin
from flask_admin.contrib.sqla import ModelView
"""Adicione a linha a seguir"""
from flask_admin.menu import MenuLink

from model.Role import Role
from model.User import User
from model.Category import Category
from model.Product import Product

from admin.Views import UserView

def start_views(app, db):
    admin = Admin(app, name='Meu Estoque', template_mode='bootstrap3')

    admin.add_view(ModelView(Role, db.session, "Funções",
category="Usuários"))
    admin.add_view(UserView(User, db.session, "Usuários",
category="Usuários"))
    admin.add_view(ModelView(Category, db.session, 'Categorias',
category="Produtos"))
    admin.add_view(ModelView(Product, db.session, "Produtos",
category="Produtos"))

    """
    Adicione a linha a seguir
    """

    admin.add_link(MenuLink(name='Logout', url='/logout'))
```

Se você recarregar a página, verá o menu de logout em seu admin. Por ora ele ainda não funciona. Mas o faremos funcionar no capítulo 10, sobre Autenticação e Segurança .

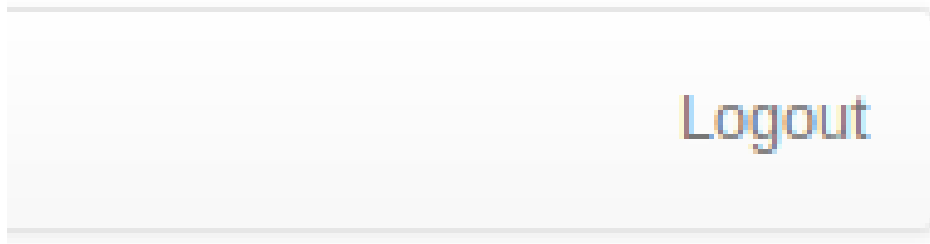


Figura 6.21: Botão de logout.

`ModelView` e `views` são coisas totalmente diferentes, tendo em vista que a `ModelView` exibe a estrutura de uma `model` no painel administrativo e a `view` pode ser qualquer coisa exibida na tela, ou seja, toda `ModelView` é uma `view`, mas nem toda `view` é uma `ModelView`.

6.3 Personalizando a home do admin

Ter um admin personalizado é algo que o Flask nos permite ter de forma simples. Veremos a seguir como podemos personalizar nossa home do painel admin.

Para personalizar nosso admin precisamos seguir alguns passos. Vamos começar criando nossa classe que sobrescreverá a tela home do admin. Abra o arquivo `views.py` e adicione as linhas a seguir.

Views.py

```
# -*- coding: utf-8 -*-
from flask_admin.contrib.sqla import ModelView
"""Adicione a linha a seguir"""
from flask_admin import AdminIndexView, expose
"""Até aqui"""
```



```

from config import app_config, app_active
config = app_config[app_active]

"""
Adicione a class `HomeView` que está a seguir.
"""

class HomeView(AdminIndexView):
    @expose('/')
    def index(self):
        return self.render('home_admin.html', data={
            'username': 'Tiago Luiz'
        })
"""
Até aqui
"""

class UserView(ModelView):
    column_exclude_list = ['password', 'recovery_code']
    ...
    """Existe mais código abaixo que é irrelevante agora"""

```

Vamos entender o que cada linha faz:

- `from flask_admin import AdminIndexView, expose` : aqui nós estamos importando a classe `AdminIndexView` que será herdada pela classe `HomeView` para que a classe `HomeView` tenha todas as características da página de `home` e possa sobrescrevê-las quando necessário, e também importamos o decorator `@expose` que será explicado a seguir;
- `@expose('/')` : usado para dizermos qual rota estamos alterando, no caso, `http://localhost:8000/admin/` , mas o `admin` não é incluso. Se a rota fosse `http://localhost:8000/admin/home` , o `@expose` seria assim `@expose('/home')` ;
- `self.render` : método que usamos para renderizar um `html` no `admin`. É similar ao método `render_template` que vimos no capítulo anterior e que será mais aprofundado no capítulo sobre `views htmls` . Como você pode perceber, passamos um argumento chamado `data` que envia um dicionário para o `html` . Por ora ele ficará estático, mas futuramente passaremos um

método que enviará o nome do usuário dinamicamente para o
html ;

Agora precisamos editar o arquivo `Admin.py` , para que ele entenda que a `HomeView` será a `view` inicial do painel administrativo. Abra o arquivo e adicione o código a seguir.

Admin.py

```
# -*- coding: utf-8 -*-
from flask_admin import Admin
from flask_admin.contrib.sqla import ModelView
from flask_admin.menu import MenuLink

from model.Role import Role
from model.User import User
from model.Category import Category
from model.Product import Product

"""Altere a linha a seguir"""
from admin.Views import UserView, HomeView

def start_views(app, db):
    """Altere a linha a seguir"""
    admin = Admin(app, name='Meu Estoque', base_template='admin/base.html',
template_mode='bootstrap3', index_view=HomeView())
    ...
```

Vamos entender o que foi alterado:

- `base_template` : parâmetro onde passamos qual será o arquivo `html` usado como página home do painel admin;
- `index_view` : parâmetro que usamos para passar ao `flask-admin` qual será a `ModelView` customizada da tela home do admin.

Com todos esses itens configurados, precisamos adicionar o código necessário ao arquivo `home_admin.html` que se encontra dentro da pasta `templates` para que ele possa funcionar corretamente.

home_admin.html

```
{% extends admin_base_template %}
{% block body %}
<h1>Seja Bem Vindo {{data.username}}</h1>
{% endblock %}
```

Vamos entender as linhas adicionadas.

- `{% extends admin_base_template %}` : essa linha é utilizada para que o arquivo `home_admin` herde o template base que existe no `admin`, ele herdará todos os arquivos `css`, `js` e também componentes como o `menu` do painel `admin`;
- `{% block body %}` : indica que aqui começa a área de corpo deste `html`. Tudo o que precisar ser criado no arquivo precisa estar abaixo dessa linha;
- `{{data.username}}` : este foi o parâmetro passado no método `self.render` dentro do arquivo `Views.py`. Podemos passar quantos parâmetros quisermos por esse método, e conseguiremos exibi-los no `html` da mesma forma;
- `{% endblock %}` : indica que aqui acaba o corpo desse arquivo `html`. Tudo o que precisa ser adicionado ao arquivo só deve ser colocado até antes dessa linha.

Com todos esses pontos configurados, teremos nossa página `home` customizada e pronta para adicionarmos o que desejarmos. Mais à frente no livro criaremos algumas requisições para adicionarmos gráficos na tela `home` do `admin`, portanto deixaremos nosso painel mais bonito e profissional. Veja a seguir o exemplo da `home` customizada.

Seja Bem Vindo Tiago Luiz

Figura 6.22: Tela Home do Admin customizada.

Criando um produto completo

Agora que temos nossas funções, usuário e categoria criados, podemos criar nosso produto sem nenhum problema.

Vamos entrar na tela de criação de um novo produto, conseguiremos criá-lo sem nenhum problema.

[List](#) [Create](#)

Usuario *

1 - tiagoluizrs

Categoria *

Calçados

Name *

Ride Skateboard

Description *

Tênis Ride Skateboard Curb Cinza
Tipo de Produto: Tênis

Qtd

10

Image

https://t-static.dafiti.com.br/eKrcBP7UqAxpSPxZfcTXhcojp_4=/fit-in/430x623/dafitistatic-a.akamaihd.net%2fp%2fride-skateboard-t%c3%aanis-ride-skateboard-curb-cinza-3612-9198302-1-zoom.jpg

Price *

149.90

Date Created

2019-03-01 14:16:00

Last Update *

2019-03-01 14:16:00

Status

1

Save

Save and Add Another

Save and Continue Editing

Cancel

Figura 6.23: Preencha para criar seu produto.

Record was successfully created.

List (1)

CreateWith selected



<input type="checkbox"/>		Usuario	Categoria	Name	Description	Qtd	Image	Price	Date Created	Last Update	Status
<input type="checkbox"/>	 	1 - tiagoluzizrs	Calçados	Ride Skateboard	Tênis Ride Skateboard Curb Cinza Tipo de Produto: Tênis Tamanho: 38	10	https://t-static.dafiti.com.br/eKrCBP7UqAxrSPxZfcTXhcojp_4=/fit-in/430x623/dafitistatic-a.akamaihd.net%2fp%2fride-skatboard-t%c3%aanis-ride-skatboard-curb-cinza-3612-9198302-1-zoom.jpg	149.90	2019-03-01 14:16:00	2019-03-01 14:16:00	1

Figura 6.24: Produto criado com sucesso!.

Pronto, agora nosso produto está criado.

6.4 Tema padrão do admin

Um tema é um conjunto de padrões que utilizamos para definir as características de nosso sistema. O Flask possui uma diversidade de temas que nos permite personalizar nosso sistema com inúmeras aparências diferentes que veremos a seguir.

Veja um exemplo de como é simples fazer isso. Abra seu arquivo `app.py` e adicione o código:

`app.py`

```

"""Este arquivo possui mais código acima que é irrelevante agora"""
...
config = app_config[app_active]
from flask_sqlalchemy import SQLAlchemy

def create_app(config_name):
    app = Flask(__name__, template_folder='templates')

    app.secret_key = config.SECRET
    app.config.from_object(app_config[config_name])
    app.config.from_pyfile('config.py')
    app.config['SQLALCHEMY_DATABASE_URI'] = config.SQLALCHEMY_DATABASE_URI

```

```

app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

"""Adicione o código a seguir"""
app.config['FLASK_ADMIN_SWATCH'] = 'paper'
"""Até aqui"""
db = SQLAlchemy(config.APP)
start_views(app,db)

db.init_app(app)

@app.route('/')

```

Como vimos no exemplo, optamos pelo tema `cerulean`, que deixará seu tema com tons cor azul céu. Você pode optar por qualquer um dos temas a seguir:

- Default;
- Cerulean;
- Cosmo;
- Cyborg;
- Darkly;
- Flatly;
- Journal;
- Lumen;
- Paper;
- Readable;
- Sandstone;
- Simplex;
- Slate;
- Spacelab;
- Superhero;
- United;
- Yeti.

Se quiser saber mais sobre cada tema, você pode acessar o site deles: <https://bootswatch.com/3/>

Fique atento: os nomes precisam estar em minúsculo quando forem adicionados no arquivo `app.py`.

Vamos optar pelo tema `paper` que traz um layout mais `clean` e mais voltado para o Design Material que o Google utiliza como design da interface do Android. Mas fique à vontade para experimentar outro.

Conclusão

Com todos esses itens configurados, temos nosso painel administrativo funcionando perfeitamente e já podemos adicionar usuários, categorias e produtos, como está proposto em nossa regra de negócios. Com o passar dos capítulos, ainda faremos outras alterações, adicionando um login seguro nele, e criando camadas de acesso para que cada tipo de usuário utilize o sistema conforme é permitido para seu nível.

Mas por ora, já podemos ver nosso sistema tomando um formato mais profissional. Isso é algo fantástico.

CAPÍTULO 7

Trabalhando com o SQLAlchemy

Neste capítulo veremos como trabalhar com uma das ferramentas mais poderosas e úteis que existem para Python, o `SQLAlchemy`, uma `lib` completa contendo padrões que facilitam o trabalho com o banco de dados de nossa aplicação.

As configurações básicas do `SQLAlchemy` já foram realizadas no capítulo 3, onde vimos sobre `models`. Nos tópicos a seguir veremos mais a fundo como executar queries de forma eficiente utilizando o `SQLAlchemy`.

7.1 Queries no SQLAlchemy

Vamos ver como é simples criar queries em nossas `models` com `SQLAlchemy`.

db.session

O `SQLAlchemy` trabalha com o conceito de sessão, que realiza a abertura de uma conexão, permitindo que realizemos as ações básicas de `insert`, `update`, `select` e `delete`.

A seguir vamos adicionar um método de `select` para pegarmos todos os produtos que estão no banco de dados. Abra o arquivo `Product.py` que está na pasta `models` e adicione o código:

Product.py

```
"""Fique atento, existe mais código acima"""
category = db.Column(db.Integer, db.ForeignKey(Category.id),
nullable=False)
usuario = relationship(User)
```



```
categoria = relationship(Category)
```

```
"""Adicione o método a seguir em seu arquivo"""
```

```
def get_all():  
    try:  
        res = db.session.query(Product).all()  
    except Exception as e:  
        res = []  
        print(e)  
    finally:  
        db.session.close()  
    return res
```

Vamos entender melhor as linhas do método do exemplo anterior.

- `db.session` : é a instância que contém a conexão aberta do banco de dados. Ela possibilitará que façamos todas as ações em nosso banco de dados.
- `query` : é o método que diz ao `SQLAlchemy` que uma query será executada. Dentro dele precisamos passar o objeto a que estamos nos referindo; no exemplo anterior, estávamos realizando uma query no objeto `Product`.
- `all()` : esse método diz ao `SQLAlchemy` que ele deverá retornar todos as linhas de resultado da query. Outro método que poderia estar no lugar desse é o `first()`, que diz ao `SQLAlchemy` para retornar apenas a primeira linha de resultado.
- `close()` : esse método é responsável por fechar a conexão ao término, ele deve ser utilizado todas as vezes que abrirmos uma conexão com o banco.

Retorno de dados do SQLAlchemy

O método `db.session.query(Product).all()` retornará uma lista com vários objetos, que nos permitem acessar seus atributos. Um exemplo seria assim:

Esses exemplos são apenas para exibir, não devem ser implementados em arquivo nenhum. Quando houver implementação, será citado o nome do arquivo.

```
try:
    produtos = db.session.query(Product).all()
except Exception as e:
    produtos = []
    print(e)
finally:
    db.session.close()
    return produtos
```

```
print(produtos[0].id)
print(produtos[0].name)
print(produtos[0].description)
print(produtos[0].qtd)
print(produtos[0].price)
print(produtos[0].date_created)
print(produtos[0].last_update)
print(produtos[0].status)
print(produtos[0].user_created)
print(produtos[0].category)
```

"""Ou através de um `for`"""

```
for produto in produtos:
    print(produto.id)
    print(produto.name)
    print(produto.description)
    print(produto.qtd)
    print(produto.price)
    print(produto.date_created)
    print(produto.last_update)
    print(produto.status)
    print(produto.user_created)
    print(produto.category)
```

Como foi possível ver, podemos acessar os resultados pelos índices, ou através de um `for`.

Se utilizarmos o método `first()` no lugar de `all()`, o resultado será apenas o objeto e não uma lista de objetos, então para acessar seria assim:

```
try:
    produto = db.session.query(Product).first()
except Exception as e:
    produto = []
    print(e)
finally:
    db.session.close()
    return produto
```

```
print(produto.id)
print(produto.name)
print(produto.description)
print(produto.qtd)
print(produto.price)
print(produto.date_created)
print(produto.last_update)
print(produto.status)
print(produto.user_created)
print(produto.category)
```

Se você quiser listar apenas alguns campos da tabela, faça como no exemplo a seguir:

```
# Query com campos específicos, usando o método first()
try:
    produto = db.session.query(Product.id, Product.name,
    Product.description).first()
except Exception as e:
    produto = []
    print(e)
finally:
    db.session.close()
    return produto
```

```
produto.id
produto.name
produto.description
```

```
# Query com campos específicos, usando o método all()
```

```
try:
    produtos = db.session.query(Product.id, Product.name,
Product.description).all()
except Exception as e:
    produtos = []
    print(e)
finally:
    db.session.close()
    return produtos
```

```
for produto in produtos:
    print(produto.id)
    print(produto.name)
    print(produto.description)
```

Você pode fazer o mesmo com o método `first()` . O conceito funciona da mesma forma.

Inserindo dados no banco

Inserir dados no banco é muito simples. Vamos ver um exemplo, alterando nossos arquivos `Product.py` da pasta `model` , `Product.py` da pasta `controller` e o `app.py` .

model/Product.py

```
"""Fique atento, existe mais código acima"""
categoria = relationship(Category)
```

```
def get_all():
    try:
```

```

        res = db.session.query(Product).all()
    except Exception as e:
        res = []
        print(e)
    finally:
        db.session.close()
    return res

"""Adicione o método a seguir em seu arquivo"""
def save(self):
    try:
        db.session.add(self)
        db.session.commit()
        return True
    except Exception as e:
        print(e)
        db.session.rollback()
        return False

```

Vamos entender o que os métodos do exemplo anterior fazem.

- `add(self)` : o método `add` adiciona um objeto na sessão, que será futuramente inserido no banco. No exemplo, ele está recebendo `self`, que representa o próprio objeto `Product`.
- `commit()` : esse é o método que executará a ação de inserir os dados no banco. Sem o `commit` a transação não ocorre até o final e o `rollback` será acionado.
- `rollback()` : esse é o método que será acionado se houver algum erro na inserção do dado. Ele vai desfazer qualquer alteração, para que a aplicação possa seguir sem nenhum problema, a única coisa que ocorrerá é que o dado não será inserido, mas o banco não ficará preso com uma transação que não deu certo.

Agora vamos editar o arquivo da `controller` e adicionar o código a seguir.

controller/Product.py

```

"""Adicione o código a seguir em seu arquivo"""
from datetime import datetime

from model.Product import Product

class ProductController():
    def __init__(self):
        self.product_model = Product()

    """Adicione o método a seguir em seu arquivo"""
    def save_product(self, obj):
        self.product_model.name = obj['name']
        self.product_model.description = obj['description']
        self.product_model.qtd = obj['qtd']
        self.product_model.price = obj['price']
        self.product_model.date_created = datetime.now()
        self.product_model.status = 1
        self.product_model.category = obj['category']
        self.product_model.user_created = obj['user_created']
        return self.product_model.save()

```

Vamos entender o que o código quer fazer.

- `self.product_model` : esse é o atributo que recebe no construtor o objeto `Product` da `model`. Através dele vamos acessar os atributos da `model`, que são `name`, `description` etc. Esses atributos armazenarão os valores vindos do arquivo `app.py` para serem salvos pelo método `save()` da `model`;
- `save()` : método que criamos na `model Product` e que salvará um novo produto com todos os dados que estão sendo adicionados nos atributos que pertencem à `model Product`.

Para finalizar, vamos modificar o arquivo `app.py` e adicionar um `import` e uma nova rota :

app.py

```

"""Adicione a linha a seguir no código lá em cima onde ficam as
importações"""
from controller.Product import ProductController

```

```

"""Adicione a rota e o método a seguir
em seu arquivo"""
@app.route('/product', methods=['POST'])
def save_products():
    product = ProductController()

    result = product.save_product(request.form)

    if result:
        message = "Inserido"
    else:
        message = "Não inserido"

    return message

return app

```

Aqui não temos muito o que explicar, pois já fizemos algo similar no capítulo de `controllers`. O `request.form` receberá os valores inseridos no formulário de inserção de produtos e os atribuirá ao método `save_product` que pertence à `controller`.

Fique atento: esse formulário não existe, nós usaremos o `postman` do mesmo modo que usamos no capítulo sobre `rotas`. Mas se você criar um arquivo `html` com um formulário que aponte para a URL dessa rota, você também conseguirá inserir um produto na base de dados.

Testando com o Postman

Siga os próximos passos para testar no `postman`.

1. Preencha os campos de URL, e o `content-type` :

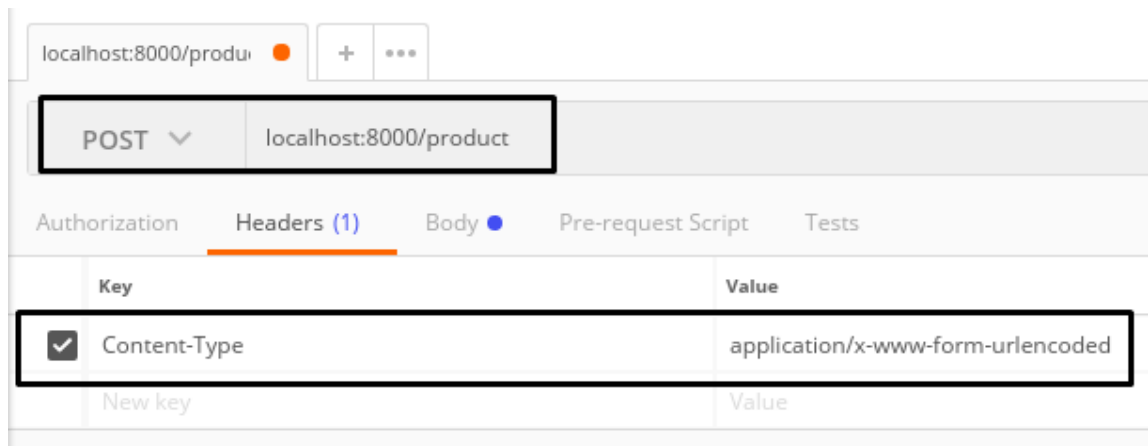


Figura 7.1: Postman configuração

2. Adicione os campos que representam o formulário pela aba Body e marque o tipo `x-www-form-urlencoded` :

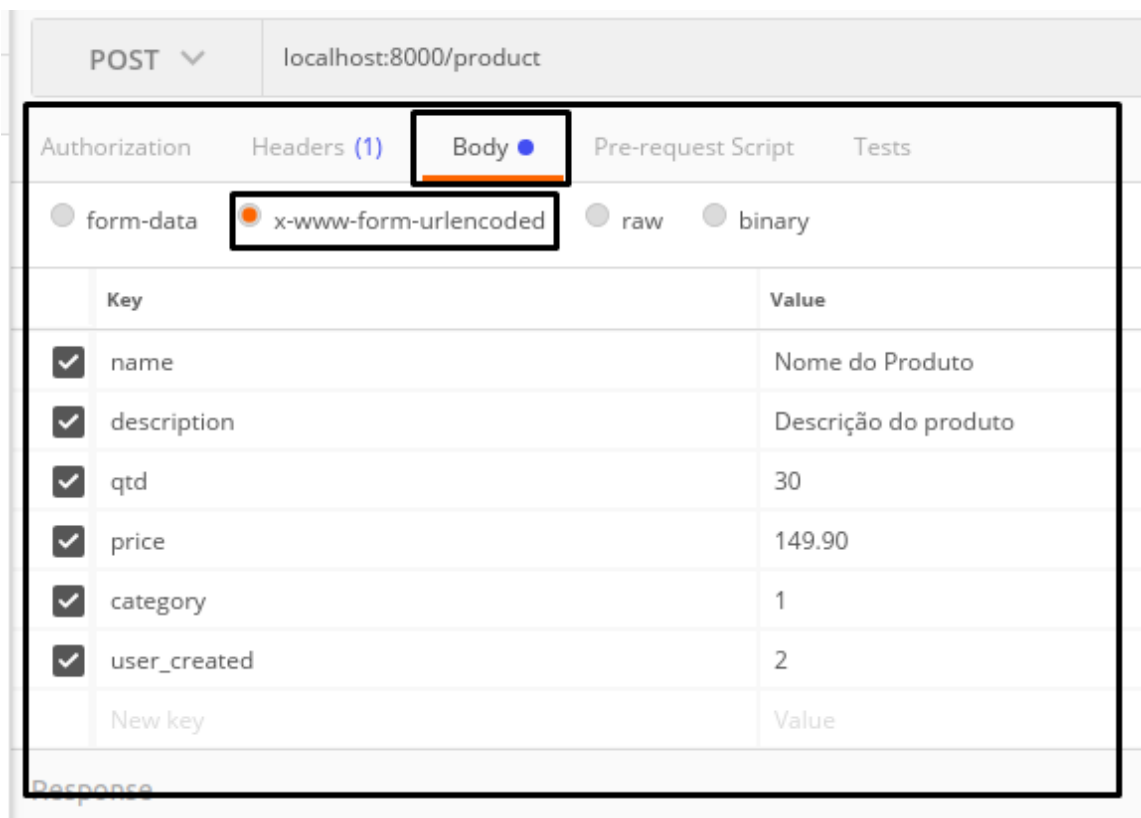


Figura 7.2: Postman preenchimento dos campos do form

Atenção aos IDs de categoria e usuário, verifique se existem. Caso só tenha um usuário, o ID terá que ser 1 e não 2.

3. Ao clicar no botão azul escrito `send` você verá o seguinte resultado:

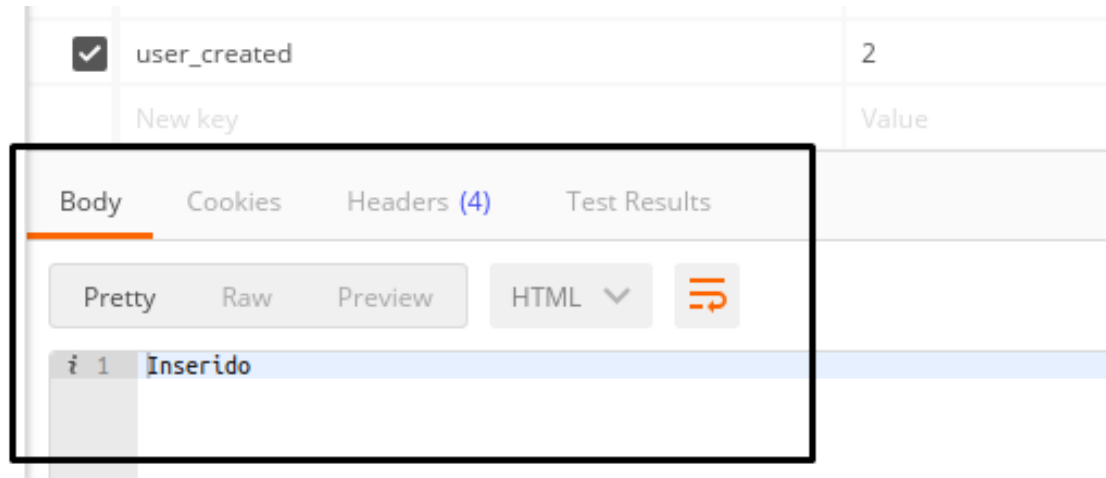


Figura 7.3: Postman requisição enviada

Com isso, se formos ao admin, veremos um item que corresponde ao que acabamos de inserir.

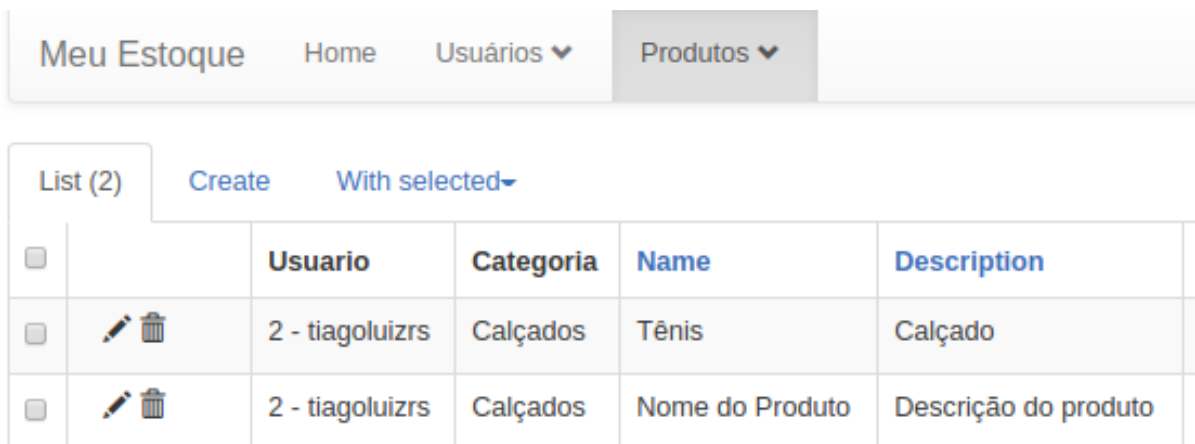


Figura 7.4: Produto inserido.

Modificando dados no banco

Para realizar mudanças em um item ou para deletá-lo não é tão diferente. Veremos um exemplo a seguir, onde editaremos um item e faremos a requisição do `postman`. Vamos lá.

model/Product.py

```
"""Fique atento, existe mais código acima"""
"""Adicione o código a seguir abaixo do método save()"""

def update(self, obj):
    try:
        res = db.session.query(Product).filter(Product.id ==
self.id).update(obj)
        db.session.commit()
        return True
    except Exception as e:
        print(e)
        db.session.rollback()
        return False
```

Vamos entender melhor os dois métodos novos que vemos no arquivo.

- `filter()` : esse é o método que usamos para filtrar o dado que desejamos modificar ou até mesmo listar. O `filter` é usado como a cláusula `WHERE` do `SQL`, então se quisermos listar um usuário de `id` igual a 1 faremos `db.session.query(User).filter(User.id==1).first()`, por exemplo. O mesmo para editarmos e deletarmos um dado específico. No caso do exemplo, `self.id` será o valor recebido na `controller`, que ainda vamos criar;
- `update()` : esse é o método que executará o `update`, que em seguida precisará do `commit()` para que a mudança ocorra efetivamente no banco. O `update` receberá um dicionário com o nome do campo e o valor da modificação. Por exemplo, se quisermos mudar somente o nome do produto, podemos fazer

```
obj = {'description': 'Nova descrição'} e passar esse valor
dentro de update() .
```

Abra o arquivo `controller/Product.py` e adicione o código a seguir, pois ele será responsável por chamar o método `update` na `model` para atualizar nosso produto:

controller/Product.py

```
"""Fique atento, existe mais código acima"""

"""Adicione o método a seguir em seu arquivo abaixo do método
save_product()"""
def update_product(self, obj):
    self.product_model.id = obj['id']
    return self.product_model.update(obj)
```

Agora precisamos criar o `endpoint` de edição de produto. Adicione um novo `endpoint` ao arquivo `app.py` .

app.py

```
"""Adicione o método e a rota a seguir
em seu arquivo após a rota de criação
de produto"""
@app.route('/product', methods=['PUT'])
def update_products():
    product = ProductController()

    result = product.update_product(request.form)

    if result:
        message = "Editado"
    else:
        message = "Não Editado"

    return message

return app
```

Para testarmos com o Postman só mudaremos alguns pequenos pontos. Veja a seguir:

1. Altere o método de POST para PUT e adicione somente os campos que deseja modificar. Adicione também o id do produto. Veja em seu banco de dados o id de um produto para usar como teste:

PUT ▾ localhost:8000/product

Authorization Headers (1) Body ● Pre-request Script Tests

☐ form-data ☒ x-www-form-urlencoded ☐ raw ☐ binary

	Key	Value
<input checked="" type="checkbox"/>	id	13
<input checked="" type="checkbox"/>	name	Nome do Produto Novo
<input checked="" type="checkbox"/>	qtd	40
<input checked="" type="checkbox"/>	price	249.90
	New key	Value

Figura 7.5: Postman: configuração para edição de produto.

2. Com isso seu produto já poderá ser alterado sem problema algum:

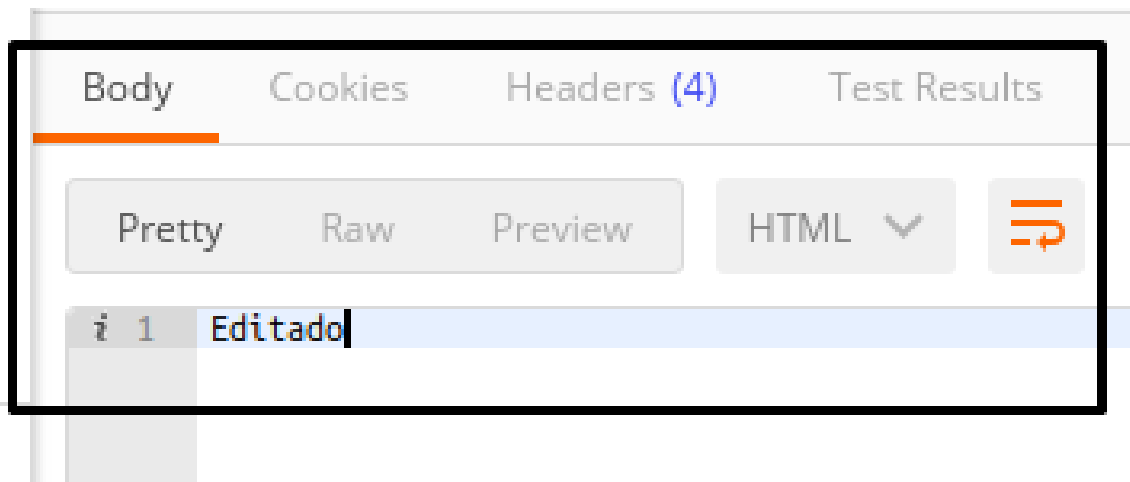


Figura 7.6: Postman: produto editado.

Novamente, não esqueça de que o id do produto precisa existir, então confira no banco de dados quais ids estão criados.

Deletando um dado do banco

Se quisermos deletar um dado do banco é mais simples ainda. Só precisamos criar um método parecido com o do exemplo a seguir em nosso `Product.py` da pasta `model`:

```
try:
    db.session.query(Product).filter(Product.id == self.id).delete()
    db.session.commit()
    return True
except Exception as e:
    db.session.rollback()
    print(e)
    return False
```

Não precisa implementar, é apenas um exemplo.

Acredito que não precisamos criar um exemplo de requisição no postman , pois o padrão se mantém.

Outro ponto interessante de se perceber é que não utilizamos o método `close()` quando fazemos alterações no banco, pois o `commit()`, ao ser executado, se der certo, executa o fechamento da conexão e, caso dê errado, o `rollback()` cancelará a transação e logo em seguida fechará a conexão. Utilizar o método `close()` nesses casos acarretará um erro no sistema.

7.2 Filtros no SQLAlchemy

Os filtros representam a cláusula `WHERE` do `SQL`, no `SQLAlchemy` podemos utilizar filtros de dois modos. Com o método `filter` ou o `filter_by`. A diferença é que o método `filter` permite que adicionemos mais de um campo para filtro, o `filter_by` não.

Esses métodos também não serão implementados no projeto, quando for para implementar o nome do arquivo será citado.

```
# Uso do filter_by
try:
    res = db.session.query(Product.id).filter_by(id=self.id).all()
except Exception as e:
    res = []
    print(e)
finally:
    db.session.close()
    return res
```

```
# Uso do filter
try:
    res = db.session.query(Product.id).filter(Product.id ==
self.id).filter(Product.price > 300).all()
except Exception as e:
    res = []
```

```
    print(e)
finally:
    db.session.close()
    return res
```

Uma outra observação é que o método `filter_by` recebe um parâmetro que representa o nome do campo da tabela e o valor, enquanto o método `filter` passa no lugar do parâmetro o objeto que representa a tabela e o campo que será usado no filtro, como o do exemplo `Product.id==self.id`.

Vamos aos exemplos.

like: podemos adicionar o `like` ao nosso filtro, para fazermos consultas mais complexas.

```
try:
    res =
db.session.query(Product).filter(Product.name.like('%tenis%')).all()
except Exception as e:
    res = []
    print(e)
finally:
    db.session.close()
    return res
```

in: podemos criar um filtro adicionando o `in`, que em SQL serve para incluir mais de uma possibilidade dentro de um mesmo `WHERE`.

```
try:
    res = db.session.query(Product).filter(Product.name.in_(['Tênis', 'Nome
do Produto Novo'])).all()
except Exception as e:
    res = []
    print(e)
finally:
    db.session.close()
    return res
```

not in: se quisermos dizer ao SQL que é para localizar qualquer coisa diferente daquilo, usamos o `not in`. Vamos ao exemplo:

```
try:
    res = db.session.query(Product).filter(~Product.name.in_(['Tênis', 'Nome
do Produto Novo'])).all()
except Exception as e:
    res = []
    print(e)
finally:
    db.session.close()
    return res
```

is null: podemos verificar se um valor é `null`.

```
try:
    res = db.session.query(Product).filter(Product.name==None).all()
except Exception as e:
    res = []
    print(e)
finally:
    db.session.close()
    return res
```

is not null: ou também se um valor não é `null`.

```
try:
    res = db.session.query(Product).filter(Product.name!=None).all()
except Exception as e:
    res = []
    print(e)
finally:
    db.session.close()
    return res
```

and: o `and` inclui mais de uma condição dentro do `WHERE`, sendo que as duas precisam ser verdadeiras para que um valor seja retornado do banco de dados. Você precisará importar o método `and_` da lib `sqlalchemy`.


```

from sqlalchemy import and_

try:
    res = db.session.query(Product).filter(and_(Product.name == 'Tênis',
Product.category == 1)).all()
except Exception as e:
    res = []
    print(e)
finally:
    db.session.close()
    return res

```

or: o `or` inclui mais de uma condição dentro do `WHERE` , sendo que apenas uma das duas precisa ser verdadeira para que um valor seja retornado do banco de dados. Você precisará importar o método `or_` da lib do `sqlalchemy` .

```

from sqlalchemy import or_

try:
    res = db.session.query(Product).filter(or_(Product.name == 'Tênis',
Product.price == 249.90, Product.category == 1)).all()
except Exception as e:
    res = []
    print(e)
finally:
    db.session.close()
    return res

```

7.3 Funções de agrupamento

O `SQLAlchemy` possui o objeto `func` , que possui métodos que permitem que utilizemos as funções básicas de agrupamento do `SQL` . Vamos ver a seguir cada um dos métodos de agrupamento e como utilizá-los, mas antes, tenha em mente que você precisará importar o método `func` do `SQLAlchemy` no topo do arquivo, assim:

```

from sqlalchemy import func .

```

- `func.count()` : utilizamos para contabilizar o total de linhas que possuem o campo descrito, no caso do exemplo, quantas linhas da tabela produto possuem `id` ;

```
from sqlalchemy import func

try:
    res = db.session.query(func.count(Product.id)).all()
except Exception as e:
    res = []
    print(e)
finally:
    db.session.close()
    return res
```

- `func.avg()` : retorna a média do campo especificado, no caso do exemplo, o `id` ;

```
from sqlalchemy import func

try:
    res = db.session.query(func.avg(Product.id)).all()
except Exception as e:
    res = []
    print(e)
finally:
    db.session.close()
    return res
```

- `func.sum()` : realiza uma soma das linhas referente ao campo especificado;

```
from sqlalchemy import func

try:
    res = db.session.query(func.sum(Product.id)).all()
except Exception as e:
    res = []
    print(e)
finally:
```

```
db.session.close()
return res
```

- `func.max()` : retorna a linha que possui o valor mais alto no campo especificado;

```
from sqlalchemy import func

try:
    res = db.session.query(func.max(Product.id)).all()
except Exception as e:
    res = []
    print(e)
finally:
    db.session.close()
    return res
```

- `func.min()` : retorna a linha que possui o valor mais baixo no campo especificado.

```
from sqlalchemy import func

try:
    res = db.session.query(func.min(Product.id)).all()
except Exception as e:
    res = []
    print(e)
finally:
    db.session.close()
    return res
```

7.4 Cláusulas avançadas do SQL

As cláusulas mais avançadas serão muito úteis para deixarmos as queries com resultados mais performáticos, possibilitando que limitemos a quantidade de linhas que desejamos retornar, agrupar os resultados por um campo específico, e realizar junções por `join`. Vamos vê-las a seguir.

limit: o limite nos permite limitar a quantidade de linhas que vamos retornar. Mas fique atento, mesmo com ele, precisamos finalizar a query com o método `all()` ou o `first()`. Não faz muito sentido você querer retornar mais de uma linha no `limit()` e usar o método `first()`, já que ele só permitirá retornar a primeira linha da query, mas tenha em mente que qualquer um dos dois métodos funciona, sem eles a query não retornará nada. Vamos aos exemplos.

```
# Query com método all()
try:
    res = db.session.query(Product).limit(10).all()
except Exception as e:
    res = []
    print(e)
finally:
    db.session.close()
    return res

# Query com método first()
try:
    res = db.session.query(Product).limit(10).first()
except Exception as e:
    res = []
    print(e)
finally:
    db.session.close()
    return res
```

order_by: o método `order_by` nos permite ordenar os dados retornados do banco de dados com base em um campo. Para isso precisaremos passar, ou o nome do campo por uma `string`, ou o objeto mais o atributo que representa o campo. Vamos ver os dois exemplos.

```
# Passando o campo como string
try:
    res = db.session.query(Product).order_by('date_created').all()
except Exception as e:
    res = []
    print(e)
```

```

finally:
    db.session.close()
    return res

# Passando o objeto e o atributo que representa o campo desejado
try:
    res = db.session.query(Product).order_by(Product.date_created).all()
except Exception as e:
    res = []
    print(e)
finally:
    db.session.close()
    return res

```

Você deve estar se perguntando como ordenar de forma decrescente, o que chamamos de `ORDER BY DESC`. Para isso precisamos importar um método chamado `desc` do `SQLAlchemy`, é bem simples:

```

from sqlalchemy import desc, asc

# Query em ordem crescente
try:
    res =
db.session.query(Product).order_by(asc(Product.date_created)).all()
except Exception as e:
    res = []
    print(e)
finally:
    db.session.close()
    return res

# Query em ordem decrescente
try:
    res =
db.session.query(Product).order_by(desc(Product.date_created)).all()
except Exception as e:
    res = []
    print(e)
finally:

```

```
db.session.close()
return res
```

Apesar de ser um padrão o `order_by` ser em ordem crescente, você pode optar por usar o método `asc()`, assim fica mais fácil para outras pessoas entenderem o que você deseja fazer na query. Legibilidade é fundamental em um sistema.

distinct: é outro dos métodos que precisam ser importados do `SQLAlchemy`, o método `distinct` é usado para eliminar duplicidades de um resultado. Por exemplo, se fazemos um `select` de produtos querendo trazer apenas as categorias que existem atualmente neles, podemos trazer com um `distinct`, para não precisarmos ver linhas duplicadas, tendo em vista que mais de um produto pode ter a mesma categoria e nosso foco é apenas trazer as categorias que estão atreladas a produtos. Vamos ver um exemplo a seguir, onde vamos trazer um `select` com um `distinct` por categoria.

```
from sqlalchemy import distinct

try:
    res = db.session.query(distinct(Product.category)).all()
except Exception as e:
    res = []
    print(e)
finally:
    db.session.close()
    return res
```

group_by: este método nos permite agrupar os dados com base em um campo da tabela. Podemos passar o nome do objeto e o atributo que representa o campo que deverá ser agrupado, ou o nome do campo como `string`.

```
# Query com o nome do objeto e o atributo que representa o campo que
# deverá ser agrupado.
try:
    res =
```

```

db.session.query(Product.category).group_by(Product.category).all()
except Exception as e:
    res = []
    print(e)
finally:
    db.session.close()
    return res

# Query com o nome do campo como string.
try:
    res = db.session.query(Product.category).group_by('category').all()
except Exception as e:
    res = []
    print(e)
finally:
    db.session.close()
    return res

```

Fique atento: o método `query` não pode receber o nome dos campos que deverão retornar como `string`. Sempre devem ser passados o nome do objeto e o atributo, por exemplo:
`query(Product.category)`, **nunca** `query('category')`.

having: a cláusula `having` é muito usada para especificar condições de filtragem em um `query` que esteja sendo agrupada. Vamos ver um exemplo a seguir.

```

try:
    res = db.session.query(Product.category,
func.sum(Product.price)).group_by(Product.category).having(func.sum(Product.price) > 150.00).all()
except Exception as e:
    res = []
    print(e)
finally:
    db.session.close()
    return res

```

join: por fim, existe a cláusula `join` . Para fazer uma query que una mais de uma tabela é muito simples. Veja o exemplo a seguir.

```
try:
    res =
db.session.query(Product,Category).join(Category).filter(Product.category=
=Category.id).all()
except Exception as e:
    res = []
    print(e)
finally:
    db.session.close()
    return res
```

Todos esses elementos podem se unir em uma única `query` , seguindo as regras que o SQL possui, ou seja, você nunca vai colocar um `order()` antes de um `join()` , por exemplo.

Algo que você precisa saber e que pode gerar confusão é que, se sua `model` possuir o método `__repr__` , o retorno da sua `query` será o que está no `__repr__` , se não ele retornará o nome do objeto. Vamos ver um exemplo.

```
"""Exemplo de uma model com o método __repr__"""
Class Category(db.Model):
...
```

```
def __repr__(self):
    return self.name

try:
    res = db.session.query(Category).first()
except Exception as e:
    res = []
    print(e)
finally:
    db.session.close()
    return res
```



```
# O resultado será algo do tipo [Calçados]

"""Exemplo de uma model sem o método __repr__"""
Class Product(db.Model):
    ...

try:
    res = db.session.query(Product).first()
except Exception as e:
    res = []
    print(e)
finally:
    db.session.close()
    return res

"""O resultado será algo do tipo
[<Product 1>, <Product 2>, <Product 3>]"""
```

O método `__repr__` é utilizado para representar o resultado da `model` de modo mais legível, mas isso não impede você de acessar os atributos da `model`. No caso da `Category` que possui esse método, nós podemos acessar o resultado do mesmo modo que na `Product`, como exemplificamos no tópico *Retorno de dados do SQLAlchemy*. É bom esclarecer isso, porque você pode acabar achando que esse retorno já é o resultado e que você ficaria impossibilitado de acessar outros atributos da tabela e isso não é verdade. Ele apenas representa o resultado da `model` uma maneira melhor.

7.5 Query execute

Por ser um framework bem completo para banco de dados `SQL`, o `SQLAlchemy` permite que criemos queries convencionais do `SQL` escrevendo-as diretamente, sem precisar passar por algum método `ORM` do framework. Ele oferece o método padrão `execute` para fazer isso. Vamos ver o exemplo a seguir.

```
try:
    products = db.session.execute('SELECT * FROM product;')
except Exception as e:
    products = []
    print(e)
finally:
    db.session.close()
    return products

for product in products:
    print(product.id)
    print(product.name)
    print(product.price)
```

Nesse caso não temos muito o que falar, afinal, dentro do método `execute` somente precisamos adicionar a `query` que desejamos executar.

Conclusão

Como podemos perceber, o `SQLAlchemy` é uma excelente ferramenta para se trabalhar. Ele agiliza e facilita nosso processo de desenvolvimento encurtando um enorme tempo que podemos dedicar a outras questões, já que ele é bem simples e nos proporciona tudo o que precisamos para que o sistema funcione corretamente.

Você não precisa utilizar o `SQLAlchemy` em seu projeto, mas ele é muito útil e o próprio Flask o utiliza em seu admin. Se preferir, pode optar por usar o modo convencional do `SQL` através do `SQLAlchemy` ou de qualquer outra `lib SQL`.

CAPÍTULO 8

Trabalhando com views

Trabalhar com `views` no Flask é incrivelmente fácil. Toda a comunicação ocorre pela `rota`. Ela é a responsável por realizar a comunicação entre a `controller` e a `view`. A seguir faremos nossa primeira tela `html`, que será a de login, e no decorrer do capítulo também alteraremos nosso arquivo `home` do admin.

8.1 Criando uma view - Tela de login

Existe um trecho de código no arquivo `app.py` que está do seguinte modo:

```
@app.route('/login/')
def login():
    return 'Aqui entrará a tela de login'
```

Será essa a rota que acionará a nossa `view` de login. Mude o trecho do código como no exemplo a seguir.

```
@app.route('/login/')
def login():
    return render_template('login.html')
```

Agora nossa rota de login está pronta para abrir o arquivo `html`.

Configurando o html

Nosso arquivo `html` já estará funcionando se quisermos testá-lo agora. Mas ele ainda não possui as configurações necessárias para apresentar uma aparência próxima do nosso tema do admin.

Como nosso template de login não faz parte diretamente da estrutura do admin, ele não possui os arquivos `css` e `js` que o

admin tem, por isso vamos precisar adicionar o Bootstrap através de uma fácil instalação e configuração. Assim nossas `views` que são independentes do admin poderão ficar tão bonitas quanto o admin.

Para instalar o Bootstrap, rode o comando a seguir:

```
(venv) tiago_luiz@ubuntu:~/livro_flask$ pip install flask-bootstrap
```

Agora abra o arquivo `app.py` e modifique como no exemplo:

app.py

```
"""Adicione a linha a seguir em seu arquivo no topo do arquivo"""
from flask_bootstrap import Bootstrap
"""Até aqui"""

    """Adicione a linha a seguir em seu arquivo antes do db.init(app)"""
    Bootstrap(app)
    """Até aqui"""
    db.init_app(app)
```

Agora que configuramos o arquivo `app.py` vamos até a pasta `templates` e vamos adicionar o código a seguir no arquivo `login.html`.

templates/login.html

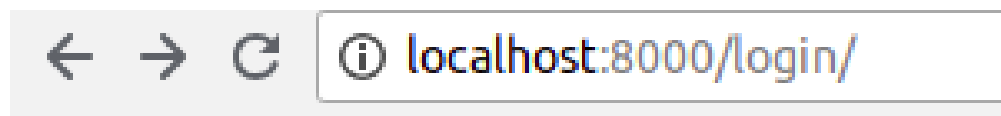
```
{% extends "bootstrap/base.html" %}
{% block title %}Login{% endblock %}

{% block content %}
    <h1>Olá mundo!!!</h1>
{% endblock %}
```

Vamos começar a entender o que cada linha faz. Mas antes disso vamos esclarecer. O Flask utiliza o `jinja2` como linguagem de template que interage entre o `python` e o `html`. Ele possui alguns códigos de sintaxe própria que fazem a interação entre essas duas linguagens.

- `{% extends "bootstrap/base.html" %}` : esse é o html base que nosso arquivo de login herda; esse arquivo base contém o arquivo `css` e `js` do `bootstrap` e o arquivo `jquery` ;
- `{% block title %}` : aqui é o local onde inserimos o título da página `html` ;
- `{% block content %}` : aqui é o local do arquivo em que inserimos todo o conteúdo que fica dentro das tags `body` ;
- `{% endblock %}` : essa é a tag que usamos no `jinja2` para fechar qualquer bloco de código `jinja2` .

Se abrirmos a tela de login na rota `http://localhost:8000/login/` veremos a seguinte tela.



Olá mundo!!!

Figura 8.1: Código-fonte da tela de Login com bootstrap.

Se abrirmos o código-fonte do `html` no navegador veremos os arquivos do `bootstrap` em nossa página de login.

```
view-source:localhost:8000/login/
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Login</title>
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6
7     <!-- Bootstrap -->
8     <link href="//cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/3.3.7/css/bootstrap.min.css" rel="stylesheet">
9     <link rel="stylesheet" href="/static/css/custom.css">
10
11   </head>
12   <body>
13
14     <h1>Olá mundo!!!</h1>
15
16
17
18
19
20     <script src="//cdnjs.cloudflare.com/ajax/libs/jquery/1.12.4/jquery.min.js"></script>
21     <script src="//cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/3.3.7/js/bootstrap.min.js"></script>
22     <script src="/static/js/custom.js"></script>
23
24   </body>
25 </html>
26
```

Figura 8.2: Código-fonte da tela de Login com bootstrap.

Agora que temos nosso arquivo com o bootstrap já configurado, vamos adicionar os arquivos `login.css` e `custom.js` que estão em nossa pasta `static`, pois eles serão fundamentais para criarmos nossas próprias regras `css` e scripts `js`. Altere seu arquivo `login.html` conforme o exemplo a seguir:

templates/login.html

```
{% extends "bootstrap/base.html" %}
{% block title %}Login{% endblock %}

""""Adicione o código a seguir""""
{% block styles %}
    {{super()}}
    <link rel="stylesheet" href="{url_for('.static',
filename='css/login.css')}">
{% endblock %}
""""Até aqui""""

{% block content %}
    <h1>Olá mundo!!!</h1>
{% endblock %}

""""Adicione o código a seguir""""
```

```
{% block scripts %}
    {{super()}}
    <script src="{{url_for('.static', filename='js/custom.js')}}"></script>
{% endblock %}
```

Vamos entender o que foi adicionado no arquivo.

- `{% block styles %}` : bloco do jinja2 que representa a área de styles do html ; quando criamos esse bloco e colocamos a tag `super()` dentro, ela traz todos os styles do template pai. No caso do `login.html` o pai é `bootstrap/base.html` ;
- `{% block scripts %}` : bloco do jinja2 que representa a área de scripts do html ; quando criamos esse bloco e colocamos a tag `super()` dentro, ela traz todos os scripts do template pai. No caso do `login.html` o pai é `bootstrap/base.html` ;
- `{{super()}}` : usamos para trazer do template pai o que desejamos. Como falamos nos dois pontos anteriores, o `super()` vai trazer tudo referente ao bloco em que ele se encontra. Assim conseguimos trazer pedaços do arquivo pai para o filho, sem precisar trazer tudo;
- `url_for('Pasta', filename='Arquivo')` : usado para localizar o arquivo que desejamos trazer para nosso html . No arquivo `login.html` colocamos `.static` para representar que queríamos acessar da pasta `./static/` .

Criando o formulário de login

Você tem liberdade para adicionar outra aparência ao seu login, mas a seguir você encontrará uma estrutura html que já será suficiente para que o sistema funcione corretamente. Abra o arquivo `login.html` e altere o código conforme a seguir:

templates/login.html

```
<!-- Retire as linhas -->
{% block content %}
    <h1>Olá mundo!!!</h1>
{% endblock %}
```

```

<!-- E no lugar dessas três linhas adicione -->
{% block content %}
    <form class="form-signin text-center" action="/login/" method="post">
        
        <h1 class="h3 mb-3 font-weight-normal">Login</h1>
        <div class="checkbox mb-3">
            <label for="inputEmail" class="sr-only">E-mail</label>
            <input name="email" type="email" id="inputEmail" class="form-
control" placeholder="E-mail" required autofocus>
        </div>
        <div class="checkbox mb-3">
            <label for="inputPassword" class="sr-only">Senha</label>
            <input name="password" type="password" id="inputPassword"
class="form-control" placeholder="Senha" required>
        </div>
        <button class="btn btn-lg btn-primary btn-block"
type="submit">Entrar</button>
    </form>
{% endblock %}

```

Agora vamos adicionar os estilos `css` em nosso arquivo `static/css/login.css`, pois serão eles que deixarão nossa tela de login mais apresentável e profissional. Adicione o código:

static/css/login.css

```

html, body {
    height: 100%;
}
body {
    display: -ms-flexbox;
    display: -webkit-box;
    display: flex;
    -ms-flex-align: center;
    -ms-flex-pack: center;
    -webkit-box-align: center;
    align-items: center;
    -webkit-box-pack: center;
    justify-content: center;
}

```



```
padding-top: 40px;
padding-bottom: 40px;
background-color: #f5f5f5;
}
.form-signin {
width: 100%;
max-width: 330px;
padding: 15px;
margin: 0 auto;
}
.form-signin .checkbox {
font-weight: 400;
}
.form-signin .form-control {
position: relative;
box-sizing: border-box;
height: auto;
padding: 10px;
font-size: 16px;
}
.form-signin .form-control:focus {
z-index: 2;
}
.form-signin input[type="email"] {
margin-bottom: -1px;
border-bottom-right-radius: 0;
border-bottom-left-radius: 0;
}
.form-signin input[type="password"] {
margin-bottom: 10px;
border-top-left-radius: 0;
border-top-right-radius: 0;
}
```

Com tudo isso adicionado, nossa tela de login ficará como a tela a seguir:

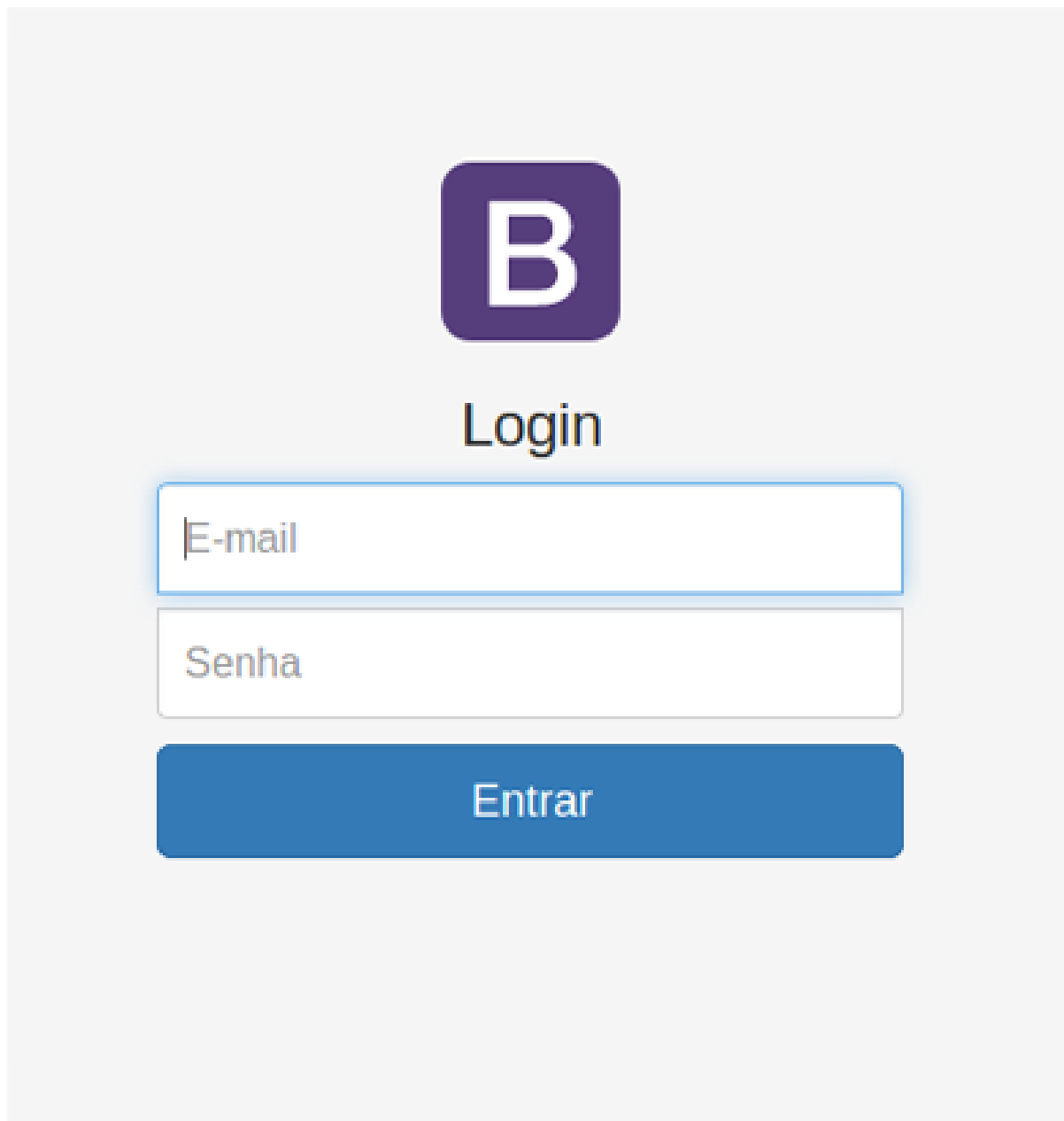


Figura 8.3: Tela de login

Argumentos por rota

Podemos passar argumentos por uma rota. Vamos ver um exemplo simples em que a `view` de login receberá uma mensagem enviada pela rota. Altere o arquivo `app.py` e adicione o código a seguir.

app.py

```
@app.route('/login/')
def login():

    """Altere o código a seguir para ficar assim"""
    return render_template('login.html', message="Essa é uma mensagem que
veio da rota")
```

Agora vamos adicionar o código a seguir no arquivo de template
login.html .

templates/login.html

```
<!-- Adicione essa linha abaixo do
h1 com texto login
-->
<h2>{{message}}</h2>
```

Sua tela de login agora exibirá a mensagem que foi passada como argumento. Para isso precisamos apenas passar a variável dentro de duas chaves, `{{nome_da_variavel}}` . Dessa forma, teremos nossa variável exibindo o valor que ela possui.



Figura 8.4: Views com argumento

Podemos passar argumentos de todos os tipos de dados. A seguir vamos editar nossa `view home` do admin e aprofundar mais sobre como lidar com outros tipos de dados que são passados como argumentos para o `html`.

8.2 Personalizando a Home Admin

Como vimos, podemos passar argumentos para o método `render_template` e nossa `view html` vai recebê-lo normalmente como variáveis. Veremos a seguir como trabalhar com esses dados de forma avançada.

Em nossa `home` do admin, vamos adicionar alguns elementos `html` para que ela não fique uma tela tão vazia como está atualmente.

Vamos inserir em nossa `home` três cards. Cada card trará o valor total de uma tabela do banco, sendo usuários, categorias e produtos. Vamos começar criando os métodos que vão calcular o total de itens que cada tabela dessas possui. Vamos modificar os arquivos `User.py`, `Category.py` e `Product.py` da pasta `model`.

model/User.py

```
"""Adicione a linha a seguir no topo"""
from sqlalchemy import func
"""Até aqui"""

"""Adicione o método a seguir dentro da
classe abaixo dos demais métodos"""
def get_total_users(self):
    try:
        res = db.session.query(func.count(User.id)).first()
    except Exception as e:
        res = []
        print(e)
    finally:
        db.session.close()
    return res
```

model/Category.py

```
"""Adicione a linha a seguir no topo"""
from sqlalchemy import func
"""Até aqui"""

"""Adicione o método a seguir dentro da
classe abaixo dos demais métodos"""
def get_total_categories(self):
    try:
        res = db.session.query(func.count(Category.id)).first()
    except Exception as e:
        res = []
        print(e)
    finally:
        db.session.close()
    return res
```

model/Product.py

```
"""Adicione a linha a seguir no topo"""
from sqlalchemy import func
"""Até aqui"""

"""Adicione o método a seguir dentro da
classe abaixo dos demais métodos"""
def get_total_products(self):
    try:
        res = db.session.query(func.count(Product.id)).first()
    except Exception as e:
        res = []
        print(e)
    finally:
        db.session.close()
    return res
```

Todos os métodos que criamos utilizam o que aprendemos no capítulo sobre SQLAlchemy . Agora, precisamos passar esses métodos em nossa ModelAndView da home . Vamos alterar o arquivo Views.py para executar os métodos e enviar os resultados na view do arquivo home_admin.html .

Views.py

```
"""Adicione o código a seguir no topo
do arquivo"""
from model.User import User
from model.Category import Category
from model.Product import Product
"""Até aqui"""
class HomeView(AdminIndexView):
    @expose('/')
    def index(self):
        """Altere seu código como o do exemplo a seguir"""
        user_model = User()
        category_model = Category()
        product_model = Product()

        users = user_model.get_total_users()
```

```

categories = category_model.get_total_categories()
products = product_model.get_total_products()

return self.render('home_admin.html', report={
    'users': 0 if not users else users[0],
    'categories': 0 if not categories else categories[0],
    'products': 0 if not products else products[0]
})
"""Até aqui"""

```

Como podemos perceber, estamos importando as `models` em nosso arquivo `Views.py` do `admin`, para assim executarmos os métodos que trazem o total de itens de cada uma dessas tabelas. Após isso, estamos passando para o argumento `report` um dicionário com os valores totais de cada tabela.

Agora precisamos adicionar nossos `cards` no arquivo `html` da `home`. Vamos adicionar o código a seguir no arquivo `home_admin.html`, pode substituir tudo que tem no arquivo.

home_admin.html

```

{% extends admin_base_template %}
{% block body %}

<div class="row mb-3">
    <div class="col-xl-4 col-lg-4 col-md-4 col-sm-4 col-xs-12 py-2">
        <div class="card bg-success text-white h-100">
            <div class="card-body bg-success">
                <div class="rotate">
                    <i class="fa fa-user fa-4x"></i>
                </div>
                <h6 class="text-uppercase">Usuários</h6>
                <h1 class="display-4">{{report.users}}</h1>
            </div>
        </div>
    </div>
    <div class="col-xl-4 col-lg-4 col-md-4 col-sm-4 col-xs-12 py-2">
        <div class="card text-white bg-info h-100">
            <div class="card-body bg-info">

```

```

        <div class="rotate">
            <i class="fa fa-list fa-4x"></i>
        </div>
        <h6 class="text-uppercase">Categorias</h6>
        <h1 class="display-4">{{report.categories}}</h1>
    </div>
</div>
</div>
<div class="col-xl-4 col-lg-4 col-md-4 col-sm-4 col-xs-12 py-2">
    <div class="card text-white bg-warning h-100">
        <div class="card-body bg-warning">
            <div class="rotate">
                <i class="fa fa-briefcase fa-4x"></i>
            </div>
            <h6 class="text-uppercase">Produtos</h6>
            <h1 class="display-4">{{report.products}}</h1>
        </div>
    </div>
</div>
</div>
</div>
{% endblock %}

```

Esse exemplo é similar ao da mensagem na tela de login. Aqui estamos recebendo o argumento `report` como um dicionário de três chaves, para ser exibido no `html`. Devemos passar a chave desejada em nosso `html`, sendo cada chave para uma finalidade: `{{report.user}}` para o total de usuários, `{{report.categories}}` para o total de categorias e `{{report.products}}` para o total de produtos.

Agora para finalizar essa etapa, precisamos adicionar estilo `css` em nossa `home`, para que os cards fiquem bem bonitos. Vamos adicionar o arquivo `home.css` que está na pasta `static` e também o arquivo de ícones do `fontawesome`, que é muito usado para adicionar ícones em sites e sistema.

Adicionar arquivos `css` em templates do admin é um pouco diferente do que já vimos, precisaremos modificar o arquivo `views.py`. Mas é bem simples, veja o exemplo a seguir.

Views.py

```
class HomeView(AdminIndexView):
    """Adicione o código a seguir"""
    extra_css = [config.URL_MAIN +
'static/css/home.css', 'https://maxcdn.bootstrapcdn.com/font-
awesome/4.7.0/css/font-awesome.min.css']
    """Até aqui"""

    @expose('/')
    def index(self):
        user_model = User()
```

Vamos entender melhor. Como podemos ver, quando queremos adicionar um arquivo `css` em nossa `view html`, passamos o atributo `extra_css` recebendo uma lista com os arquivos. No exemplo anterior, adicionamos um arquivo local utilizando o caminho `config.URL_MAIN + 'static/css/home.css'` que contém a concatenação da `url` padrão do sistema e o caminho estático do arquivo `home.css`. Adicionamos também um segundo arquivo, porém ele está em outro servidor, então passamos a `url` dele. Assim os dois foram adicionados à `home` do `admin` e podemos personalizar nossa página usando os ícones do `fontawesome` e adicionar novos estilos `css` no arquivo `home.css`.

Adicione o código a seguir no arquivo `home.css`.

static/css/home.css

```
body, html {
    height:100%;
}
.py-2{
    overflow: hidden;
}
.card {
    overflow: hidden;
    padding: 0 10px;
    box-sizing: border-box;
    border-radius: 5px;
```

```

}
.card-body .rotate {
  z-index: 8;
  float: right;
  height: 100%;
}

.card-body .rotate i {
  color: rgba(20, 20, 20, 0.15);
  position: absolute;
  left: auto;
  right: 21px;
  bottom: 0px;
  display: block;
  -webkit-transform: rotate(-44deg);
  -moz-transform: rotate(-44deg);
  -o-transform: rotate(-44deg);
  -ms-transform: rotate(-44deg);
  transform: rotate(-44deg);
}

```

Com tudo isso feito, se entrarmos em nossa página `home`, veremos a seguinte tela.

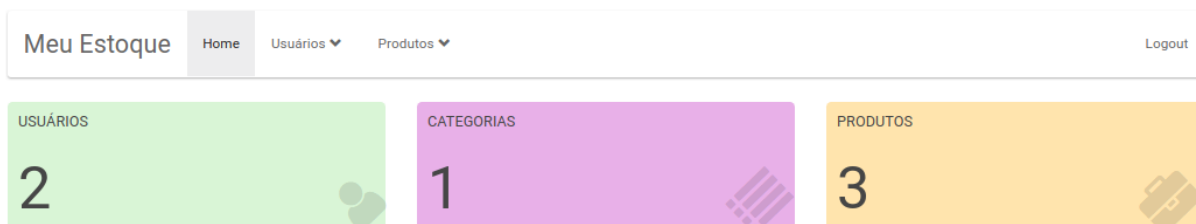


Figura 8.5: View home customizada com cards de quantidade de itens.

Agora nossa `home` está bem melhor do que antes. Podemos ver que é bem simples personalizar uma `view` html no Flask.

Por fim, para entendermos mais a fundo sobre a interação entre o back-end e a `view` do Flask, vamos listar os 5 últimos produtos criados em nosso sistema e adicionar em uma tabela visual em nossa `home`, logo abaixo dos cards. Para isso, precisamos criar um

método no arquivo `Product.py` da `model` com um `limit` de 5 e o `order_by` na data de criação em ordem decrescente, que fará a ordenação do mais novo para o mais antigo.

Abra o arquivo `User.py` da `model` e adicione o código a seguir abaixo dos demais métodos.

model/Product.py

```
def get_total_products(self):
    try:
        res = db.session.query(func.count(Product.id)).first()
    except Exception as e:
        res = []
        print(e)
    finally:
        db.session.close()
    return res

"""Adicione o código a seguir"""
def get_last_products(self):
    try:
        res =
db.session.query(Product).order_by(Product.date_created).limit(5).all()
    except Exception as e:
        res = []
        print(e)
    finally:
        db.session.close()
    return res
```

Agora vamos adicionar o método na `Views.py`, para ela poder enviar o resultado para o `html`.

Views.py

```
...
"""Existe mais código anterior, que é irrelevante para o exemplo"""

@expose('/')
def index(self):
```

```

user_model = User()
category_model = Category()
product_model = Product()

users = user_model.get_total_users()
categories = category_model.get_total_categories()
products = product_model.get_total_products()

"""Adicione o código a seguir"""
last_products = product_model.get_last_products()
"""Até aqui"""

"""Altere o seu método self.render pelo que está a seguir"""
return self.render('home_admin.html', report={
    'users': users[0],
    'categories': categories[0],
    'products': products[0]
}, last_products=last_products)

```

Com tudo isso certo, vamos agora para a `view` da `home`. Altere seu arquivo `home_admin.html` e adicione o `html` a seguir após o `html` dos cards. Não apague o `html` dos cards.

templates/home_admin.html

```

<div class="row mb-3">
    ...
    <!-- O código dos cards está aqui -->
    ...
</div>

<div class="row">
    <div class="col-xl-12 col-lg-12 col-md-12 col-sm-12 col-xs-12">
        <h3>Últimos produtos</h3>
        <table class="table table-striped table-hover ">
            <thead>
                <tr>
                    <th>ID</th>
                    <th>Nome</th>
                    <th>Preço</th>
                    <th>Quantidade</th>

```

```

        <th>Data de Criação</th>
    </tr>
</thead>
<tbody>
    {% for product in last_products %}
        <tr>
            <td>{{product.id}}</td>
            <td>{{product.name}}</td>
            <td>{{product.price}}</td>
            <td>{{product.qtd}}</td>
            <td>{{product.date_created}}</td>
        </tr>
    {% endfor %}
</tbody>
</table>
</div>
</div>

```

Como o resultado que esperávamos era uma lista com vários produtos, foi possível passar essa lista por um `for` e iterá-lo de modo que gerasse uma tabela com todos os produtos. Se você perceber, não é muito diferente do `for` padrão do `python`, tendo apenas a questão da sintaxe do `jinja2`, que possui `{%}` entre o código e também o `endfor`, que precisa existir.

Pronto. Agora nossa `home` está bem mais profissional e completa do que antes. Fique à vontade para fazer mais personalizações se quiser. Se atualizarmos a página teremos uma `home` assim.

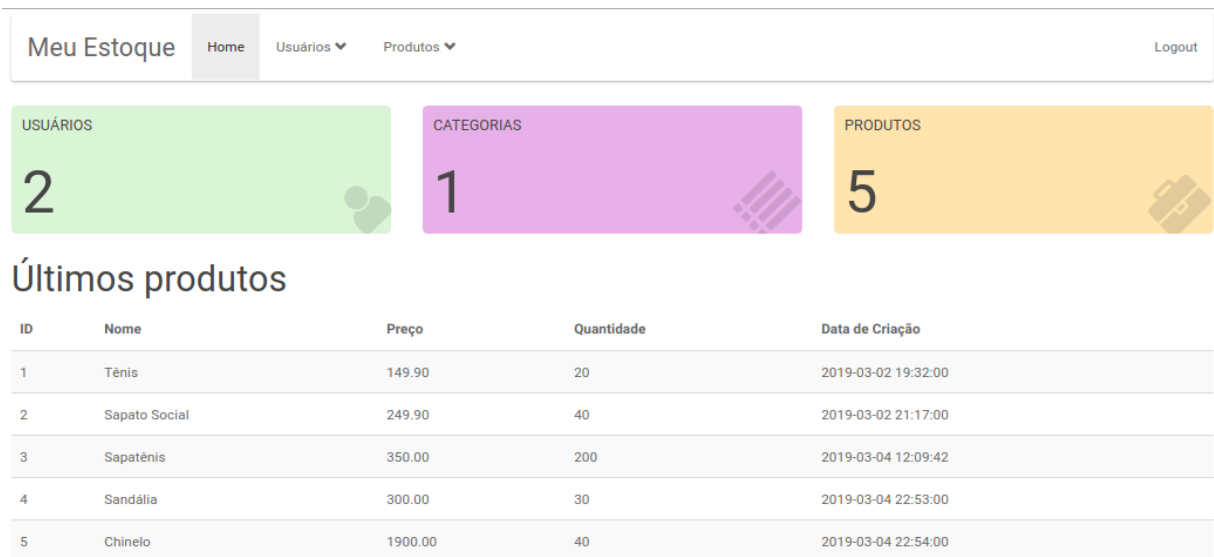


Figura 8.6: View home customizada com tabela de últimos itens criados.

Conclusão

Como podemos ver, trabalhar com `views` no Flask não é algo complicado. Além de ser simples é bem útil personalizar nossas `views` da maneira que melhor desejarmos. Isso é fundamental para que nosso sistema tenha as características que são necessárias segundo sua regra de negócios. Este é dos pontos positivos do Flask, a personalização completa de seu sistema.

Autenticação e requisição segura

Desenvolver uma aplicação web segura é algo fundamental e tem se tornado mais simples a cada dia. O Python possui ferramentas excelentes que em conjunto com o Flask permitem que a aplicação web fique inteiramente segura.

Nesta parte aprenderemos a trabalhar com autenticação segura em serviços web, falaremos sobre utilização de decorators para reforçar a segurança do projeto, criação de uma área administrativa segura e criptografia para uso em senhas (autenticação de usuários) e tokens para API e recuperação de senha.



Figura 1: Autenticação e requisição segura

CAPÍTULO 9

API Rest no Flask

Um dos pontos mais fortes do Flask é sua simplicidade para a criação de uma `API Rest` .

Neste capítulo, criaremos nossa `API Rest`, que permitirá aos usuários acessarem os dados do nosso sistema através de rotas que chamamos de `endpoints`.

`Endpoints` são rotas/urls que criamos para que um `client`, ou seja, uma aplicação externa, possa acessar os dados do nosso serviço.

9.1 Criando um Endpoint

Vamos começar criando nossa `model` e `controller`, após isso criaremos 3 `endpoints`, que nos permitirão ter acesso à listagem de produtos.

Vamos começar criando as `queries` dentro dos arquivos das `models`. Abra os respectivos arquivos `Product.py` e `User.py` da pasta `model`.

No arquivo `Product.py` modificaremos o método `get_all()` para passar a receber um limite de linhas. Nosso método vai verificar se `limit` é nulo ou não, se for nulo ele listará todos os produtos, se não for, ele listará a quantidade limite. Usaremos o método `order_by` junto ao `limit` para que nesse caso ele seja ordenado pela data de criação.

Criaremos também um método que trará o produto baseado em seu `id`. Vamos chamá-lo de `get_product_by_id()`.

model/Product.py

```
"""Modifique o método get_all do seu projeto para ficar como o código a seguir"""
def get_all(self, limit):
    try:
        if limit is None:
            res = db.session.query(Product).all()
        else:
            res =
```

```

db.session.query(Product).order_by(Product.date_created).limit(limit).all(
)
    except Exception as e:
        res = []
        print(e)
    finally:
        db.session.close()
    return res

"""Adicione o método a seguir no arquivo, ele ainda não existe"""
def get_product_by_id(self):
    try:
        res = db.session.query(Product).filter(Product.id==self.id).first()
    except Exception as e:
        res = None
        print(e)
    finally:
        db.session.close()
    return res

```

No arquivo `User.py` já existem 2 métodos que precisam ser preenchidos e ainda não foram, um deles veremos no próximo capítulo, de autenticação, que é o `get_user_by_email`, mas vamos adicionar o código a seguir no método `get_user_by_id`.

model/User.py

```

def get_user_by_id(self):
    """Adicione o código a seguir no método"""
    try:
        res = db.session.query(User).filter(User.id==self.id).first()
    except Exception as e:
        res = None
        print(e)
    finally:
        db.session.close()
    return res

```

Agora que criamos e modificamos os arquivos da `model`, precisamos criar nossos métodos nas `controllers`. Para isso, vamos editar os

arquivos `Product.py` e `User.py` que estão na pasta `controller` e adicionar os códigos a seguir.

controller/Product.py

```
def get_products(self, limit):
    result = []
    try:
        res = self.product_model.get_all(limit=limit)

        for r in res:
            result.append({
                'id': r.id,
                'name': r.name,
                'description': r.description,
                'qtd': str(r.qtd),
                'price': str(r.price),
                'image': r.image,
                'date_created': r.date_created
            })
        status = 200
    except Exception as e:
        print(e)
        result = []
        status = 400
    finally:
        return {
            'result': result,
            'status': status
        }

def get_product_by_id(self, product_id):
    result = {}
    try:
        self.product_model.id = product_id
        res = self.product_model.get_product_by_id()

        result = {
            'id': res.id,
            'name': res.name,
            'description': res.description,
```

```

        'qtd': str(res.qtd),
        'price': str(res.price),
        'image': res.image,
        'date_created': res.date_created
    }

    status = 200
except Exception as e:
    print(e)
    result = []
    status = 400
finally:
    return {
        'result': result,
        'status': status
    }

```

Você com certeza percebeu que o método que retorna uma lista de produtos passou por um `for`, onde construímos um dicionário baseado no resultado. Isso acontece, pois precisaremos enviar para o `client` um resultado `json` e, para isso, só podemos fazer enviando uma lista de dicionários, que futuramente será convertida para um `json`.

O resultado padrão que o `SQLAlchemy` entrega é uma lista de objetos, como falamos em capítulos anteriores, algo parecido com isso:

`[<Product 1>, <Product 2>, <Product 3>]`. Só que isso é impossível de ser convertido em `json`, então precisamos fazer um `for` que vai acessar cada objeto, e quando acessamos cada objeto, através de seus atributos, conseguimos acessar cada campo da tabela correspondente, e assim construímos nossa lista de dicionários.

Se você reparar, o mesmo se aplica ao método que retorna apenas um produto. A diferença é que ele retorna apenas o objeto e não uma lista de objetos. Ficaria assim: `<Product 1>`.

Vamos adicionar essas alterações também no arquivo `User.py` da `controller`.

controller/User.py

```
def get_user_by_id(self, user_id):
    result = {}
    try:
        self.user_model.id = user_id
        res = self.user_model.get_user_by_id()
        result = {
            'id': res.id,
            'name': res.username,
            'email': res.email,
            'date_created': res.date_created
        }
        status = 200
    except Exception as e:
        print(e)
        result = []
        status = 400
    finally:
        return {
            'result': result,
            'status': status
        }
```

Agora que as `models` e as `controllers` estão com tudo pronto, vamos ao principal, que é a criação dos `endpoints`. Abra o arquivo `app.py` e vamos adicionar as rotas a seguir dentro dele.

app.py

```
"""Altera o import do topo do arquivo"""
from flask import Flask, request, redirect, render_template, Response,
json

"""Insira o código a seguir após a última
rota criada e antes do return app"""
@app.route('/products/', methods=['GET'])
@app.route('/products/<limit>', methods=['GET'])
def get_products(limit=None):
    header = {}
```

```

    product = ProductController()
    response = product.get_products(limit=limit)
    return Response(json.dumps(response, ensure_ascii=False),
mimetype='application/json'), response['status'], header

@app.route('/product/<product_id>', methods=['GET'])
def get_product(product_id):
    header = {}

    product = ProductController()
    response = product.get_product_by_id(product_id = product_id)

    return Response(json.dumps(response, ensure_ascii=False),
mimetype='application/json'), response['status'], header

@app.route('/user/<user_id>', methods=['GET'])
def get_user_profile(user_id):
    header = {}

    user = UserController()
    response = user.get_user_by_id(user_id=user_id)

    return Response(json.dumps(response, ensure_ascii=False),
mimetype='application/json'), response['status'], header

"""O return app sempre será a última linha"""
return app

```

Vamos entender o que há de novo no código.

- `Response(json, mimetype, status, header)` : esse é o método responsável por enviar um resultado para o `client`. Ele recebe 4 argumentos, que são essenciais para o retorno funcionar corretamente:
 - *json*: aqui passamos o JSON com o corpo de tudo que desejamos enviar para o `client`. Nos exemplos, estamos enviando um `json` de uma lista de produtos, de um único

produto e de um único usuário, e para isso precisamos do método `json.dumps`, que explicaremos a seguir;

- *mimetype*: é o argumento que usamos para dizer qual o formato de arquivo que estamos enviando para o `client`. No exemplo estamos enviando de tipo `json`;
 - *status*: status do envio para o `client`. Usado para dizer ao requisitor se a requisição funcionou ou não e, baseado no código enviado, o que o requisitor deve fazer. A lista de códigos `HTTP` que existe pode ser encontrada facilmente na internet, mas deixarei no final do capítulo uma lista com os códigos de status `HTTP`;
 - *header*: também um JSON, onde enviamos dados mais restritos como tokens de segurança, o tipo da autenticação e outros itens à nossa escolha. Veremos mais a fundo no capítulo de autenticação, por ora estamos enviando-o vazio, `{}`.
- `json.dumps()`: método que utilizamos para converter nosso dicionário do `python` em uma string `json` que poderá ser enviada pelo método `Response` ao requisitor, no caso, o `client`.

Você deve ter percebido que colocamos duas rotas dentro de um método. Isso também é possível e muito útil. No caso do exemplo, criamos duas rotas, uma que passa um `limit` e a outra que não passa, o que permite que passemos ou não o valor de `limit`, sem que ocorra erro.

Por fim, precisamos dizer à nossa API que estamos autorizando que requisições externas sejam feitas a ela, ou seja, URLs que não são a do nosso serviço vão requisitá-lo e ele precisa autorizar isso. Para isso, adicione o código a seguir, abaixo do código de configuração da `app.py`.

```
db.init_app(app)
```

```
"""Adicione o código a seguir abaixo do db.init_app(app).  
Ele deve ficar antes de qualquer rota"""  
@app.after_request
```

```
def after_request(response):  
    response.headers.add('Access-Control-Allow-Origin', '*')  
    response.headers.add('Access-Control-Allow-Headers', 'Content-Type')  
    response.headers.add('Access-Control-Allow-Methods',  
'GET,PUT,POST,DELETE,OPTIONS')  
    return response
```

Se testarmos qualquer um dos endpoints pelo postman obteremos os resultados esperados. Vamos ver.

GET

http://localhost:8000/products/

Authorization

Headers

Body

Pre-request Script

Tests

Key	Value
New key	Value

Body

Cookies

Headers (4)

Test Results

Pretty

Raw

Preview

JSON

```

1 {
2   "result": [
3     {
4       "date_created": "Sat, 02 Mar 2019 19:32:00 GMT",
5       "description": "Calçado",
6       "id": 1,
7       "image": "",
8       "name": "Tênis",
9       "price": "149.90",
10      "qtd": "20"
11     },
12     {
13       "date_created": "Sat, 02 Mar 2019 21:17:00 GMT",
14       "description": "Calçado",
15       "id": 2,
16       "image": "",
17       "name": "Sapato Social",
18       "price": "249.90",
19       "qtd": "40"
20     },
21     {
22       "date_created": "Mon, 04 Mar 2019 12:09:42 GMT",
23       "description": "Calçado",

```

Figura 9.1: Teste do endpoint de todos os produtos. Rota <http://localhost:8000/products/>

GET

http://localhost:8000/products/2

Authorization

Headers

Body

Pre-request Script

Tests

Key	Value
New key	Value

Body

Cookies

Headers (4)

Test Results

Pretty

Raw

Preview

JSON

```

1 {
2   "result": [
3     {
4       "date_created": "Sat, 02 Mar 2019 19:32:00 GMT",
5       "description": "Calçado",
6       "id": 1,
7       "image": "",
8       "name": "Tênis",
9       "price": "149.90",
10      "qtd": "20"
11    },
12    {
13      "date_created": "Sat, 02 Mar 2019 21:17:00 GMT",
14      "description": "Calçado",
15      "id": 2,
16      "image": "",
17      "name": "Sapato Social",
18      "price": "249.90",
19      "qtd": "40"
20    }
21  ],
22  "status": 200
23 }

```

Figura 9.2: Teste do endpoint de todos os produtos com limit. Rota <http://localhost:8000/products/2>

GET

http://localhost:8000/product/1

Authorization

Headers

Body

Pre-request Script

Test Results

Key	Value
New key	Value

Body

Cookies

Headers (4)

Test Results

Pretty

Raw

Preview

JSON

```
1 {
2   "result": {
3     "date_created": "Sat, 02 Mar 2019 19:32:00 GMT",
4     "description": "Calçado",
5     "id": 1,
6     "image": "",
7     "name": "Tênis",
8     "price": "149.90",
9     "qtd": "20"
10  },
11  "status": 200
12 }
```

Figura 9.3: Teste do endpoint de produto por id. Rota `http://localhost:8000/product/1`

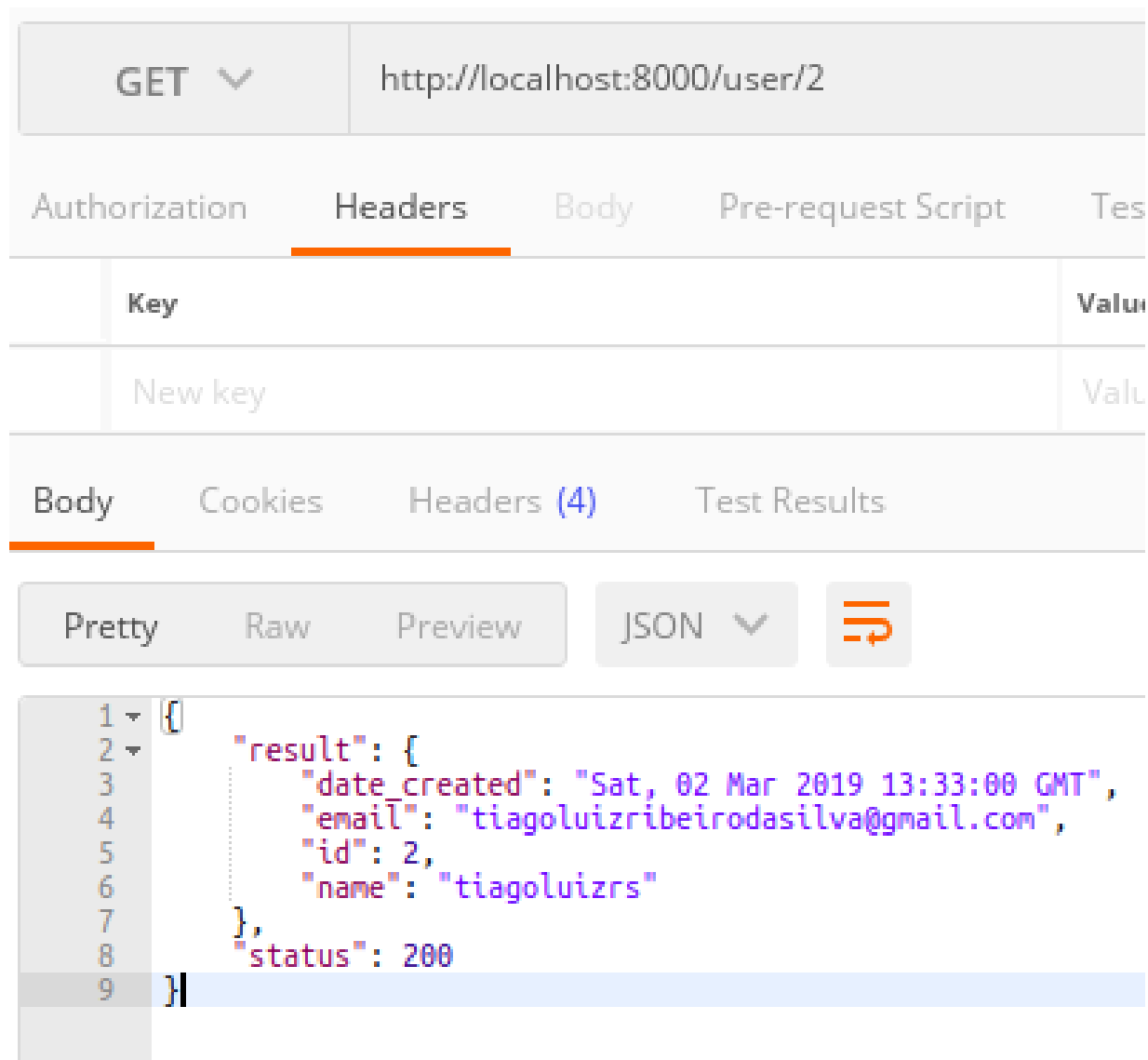


Figura 9.4: Teste do endpoint de usuário por id. Rota `http://localhost:8000/user/2`

Não se esqueça que os IDs precisam existir para a consulta não retornar vazia.

9.2 Recebendo dados JSON

Assim como enviamos valores de um formulário, podemos enviar valores via `json`. Não é nada tão diferente do que fazemos com formulários, precisamos apenas mudar o método `request`. Vamos ver um pequeno exemplo a seguir, pois faremos algo mais complexo no próximo capítulo.

```
"""Quando recebemos valores vindos de um formulário"""  
email = request.form['email']
```

```
"""Quando recebemos valores vindos de um body json"""  
email = request.json['email']
```

```
"""Quando recebemos valores vindos de um header json"""  
token = request.header['token']
```

Lista de códigos de estado HTTP

Como prometido, a seguir temos a lista com todos os status HTTP que existem.

Código	Significado
100	Continuar
101	Mudando protocolos
102	Processamento (WebDAV) (RFC 2518)
122	Pedido-URI muito longo
2xx	Sucesso
200	Ok
201	Criado
202	Aceito
203	Não autorizado (desde HTTP/1.1)
204	Nenhum conteúdo

Código	Significado
205	Reset
206	Conteúdo parcial
207	Status Multi (WebDAV) (RFC 4918)
3xx	Redirecionamento
300	Múltipla escolha
301	Movido
302	Encontrado
303	Consulte outros
304	Não modificado
305	Use Proxy (desde HTTP/1.1)
306	Proxy Switch
307	Redirecionamento temporário (desde HTTP/1.1)
308	Redirecionamento permanente
4xx	Erro de cliente
400	Requisição inválida
401	Não autorizado
402	Pagamento necessário
403	Proibido
404	Não encontrado
405	Método não permitido
406	Não aceitável
407	Autenticação de proxy necessária

Código	Significado
408	Tempo de requisição esgotou (Timeout)
409	Conflito geral
410	Gone
411	Comprimento necessário
412	Pré-condição falhou
413	Entidade de solicitação muito grande
414	Pedido-URI Too Long
415	Tipo de mídia não suportado
416	Solicitada de faixa não satisfatória
417	Falha na expectativa
418	Eu sou um bule de chá
422	Entidade improcessável (WebDAV) (RFC 4918)
423	Fechado (WebDAV) (RFC 4918)
424	Falha de dependência (WebDAV) (RFC 4918)
425	Coleção não ordenada (RFC 3648)
426	Upgrade obrigatório (RFC 2817)
450	Bloqueados pelo Controle de Pais do Windows
499	Cliente fechou Pedido (utilizado em ERPs/VPSA)
5xx	Outros erros
500	Erro interno do servidor (Internal Server Error)
501	Não implementado (Not implemented)
502	Bad Gateway

Código	Significado
503	Serviço indisponível (Service Unavailable)
504	Gateway Time-Out
505	HTTP Version not supported

Para saber mais detalhes sobre cada código de status, acesse:
https://pt.wikipedia.org/wiki/Lista_de_c%C3%B3digos_de_estado_HTTP/

Conclusão

Como podemos ver, criar uma API no Flask não é nada difícil, por isso ele é escolhido na maioria dos casos que necessitam que o sistema forneça uma API, ou até mesmo para ser usado somente como API sem um sistema admin por trás. A flexibilidade do Flask é um ponto fundamental que o torna uma ferramenta excelente.

No próximo capítulo, aprenderemos a deixar nossa API mais segura com alguns recursos que são fundamentalmente importantes.

CAPÍTULO 10

Autenticação e segurança no Flask

Neste capítulo, falaremos sobre um tema que é de grande importância quando criamos uma API, a autenticação segura. Até o momento, estamos permitindo que acessos externos realizem o consumo de dados de nossa API sem que haja uma devida segurança no meio disso.

Daqui em diante, criaremos um modelo de segurança em nossa aplicação que permitirá que somente usuários autenticados no sistema acessem nossos dados pela API. Para isso, precisaremos criar um login que nos entregará um token que servirá para autorizar o usuário a acessar as áreas da API que hoje estão abertas a qualquer um.

Para isso usaremos o padrão `JWT`, que permite que façamos o envio de dados de forma segura através de objetos JSON.

10.1 JWT

Antes de seguirmos para a prática, precisamos entender o conceito do padrão `JWT`. Não será nada enorme, mas sim essencial.

Descrição

O JSON Web Token (`JWT`) é um padrão aberto (RFC 7519) que define uma maneira compacta e autocontida para transmitir com segurança informações entre as partes como um objeto JSON.

Na linguagem popular, o `JWT` é uma maneira que utilizaremos de transitar os dados JSON de forma segura. Então, quando um usuário tentar acessar a lista de produtos no sistema ou um perfil de

usuário, ele precisará estar autenticado seguindo os padrões do JWT . Veremos esses padrões mais a seguir.

Aplicando o JWT

Aplicar o padrão JWT no python é muito simples, pois ele possui uma lib chamada PyJWT que facilita bastante nosso trabalho. Para começar, vamos instalar a lib em nosso projeto.

```
(venv) tiago_luiz@ubuntu:~/livro_flask$ pip install pyjwt
```

Agora que tudo está instalado, precisamos abrir o arquivo `User.py` da pasta `model` e adicionar os códigos a seguir no método `get_user_by_email()` .

model/User.py

```
def get_user_by_email(self):
    try:
        res = db.session.query(User).filter(User.email==self.email).first()
    except Exception as e:
        res = None
        print(e)
    finally:
        db.session.close()
    return res
```

Esse método estava esperando para ser preenchido desde o capítulo sobre `controllers` . Não é necessário falarmos a fundo sobre ele, pois ele funciona igual ao método `get_user_by_id()` . A diferença é que seu filtro é por `email` e não por `id` .

Agora que a `model` está ok, vamos abrir o arquivo `User.py` da pasta `controller` e adicionar o código a seguir.

controller/User.py

```
from model.User import User
```

```
"""Adicione as linhas a seguir em seu arquivo"""
```

```

from datetime import datetime, timedelta
import hashlib, base64, json, jwt
from config import app_config, app_active
config = app_config[app_active]
"""Até aqui"""

class UserController():
    def __init__(self):
        self.user_model = User()

    """Adicione os métodos a seguir em seu arquivo,
    abaixo dos demais métodos"""
    def verify_auth_token(self, access_token):
        status = 401
        try:
            jwt.decode(access_token, config.SECRET, algorithm='HS256')
            message = 'Token válido'
            status = 200
        except jwt.ExpiredSignatureError:
            message = 'Token expirado, realize um novo login'
        except:
            message = 'Token inválido'

        return {
            'message': message,
            'status': status
        }

    def generate_auth_token(self, data, exp=30, time_exp=False):
        if time_exp == True:
            date_time = data['exp']
        else:
            date_time = datetime.utcnow() + timedelta(minutes=exp)

        dict_jwt = {
            'id': data['id'],
            'username': data['username'],
            "exp": date_time
        }
        access_token = jwt.encode(dict_jwt, config.SECRET,
algorithm='HS256')

```

```
return access_token
```

Agora que colocamos todo o código necessário para o arquivo `User.py` da `controller` funcionar, vamos entender melhor o que nossos novos métodos vão fazer.

- `import jwt` : essa é a lib do padrão JWT que o python possui. Ela nos fornece 2 métodos muito importantes: `jwt.decode` e `jwt.encode` , falaremos deles a seguir.
- `jwt.decode(payload, secret, algorithm)` : esse é o método que usaremos para criar o `token` . Para que ele funcione, precisaremos passar os seguintes parâmetros:
 - `payload` : aqui é passado um dicionário com os dados que você deseja usar para criar o token; esses dados podem ser aleatórios ou não. Recomenda-se que sejam dados como `email` , `username` etc. Dentro desse dicionário pode haver também uma chave chamada `exp` , onde passamos o horário em que o token expirará. Se a chave `exp` não for passada, o token não terá expiração;
 - `secret` : essa é uma `hash` secreta que precisa ser passada para que o token seja criado com segurança. É comum colocarmos a `secret_key` que criamos no arquivo `config.py` ;
 - `algorithm` : aqui é o tipo de algoritmo `hash` de criptografia que usaremos. Ele aceita os seguintes algoritmos:
HS256 , HS384 , HS512 , ES256 , ES384 , ES512 , RS256 , RS384 , RS512 , PS256 , PS384 , PS512 . Usaremos o HS256 , por ser considerado bem seguro.
- `jwt.encode(jwt_token, secret, algorithm)` : esse é o método que usamos para pegar o token criado com o método `jwt.decode` e devolvê-lo ao formato `dicionário` que ele possuía antes de virar token.
- `from datetime import datetime, timedelta` : essas são as classes que permitirão que trabalhem com datas e horários em nosso arquivo:
 - `datetime` : usaremos para capturar o horário atual;

- `timedelta` : usaremos para incrementar mais tempo ao horário atual, ou seja, pegaremos o horário atual e incrementaremos mais `x` minutos, que será o tempo de expiração do token;
- `jwt.ExpiredSignatureError` : essa é a exceção que o `PyJWT` nos retorna quando a chave `exp` está com o horário inferior ao atual. Se passamos o horário `x` na chave `exp` e nosso horário atual já é após o horário `x`, a exceção dirá ao `python` que o token expirou, dessa forma poderemos dizer à aplicação que é necessário refazer o login.

Toda vez que o usuário efetuar login ele receberá um novo token que será gerado pelo método `generate_auth_token()`.

E toda vez que o usuário precisar acessar um `endpoint` ele precisará passar esse `token`, que será verificado no método `verify_auth_token()`. Esse método pegará o `token` e passará pelo método `jwt.encode()`, que poderá retornar uma exceção quando o `token` estiver expirado ou quando o `token` for inválido. Se nenhuma das duas situações ocorrer ele retornará sucesso e permitirá ao usuário continuar consumindo os dados da API.

Agora que entendemos o que os métodos fazem, vamos para a parte final dessa etapa. Vamos abrir o arquivo `app.py` e adicionar as seguintes modificações.

app.py

```
# -*- coding: utf-8 -*-

"""Adicione o método abort no import do flask"""
from flask import Flask, request, redirect, render_template, Response,
json, abort
from flask_sqlalchemy import SQLAlchemy

"""Adicione o import a seguir no arquivo"""
from functools import wraps

"""Código minimizado. Ele não será modificado nesse momento"""
```

```

def create_app(config_name):
    """Código minimizado. Ele não será modificado nesse momento"""

    """Adicione o método a seguir após o after_request"""
    def auth_token_required(f):
        @wraps(f)
        def verify_token(*args, **kwargs):
            user = UserController()
            try:
                result = user.verify_auth_token(request.headers['access_token'])
                if result['status'] == 200:
                    return f(*args, **kwargs)
                else:
                    abort(result['status'], result['message'])
            except KeyError as e:
                abort(401, 'Você precisa enviar um token de acesso')
            return verify_token
        return auth_token_required

    """Até aqui"""

    """Código minimizado. Ele não será modificado nesse momento"""

    @app.route('/products/', methods=['GET'])
    @app.route('/products/<limit>', methods=['GET'])
    """Adicione o decorator @auth_token_required antes do método
get_products()"""
    @auth_token_required
    def get_products(limit=None):
        """Altere seu código para que a header fique como a do código a
seguir"""
        header = {
            'access_token': request.headers['access_token'],
            "token_type": "JWT"
        }

        product = ProductController()
        response = product.get_products(limit=limit)
        return Response(json.dumps(response, ensure_ascii=False),
mimetype='application/json'), response['status'], header

    @app.route('/product/<product_id>', methods=['GET'])

```

```

        """Adicione o decorator @auth_token_required antes do método
get_product()"""
        @auth_token_required
        def get_product(product_id):
            """Altere seu código para que a header fique como a do código a
seguir"""
            header = {
                'access_token': request.headers['access_token'],
                "token_type": "JWT"
            }

            product = ProductController()
            response = product.get_product_by_id(product_id = product_id)

            return Response(json.dumps(response, ensure_ascii=False),
mimetype='application/json'), response['status'], header

        @app.route('/user/<user_id>', methods=['GET'])
        """Adicione o decorator @auth_token_required antes do método
get_user_profile()"""
        @auth_token_required
        def get_user_profile(user_id):
            """Altere seu código para que a header fique como a do código a
seguir"""
            header = {
                'access_token': request.headers['access_token'],
                "token_type": "JWT"
            }

            user = UserController()
            response = user.get_user_by_id(user_id=user_id)

            return Response(json.dumps(response, ensure_ascii=False),
mimetype='application/json'), response['status'], header

        """Adicione o método a seguir em seu código"""
        @app.route('/login_api/', methods=['POST'])
        def login_api():
            header = {}
            user = UserController()

```

```

email = request.json['email']
password = request.json['password']

result = user.login(email, password)
code = 401
response = {"message": "Usuário não autorizado", "result": []}

if result:
    if result.active:
        result = {
            'id': result.id,
            'username': result.username,
            'email': result.email,
            'date_created': result.date_created,
            'active': result.active
        }

        header = {
            "access_token": user.generate_auth_token(result),
            "token_type": "JWT"
        }
        code = 200
        response["message"] = "Login realizado com sucesso"
        response["result"] = result

    return Response(json.dumps(response, ensure_ascii=False),
mimetype='application/json'), code, header
    """"Até aqui""""

return app

```

Como podemos ver, o método `generate_auth_token()` recebe o objeto de resultado do usuário como havíamos dito e entregará um token como resultado. Esse token será passado para que o usuário possa sempre enviá-lo no header da requisição.

Antes dos métodos de cada `endpoint`, adicionamos um `decorator` chamado `@auth_token_required`. Um `decorator` basicamente é um método que precede o método principal. Vamos usar como exemplo

`get_products()` . Toda vez que o `endpoint` for acessar o método principal `get_products()` , antes, ele acessará o método `auth_token_required()` . Esse método é responsável por pegar o `request.headers['access_token']` enviado pelo usuário e verificar se esse `token` é válido. Isso é feito através do método `verify_auth_token()` , passando o `token` como parâmetro. Se o `token` estiver certo, ele retornará status 200 e o usuário poderá continuar a acessar a API; se o `token` estiver expirado ou inválido, o usuário será redirecionado para uma página de código 401 com uma mensagem informando `token inválido` OU `token expirado` e o usuário deverá realizar novo login para receber um novo `token` .

Agora vamos realizar um exemplo de acesso com token de acesso.

1. Acesse a URL para efetuar login e coloque os dados de acesso, aperte `send` e o login será realizado:

http://localhost:8000/login_api/

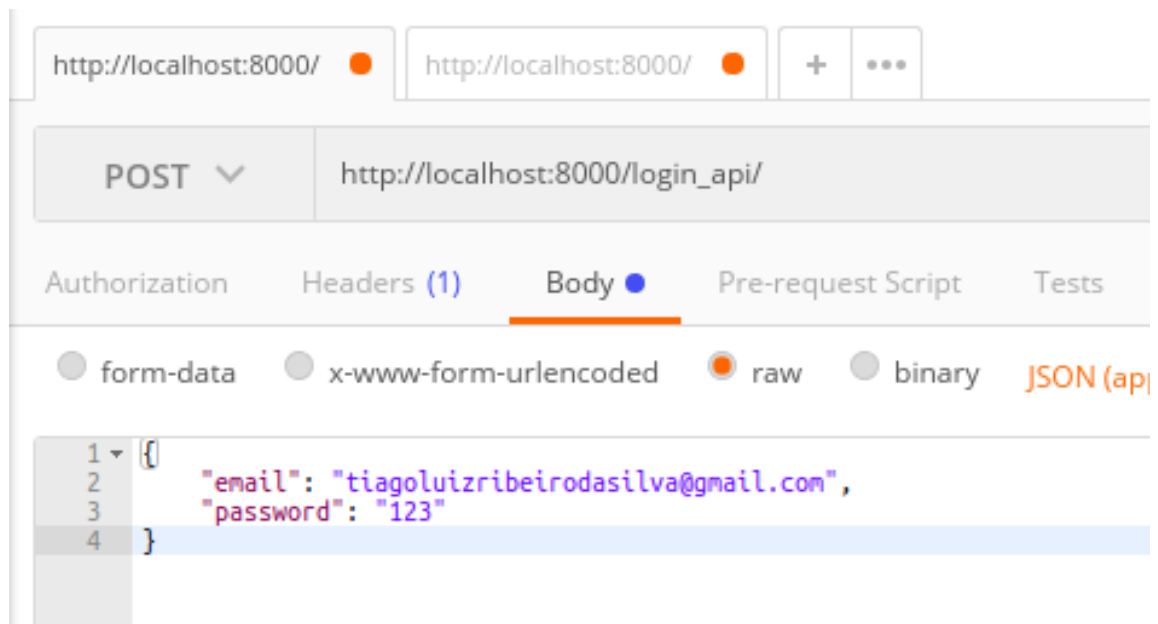


Figura 10.1: Passo 1.1 envio dos dados via post

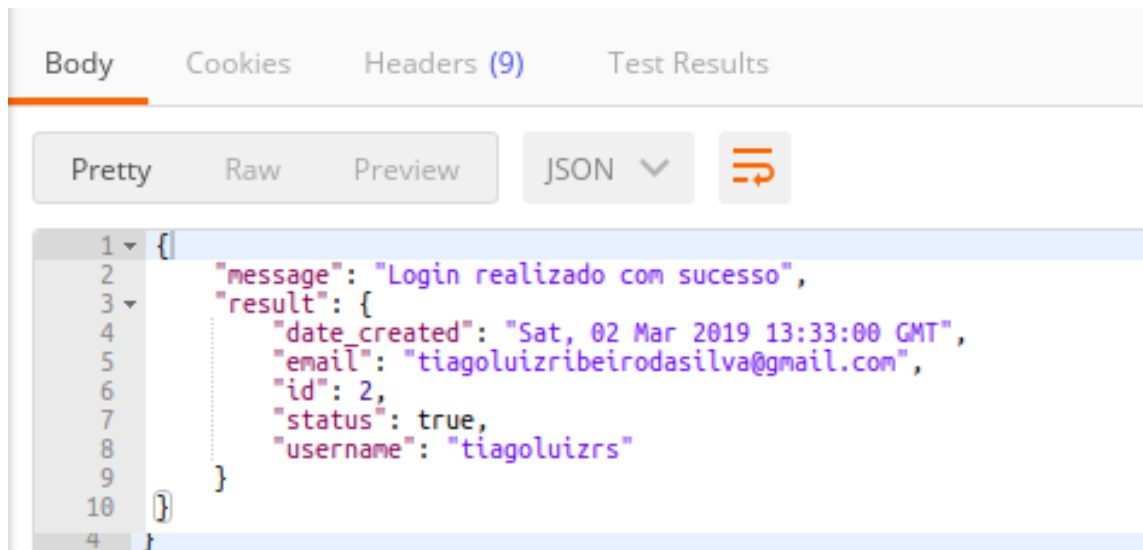


Figura 10.2: Passo 1.2 resposta da API

2. Vá até a aba `headers` e procure a chave `access_token`, copie o valor dela:

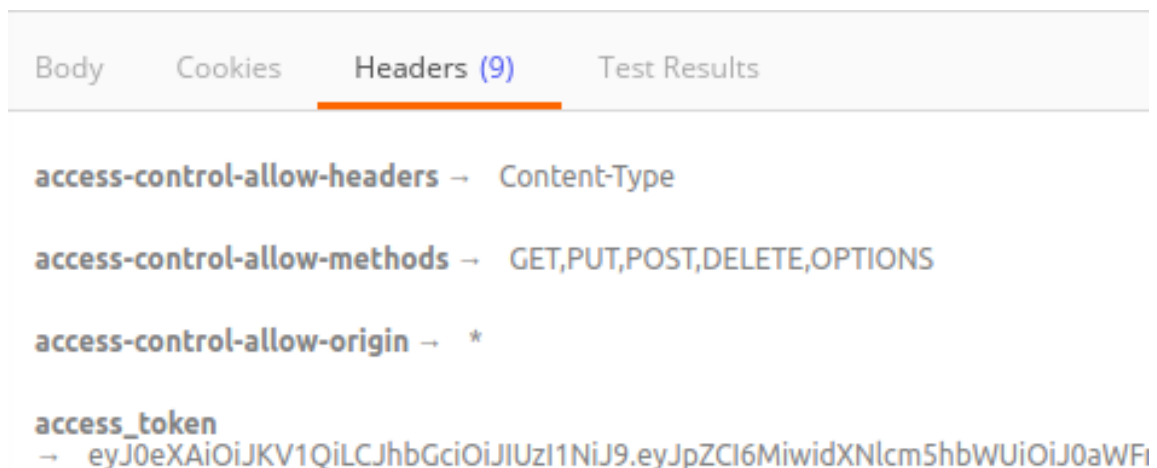


Figura 10.3: Passo 2 para testar login com token

3. Agora acesse o `endpoint` de produtos como a imagem mostra, adicionando uma chave na área `headers` com o nome `access_token`. O valor deverá ser a chave que você copiou da `header` de login. Clique em `send` e pronto. Você conseguirá acessar normalmente a API.

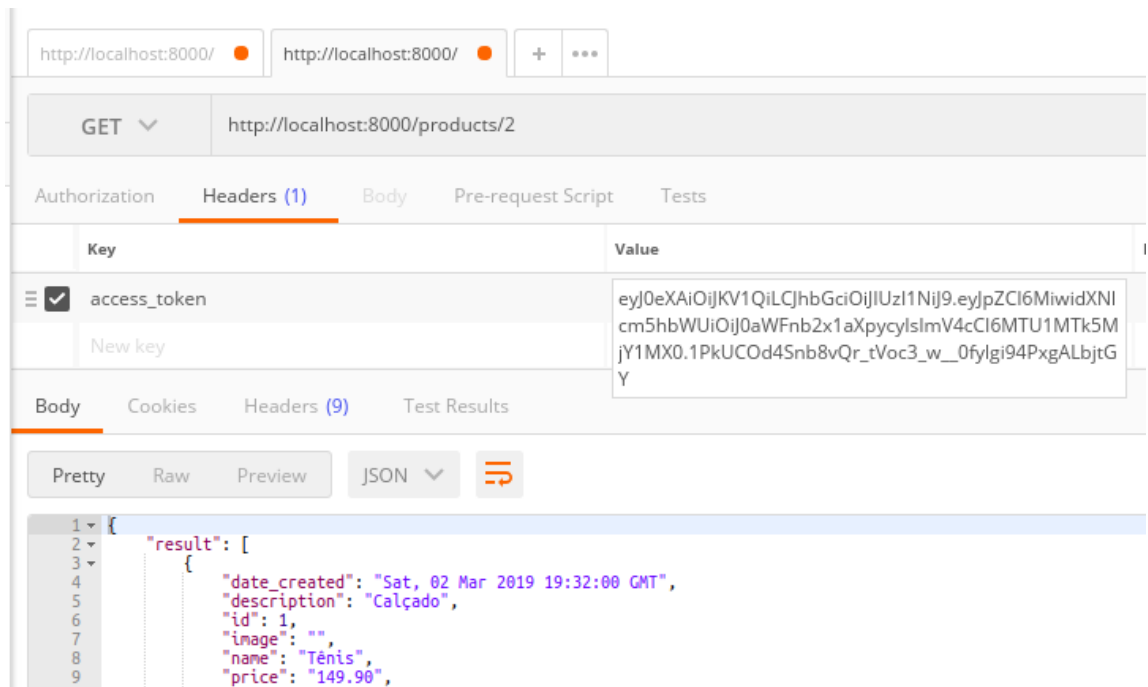


Figura 10.4: Passo 3 para testar login com token

- Se você não enviar um token , receberá uma mensagem como a da imagem a seguir.

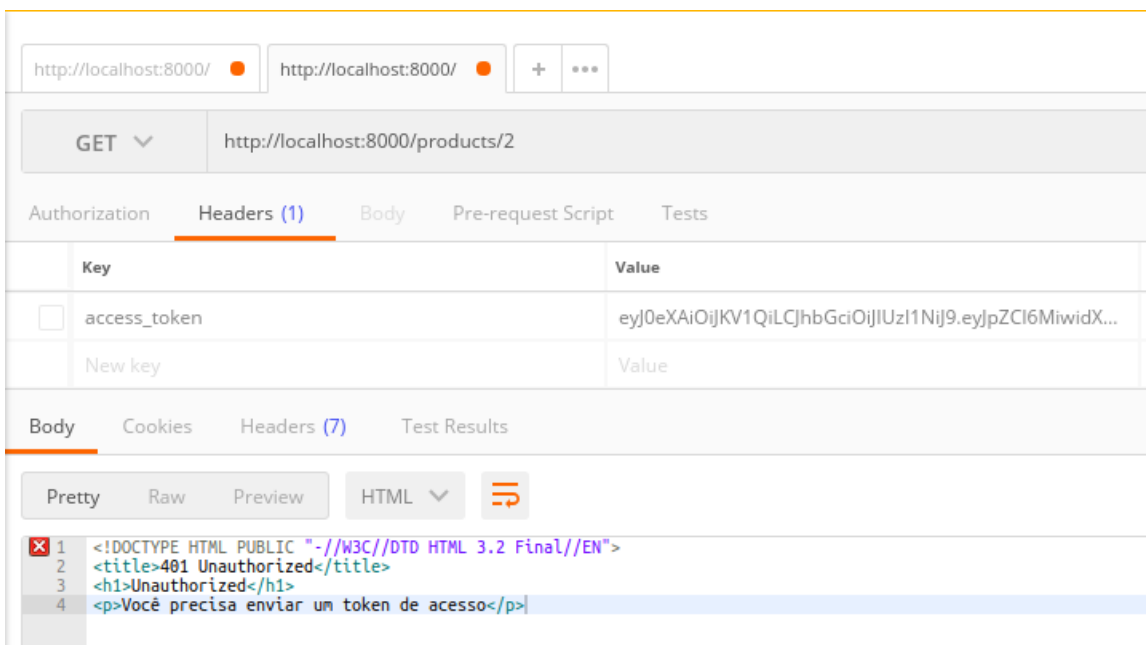


Figura 10.5: Teste de requisição à API sem token

5. Se você enviar um token inválido, receberá uma mensagem como a da imagem a seguir.

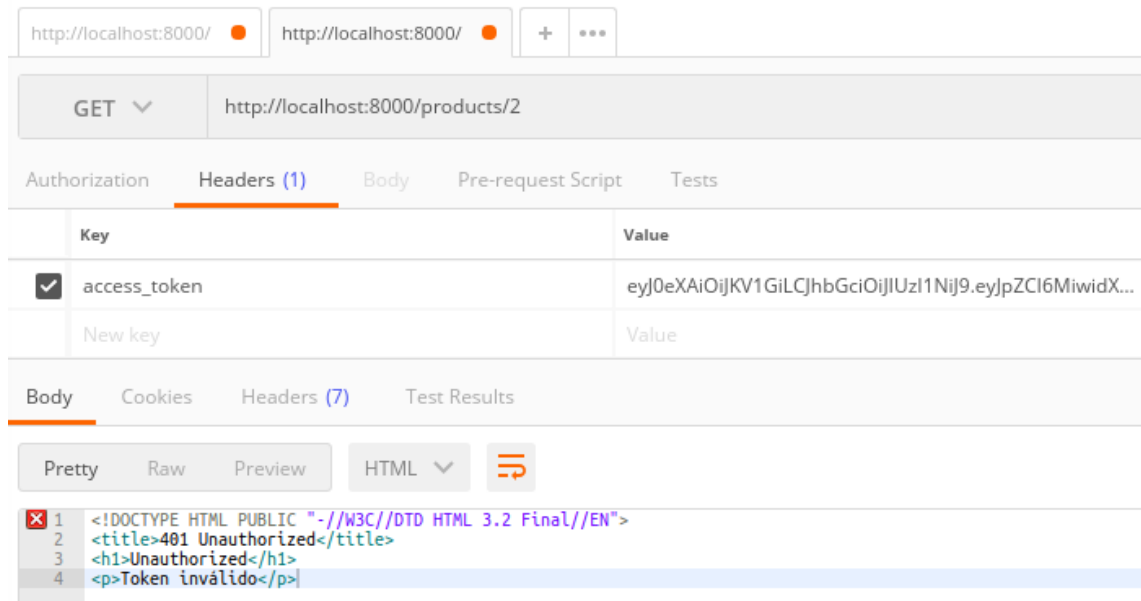


Figura 10.6: Teste de requisição à API com token inválido

10.2 Restringindo o painel Administrativo

Nessa etapa, pegaremos a tela de login que fizemos para nosso admin e colocaremos para funcionar.

Faremos todas as restrições que são necessárias em nosso painel, para que cada tipo de usuário possa fazer apenas o que é permissível a ele.

Antes de continuar, crie um usuário de cada tipo no painel admin, para que possamos utilizá-los futuramente neste capítulo.

Meu Estoque

Home

Usuários

Produtos

List (4)

Create

Export

Add Filter

20 items

WITH SELECTED

Search: Nome de usuário, E-m













<input type="checkbox"/>		Função	Nome de usuário	E-mail	Data de Criação
<input type="checkbox"/>	  	Cliente	usuario	usuario@gmail.com	2019-03-07 17:53
<input type="checkbox"/>	  	Admin	tiagoluizrs	tiagoluizribeirodasilva@gmail.com	2019-03-02 13:33
<input type="checkbox"/>	  	Logista	loginsta	logista@gmail.com	2019-03-07 17:53
<input type="checkbox"/>	  	Gerente	gerente	gerente@gmail.com	2019-03-07 17:53

Figura 10.7: Tela de usuários do admin

Pronto, agora podemos seguir.

Para criar o login no Flask, usaremos a `lib flask_login`. Execute o comando a seguir para instalar a `lib`.

```
(venv) tiago_luiz@ubuntu:~/livro_flask$ pip install flask_login
```

Agora que a `lib` está instalada, vamos começar a editar nossos arquivos para adicionar nosso código.

Abra o arquivo `User.py` da pasta `model` e altere as linhas a seguir.

model/User.py

```
# -*- coding: utf-8 -*-
"""Importe o UserMixin do flask_login no topo do arquivo"""
from flask_login import UserMixin

"""Adicione o UserMixin na linha a seguir de sua class User"""
class User(db.Model, UserMixin):
```

A classe `UserMixin` implementa todos os métodos necessários do `flask_login` em nossa `model`. Com ele, poderemos salvar o usuário

na sessão, usando a `model` selecionada e verificar se ele está logado ou não.

Agora abra o arquivo `app.py` para adicionarmos as modificações a seguir.

```
"""Import dos elementos do flask_login no
topo do arquivo"""
from flask_login import LoginManager, login_user, logout_user

"""Código minimizado. Ele não será modificado nesse momento"""

def create_app(config_name):
    app = Flask(__name__, template_folder='templates')

    """Adicione as duas linhas a seguir no arquivo"""
    login_manager = LoginManager()
    login_manager.init_app(app)
    """Até aqui"""

    app.secret_key = config.SECRET
    app.config.from_object(app_config[config_name])
```

Vamos entender melhor os itens novos do código:

- `from flask_login import LoginManager, login_user, logout_user` :
esses são os elementos importados da `lib flask_login` :
 - `LoginManager` : objeto que instanciamos para habilitar o `flask_login` em nossa aplicação;
 - `login_user` : método que usaremos para salvar os dados do usuário na sessão quando efetuarmos o login;
 - `logout_user` : método que usaremos para efetuar o logout do usuário e apagar seus dados da sessão da aplicação.
- `login_manager.init_app(app)` : esse é o momento em que a instância do objeto `LoginManager()` é ativada em nossa aplicação.

Agora que configuramos o `flask_login` em nosso `app.py`, vamos alterar o método de `login` que usaremos para entrar no `admin`,

aquele que criamos lá no início do livro.

app.py

```
"""Código minimizado. Ele não será modificado nesse momento"""

@app.route('/login/')
def login():
    """Altere seu método padrão da tela de login para renderizar o html
    com o parâmetro `data` contendo o seguinte dicionário"""
    return render_template('login.html', data={'status': 200, 'msg': None,
'type': None})

@app.route('/login/', methods=['POST'])
def login_post():
    user = UserController()

    email = request.form['email']
    password = request.form['password']

    result = user.login(email, password)

    """Modifique o método POST de login para ficar como o da linha a
    seguir"""
    if result:
        if result.role == 4:
            return render_template('login.html', data={'status': 401, 'msg':
'Seu usuário não tem permissão para acessar o admin', 'type': 2})
        else:
            login_user(result)
            return redirect('/admin')
    else:
        return render_template('login.html', data={'status': 401, 'msg':
'Dados de usuário incorretos', 'type': 1})

"""Código minimizado. Ele não será modificado nesse momento"""
```

As pequenas alterações que fizemos foram para melhorar e deixar nosso login mais fácil de entender. Primeiro, para o atributo `data`, criamos um dicionário que terá um `status 200` quando for sucesso e

401 quando o usuário errar a senha ou não existir ou quando ele não estiver autorizado a entrar.

Para cada caso temos uma mensagem diferente e um `type` diferente. O `type 1` é para o tipo de alerta de erro, o `type 2` é para uma chamada de atenção e o `3` para uma informação. Esses tipos serão usados para escolhermos as cores do balão `html` que usaremos para informar ao usuário a mensagem. Veremos isso quando editarmos o arquivo `login.html`.

O status, como dito, ficou entre `200` e `401`. No arquivo `html` verificaremos isso antes de emitir qualquer mensagem.

Podemos verificar que, se o usuário tiver a `role` de tipo `4`, ou seja, cliente, ele não poderá entrar no painel admin, como descrevemos em nossa regra de negócios no início do livro.

Vamos alterar o trecho de código necessário em nosso arquivo `html` para entendermos melhor o que desejamos fazer.

templates/login.html

```
<button class="btn btn-lg btn-primary btn-block"
type="submit">Entrar</button>
<!-- Adicione as linhas a seguir após o botão entrar do html -->
{% if data.status == 401 %}
    {% if data.type == 1 %}
        <div class="alert alert-danger" role="alert">
            {{data.msg}}
        </div>
    {% elif data.type == 2 %}
        <div class="alert alert-warning" role="alert">
            {{data.msg}}
        </div>
    {% endif %}
{% elif data.status == 200 %}
    {% if data.type == 3 %}
        <div class="alert alert-success" role="alert">
            {{data.msg}}
        </div>
```



```

        {% endif %}
    {% endif %}
    <!-- Até aqui. Onde fica a tag de fechamento do form -->
</form>

```

Como podemos ver, antes de emitir qualquer mensagem, verificamos o status e o tipo de mensagem. Assim conseguimos alterar a cor do balão de informações e deixar nossa tela de login mais informativa.

Agora precisamos criar dois métodos que serão muito importantes. O método que realizará o `logout` do usuário no painel `admin` e o método padrão que o `flask_login` solicita que seja criado, para o `flask_login` poder retornar o objeto do usuário quando necessário.

```

"""Crie a rota a seguir em seu arquivo após a última rota criada para
que tenhamos um logout"""
@app.route('/logout')
def logout_send():
    logout_user()
    return render_template('login.html', data={'status': 200, 'msg':
'Usuário deslogado com sucesso!', 'type':3})

"""Crie a rota a seguir em seu arquivo para que tenhamos um logout"""
@login_manager.user_loader
def load_user(user_id):
    user = UserController()
    return user.get_admin_login(user_id)
"""Até aqui"""

return app

```

Basicamente, a primeira rota é para realizar o `logout`. Ela é utilizada para chamar o método `logout_user()` e realizar todo o processo de `logout` que o próprio `flask_login` realiza. Após isso, ele renderiza a tela de login com uma mensagem de tipo 3 (info) dizendo que o login foi realizado com sucesso.

O método `load_user()` é chamado automaticamente no momento em que o método `login_user()` é chamado pela nossa aplicação no

momento do login.

Observe na rota `login` que `login_user(result)` passa `result` como resultado da tentativa de login que o usuário faz no `login.html`. O método `load_user()` possui o decorator `@login_manager.user_loader` que é criado pelo `flask_login`. Ele basicamente chama o método `get_admin_login()` que criaremos na `controller User.py`. Esse método verificará se o usuário existe, baseado no id enviado como parâmetro; caso exista, ele retorna o objeto do usuário para o método que o chamou, no caso, `login_user()`.

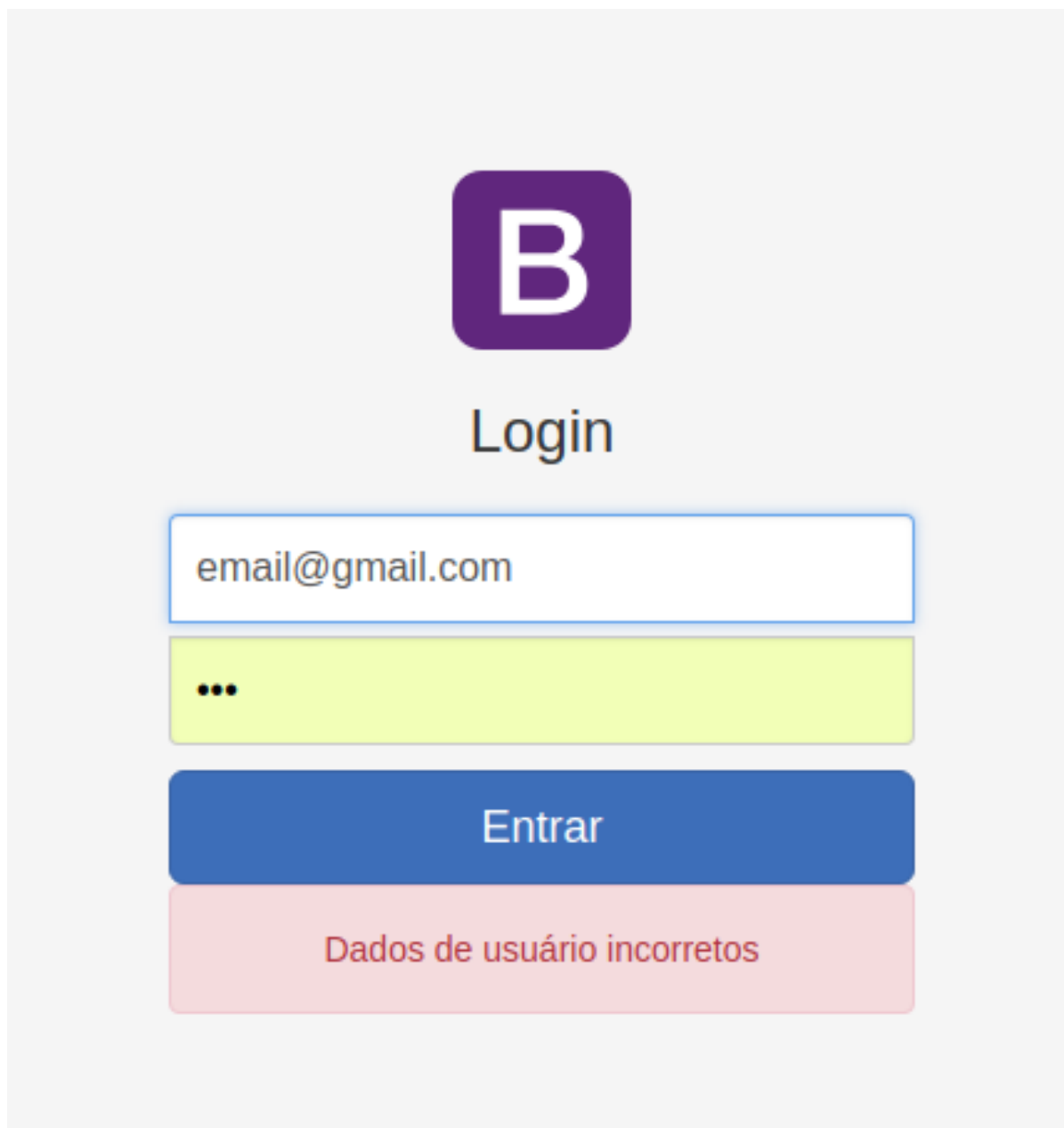
Vamos criar o método `get_admin_login()` no arquivo `User.py` da `controller`.

controller/User.py

```
"""Adicione o método a seguir em seu arquivo"""
def get_admin_login(self, user_id):
    self.user_model.id = user_id

    response = self.user_model.get_user_by_id()
    return response
```

Se tentarmos entrar com um usuário não existente, ele não funcionará e informará que o usuário não existe.



B

Login

email@gmail.com

...

Entrar

Dados de usuário incorretos

Figura 10.8: Teste de login incorreto.

Se tentarmos entrar com um usuário de tipo 4, ou seja, role cliente, ele não terá permissão para acessar o admin.

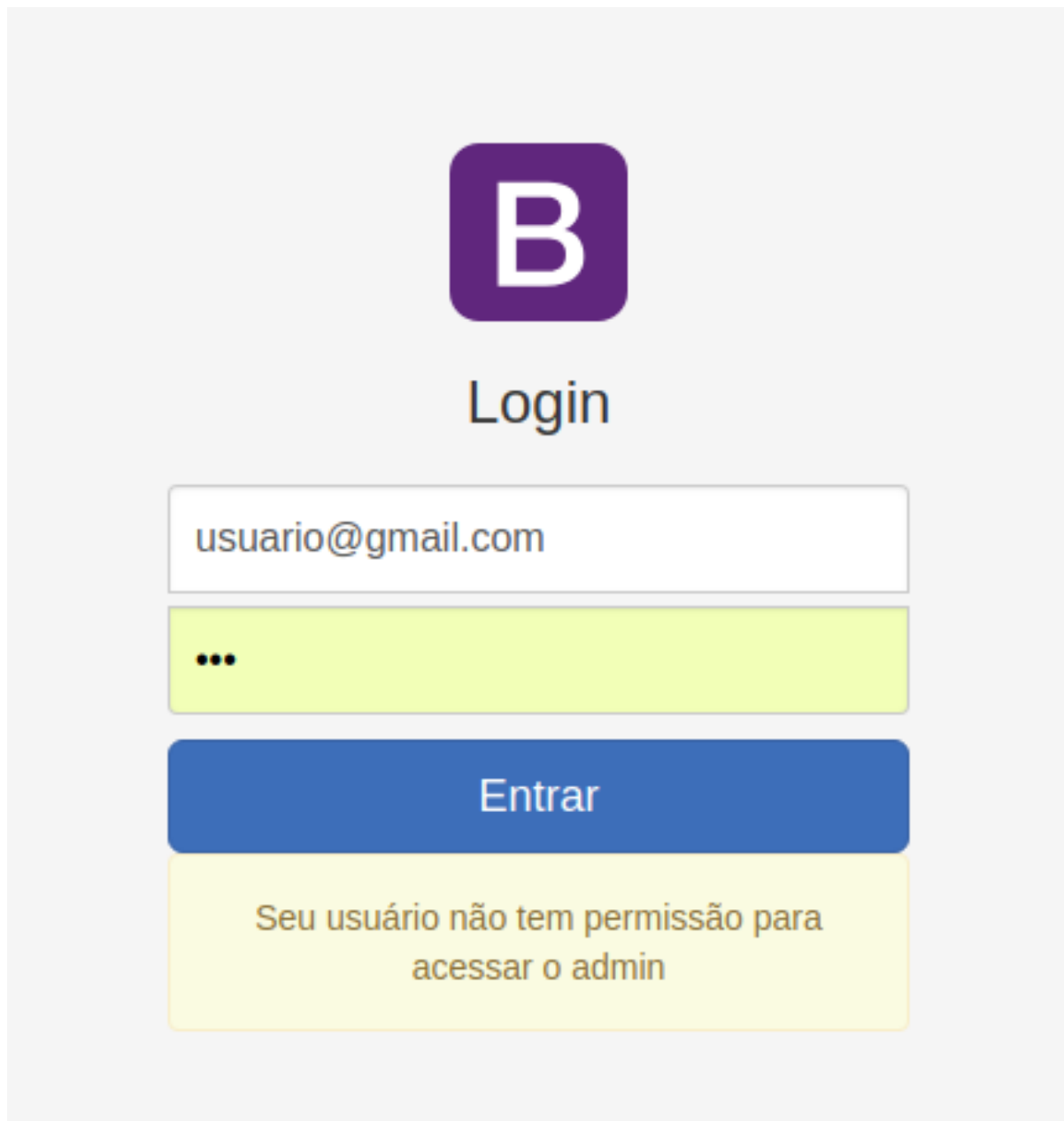


Figura 10.9: Teste de login não permitido.

Claro que se entrarmos pela URL do admin ainda conseguiremos entrar nela. Isso é um problema. Para resolver isso, vamos modificar algumas coisas nos arquivos `Admin.py` e `Views.py` da pasta `admin`. Vamos lá.

Abra o arquivo `Views.py`. Agora que aplicaremos níveis de acesso ao nosso admin, cada `view` precisa ter suas próprias regras e para

isso criaremos uma `ModelView` customizada para cada `view`.

Views.py

```
# -*- coding: utf-8 -*-
from flask_admin import AdminIndexView, expose
from flask_admin.contrib.sqla import ModelView
"""Adicione os imports a seguir em seu arquivo"""
from flask_login import current_user
from flask import redirect

"""Código minimizado. Ele não será modificado nesse momento"""

class HomeView(AdminIndexView):
    extra_css = [config.URL_MAIN +
'static/css/home.css', 'https://maxcdn.bootstrapcdn.com/font-
awesome/4.7.0/css/font-awesome.min.css']

    """Código minimizado. Ele não será modificado nesse momento"""

    """Adicione os métodos a seguir em sua ModelView"""
    def is_accessible(self):
        return current_user.is_authenticated

    def inaccessible_callback(self, name, **kwargs):
        if current_user.is_authenticated:
            return redirect('/admin')
        else:
            return redirect('/login')

class UserView(ModelView):
    """Código minimizado. Ele não será modificado nesse momento"""

    """Adicione os métodos a seguir em sua ModelView"""
    def is_accessible(self):
        return current_user.is_authenticated

    def inaccessible_callback(self, name, **kwargs):
        if current_user.is_authenticated:
            return redirect('/admin')
        else:
```

```

        return redirect('/login')

class RoleView(ModelView):
    """Adicione os métodos a seguir em sua ModelView"""
    def is_accessible(self):
        return current_user.is_authenticated

    def inaccessible_callback(self, name, **kwargs):
        if current_user.is_authenticated:
            return redirect('/admin')
        else:
            return redirect('/login')

class CategoryView(ModelView):
    """Adicione os métodos a seguir em sua ModelView"""
    def is_accessible(self):
        return current_user.is_authenticated

    def inaccessible_callback(self, name, **kwargs):
        if current_user.is_authenticated:
            return redirect('/admin')
        else:
            return redirect('/login')

class ProductView(ModelView):
    """Adicione os métodos a seguir em sua ModelView"""
    def is_accessible(self):
        return current_user.is_authenticated

    def inaccessible_callback(self, name, **kwargs):
        if current_user.is_authenticated:
            return redirect('/admin')
        else:
            return redirect('/login')

```

Vamos entender a mágica desses códigos:

- `is_accessible(self)` : esse método é usado para verificar se o usuário tem permissão para acessar a `view` ; se não tiver

permissão, ele não poderá acessar nem verá o menu que envia para essa página.

- `current_user` : esse é o objeto que possui os dados do usuário logado. Dados que estão salvos em sessão, além de outros atributos como o `is_authenticated` de que falaremos a seguir;
- `is_authenticated` : esse é o atributo que usamos para retornar se o usuário está autenticado ou não para acessar a página em questão. Se eu quiser que um usuário de role 1 acesse a `view` mas um usuário de role 2 não acesse, só retornaremos o `is_authenticated` se a role for igual a 1;
- `inaccessible_callback` : esse é o método que usamos quando um usuário não tiver permissão para acessar a página. O método `is_accessible` chamará esse método e verificará, se o usuário estiver autenticado corretamente, ele redirecionará o usuário para o painel do admin, caso ele não esteja logado corretamente ele será direcionado para a tela de login.

Agora precisamos modificar nosso arquivo `Admin.py` , para usarmos as novas `ModelViews` . Abra o arquivo e adicione as modificações a seguir.

admin/Admin.py

```
# -*- coding: utf-8 -*-
from flask_admin import Admin
# Capítulo 10 - Remover
from flask_admin.contrib.sqla import ModelView
from flask_admin.menu import MenuLink

from model.Role import Role
from model.User import User
from model.Category import Category
from model.Product import Product

"""Adicione as novas ModelViews no import a seguir"""
from admin.Views import UserView, HomeView, RoleView, CategoryView,
ProductView

def start_views(app, db):
```

```
admin = Admin(app, name='Meu Estoque', base_template='admin/base.html',
template_mode='bootstrap3', index_view=HomeView())
```

```
"""Altere as linhas do seu código que usavam ModelView para RoleView,
CategoryView e ProductView, como no código a seguir"""
```

```
admin.add_view(RoleView(Role, db.session, "Funções",
category="Usuários"))
admin.add_view(UserView(User, db.session, "Usuários",
category="Usuários"))
admin.add_view(CategoryView(Category, db.session, 'Categorias',
category="Produtos"))
admin.add_view(ProductView(Product, db.session, "Produtos",
category="Produtos"))
```

```
admin.add_link(MenuLink(name='Logout', url='/logout'))
```

Agora nosso admin já está utilizando as `ModelViews` customizadas.

No momento, ainda não usamos esses dois métodos para restringir o acesso pelo tipo de usuário, mas já não será possível acessar o admin sem efetuar o login. No próximo tópico criaremos os níveis de acesso, usando os métodos que acabamos de ver.

10.3 Níveis de acesso no admin

Nessa etapa, vamos adicionar nossos níveis de acesso, utilizando os métodos do `flask_login` que acabamos de ver no tópico anterior.

Basicamente vamos alterar o método `is_accessible()` de algumas `ModelViews`. Veja o código a seguir e faça as modificações necessárias em seu arquivo `Views.py`.

Views.py

```
class UserView(ModelView):
    """Código minimizado. Ele não será modificado nesse momento"""
```



```
"""Altere o método is_accessible para ficar como o código a seguir"""
```

```
def is_accessible(self):  
    if current_user.is_authenticated:  
        role = current_user.role  
        if role == 1:  
            self.can_create = True  
            self.can_edit = True  
            self.can_delete = True  
        return current_user.is_authenticated
```

```
"""Código minimizado. Ele não será modificado nesse momento"""
```

```
class RoleView(ModelView):
```

```
    """Altere o método is_accessible para ficar como o código a seguir"""
```

```
    def is_accessible(self):  
        if current_user.is_authenticated:  
            role = current_user.role  
            if role == 1:  
                self.can_create = True  
                self.can_edit = True  
                self.can_delete = True  
            return current_user.is_authenticated
```

```
    """Código minimizado. Ele não será modificado nesse momento"""
```

```
class CategoryView(ModelView):
```

```
    """Altere o método is_accessible para ficar como o código a seguir"""
```

```
    can_view_details = True
```

```
    def is_accessible(self):  
        if current_user.is_authenticated:  
            role = current_user.role  
            if role == 1:  
                self.can_create = True  
                self.can_edit = True  
                self.can_delete = True  
                return current_user.is_authenticated  
            elif role == 2:  
                self.can_create = True  
                self.can_edit = True  
                self.can_delete = True
```

```

        return current_user.is_authenticated

    """Código minimizado. Ele não será modificado nesse momento"""

class ProductView(ModelView):
    """Altere o método is_accessible para ficar como o código a seguir"""
    can_view_details = True

    def is_accessible(self):
        if current_user.is_authenticated:
            role = current_user.role
            if role == 1:
                self.can_create = True
                self.can_edit = True
                self.can_delete = True
            elif role == 2:
                self.can_create = True
                self.can_edit = True
                self.can_delete = True
            elif role == 3:
                self.can_create = True
                self.can_edit = True
                self.can_delete = False

        return current_user.is_authenticated

    """Código minimizado. Ele não será modificado nesse momento"""

```

Com tudo isso feito. Nosso admin agora tem níveis de acesso. Vendo as imagens a seguir conseguimos perceber que cada tipo de usuário consegue ver apenas as telas que são permitidas para ele e se tentar acessar pela URL também será redirecionado para a home do admin ou para a tela de login, se não tiver realizado login.

Se percebermos, em algumas `ModelViews` só retornamos `current_user.is_authenticated` para alguns tipos de `roles`, ou seja, cada `view` permite ou não que um tipo de usuário a acesse.

Vamos ver um exemplo de cada um dos tipos de `roles`, exceto o de tipo `cliente`, que não tem acesso ao `admin`.

1. Login com role de tipo Admin:

Meu Estoque Home Usuários ▾ Produtos ▾ Logout

Funções
Usuários

Categorias
Produtos

USUÁRIOS
4

CATEGORIAS
1

PRODUTOS
5

Últimos produtos

ID	Nome	Preço	Quantidade	Data de Criação
1	Tênis	149.90	20	2019-03-02 19:32:00
2	Sapato Social	249.90	40	2019-03-02 21:17:00
3	Sapatênis	350.00	200	2019-03-04 12:09:42
4	Sandália	300.00	30	2019-03-04 22:53:00
5	Chinelo	1900.00	40	2019-03-04 22:54:00

Figura 10.10: Login com role de tipo Admin.

2. Login com role de tipo Gerente:

Meu Estoque Home Produtos ▾ Logout

Categorias
Produtos

USUÁRIOS
4

CATEGORIAS
1

PRODUTOS
5

Últimos produtos

ID	Nome	Preço	Quantidade	Data de Criação
1	Tênis	149.90	20	2019-03-02 19:32:00
2	Sapato Social	249.90	40	2019-03-02 21:17:00
3	Sapatênis	350.00	200	2019-03-04 12:09:42
4	Sandália	300.00	30	2019-03-04 22:53:00
5	Chinelo	1900.00	40	2019-03-04 22:54:00

Figura 10.11: Login com role de tipo Gerente.

3. Login com role de tipo Logista:

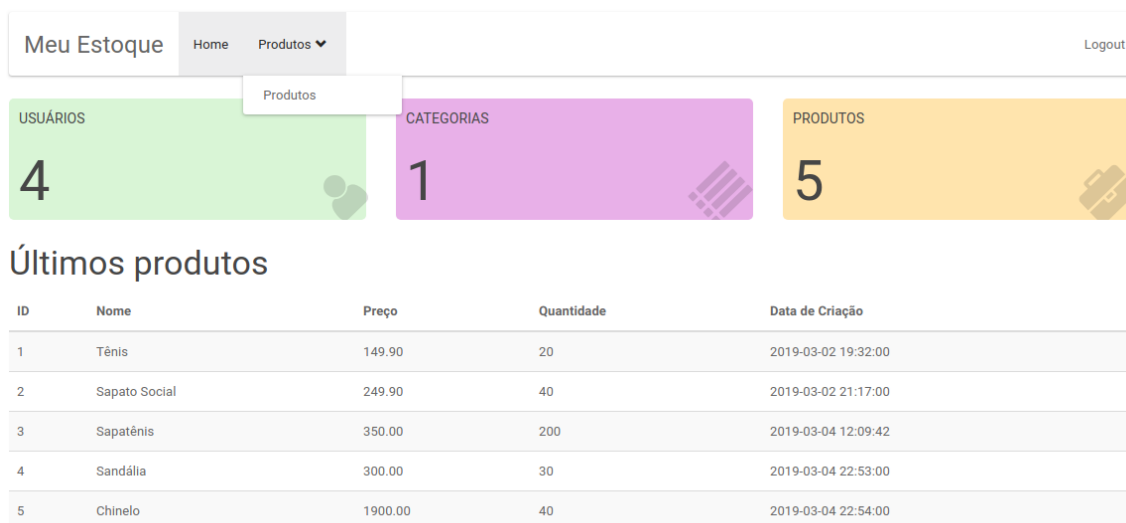


Figura 10.12: Login com role de tipo Logista.

Agora nosso admin está protegido e com seus níveis de acesso criados.

Conclusão

Neste capítulo aprendemos como deixar nossa API e nosso admin mais seguros, através de ferramentas que o Flask possui, que são excelentes e muito fáceis de se utilizar. Com tudo que temos até agora, já é possível criar um sistema completamente robusto e consistente.

Daqui em diante, veremos um conteúdo adicional, para que nosso sistema possa realizar a recuperação de senhas de usuário através do recurso de envio de e-mail.

CAPÍTULO 11

Trabalhando com serviços de e-mail

11.1 Introdução

Neste capítulo, aprenderemos sobre serviço de envio por e-mail. Esse tipo de serviço é algo muito importante e muito utilizado em sistemas que precisam de recuperação de senha, confirmação de e-mail, notificação, entre outros, que geralmente ocorrem via envio por e-mail.

Para nosso sistema, utilizaremos o serviço de e-mail `SendGrid`, que é gratuito e poderemos utilizar tranquilamente. Ele possui planos pagos, mas não é necessário para usar o serviço de que precisaremos agora.

Antes de continuar, você precisará se cadastrar no site <https://signup.sendgrid.com/>. Se você desejar ver os planos do serviço, acesse <https://sendgrid.com/pricing/>.

Com o cadastro realizado, podemos seguir em frente.

11.2 Primeiros passos

Para que nosso serviço do `SendGrid` funcione, precisamos fazer a instalação da `lib dele` no `python`. Para isso, rode o comando a seguir.

```
(venv) tiago_luiz@ubuntu:~/livro_flask$ pip install sendgrid==6.0.5
```

Agora que temos nosso cadastro no `SendGrid` e a `lib dele` instalada, podemos configurá-lo em nosso projeto. A primeira coisa

de que precisamos é `API_KEY` do `SendGrid` , para isso precisamos seguir os próximos passos:

1. Efetue login no `SendGrid` e acesse o link https://app.sendgrid.com/settings/api_keys/ para criar a chave de acesso do seu projeto:
2. Clique no botão `Create API Key` para criar a chave de API para seu projeto.

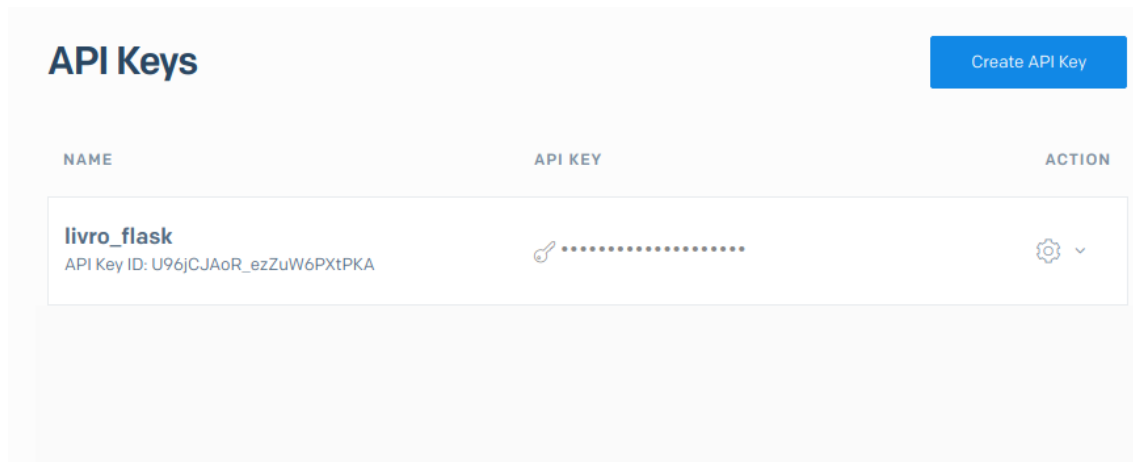




Figura 11.1: Criando uma api key SendGrid.

3. Adicione o nome da API , selecione `Full Access` e clique no botão `Create & View` .

Create API Key

API Key Name • flask_admin ⓘ

API Key Permissions • ⓘ

Full Access
Allows the API key to access GET, PATCH, PUT, DELETE, and POST endpoints for all parts of your account, excluding billing.

Figura 11.2: Nome da api key.

4. Após criar você receberá a `API_KEY`. Copie esse código, pois o usaremos nos próximos passos.

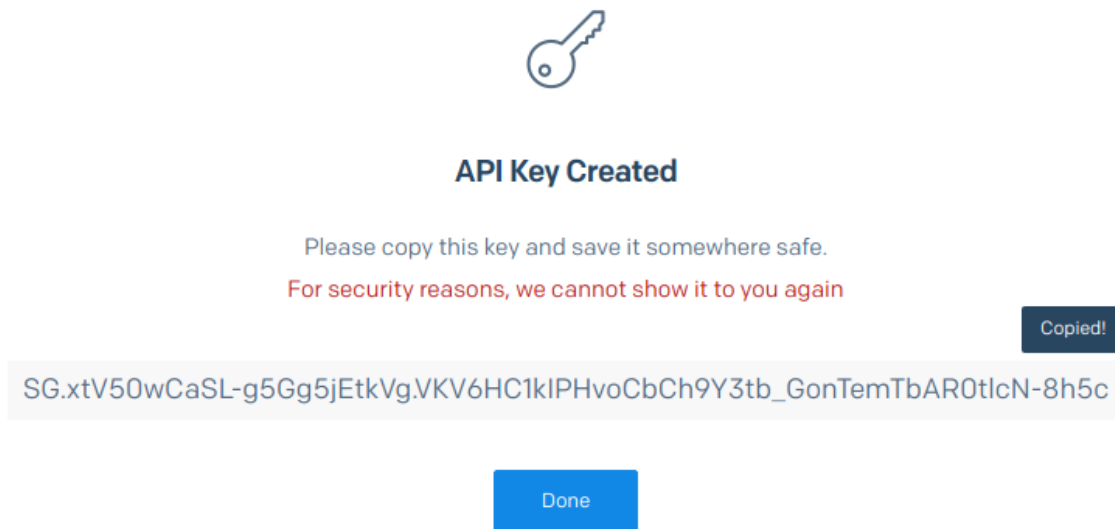


Figura 11.3: Código da api key.

Config.py

```
class Config(object):
    CSRF_ENABLED = True
    SECRET = 'ysb_92=qe#dgjf8%0ng+a*#4rt#5%3*4kw5%i2bck*gn@w3@f&-&'
    TEMPLATE_FOLDER =
os.path.join(os.path.dirname(os.path.abspath(__file__)), 'templates')
    ROOT_DIR = os.path.dirname(os.path.abspath(__file__))
    APP = None
    SQLALCHEMY_DATABASE_URI =
'mysql+mysqldb://root:123@localhost:3306/livro_flask'
    """
    Adicione a seguir o código da sua API.
    """
    SENDGRID_API_KEY = 'API_KEY'
```

Com nosso projeto contendo a `API_KEY` podemos seguir. Agora está tudo pronto para começarmos a criar nosso serviço de envio de e-mail pelo sistema.

11.3 Criando o serviço de envio

Vamos começar criando um arquivo que nos permitirá enviar por e-mail qualquer mensagem que desejarmos.

Abra o arquivo `Email.py` que fica dentro da pasta `controller` e adicione o código a seguir:

controller/Email.py

```
import os
from sendgrid import SendGridAPIClient
from sendgrid.helpers.mail import Mail

from config import app_active, app_config
config = app_config[app_active]

class EmailController():
    def send_email(self, t_email, subject, content_text,
f_email="contato@site.com.br"):
        message = Mail(
            from_email=f_email,
            to_emails=t_email,
            subject=subject,
            html_content=content_text)
        try:
            sg = SendGridAPIClient(config.SENDGRID_API_KEY)
            response = sg.send(message)

            return {
                'status_code' : response.status_code,
                'body' : response.body,
                'headers' : response.headers
            }
        except Exception as e:
            print(e.message)
            raise e
```

Vamos entender melhor cada linha:

- `from sendgrid import SendGridAPIClient` : método importado da lib `sendgrid` que será usado para comunicar ao serviço web qual a API que está sendo usada nesse projeto, dentro do método passamos a `API_KEY` através do parâmetro `apikey` , no caso do código que escrevemos, estamos passando como valor a variável que está no arquivo de configuração `config.SENDGRID_API_KEY` , pois ela contém a chave da nossa `API_KEY` ;
- `from sendgrid.helpers.mail import Mail` : lib do `SendGrid` que será usada para criar a estrutura de e-mail que será enviada, contendo os seguintes atributos:
 - `from_email` : e-mail que aparecerá como responsável pelo envio. Você pode escolher qualquer e-mail, pois isso só aparecerá como um rótulo;
 - `to_email` : e-mail que deverá receber a mensagem;
 - `subject` : assunto deste e-mail;
 - `html_content` : conteúdo deste e-mail, podendo ser um texto normal ou um `html` ;
- `SendGridAPIClient` : classe onde passamos a chave de API que usaremos no projeto, ele retornará para a variável `sg` uma instância do objeto `SendGridAPIClient` que será usado mais à frente para enviar o e-mail;
- `sg.send` : aqui temos o objeto `sg` e o método `send` que realiza o envio do e-mail para seu destinatário;
- `response` : O `response` contém o resultado do envio, sendo ele bem-sucedido ou não; dentro dele teremos 3 elementos:
 - `status_code` : código do envio, 200 ou 202 para envio com sucesso;
 - `body` : mensagem que a API retorna após o envio ser realizado;
 - `headers` : elementos do cabeçalho que a API retorna, sendo o tipo de envio, token etc.

Com essa classe criada, temos nosso serviço de e-mail pronto e disponível para ser usado em qualquer parte do sistema. Agora começaremos a colocá-lo em uso dentro do sistema. Mãos à obra!

11.4 Solicitação de recuperação de senha

Antes de fazermos a rota de "Esqueci minha senha", vamos adicionar em nossa `view` de login um link que nos levará para a tela de "Esqueci minha senha". Para isso, basta alterar o código do arquivo `login.html` como no exemplo a seguir:

```
<button class="btn btn-lg btn-primary btn-block"
type="submit">Entrar</button>
<!-- Adicione o link a seguir após o botão `Entrar` -->
<a href="http://localhost:8000/recovery-password/">Recuperar senha</a>
<!-- Até aqui, o código a seguir já existe, mantenha-o -->
{% if data.status == 401 %}
    {% if data.type == 1 %}
        <div class="alert alert-danger" role="alert">
```

Agora que temos um botão de recuperar senha em nossa tela de login, o usuário poderá ser redirecionado com mais facilidade. Vamos seguir para o próximo passo.

Nesse momento, criaremos o método em nossa `controller`, que será responsável por gerar um `token` em nosso sistema. Ele será atrelado ao usuário através do campo `recovery_code` que existe na tabela `users`. Esse mesmo `token` será usado mais à frente para o usuário acessar a tela de troca de senha. Abra o arquivo `User.py` da pasta `controller` e adicione os itens a seguir.

controller/User.py

```
from model.User import User

from datetime import datetime, timedelta
import jwt
from config import app_config, app_active

"""Importe o arquivo a seguir"""
from controller.Email import EmailController

config = app_config[app_active]
```

```

class UserController():
    def __init__(self):
        self.user_model = User()
        """Adicione a linha a seguir em seu método __init__"""
        self.email_controller = EmailController()

    """Adicione os métodos a seguir em seu arquivo"""
    def recovery(self, to_email):
        self.user_model.email = to_email
        res = self.user_model.get_user_by_email()

        if res is not None:
            user_id = res.id
            username = res.username

            recovery_code = self.generate_auth_token({
                'id': user_id,
                'username': username
            }, exp=5)
            recovery_code = recovery_code.decode("utf-8")
            try:
                self.user_model.id = res.id
                res = self.user_model.update({
                    'recovery_code': recovery_code
                })

                if res:
                    content_text = 'Olá %. Para realizar a alteração de senha,
você precisa acessar a seguinte url: %snew-password/%s' % (username,
config.URL_MAIN, recovery_code)
                else:
                    return {
                        'status_code' : 401,
                        'body' : 'Erro ao gerar código de envio',
                    }

            except:
                return {
                    'status_code' : 401,

```

```

        'body' : 'Erro ao gerar código de envio',
    }

    try:
        result = self.email_controller.send_email(to_email, 'Recuperação
de senha', content_text)
    except:
        return {
            'status_code' : 401,
            'body' : 'Erro no serviço de e-mail. Por favor. Entre em
contato com o administrador.',
        }
    else:
        result = {
            'status_code' : 401,
            'body' : 'Usuário inexistente',
        }

    return result

```

Vamos entender o que foi adicionado ao código.

- `EmailController` : aqui importamos nosso serviço de e-mail, aquele que criamos no início do capítulo;
- `self.user_model.email` : aqui estamos pegando como parâmetro o e-mail que será passado através do formulário da `view` ;
- `self.user_model.get_user_by_email()` : agora que temos o e-mail do usuário e já o passamos para o atributo `self.user_model.email` pegaremos todos os dados necessários do usuário através desse método `get_user_by_email` , para podermos gerar um `token` válido para esse usuário realizar sua troca de senha;
- `self.generate_auth_token` : aqui pegaremos os dados do usuário como `id` e `username` para criar o token do usuário, com um tempo de expiração de 5 minutos;
- `recovery_code.decode("utf-8")` : aqui nós preparamos o código para ser enviado pela `url` ; para isso, ele deve ser uma `string` `utf-8` e não um conjunto de `bytes` , que é o formato padrão em que o método `generate_auth_token` gera o `token` ;

- `self.user_model.update` : aqui nós vamos armazenar o `token` que foi criado no banco de dados, atualizando a coluna `recovery_code` daquele usuário específico;
- `content_text` : aqui temos a variável que armazena o texto que enviaremos por e-mail. Esse texto conterá o caminho que usuário deverá acessar para fazer a alteração de senha. Fique tranquilo quanto à `url` , vamos criá-la mais à frente;
- `self.email_controller.send_email` : aqui temos o método que criamos dentro do serviço de envio de e-mail, onde passaremos como parâmetro o e-mail do receptor, o título da mensagem e o corpo do texto, no caso aqui, `context_text` .

Com tudo isso feito, temos nosso método pronto para receber um e-mail e enviar um token de alteração de senha!

Agora vamos criar as `views` de recuperação de senha do nosso sistema. Abra o arquivo `app.py` e adicione os itens a seguir.

app.py

```
# Esses métodos já existem, sobrescreva-os
@app.route('/recovery-password/')
def recovery_password():
    return render_template('recovery.html', data={'status': 200, 'msg':
None, 'type': None})

@app.route('/recovery-password/', methods=['POST'])
def send_recovery_password():
    user = UserController()

    result = user.recovery(request.form['email'])

    if result['status_code'] == 200 or result['status_code'] == 202:
        return render_template('recovery.html', data={'status':
result['status_code'], 'msg': 'Você receberá um e-mail em sua caixa para
alteração de senha.', 'type': 3})
    else:
        return render_template('recovery.html', data={'status':
result['status_code'], 'msg': result['body'], 'type': 1})
```

Como podemos ver, a primeira rota carregará um arquivo `recover.html` que criaremos adiante e a segunda rota será responsável por enviar a requisição `post` que fará a solicitação da troca de senha. Nessa rota, temos o método `recovery` que acabamos de criar dentro do arquivo `User.py` da pasta `controller`. O resultado desse método é o `status_code` que o serviço de e-mail nos enviará, aquele que falamos no tópico que descrevemos o serviço `Email.py`.

Podemos perceber também que, de acordo com cada `status_code`, teremos um `type` diferente, exatamente como fizemos na `view` de `login`.

Agora que nossas rotas estão prontas, vamos adicionar o próximo código em nosso arquivo `recovery.html`, que está dentro da pasta `templates`.

```
{% extends "bootstrap/base.html" %}
{% block title %}Login{% endblock %}

{% block styles %}
{{super()}}
<link rel="stylesheet" href="{url_for('.static',
filename='css/login.css')}">
{% endblock %}

{% block content %}
<form class="form-signin text-center" action="/recovery-password/"
method="post">
    
    <h1 class="h3 mb-3 font-weight-normal">Solicitar nova senha</h1>
    <div class="checkbox mb-3">
        <label for="inputEmail" class="sr-only">E-mail</label>
        <input name="email" type="email" id="inputEmail" class="form-
control" placeholder="E-mail" required autofocus>
    </div>
    <button class="btn btn-lg btn-primary btn-block" type="submit">Enviar
```

```

solicitação</button>
    {% if data.status == 200 or data.status == 202 %}
        {% if data.type == 3 %}
            <div class="alert alert-success" role="alert">
                {{data.msg}}
            </div>
        {% endif %}
    {% else %}
        {% if data.type == 1 %}
            <div class="alert alert-danger" role="alert">
                {{data.msg}}
            </div>
        {% endif %}
    {% endif %}
</form>
{% endblock %}

{% block scripts %}
    {{super()}}
    <script src="{{url_for('.static', filename='js/custom.js')}}"></script>
{% endblock %}

```

Esse arquivo `html` terá um input de tipo e-mail que será enviado como método `post` para nosso servidor e acionará o `endpoint` `/recovery-password/` dessa forma conseguiremos enviar o e-mail com o link de alteração de senha para o usuário que o solicitou.

Com tudo isso criado, temos nossa solicitação de alteração de senha finalizada e funcionando. Sua `view` ficará parecida com a da imagem:



Solicitar nova senha

Figura 11.4: Tela de recuperação de senha.

Se tentarmos enviar uma solicitação usando um e-mail cadastrado no sistema o resultado será esse:



Solicitar nova senha

tiagoluzribeirodasilva@gmail.com

Enviar solicitação

Você receberá um e-mail em sua caixa
para alteração de senha.

Figura 11.5: Solicitação de recuperação de senha bem sucedida.

Caso o e-mail não exista, receberemos a seguinte mensagem:



Figura 11.6: Solicitação de recuperação de senha mal sucedida.

Nesse exato momento, se você tentar enviar uma solicitação de recuperação de senha, ela já vai funcionar. Você já receberá um e-mail com o `token`, como visto na próxima tela:

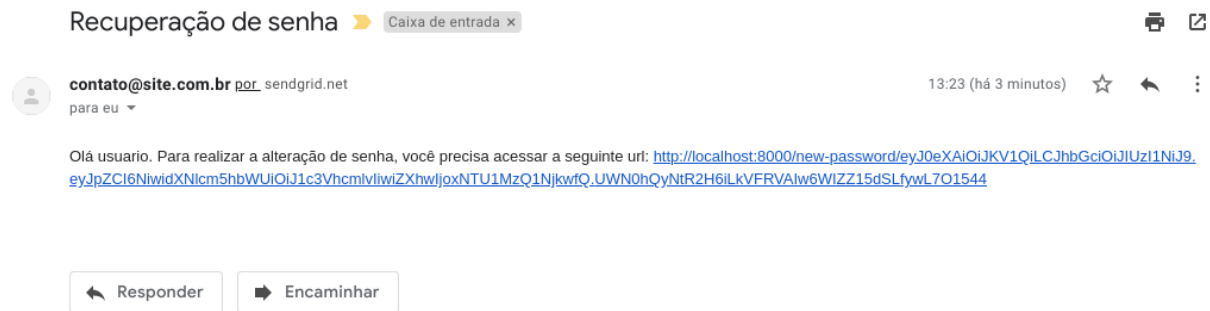


Figura 11.7: E-mail de recuperação de senha.

Mas esse link não funcionará ainda, pois ainda não existe um serviço que verifique se o `token` é válido para que a alteração de senha seja realizada, mas fique tranquilo, pois esse é nosso próximo passo!

11.5 Validando o token de recuperação

Nessa etapa, vamos pegar a `url` recebida no e-mail e verificar se o `token` que está nela está é válido. A `url` seguirá um padrão parecido com esse `http://localhost:8000/new-password/eyJ0eXAiOiJKV1QiL...` e poderá ou não ser válida, tendo em vista que o tempo de expiração é de 5 minutos. Se o usuário demorar mais tempo que isso, precisará realizar outra solicitação, para que um novo `token` seja gerado. Então mãos à obra.

Abra o arquivo `User.py` da pasta `model` e adicione os métodos a seguir:

model/User.py

```
def get_user_by_recovery(self):
    try:
        res =
```

```

db.session.query(User).filter(User.recovery_code==self.recovery_code).first()
    except Exception as e:
        res = None
        print(e)
    finally:
        db.session.close()
        return res

"""Esse método já existe, complete-o
conforme o exemplo a seguir"""
def update(self, obj):
    try:
        res = db.session.query(User).filter(User.id == self.id).update(obj)
        db.session.commit()
        return True
    except Exception as e:
        print(e)
        db.session.rollback()
        return False

```

Esse método fará um `select` no banco de dados, verificando se o `token` passado no atributo `recovery_code` existe na base de dados, retornando os dados do usuário; caso contrário retornará `None`.

Agora precisamos criar os métodos do arquivo `User.py` da pasta `controller`, para podermos validar o `token` e atualizar a senha. Adicione os métodos a seguir no arquivo.

controller/User

```

def get_user_by_recovery(self, recovery_password):
    self.user_model.recovery_code = recovery_password
    return self.user_model.get_user_by_recovery()

def new_password(self, user_id, password):
    self.user_model.set_password(password)
    self.user_model.id = user_id

    return self.user_model.update({

```

```
        'password': self.user_model.password
    })
```

O método `get_user_by_recovery` chamará o método que criamos na `model` e retornará os dados do usuário ou `None` (caso esse token não exista) para a rota que criaremos. O método `new_password` atualizará a senha do usuário. Vamos abrir o arquivo `app.py` e adicionar as rotas que nos permitirão realizar essas tarefas.

```
@app.route('/new-password/<recovery_code>')
def new_password(recovery_code):
    user = UserController()
    result = user.verify_auth_token(recovery_code)

    if result['status'] == 200:
        res = user.get_user_by_recovery(str(recovery_code))
        if res is not None:
            return render_template('new_password.html', data={'status':
result['status'], 'msg': None, 'type': None, 'user_id': res.id})
        else:
            return render_template('recovery.html', data={'status': 400,
'msg': 'Erro ao tentar acessar os dados do usuário. Tente novamente mais
tarde.', 'type': 1})
        else:
            return render_template('recovery.html', data={'status':
result['status'], 'msg': 'Token expirado ou inválido, solicite novamente a
alteração de senha', 'type': 1})

@app.route('/new-password/', methods=['POST'])
def send_new_password():
    user = UserController()
    user_id = request.form['user_id']
    password = request.form['password']

    result = user.new_password(user_id, password)

    if result:
        return render_template('login.html', data={'status': 200, 'msg':
'Senha alterada com sucesso!', 'type': 3, 'user_id': user_id})
    else:
```

```
return render_template('new_password.html', data={'status': 401,
'msg': 'Erro ao alterar senha.', 'type': 1, 'user_id': user_id})
```

A primeira rota receberá como parâmetro `<recovery_code>` . Quando pegarmos o valor de `recovery_code` , antes de verificarmos se existe na base de dados, chamaremos o método `verify_auth_token` .

Analisaremos o `token` para sabermos se é válido dentro das diretrizes do padrão `jwt` . Após isso, veremos também se ele não está expirado, e caso esteja, o usuário será redirecionado para a `view recovery.html` ; já caso esteja tudo válido, o método `get_user_by_recovery` será chamado e pegaremos os dados do usuário através do `token` que foi passado, assim saberemos qual usuário terá sua senha alterada.

Com os dados do usuário capturados, carregaremos a rota `new_password.html` passando o `user_id` como um dos parâmetros e usaremos esse id mais à frente para atualizar a senha do usuário.

A segunda rota será usada para realizarmos a requisição `post` que o usuário fará para enviar sua nova senha para o sistema. Para atualizar a senha, chamaremos o método `new_password` que criamos dentro da `model User.py` . Se o método realizar a atualização com sucesso, uma mensagem de tipo 3 será emitida informando que tudo ocorreu bem, dentro de uma box verde; caso algum erro ocorra, outra mensagem será emitida, mas de tipo 1, ou seja, com erro dentro de uma caixa vermelha, bem similar ao que ocorre na tela de login e na tela de solicitação de nova senha.

Agora precisamos configurar a `view new_password.html` para que nosso usuário possa realizar de verdade a mudança de senha. Para isso, precisaremos adicionar os códigos a seguir dentro do arquivo `new_password.html` que está na pasta `templates` .

```
{% extends "bootstrap/base.html" %}
{% block title %}Login{% endblock %}
```

```
{% block styles %}
```

```

{{super()}}
<link rel="stylesheet" href="{{url_for('.static',
filename='css/login.css')}}">
{% endblock %}

{% block content %}
    <form class="form-signin text-center" action="/new-password/"
method="post">
        
        <h1 class="h3 mb-3 font-weight-normal">Nova Senha</h1>
        <div class="checkbox mb-3">
            <label for="inputPassword" class="sr-only">Senha</label>
            <input name="password" type="password" id="inputPassword"
class="form-control" placeholder="Senha" required>
        </div>
        <input name="user_id" type="hidden" id="inputUser_id" class="form-
control" value="{{data.user_id}}">
        <button class="btn btn-lg btn-primary btn-block" type="submit">Alterar
senha</button>
        {% if data.status == 401 %}
            {% if data.type == 1 %}
                <div class="alert alert-danger" role="alert">
                    {{data.msg}}
                </div>
            {% elif data.type == 2 %}
                <div class="alert alert-warning" role="alert">
                    {{data.msg}}
                </div>
            {% endif %}
        {% elif data.status == 200 %}
            {% if data.type == 3 %}
                <div class="alert alert-success" role="alert">
                    {{data.msg}}
                </div>
            {% endif %}
        {% endif %}
    </form>
{% endblock %}

```

```
{% block scripts %}  
{{super()}}  
<script src="{{url_for('.static', filename='js/custom.js')}}"></script>  
{% endblock %}
```

Com tudo concluído, se tentarmos realizar a solicitação de alteração de senha, receberemos um novo e-mail e, ao clicar no link, uma tela como a imagem adiante será exibida.


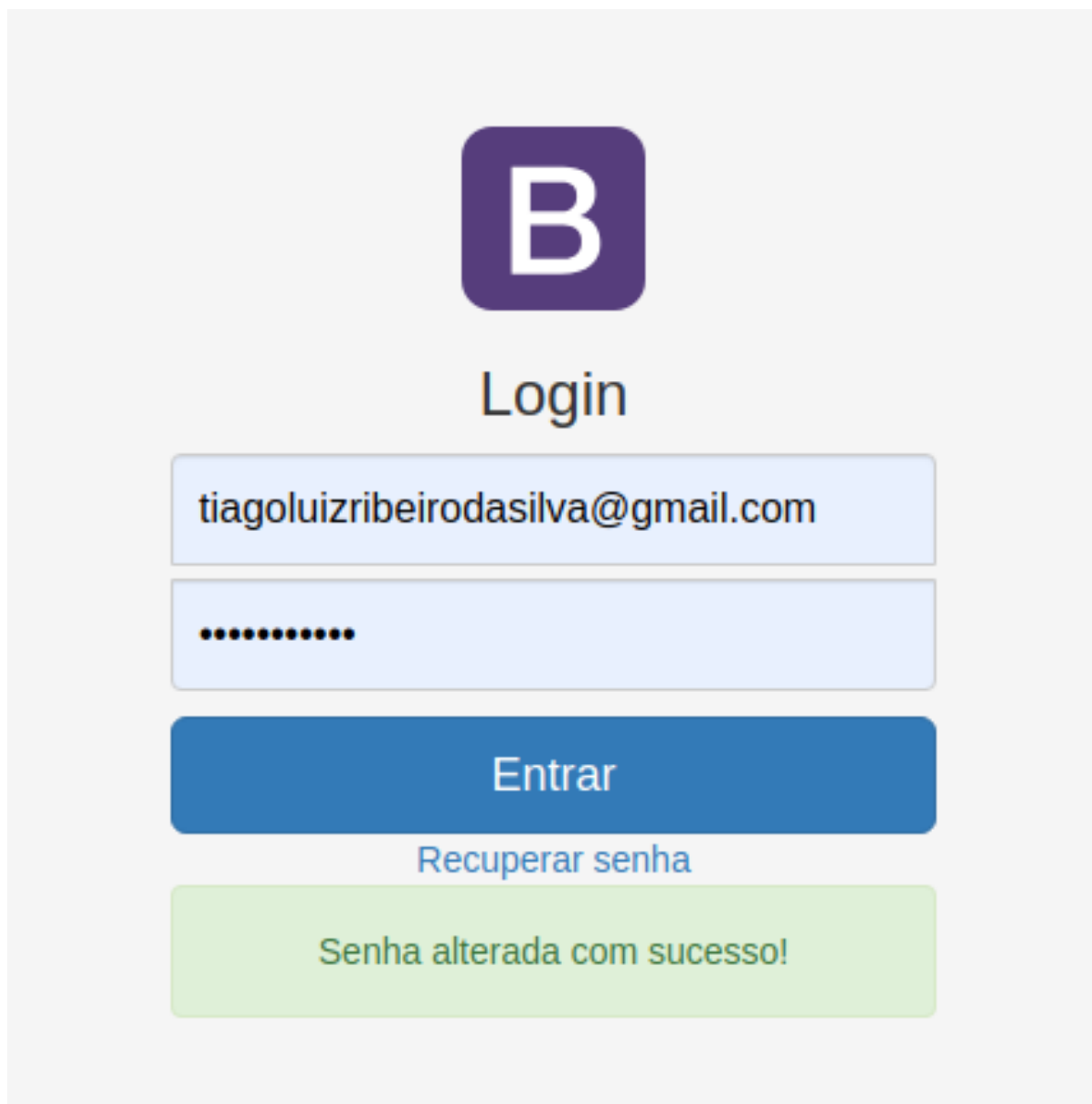
A imagem mostra uma interface web para alterar a senha. No topo, há um ícone de perfil: uma letra 'B' branca dentro de um quadrado arredondado de cor roxa. Abaixo do ícone, o texto 'Nova Senha' é exibido em uma fonte sans-serif. Logo abaixo, há um campo de entrada de texto branco com uma borda azulada; o conteúdo do campo é representado por dez pontos cinza e um cursor de texto no final. Abaixo do campo, há um botão retangular de cor azul com o texto 'Alterar senha' em branco.

Figura 11.8: Alterando a senha.

Ao realizar a alteração da senha, o usuário será redirecionado para a tela de login e uma mensagem informará que a senha foi alterada com sucesso.



The image shows a login interface. At the top is a purple square logo with a white letter 'B'. Below it is the word 'Login' in a large, dark font. There are two light blue input fields: the first contains the email 'tiagoluizribeirodasilva@gmail.com' and the second contains a series of dots representing a password. Below the password field is a blue button with the text 'Entrar'. Underneath the button is a link that says 'Recuperar senha' in a smaller, blue font. At the bottom is a green rectangular box with the text 'Senha alterada com sucesso!' in a bold, green font.

Figura 11.9: Senha alterada com sucesso.

Caso o usuário tenha seu `token` inválido ou expirado, ele será redirecionado para a tela de solicitação de nova senha e precisará solicitar novamente, para receber outro token via e-mail.



The image shows a web interface for requesting a new password. At the top is a purple square logo with a white letter 'B'. Below the logo is the title 'Solicitar nova senha' in a dark blue font. There is a white input field with the placeholder text 'E-mail'. Below the input field is a blue button with the text 'Enviar solicitação'. At the bottom is a light red rectangular box containing the error message 'Token expirado ou inválido, solicite novamente a alteração de senha' in a red font.

Figura 11.10: Token expirado.

Com isso temos nosso serviço de recuperação de senha concluído e assim também um sistema seguro e robusto.

Conclusão

Este é o momento e que podemos analisar tudo que foi feito no decorrer de todo o livro. Criamos um sistema completo, robusto e seguro, utilizando um dos melhores frameworks para `python` que existe.

Essa sem dúvidas foi uma experiência única para todos nós. Espero que tenha gostado e que tire um ótimo proveito de cada linha que fora colocada neste livro. Foi com um imenso carinho e dedicação que eu e você chegamos até essa etapa, a última página.

Daqui em diante você está totalmente apto a criar sistemas web seguros e completos utilizando Flask Framework.

Até a próxima!!!