



Flask

CURSO WEB PYTHON

FRAMEWORK FLASK



Jonathas H. Miente

@campcodebrasil

Introdução ao Flask

Desenvolvimento Web Simples e Poderoso

Bem-vindo à sua jornada no mundo do desenvolvimento web com Flask! Se você está buscando construir aplicações web de forma rápida, eficiente e com uma curva de aprendizado amigável, você está no lugar certo.

Flask é um framework web em Python que se destaca por sua simplicidade e flexibilidade. Criado para ser fácil de entender e rápido para começar, Flask é a escolha ideal para desenvolvedores que desejam construir aplicações web robustas sem a complexidade desnecessária.

Nesta apostila, vamos explorar os fundamentos do Flask, desde a configuração inicial até a criação de aplicações web dinâmicas e interativas. Ao longo do caminho, vamos abordar conceitos cruciais, como roteamento, templates, interação com bancos de dados e autenticação de usuários

Sumário

1. Módulo Básico.....	4
Preparação de Ambiente.....	5
Instalando o Flask.....	5
Exemplo Básico de uma aplicação Flask.....	6
Estrutura de um Projeto Flask.....	7
Primeiro Projeto (Projeto Básico).....	9
Reutilizando Arquivos HTML.....	12
Função extends.....	12
Função include.....	13
Trabalhando com arquivos Estáticos.....	14
Enviando dados do Servidor para o Navegador.....	17
Utilizando Link Dinâmico para rotas.....	20
Rotas com parâmetros.....	21
2. Módulo Intermediário.....	23
Banco de Dados.....	24
Models.....	26
Formulários.....	27
Campos.....	29
Validações.....	29

Salvando Informações no Banco de Dados.....	30
Trabalhando um pouco com Front-End	33
Recuperando Informações no Banco de Dados	36
Módulo Avançado	39
Controle de Login	39
Cadastro de Usuário	41
Criando rota para Login	43
Criando Rota para Logout	45
Estrutura de Decisão if no HTML com Flask	46
Permissão de Acesso para Usuários Logados	47
Vincular registros a outras tabelas (FK)	47
Trabalhando com Tipo de dados de arquivos	50
Trabalhando com Variável de Ambiente.....	53
GitHub.....	54
Instalação	54
Criando Repositório.....	55
Preparando Repositório e Arquivo .gitignore	56
Atualizando Repositório	57
Deploy	58
Render.com (Opção Gratuita)	58
Utilizando CPanel	65

1. Módulo Básico

O que iremos aprender nesse capítulo?

Preparação do Ambiente:

Configuração do ambiente de desenvolvimento para Flask, incluindo a instalação do Flask, criação de um ambiente virtual e configuração do ambiente de trabalho.

Estrutura de um Projeto Flask:

Exploração da estrutura básica de pastas e arquivos em um projeto Flask, incluindo o entendimento dos diretórios static, templates e arquivos essenciais como main.py.

Reutilização de Arquivos HTML:

Uso de templates para reutilizar componentes HTML em várias páginas, permitindo uma estrutura modular e consistente em todo o projeto.

Arquivos Estáticos:

Utilização de arquivos estáticos, como CSS, JavaScript e imagens, para melhorar a apresentação e a funcionalidade das páginas web.

Envio de Dados do Servidor para o Navegador:

Compreensão de como enviar dados dinâmicos do servidor Flask para o navegador, possibilitando a exibição de informações atualizadas nas páginas web.

Links Dinâmicos e Rotas com Parâmetros:

Implementação de rotas dinâmicas que respondem a diferentes URLs e aceitam parâmetros variáveis, permitindo a criação de páginas dinâmicas com informações específicas.

Objetivos do Módulo:

- Estabelecer uma base sólida para o desenvolvimento web usando Flask.
- Capacitar os alunos a criarem uma estrutura de projeto organizada e eficiente.
- Entender a importância e aplicação dos arquivos estáticos e templates para a construção de interfaces dinâmicas.
- Facilitar a comunicação entre o servidor e o navegador para exibir dados dinâmicos e criar páginas dinâmicas.

Preparação de Ambiente

Antes de tudo, é necessário ter o Python instalado. Flask é um framework web para Python. Você pode verificar a presença do Python em seu sistema digitando `python --version` no terminal ou prompt de comando. Caso não possua o Python instalado, pode obter o instalador no site oficial, www.python.org/

Após instalado o Python, deve ser criado um ambiente virtual para seu projeto, para isso no terminal digite o seguinte comando:

```
python -m venv .venv
```

Com isso irá criar em seu projeto uma pasta chamada “.venv” contendo uma versão do python e isolando as bibliotecas necessárias para seu projeto.

Para ativar o ambiente virtual basta digitar no terminal o seguinte comando:

```
.venv\Scripts\activate
```

Instalando o Flask

Antes de iniciarmos a construção de nossa aplicação, devemos instalar a biblioteca em nosso ambiente, para isso após ativar o ambiente virtual, digite o seguinte comando no terminal:

```
pip install flask
```

Exemplo Básico de uma aplicação Flask

Abaixo veremos como é construído uma aplicação básica com flask.

```
1. from flask import Flask
2.
3. app = Flask(__name__)
4.
5. @app.route('/')
6. def homepage():
7.     return 'Minha Primeira Página Flask'
8.
9. if __name__ == '__main__':
10.     app.run(debug=True)
```

Vamos entender o código acima:

Na linha 1, estamos realizando a classe `Flask` do framework `flask`, será essa classe que será responsável por instanciar nossa aplicação.

Na linha 3, estamos criando o aplicativo, é passado como parâmetro a variável `__name__` que serve para indicar que o nome da aplicação será o nome do arquivo `py`

Na linha 5, estamos definindo a rota para a view (entenderemos com mais detalhes no módulo VIEW), no exemplo, estamos definindo a rota raiz da aplicação.

Na linha 6, estamos iniciando a função da view

Na linha 7, estamos retornando o texto `'Minha Primeira Página Flask'` para ser exibido quando a aplicação estiver executando e a rota raiz for chamada

Na linha 9, estamos verificando se o arquivo que está sendo executado é o arquivo `main.py`, se for irá seguir para linha 10

Na linha 10, estamos iniciando a aplicação, definindo como parâmetro o `debug=True`, que serve para depurar o código, ou seja, sempre que houver uma alteração no código, a aplicação será reiniciada.

Estrutura de um Projeto Flask

Estrutura de Arquivos

No exemplo acima, construímos uma aplicação Flask sem organizar na estrutura de um projeto Flask, a partir de agora iremos começar a organizar essa estrutura.

A estrutura de um projeto Flask é construída separando os arquivos de acordo com suas funcionalidades, sendo que na raiz do projeto, iremos manter o arquivo `main.py` (podendo ser outro nome se desejar, por convenção o arquivo normalmente é chamado como `main.py` ou `run.py`), esse arquivo será responsável por iniciar a aplicação.

O restante do projeto será incluso dentro de uma pasta com o nome da aplicação, ou podendo chamar de “app”, dentro dessa pasta teremos a seguinte estrutura

- `__init__.py` - Arquivo responsável por inicializar o módulo, será incluso nele a construção da aplicação flask
- `form.py` - Arquivo responsável por manter os formulários da aplicação
- `models.py` - Arquivo responsável por manter os modelos (comunicação com o banco de dados) da aplicação
- `routes.py` - Arquivo responsável por manter as views da aplicação
- `templates` - Pasta responsável por manter os arquivos HTML
- `static` - Pasta responsável por manter os arquivos estáticos (css, img, js, etc...)

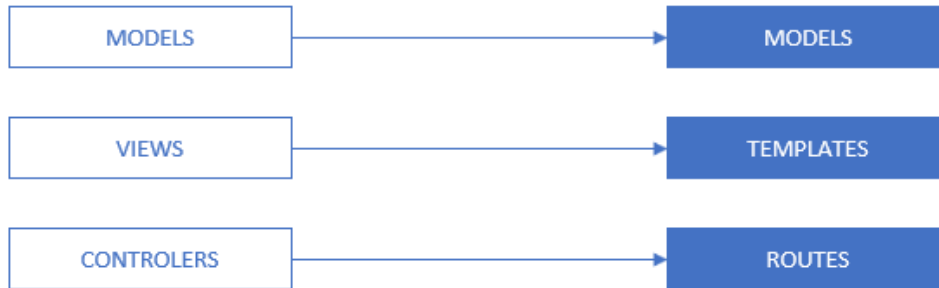
Um projeto Flask pode conter uma ou mais aplicações, para cada aplicação deve ser respeitado a estrutura acima, mantendo uma pasta na raiz do projeto para cada aplicação.

A estrutura final de um projeto com uma aplicação seria da seguinte forma:

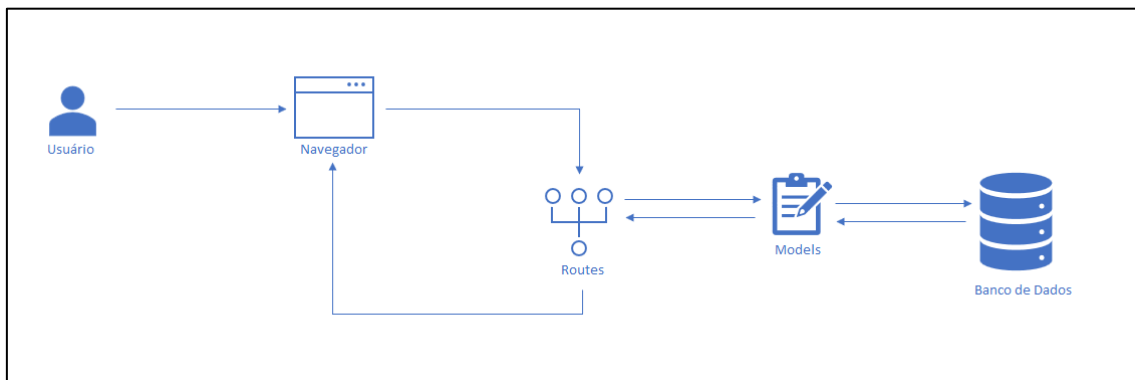
```
Meu_projeto/
|-- app/
|   |-- static/
|   |   |-- css/
|   |   |   |-- style.css
|   |   |-- templates
|   |       |-- base.html
|   |       |-- index.html
|   |       |-- about.html
|   |-- __init__.py
|   |-- form.py
|   |-- models.py
|   |-- routes.py
|-- main.py
```

Estrutura MVC

O Flask não possui uma estrutura de MVC (Model-View-Controller) rígida como alguns outros frameworks. No entanto, você pode aplicar conceitos semelhantes de organização do código para alcançar uma separação de preocupações semelhante.



Comportamento de um projeto Flask



Vamos entender a imagem acima:

1. O Usuário interage no navegador
 - Digitando a url e rota (a rota é tudo após a primeira / após a url) (Método GET)
 - Enviando dados em Formulários (Método POST)
2. O navegador envia os dados para a Rota, que “chama” sua função
3. Na função da rota, pode ou não se comunicar com o Models (caso precise utilizar algum dado que esteja armazenado em banco de dados)
4. O Models é responsável pela comunicação com o banco de dados
5. Após a função da rota processar todas as informações, devolve para o navegador renderizar a nova página para o usuário.

Primeiro Projeto (Projeto Básico)

Para esse projeto, iremos construir um projeto já com a estrutura recomendada, ou seja, vamos construir de acordo com o tópico anterior, para isso, vamos iniciar com os seguintes passos:

1. Iniciando Projeto

Abra essa pasta com seu editor de código preferido, recomendamos o uso do PyCharm ou VS Code

Dentro dessa pasta repita o processo da preparação de ambiente e instalação do Flask

2. Preparando Projeto

Dentro dessa pasta iremos criar algumas estruturas e arquivos:

- O primeiro arquivo que iremos criar será o arquivo `__init__.py`

```
from flask import Flask

app = Flask(__name__)
```

- O segundo arquivo que iremos criar será o `routes.py`

```
from flask import render_template
from app import app

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

Neste arquivo, estamos importando a função `render_template` que serve para renderizar um arquivo HTML.

Após importamos o aplicativo `app` que está dentro do módulo `app` (criado na etapa anterior)

Depois definimos a rota raiz e criamos a função com o retorno da renderização do arquivo `'index.html'`

- O terceiro arquivo que iremos criar, precisaremos criar a pasta templates, dentro da pasta app, depois iremos criar o arquivo HTML dentro da pasta templates.

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
  <meta charset="UTF-8">
  <title>Primeiro Projeto Flask</title>
</head>
<body>

  <h1>Bem Vindos ao Primeiro Projeto Flask</h1>

  <p>Está é a página Raiz do projeto</p>

</body>
</html>
```

Esse é um arquivo básico de HTML, não vamos entrar em detalhes no front-end da aplicação nessa apostila, caso não conheça as ferramentas de desenvolvimento Front-End, indico estudar as seguintes linguagens:

- HTML
 - CSS
 - JavaScript
- Agora iremos voltar no arquivo `__init__.py`, precisamos importar a view homepage dentro desse arquivo

```
from flask import Flask

app = Flask(__name__)

from app.routes import homepage
```

A importação das views deve ser realizada após a criação do app, pois as views dependem do app para funcionar, caso seja importado antes, sua aplicação irá “quebrar” pois estará com referência circular.

- Por fim devemos voltar na raiz da aplicação e criar o arquivo main.py

```
from app import app

if __name__ == '__main__':
    app.run(debug=True)
```

Nesse arquivo iremos importar o app e inicializar a aplicação

Agora podemos rodar o arquivo main.py, onde será inicializado o servidor, criando assim nosso primeiro projeto Flask

```
* Serving Flask app 'app'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
```

Bem Vindos ao Primeiro Projeto Flask

Esta é a página Raiz do projeto

No final obteremos a seguinte estrutura de arquivos:

```
projeto_basico/
|-- app/
|--     |-- templates
|--     |--     |-- index.html
|--     |-- __init__.py
|--     |-- routes.py
|-- main.py
```

Reutilizando Arquivos HTML

Função extends

A função `extends` serve para reutilizar um arquivo HTML, ou seja, podemos criar um arquivo base e estender sua estrutura para outros arquivos, assim reduzindo drasticamente os códigos, melhorando a manutenção.

Para isso, vamos copiar o conteúdo do arquivo `index.html` em um novo arquivo, chamado `base.html`.

Após isso, vamos definir blocos onde poderá ser editado o conteúdo em outros arquivos, esses blocos tem a seguinte sintaxe: `{% block NOME_BLOCO %} CONTEÚDO DEFAULT {% endblock %}`

No conteúdo default pode criar conteúdos, caso o bloco não seja chamado no arquivo que está estendendo o código, o conteúdo padrão será exibido, esse conteúdo não é obrigatório, podendo não possuir nenhum conteúdo.

No nosso arquivo base, vamos criar dois blocos, um para o título da página e outro para o conteúdo da página.

O arquivo `base.html` ficará da seguinte forma:

```
1 <!DOCTYPE html>
2 <html lang="pt-br">
3 <head>
4 |   <meta charset="UTF-8">
5 |   <title>{% block title %}Primeiro Projeto Flask{% endblock %}</title>
6 </head>
7 <body>
8 |   {% block content %} {% endblock %}
9 </body>
10 </html>
```

Note que no bloco `'title'` criamos um conteúdo padrão, com isso caso não “chamarmos” esse bloco no arquivo que está estendendo o arquivo base, manteremos o padrão.

Quando “chamamos” o bloco, todo conteúdo default do bloco é apagado e substituído pelo que você está incluindo no arquivo novo.

Agora vamos editar nosso arquivo `index`, removendo todos os conteúdos em comum com o base e alterando o conteúdo do bloco `'content'`, o arquivo `index.html` ficará da seguinte forma:

```
1 {% extends 'base.html' %}
2
3 {% block content %}
4 |   <h1>Bem Vindos ao Primeiro Projeto Flask</h1>
5
6 |   <p>Está é a página Raiz do projeto</p>
7 {% endblock %}
```

Note que na primeira linha utilizamos a sintaxe `{% extends 'base.html' %}` que serve para informarmos que esse arquivo irá estender o arquivo `base.html`

Se executarmos nosso código, não haverá nenhuma alteração no conteúdo, porém todos os arquivos que estenderem o arquivo base.html serão atualizados quando modificarmos o arquivo base.html.

Função include

Outra forma de reutilizar arquivos html é utilizando a função include, que serve para incluir o conteúdo de outro arquivo html, para isso vamos criar um arquivo chamado lista.html, esse arquivo irá conter uma lista das funcionalidades do flask no HTML que estamos aprendendo, por hora iremos manter dois itens nessa lista, sendo as funções que acabamos de aprender.

Arquivo lista.html

```
1 <hr>
2 <p>Lista de funções Flask no HTML</p>
3 <ul>
4     <li>extends</li>
5     <li>include</li>
6 </ul>
```

Agora iremos incluir essa lista dentro do arquivo index.html, dentro do bloco contente iremos incluir a seguinte linha:

```
{% include 'lista.html' %}
```

Isso fará que o conteúdo do arquivo lista.html seja incorporado no arquivo index.html

```
1 {% extends 'base.html' %}
2
3 {% block content %}
4     <h1>Bem Vindos ao Primeiro Projeto Flask</h1>
5
6     <p>Está é a página Raiz do projeto</p>
7
8     {% include 'lista.html' %}
9 {% endblock %}
```

Ao executarmos nosso projeto, teremos o seguinte resultado:

Bem Vindos ao Primeiro Projeto Flask

Está é a página Raiz do projeto

Lista de funções Flask no HTML

- extends
- include

Trabalhando com arquivos Estáticos

Os arquivos estáticos desempenham um papel crucial no desenvolvimento web, aprimorando a experiência do usuário e fornecendo recursos visuais e funcionais para uma aplicação. No contexto do Flask, uma estrutura web em Python, os arquivos estáticos desempenham um papel importante na construção de interfaces web dinâmicas e atraentes.

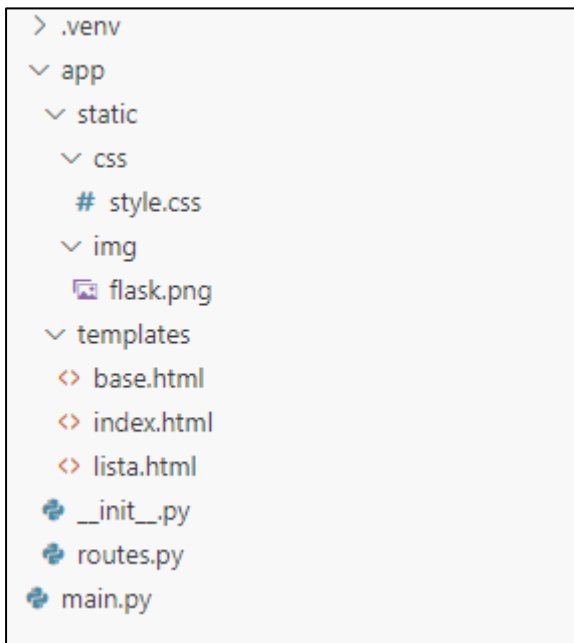
O que são Arquivos Estáticos?

Arquivos estáticos consistem em recursos que não são dinamicamente gerados pelo servidor web. Esses recursos incluem folhas de estilo (CSS), scripts (JavaScript), imagens, fontes e outros tipos de arquivos que são carregados diretamente pelo navegador do usuário.

Para trabalharmos com arquivos estáticos, devemos criar na pasta do projeto (app) uma pasta se chamada 'static'

Dentro dessa pasta, separamos os arquivos por categoria, uma pasta para cada, sendo img para imagens, css para arquivos css e js para arquivos Javascript (essa abordagem não é obrigatória, porém é uma boa prática)

Para o projeto que estamos construindo, vamos criar a pasta para arquivos de imagens e para arquivos css, ficando com a seguinte estrutura de arquivos:



Note que já incluímos um arquivo em cada pasta, sendo na pasta css é um arquivo em branco, onde estaremos incluindo algumas classes para o HTML estar utilizando e na pasta img, incluímos um arquivo de imagem com o logo do Flask.

Utilizando arquivos estáticos

Para utilizarmos arquivos estáticos, precisamos primeiro atualizar nosso arquivo de rotas, incluindo a importação da função `url_for`

O arquivo ficará da seguinte forma:

```
1 from flask import render_template, url_for
2 from app import app
3
4 @app.route('/')
5 def homepage():
6     return render_template('index.html')
7
```

Após isso, vamos abrir o arquivo `index.html` e incluir o logo do flask, para isso, vamos incluir a seguinte linha em nosso código:

```

```

A tag `img`, informa que iremos incluir uma imagem no HTML e o atributo `src` é o local da imagem, utilizamos o `url_for` para buscar o caminho do arquivo estático, passando como parâmetro o nome da pasta e o nome do arquivo (incluindo a subpasta)

O arquivo `index.html` ficará da seguinte forma:

```
1 {% extends 'base.html' %}
2
3 {% block content %}
4
5     
6
7     <h1>Bem Vindos ao Primeiro Projeto Flask</h1>
8
9     <p>Está é a página Raiz do projeto</p>
10
11     {% include 'lista.html' %}
12
```

Resultado no navegador:



Iremos incluir também o arquivo css, porém iremos fazer isso no arquivo base.html

Dentro do bloco HEAD vamos incluir a linha:

```
<link rel="stylesheet" href="{{ url_for('static', filename='css/style.css') }}">
```

Na tag BODY vamos incluir o atribulo class="text-blue1"

Ficando com o Código:

```
1 <!DOCTYPE html>
2 <html lang="pt-br">
3 <head>
4     <meta charset="UTF-8">
5     <title>{% block title %}Primeiro Projeto Flask{% endblock %}</title>
6     <link rel="stylesheet" href="{{ url_for('static', filename='css/style.css') }}">
7 </head>
8 <body class="text-blue1">
9     {% block content %} {% endblock %}
10 </body>
11 </html>
12
```

No Arquivo style.css vamos criar a classe text-blue1

```
1 .text-blue1{
2     color: #184cc4;
3 }
4
```

Com essas alterações teremos o seguinte resultado:



Enviando dados do Servidor para o Navegador

Uma das funcionalidades mais importantes em uma aplicação web é a capacidade de transmitir dados do servidor (back-end) para o navegador do usuário (front-end). No Flask, isso pode ser alcançado de várias maneiras, permitindo que dados dinâmicos sejam exibidos nas páginas da web.

Neste tópico iremos trabalhar com dois formatos, sendo:

1. *Renderização de Templates com Dados*

O Flask permite que você passe variáveis para os templates durante a renderização, tornando possível exibir dados dinâmicos nas páginas HTML.

Para isso, dentro da função da rota, podemos criar a variável que iremos enviar para o navegador (essa variável pode estar sendo recuperada de um banco de dados por exemplo)

Após criar a variável, vamos incluir no retorno do `render_template`, ficando com a função da rota da seguinte forma:

```
4 @app.route('/')
5 def homepage():
6     usuario = 'Jonathas'
7     return render_template('index.html', usuario=usuario)
8
```

No arquivo HTML, podemos recuperar o dado utilizando duas chaves, com o nome da variável dentro das chaves, ex: `{{ usuário }}`

Atualizando o `index.html`, teremos o resultado:

```
1 {% extends 'base.html' %}
2
3 {% block content %}
4
5     
6
7     <h1>Bem Vindos ao Primeiro Projeto Flask</h1>
8
9     <p>Está é a página Raiz do projeto</p>
10
11     <p>Usuário: {{ usuario }}</p>
12
13     {% include 'lista.html' %}
14
15 {% endblock %}
```

Para cada variável, temos que incluir no retorno, caso tenha muitas variáveis isso acaba sendo inviável, uma solução é criar uma variável do tipo dicionário e para cada posição desse dicionário é um dado que você precisa passar para o navegador.

Função da rota

```
4 @app.route('/')
5 def homepage():
6     context = {
7         'usuario': 'Jonathas',
8         'idade': 34
9     }
10
11     return render_template('index.html', context=context)
12
```

Index.html

```
1  {% extends 'base.html' %}
2
3  {% block content %}
4
5      
6
7      <h1>Bem Vindos ao Primeiro Projeto Flask</h1>
8
9      <p>Está é a página Raiz do projeto</p>
10
11      <p>Usuário: {{ context['usuario'] }}</p>
12      <p>Idade: {{ context['idade'] }}</p>
13
14      {% include 'lista.html' %}
15
16  {% endblock %}
```

Resultado:



Flask

Bem Vindos ao Primeiro Projeto Flask

Está é a página Raiz do projeto

Usuário: Jonathas

Idade: 34

Lista de funções Flask no HTML

- extends
- include

2. APIs JSON

O Flask pode fornecer endpoints que respondem com dados no formato JSON. Isso permite que o front-end faça solicitações AJAX para buscar dados dinâmicos.

O primeiro passo é criar uma rota para fornecer os dados para o HTML, para essa função iremos utilizar o `jsonify` como retorno, para isso devemos importar essa função

```
from flask import jsonify
```

Depois vamos criar uma função que retorna um dicionário

```
14 @app.route('/api/dados')
15 def dados():
16     dados = {"mensagem": "Dados do back-end!"}
17     return jsonify(dados)
18
```

Após criado a rota e a função, vamos incluir um parágrafo no HTML

```
<p>Mensagem: <span id="msg_back"></span></p>
```

E adicionar o JavaScript para recuperar os dados desse endpoint

```
1  {% extends 'base.html' %}
2
3  {% block content %}
4
5      
6
7      <h1>Bem Vindos ao Primeiro Projeto Flask</h1>
8
9      <p>Está é a página Raiz do projeto</p>
10
11      <p>Usuário: {{ context['usuario'] }}</p>
12      <p>Idade: {{ context['idade'] }}</p>
13
14      <p>Mensagem: <span id="msg_back"></span></p>
15
16      {% include 'lista.html' %}
17
18      <script>
19          const msg_back = document.getElementById('msg_back')
20          fetch('/api/dados')
21              .then(response => response.json())
22              .then(data => {
23                  msg_back.innerHTML = data.mensagem;
24              });
25      </script>
26
27  {% endblock %}
28
```

Obtendo o seguinte resultado:



Utilizando Link Dinâmico para rotas

No exemplo acima utilizamos uma rota para buscar dados através do Javascript, porém por algum motivo, pode ser que a rota seja alterada (já o nome da função é mais difícil de acontecer), com isso o ideal seria utilizar a rota dinâmica.

Para usar a rota dinâmica vamos utilizar a função `url_for`, sendo que o conceito é muito próximo de usar arquivos estático.

No index html, vamos alterar a linha 20:

De: `fetch('/api/dados')`

Para: `fetch('{{ url_for("dados") }}')`

Nesse caso estamos recuperando os dados da rota através do nome da função dessa rota, assim se alterarmos o caminho da rota, não haverá problema.

Essa alteração não terá nenhum impacto visual, pois estamos alterando o comportamento para recuperar dinamicamente o caminho da rota.

Como boa prática, vamos estar utilizando essa abordagem sempre que precisarmos utilizar o caminho da rota dentro do HTML.

Vale reforçar também que para utilizarmos a função `url_for` no HTML, o arquivo da rota deve ter importado a função `url_for` (vimos isso quando aprendemos a trabalhar com arquivos estáticos)

Rotas com parâmetros

As rotas com parâmetros no Flask são uma maneira poderosa de criar URLs dinâmicas que podem aceitar valores variáveis. Isso é útil quando você precisa criar rotas que lidam com diferentes recursos ou entidades com base em identificadores únicos.

Para criar uma rota com parâmetro, devemos adicionar na rota o nome da variável dentro dos sinais <> e passar essa mesma variável como parâmetro da função, com isso conseguimos trabalhar com esse dado dentro da função e retornando os dados desejados após tratados.

Para o exemplo, vamos criar uma rota que recebe um número em direto na rota e iremos dobrar o valor, no navegador será exibido o número que está na rota e o valor dobrado.

routes.py

```
20 @app.route('/calcular/<int:valor>')
21 def calcular(valor):
22     context = {
23         'valor': valor,
24         'valor_dobro': valor * 2
25     }
26     return render_template('calcular.html', context=context)
27
```

*** Incluso as linhas 20 à 26

calcular.html

```
1 {% extends 'base.html' %}
2
3 {% block content %}
4
5     
6
7     <h1>Rota com Parâmetro</h1>
8
9     <p>Valor Original: {{ context['valor'] }}</p>
10    <p>Dobro: {{ context['valor_dobro'] }}</p>
11
12    <a href="{{ url_for('homepage') }}"><button>Voltar</button></a>
13 {% endblock %}
14
```

index.html

```
16 <a href="{{ url_for('calcular', valor=50) }}"><button>Calcular Valor 50</button></a>
```

*** Incluso a linha 16

Resultado Page



Flask

Bem Vindos ao Primeiro Projeto Flask

Está é a página Raiz do projeto

Usuário: Jonathas

Idade: 34

Mensagem: Dados do back-end!

Calcular Valor 50

Lista de funções Flask no HTML

- extends
- include

Ao clicar no botão “calcular Valor 50” será redirecionado para a rota /calcular/50



Flask

Rota com Parâmetro

Valor Original: 50

Dobro: 100

Voltar

*** Note que na URL está com a rota /calcular/50, se alterarmos o valor de 50 para outro, o valor será alterado na página dinamicamente

127.0.0.1:5000/calcular/50

2. Módulo Intermediário

O que iremos ver nesse Capítulo?

Segurança:

Garantir a segurança do aplicativo é essencial. Isso inclui proteger contra ataques comuns como injeção de SQL, Criptografia de dados sensíveis e Cross-Site Request Forgery (CSRF).

O Flask oferece métodos e extensões para ajudar na segurança, como o Flask-SQLAlchemy para proteger contra injeção de SQL e o Flask-WTF para prevenir CSRF.

Banco de Dados (SQLite):

O uso de banco de dados é essencial para aplicativos mais complexos. O SQLite é uma excelente opção para começar, pois é simples e incorporado ao Flask.

Você pode criar, manipular e consultar dados utilizando o SQLite com a ajuda de extensões como SQLAlchemy ou diretamente com o módulo SQLite3.

Models (Modelos):

Models representam a estrutura dos dados em um banco de dados. No Flask, eles são implementados usando classes Python que interagem com o banco de dados.

Models são úteis para definir tabelas, relações entre os dados e realizar operações de leitura e escrita.

Formulários:

O Flask-WTF (Flask-WTForms) é uma extensão comumente usada para lidar com formulários. Ele simplifica a criação de formulários HTML e a validação de dados enviados pelo usuário.

Os formulários são importantes para coletar informações do usuário e processá-las no servidor.

Controle de Loop:

Vamos aprender implementar uma estrutura de Loop nos templates

Objetivos do Módulo:

- Compreender como integrar um banco de dados ao Flask usando SQLite.
- Criar e manipular modelos para representar dados de maneira estruturada.
- Utilizar formulários para coletar e validar dados dos enviados pelo usuário.
- Garantir a segurança do aplicativo aplicando práticas de segurança recomendadas.

Banco de Dados

Vamos começar a configurar nosso banco de dados, para isso vamos precisar instalar duas novas ferramentas, sendo:

Flask-SQLAlchemy: Será o ORM que iremos trabalhar, esse framework permite criarmos e manipularmos o banco de dados diretamente do Python, através de classes.

Flask-Migrate: Servirá para o controle de alterações no banco de dados.

Para instalarmos essas ferramentas usaremos o pip

```
pip install Flask-SQLAlchemy Flask-Migrate
```

Após instalado, vamos precisar configurar o banco de dados, vamos utilizar o SQLite, porém também iremos aprender a conectar em banco de dados PostgreSQL e MySQL.

Conectando Banco de Dados SQLite

No arquivo `__init__.py` dentro da pasta `app`, vamos realizar algumas configurações, sendo:

- Importação do Flask-SQLAlchemy e Flask-Migrate
- Configuração da URI do banco de dados (String de conexão)
- Configuração do Migrate
- As configurações devem ser antes da importação das rotas

```
1  from flask import Flask
2  from flask_sqlalchemy import SQLAlchemy
3  from flask_migrate import Migrate
4
5  app = Flask(__name__)
6
7
8  app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///database.db'
9  app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
10
11  db = SQLAlchemy(app)
12  migrate = Migrate(app, db)
13
14
15  from app.routes import homepage, dados
16
```


Vamos entender cada linha de código

- Na linha 2 e 3, estamos realizando a importação das ferramentas necessárias para configuração do SQLAlchemy e Flask Migrate
- Na linha 9, estamos configurando a conexão com o banco de dados
- Na linha 10, estamos desativando o rastreo de modificações, em resumo, ao definir `SQLALCHEMY_TRACK_MODIFICATIONS` como `False`, você desabilita o rastreamento automático de modificações de objetos SQLAlchemy, o que pode resultar em melhor desempenho, especialmente em ambientes de produção.
- Na linha 12, estamos instanciando o objeto do banco de dados
- Na linha 13, estamos instanciando o objeto do migrate

Com essas configurações, preparamos nosso projeto para acesso ao banco de dados SQLite

Conectando Banco de Dados PostgreSQL / MySQL

Para conectar um banco de dados PostgreSQL / MySQL, temos que fazer todos os passos acima, a única diferença é que temos que instalar a biblioteca para conexão com o banco de dados e a URI é um pouco diferente

Para o PostgreSQL, temos que instalar a ferramenta `psycopg2-binary`

```
pip install psycopg2-binary
```

```
9 app.config['SQLALCHEMY_DATABASE_URI'] = 'postgresql://seu_usuario:senha@localhost/nome_do_banco'
```

Para o MySQL, temos que instalar a ferramenta `mysqlclient`

```
pip install mysqlclient
```

```
9 app.config['SQLALCHEMY_DATABASE_URI'] = 'mysql://seu_usuario:senha@localhost/nome_do_banco'
```

Models

Em Flask, os "models" se referem à representação das estruturas de dados que são mapeadas em um banco de dados utilizando o SQLAlchemy ou outro ORM (Object-Relational Mapping). Os models são classes Python que definem a estrutura das tabelas em um banco de dados relacional.

Cada model representa uma tabela no banco de dados e os atributos da classe representam colunas nessa tabela.

Na pasta app do nosso projeto, vamos criar um novo arquivo, models.py

```
1 from app import db
2 from datetime import datetime
3
4 class Filmes(db.Model):
5     id = db.Column(db.Integer, primary_key=True)
6     data_criacao = db.Column(db.DateTime, nullable=True, default=datetime.utcnow())
7     titulo = db.Column(db.String, nullable=True)
8     ano = db.Column(db.Integer, nullable=True)
9     resumo = db.Column(db.String, nullable=True)
10
```

Após a criação da classe, precisamos importá-la no arquivo __init__.py da pasta app

```
10
11 db = SQLAlchemy(app)
12 migrate = Migrate(app, db)
13
14
15 from app.routes import homepage, dados
16 from app.models import Filmes
17
```

A importação deve ocorrer após a criação das variáveis db e migrate

O próximo passo é iniciar as migrações, para isso precisamos digitar no terminal os seguintes comandos:

1. Iniciar as configurações de migração (utiliza apenas uma vez no projeto).

```
flask db init
```

2. Verificar as alterações no banco de dados e criar uma migração com base nas alterações nos modelos (apenas os modelos importados no arquivo __init__.py sofrerão as atualizações)

```
flask db migrate -m "mensagem descrevendo a alteração"
```

3. Aplica as alterações pendentes no banco de dados.

```
flask db upgrade
```

Note que após executar os passos acima, será criado uma pasta chamada migrations e uma chamada instance.

Os passos 2 e 3 devem ser repetidos sempre que houver alteração no Models

Formulários

Os "forms" são estruturas que permitem criar e validar formulários de maneira mais conveniente e segura. O Flask, por si só, não possui um módulo de formulários incorporado, mas muitos desenvolvedores usam extensões como o Flask-WTF para lidar com formulários de maneira eficiente.

Abaixo está um resumo dos formulários no Flask:

Flask-WTF: É uma extensão que simplifica a criação e validação de formulários no Flask. Ele facilita a geração de campos de formulário HTML e lida com a validação de dados submetidos.

Funcionamento Básico: Com Flask-WTF, você define classes Python para representar seus formulários. Cada campo de entrada no formulário é representado por um atributo na classe do formulário, e esses atributos são geralmente instâncias de classes de campos (como StringField, IntegerField, etc.).

Validação de Dados: O Flask-WTF fornece funcionalidades para validar automaticamente os dados submetidos pelos usuários. Ele pode verificar se os campos são obrigatórios, se estão no formato correto, entre outras validações.

Proteção contra CSRF: O Flask-WTF ajuda a proteger os formulários contra ataques CSRF (Cross-Site Request Forgery) adicionando automaticamente um campo de token CSRF aos formulários.

Renderização de HTML: Além da validação, o Flask-WTF também ajuda na renderização de formulários HTML. Você pode usar esses objetos de formulário em templates Jinja2 para gerar campos de formulário e mensagens de erro de maneira fácil.

Antes de criarmos nosso primeiro Form, precisamos instalar a ferramenta flask_wtf, no terminal vamos digitar a seguinte linha de comando:

```
pip install flask_wtf
```

Após vamos criar um arquivo chamado forms.py dentro da pasta app do projeto.

```
1 from flask_wtf import FlaskForm
2 from wtforms import StringField, SubmitField, TextAreaField
3 from wtforms.validators import DataRequired, ValidationError
4
5 from app.models import Filmes
6
7 class FilmeForm(FlaskForm):
8     titulo = StringField('Titulo', validators=[DataRequired()])
9     ano = StringField('Ano', validators=[DataRequired()])
10    resumo = TextAreaField('Resumo', validators=[DataRequired()])
11    submit = SubmitField('Salvar')
12
13    def validate_titulo(self, titulo):
14        if Filmes.query.filter_by(titulo=titulo.data).first():
15            raise ValidationError('Este Filme já foi incluído!!!')
16
```

Vamos entender melhor os códigos do arquivo forms.py

- Linha 1, importamos a classe FlaskForm
- Linha 2, importamos os campos que iremos trabalhar
- Linha 3, importamos os validadores que iremos trabalhar
- Linha 5, importamos a classe Filmes de nosso Model
- Linha 7, criamos a classe FilmeForm herdando as configurações da classe FlaskForm
- Linha 8 à 10, criamos nossos campos, definindo um validador de campo requerido
- Linha 11, criamos nosso botão para enviar o formulário
- Linha 13 à 15, criamos uma função que serve para validar se já existe o título digitado, criando um campo único.

Por último precisamos criar um secret_key para a validação do csrf_token, como esse dado é extremamente sensível, não podemos deixá-lo exposto em nosso código, por isso vamos criar um arquivo na raiz do nosso projeto, chamado local_settings.py, esse arquivo não pode ser compartilhado publicamente, por exemplo se você estiver compartilhando seu projeto no github. Certifique-se que esse arquivo seja configurado para não ser realizado upload.

Uma outra forma de trabalharmos com esse dado é utilizando uma variável de ambiente.

Para gerar uma chave aleatória secreta, podemos utilizar a biblioteca secrets, em um arquivo python em branco digite o código:

```
1 import secrets
2
3 print(secrets.token_hex(16))
4
```

Copie o resultado que for exibido no terminal na variável SECRET_KEY no arquivo local_settings.py

```
1
2 SECRET_KEY = '99aaf492fdcdc8dd9b2e9c23301c55aa'
3
```

Importe essa variável no arquivo __init__.py da pasta app e configure a Secret Key do APP

```
1 from flask import Flask
2 from flask_sqlalchemy import SQLAlchemy
3 from flask_migrate import Migrate
4
5 from local_settings import SECRET_KEY
6
7 app = Flask(__name__)
8
9 app.config['SECRET_KEY'] = SECRET_KEY
10
11 app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///database.db'
12 app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
13
14 db = SQLAlchemy(app)
15 migrate = Migrate(app, db)
```

Campos

Segue abaixo os principais campos do wtforms

Campo	Descrição
StringField	Campo de Texto Simples
IntegerField	Campo de Números Inteiros
FloatField	Campo de Números Decimais
BooleanField	Campo de Valores Verdadeiro ou Falso
DateField	Campo de Data
DateTimeField	Campo de Data e Hora
SelectField	Campo de Dropdown para Selecionar Opções
RadioField	Campo de Seleção de Opções
TextAreaField	Campo de Textos Multiplaslinhas
FileField	Campo de Arquivos
PasswordField	Campo de Captura de Senhas
SubmitField	Botão de Envio

Validações

Como vimos no exemplo, podemos utilizar validações prontas do `wtforms.validators` ou criar nossos próprios validators, para criar um validator, basta criar uma função com o seguinte nome: `validade_nomeAtributo`, os parâmetros devem ser sempre o `self` e o atributo.

Para utilizar os validators do `wtforms.validators` deve ser passado como parâmetro do campo criado através do `validators = []`, onde cada item da lista deve ser um validator

Segue abaixo os principais validators do `wtforms.validators`:

Validator	Descrição
DataRequired	Garante que um campo não seja enviado em branco
Length(min=x, max=y)	Define limites para o tamanho de um campo (mínimo e máximo)
Email	Verifica se o campo possui um formato de e-mail válido
EqualTo(fieldname)	Garante que dois campos tenham valores iguais, útil para campos de confirmação de senha
NumberRange(min=x, max=y)	Limita os valores numéricos aceitáveis em um intervalo
Regexp(regex, message=None)	Usa expressões regulares para validar o formato do campo
AnyOf(choices)	Verifica se o valor do campo está entre um conjunto específico de valores
NoneOf(values)	Garante que o valor do campo não esteja em um conjunto específico de valores
URL	Verifica se o campo possui um formato de URL válido
Optional	Torna um campo opcional (não requerido). Se o campo estiver vazio, a validação passa
FileAllowed(upload_set, message=None)	Valida se o arquivo enviado pertence a um conjunto específico de tipos permitidos

Salvando Informações no Banco de Dados

Agora que já preparamos o banco de dados, o Models e o Forms, vamos implementar a rota e o template para salvar as informações no banco de dados.

O primeiro passo, é criar uma rota para salvar os dados no banco de dados, por hora não vamos nos preocupar com a lógica de salvar os dados, apenas vamos nos preocupar em adicionar o formulário no navegador.

No arquivo routes.py, vamos adicionar a seguinte função:

```
30 @app.route('/filmes/novo')
31 def novoFilme():
32     form = FilmeForm()
33     return render_template('filme_novo.html', form=form)
34
```

Agora precisamos criar o arquivo filme_novo.html na pasta templates

```
1 {% extends 'base.html' %}
2
3 {% block content %}
4     <center>
5         <h1>Novo Filme</h1>
6         <hr><br>
7
8         <form method="post">
9             {{form.csrf_token}}
10
11             {{form.titulo.label()}}
12             {{form.titulo}}
13
14             {{form.ano.label()}}
15             {{form.ano}}
16
17             <br><br>
18
19             {{form.resumo.label()}} <br>
20             {{form.resumo(rows=10, cols=50)}}
21
22             <br><br>
23             {{form.submit()}}
24         </form>
25     </center>
26 {% endblock %}
```

Nesse arquivo adicionamos o form com método POST, pois iremos trabalhar com envio de informações para o servidor, sempre que trabalhamos com esse tipo de formulário, a primeira informação que temos que adicionar dentro do FORM é o csrf_token.

Nas linhas 11 à 20 estamos incluindo os campos configurado no nosso form, podendo ser passado como parâmetros do campo, os parâmetros do HTML.

*** Lembrando que não estamos preocupados nesse momento com o visual (Front-End) da aplicação e sim com suas funcionalidades, caso desejem deixar o HTML mais agradável, fiquem à vontade para fazer as alterações que desejarem

Também vamos adicionar o link para essa nova rota no arquivo index.html

```
<a href="{{ url_for('novoFilme')}}"><button>Novo Filme</button></a>
```

Com isso teremos o seguinte resultado

Novo Filme

Resumo

O formulário ainda não está funcionando, pois devemos configurar a rota para receber solicitações do tipo POST e o que será feito com o formulário.

O próximo passo é configurar o arquivo form, incluindo a função save, para salvar as informações no banco de dados.

Para isso precisamos importar a variável db do nosso app e criar a nova função do formulário

```
5 from app import db
```

```
8 class FilmeForm(FlaskForm):
9     titulo = StringField('Titulo', validators=[DataRequired()])
10    ano = StringField('Ano', validators=[DataRequired()])
11    resumo = TextAreaField('Resumo', validators=[DataRequired()])
12    submit = SubmitField('Salvar')
13
14    def validate_titulo(self, titulo):
15        if Filmes.query.filter_by(titulo=titulo.data).first():
16            raise ValidationError('Este Filme já foi incluso!!!')
17
18    def save(self):
19        filme = Filmes(
20            titulo = self.titulo.data,
21            ano = self.ano.data,
22            resumo = self.resumo.data
23        )
24
25        db.session.add(filme)
26        db.session.commit()
```

O próximo passo é configurar a rota, devemos incluir no app.route o parâmetro methods=['GET', 'POST'] e incluir a função de validação do post.

Também vamos importar o método redirect, para se tudo ocorrer certo, após salvar os dados a página ser redirecionada

```
1 from flask import render_template, url_for, jsonify, redirect
```

```

29
30 @app.route('/filmes/novo', methods=['GET', 'POST'])
31 def novoFilme():
32     form = FilmeForm()
33     if form.validate_on_submit():
34         form.save()
35         return redirect(url_for('homepage'))
36
37     return render_template('filme_novo.html', form=form)
38

```

Vamos realizar um teste, preencheremos um filme, ao clicar em salvar, se tudo ocorrer bem, vamos ser redirecionados para a página inicial.

Novo Filme

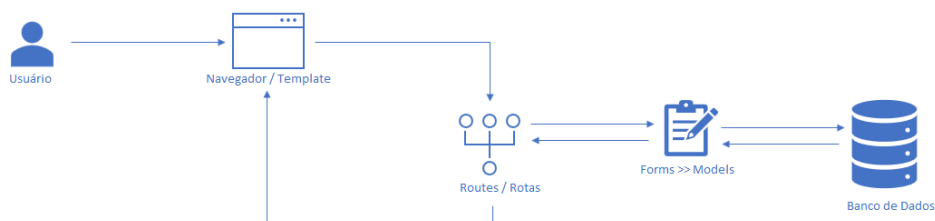
Título Ano

Resumo

Após um feitiço dar errado, Peter Parker busca a ajuda do Doutor Estranho para corrigir a exposição de sua identidade secreta. Isso desencadeia a chegada de vilões de universos paralelos, levando Peter a buscar versões anteriores do Homem-Aranha para lidar com a crise. Ele enfrenta conflitos internos sobre o preço a pagar para restaurar a normalidade.

Em resumo, sempre que precisamos incluir dados no banco de dados, precisamos configurar:

- um models, que será nossa tabela no banco de dados,
- um formulário, que será servirá para garantir a segurança, validações e comunicação com o models
- uma rota, será responsável por renderizar o template e criar a comunicação entre o usuário e o models
- um template, que será o que será exibido no navegador do usuário.



Trabalhando um pouco com Front-End

Antes de prosseguir para a próxima etapa, vamos melhorar um pouco nosso template, não iremos nos aprofundar no tema, por isso recomendamos um estudo aprofundado no tema.

Para facilitar, não vamos construir os estilos do zero, vamos utilizar o framework front-end bootstrap, para adicionar em nosso projeto precisamos alterar nosso arquivo base.html

Link do framework: <https://getbootstrap.com/docs/5.0/getting-started/introduction/>

base.html

```
1 <!DOCTYPE html>
2 <html lang="pt-br">
3 <head>
4   <meta charset="UTF-8">
5   <title>{% block title %}Primeiro Projeto Flask{% endblock %}</title>
6   <link rel="stylesheet" href="{{ url_for('static', filename='css/style.css') }}">
7   <link
8     href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/css/bootstrap.min.css"
9     rel="stylesheet"
10    integrity="sha384-EVSTQN3/azprG1Anm3QDgpJLIm9Nao0Yz1ztcQTWfSdp3yD65VohhpuuCOMLASjC"
11    crossorigin="anonymous"
12  >
13 </head>
14 <body class="text-blue1">
15   <div class="container">
16     {% block content %} {% endblock %}
17   </div>
18 </body>
19 </html>
20
```

Também iremos alterar o arquivo index.html, nesse arquivo vamos retirar os links e criar um novo arquivo nav.html, que será responsável por manter os links, vamos inclusive incluir o nav no arquivo base.html, assim surtindo efeito em todas as páginas de nosso projeto

nav.html

```
1 <nav class="row justify-content-center text-center mt-4 mb-4">
2
3   <div class="col-12">
4     
5   </div>
6
7   <div class="col-12 mt-3 text-start">
8     <a href="{{ url_for('novoFilme') }}" class="btn btn-primary col-3">Novo Filme</a>
9     <a href="{{ url_for('calcular', valor=50) }}" class="btn btn-dark col-3">Calcular Valor 50</a>
10  </div>
11 </nav>
12
```

base.html

```
14 <body class="text-blue1">
15   <div class="container">
16     {% include 'nav.html' %}
17     {% block content %} {% endblock %}
18   </div>
19 </body>
```

index.html

```
1  {% extends 'base.html' %}
2
3  {% block content %}
4
5      <h1>Bem Vindos ao Primeiro Projeto Flask</h1>
6
7      <p>Está é a página Raiz do projeto</p>
8
9      <p>Usuário: {{ context['usuario'] }}</p>
10     <p>Idade: {{ context['idade'] }}</p>
11
12     <p>Mensagem: <span id="msg_back"></span></p>
13
14     {% include 'lista.html' %}
15
16     <script>
17         const msg_back = document.getElementById('msg_back')
18         fetch('{{ url_for("dados") }}')
19             .then(response => response.json())
20             .then(data => {
21                 msg_back.innerHTML = data.mensagem;
22             });
23     </script>
24
25 {% endblock %}
26
```

lista.html

```
1  <div class="border shadow mt-3 p-4">
2      <p>Lista de funções Flask no HTML</p>
3      <ul>
4          <li>extends</li>
5          <li>include</li>
6      </ul>
7  </div>
```

filme_novo.html

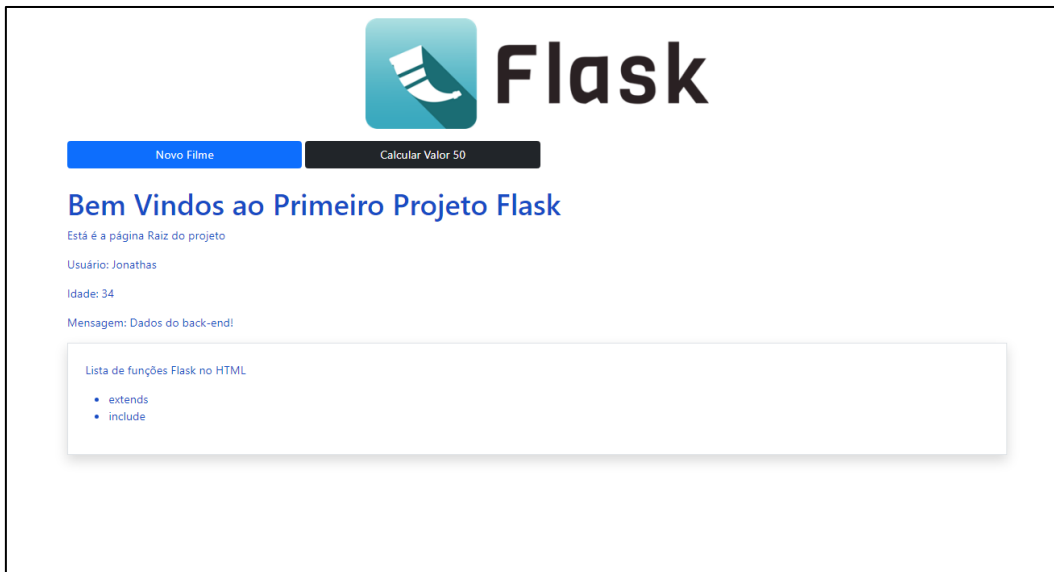
```
1  {% extends 'base.html' %}
2
3  {% block content %}
4      <h1>Novo Filme</h1>
5
6      <form method="post" class="form mt-2 p-4 border shadow rounded">
7          {{ form.csrf_token }}
8
9          {{ form.titulo.label() }}
10         {{ form.titulo(class='form-control') }}
11
12         {{ form.ano.label(class='mt-3') }}
13         {{ form.ano(class='form-control') }}
14
15         {{ form.resumo.label(class='mt-3') }} <br>
16         {{ form.resumo(class='form-control', rows=10) }}
17
18         <br><br>
19         <div class="row justify-content-end">
20             <div class="col-4">
21                 {{ form.submit(class='btn btn-primary col-12') }}
22             </div>
23         </div>
24     </form>
25
26 {% endblock %}
27
```

Em resumo, utilizamos o framework bootstrap para estilizar nosso projeto, adicionando classes para itens, assim deixando mais agradável.

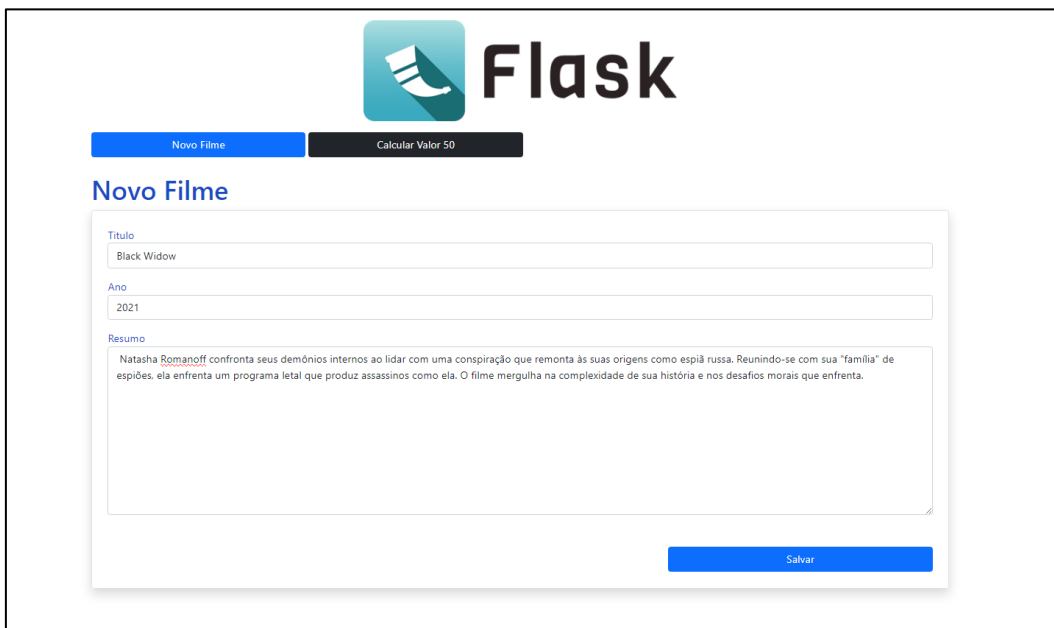
calcular.html

```
1 {% extends 'base.html' %}
2
3 {% block content %}
4
5     <h1>Rota com Parâmetro</h1>
6
7     <p>Valor Original: {{ context['valor'] }}</p>
8     <p>Dobro: {{ context['valor_dobro'] }}</p>
9
10    <a href="{{ url_for('homepage') }}" class="btn btn-primary col-3">Voltar</a>
11 {% endblock %}
12
```

Resultado:



The screenshot shows the Flask homepage. At the top, there's a blue button labeled "Novo Filme" and a black button labeled "Calcular Valor 50". Below these, the heading "Bem Vindos ao Primeiro Projeto Flask" is displayed. Underneath, it says "Está é a página Raiz do projeto". The user information "Usuário: Jonathas" and "Idade: 34" is shown. A message "Mensagem: Dados do back-end!" is present. A box titled "Lista de funções Flask no HTML" contains a bulleted list: "• extends" and "• include".



The screenshot shows the "Novo Filme" form. It has a blue button labeled "Novo Filme" and a black button labeled "Calcular Valor 50". The form fields are: "Titulo" with the value "Black Widow", "Ano" with the value "2021", and "Resumo" with a text area containing a paragraph about Natasha Romanoff. A blue "Salvar" button is at the bottom right.

Recuperando Informações no Banco de Dados

Agora que aprendemos a salvar informações em nosso banco de dados, precisamos aprender em como recuperar e exibir para o usuário.

Vamos criar duas rotas para exibir os dados salvos, uma rota que irá listar todos os dados da tabela e outra rota que irá exibir o detalhe, vamos iniciar com a rota para listar todos os dados.

No arquivo routes.py, vamos incluir a rota listaFilmes

```
41 @app.route('/filme/lista')
42 def listaFilme():
43     filmes = Filmes.query.order_by(Filmes.titulo.desc()).all()
44     return render_template('filme_lista.html', objects=filmes)
45
```

Na pasta templates, vamos criar o arquivo filme_lista.html

```
1  {% extends 'base.html' %}
2
3  {% block content %}
4      <h1>Lista de Filme</h1>
5
6      <div class="row mt-4 border rounded shadow">
7
8          <table class="table">
9              <thead class="table-primary">
10                 <tr>
11                     <th>Titulo</th>
12                     <th>Ano</th>
13                     <th></th>
14                 </tr>
15             </thead>
16             <tbody>
17                 {% for o in objects %}
18                 <tr>
19                     <td>{{o.titulo}}</td>
20                     <td>{{o.ano}}</td>
21                     <td></td>
22                 </tr>
23                 {% endfor %}
24             </tbody>
25         </table>
26
27     </div>
28
29 {% endblock %}
```

Na linha 17 à 23, utilizamos o operador de loop com o Flask no HTML, onde estamos interagindo todos os itens da lista objects

No Arquivo nav.html, vamos incluir duas rotas, uma para voltar para o início e uma para lista de filmes

```
8 <a href="{{ url_for('homepage')}}" class="btn btn-success col-3">Home</a>
9 <a href="{{ url_for('listaFilme')}}" class="btn btn-info col-3">Lista Filme</a>
```

Ao clicarmos na rota da lista de filmes, teremos todos os filmes cadastrados



Agora iremos criar a rota para exibir cada filme, para isso vamos criar uma rota detailFilme no arquivo routes.py

```
47 @app.route('/filme/<int:id>')
48 def detailFilme(id):
49     filme = Filmes.query.get(id)
50     return render_template('filme_detail.html', object=filme)
```

Vamos criar o arquivo filme_detail.html na pasta de templates

```
1 {% extends 'base.html' %}
2
3 {% block content %}
4     <h1>Filme: {{object.titulo}}</h1>
5
6     <div class="row mt-4 border rounded shadow p-4">
7
8         <div class="col-12">
9             <strong>Ano:</strong> {{object.ano}}
10        </div>
11
12        <div class="col-12 mt-3">
13            <strong>Resumo</strong>
14            <p>
15                {% for r in object.resumo.split('\n') %}
16                    {{r}} <br>
17                {% endfor %}
18            </p>
19        </div>
20    </div>
21
22 {% endblock %}
23
24
```

Na linha 15 a 17, iniciamos um loop para criar uma linha para cada quebra de linha do resumo.

Vamos editar a tabela do arquivo filme_lista.html

```
<tbody>
  {% for o in objects %}
  <tr>
    <td>{{o.titulo}}</td>
    <td>{{o.ano}}</td>
    <td class="text-end"><a href="{{url_for('detailFilme', id=o.id)}}" class="btn btn-secondary">Detalhes</a></td>
  </tr>
  {% endfor %}
</tbody>
```

Teremos o seguinte resultado



Flask

[Home](#) [Lista Filme](#) [Novo Filme](#) [Calcular Valor 50](#)

Bem Vindos ao Primeiro Projeto Flask

Está é a página Raiz do projeto


Usuário: Jonathas

Idade: 34

Mensagem: Dados do back-end!

Lista de funções Flask no HTML

- extends
- include



Flask

[Home](#) [Lista Filme](#) [Novo Filme](#) [Calcular Valor 50](#)

Lista de Filme

Titulo	Ano	
Spider-Man: No Way Home	2021	Detalhes
Shang-Chi and the Legend of the Ten Rings	2021	Detalhes
Black Widow	2021	Detalhes



Flask

[Home](#) [Lista Filme](#) [Novo Filme](#) [Calcular Valor 50](#)

Filme: Shang-Chi and the Legend of the Ten Rings

Ano: 2021

Resumo
Shang-Chi, treinado desde jovem por seu pai, o possuidor dos Dez Anéis, foge para viver uma vida normal. No entanto, é confrontado pelo passado ao ser forçado a confrontar seu pai e impedir uma entidade antiga de ressurgir. A jornada o leva a explorar sua identidade, herança e poderes únicos.

Módulo Avançado

Objetivo do Módulo

- Implementar controle de login para autenticação.
- Implementar controle de acesso para usuários autenticados.
- Implementar controle de logout
- Estrutura de decisão if no HTML
- Vincular registros a outras tabelas (FK)
- Tipo de dados de arquivos
- Trabalhar com Variáveis de Ambiente
- GitHub
- Deploy

Controle de Login

Vamos implementar o controle de login em nossa aplicação, assim podemos definir algumas páginas somente usuários logados podem acessar, para isso vamos primeiro instalar a ferramenta flask-login, flask-bcrypt e email_validator

```
pip install flask-login flask-bcrypt email_validator
```

Depois precisamos alterar algumas configurações no arquivo `__init__.py` do app

Onde vamos importar a classe `LoginManager` do `flask_login` e `Bcrypt` do `flask-bcrypt`, também vamos criar a variável `login_manager` e a variável `bcrypt`

Com a variável `login_manager`, vamos definir qual a rota para login

`__init__.py`

```
1 from flask import Flask
2 from flask_sqlalchemy import SQLAlchemy
3 from flask_migrate import Migrate
4 from flask_login import LoginManager
5 from flask_bcrypt import Bcrypt
6
7 from local_settings import SECRET_KEY
8
9 app = Flask(__name__)
10
11 app.config['SECRET_KEY'] = SECRET_KEY
12
13 app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///database.db'
14 app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
15
16 db = SQLAlchemy(app)
17 migrate = Migrate(app, db)
18 login_manager = LoginManager(app)
19 login_manager.login_view = 'login'
20 bcrypt = Bcrypt(app)
21
```

Após configurado o app, precisamos configurar o model de usuário e definir qual model pode realizar o login, no arquivo `models.py` vamos importar a variável `login_manager` que criamos e também vamos importar a classe `UserMixin` do `flask_login`

Vamos criar a classe `User`, definindo o `UserMixin` nessa classe, isso informa que essa classe será utilizada para o login, também precisamos implementar uma função `load_user` para definir o carregamento do usuário.

`models.py`

```
1  from app import db, login_manager
2  from datetime import datetime
3  from flask_login import UserMixin
4
5
6  @login_manager.user_loader
7  def load_user(user_id):
8      return User.query.get(int(user_id))
9
10
11  class User(db.Model, UserMixin):
12      id = db.Column(db.Integer, primary_key=True)
13      email = db.Column(db.String(100), unique=True)
14      username = db.Column(db.String(100))
15      nome = db.Column(db.String(100))
16      sobrenome = db.Column(db.String(100))
17      password = db.Column(db.String(100))
18
```

Precisamos fazer o migrate agora, para isso vamos digitar no shell o comando de migrate

```
flask db migrate
```

```
flask db upgrade
```

Agora vamos criar três rotas, uma para cadastro de novos usuários, uma para o login e outra para o logout.

Cadastro de Usuário

Vamos iniciar com o cadastro de usuário, primeiro vamos definir a classe no forms.py

```
15 class UserForm(FlaskForm):
16     email = StringField('E-Mail', validators=[DataRequired(), Email()])
17     username = StringField('Username', validators=[DataRequired()])
18     nome = StringField('Nome', validators=[DataRequired()])
19     sobrenome = StringField('Sobrenome', validators=[DataRequired()])
20     senha = PasswordField('Senha', validators=[DataRequired()])
21     senha_confirmacao = PasswordField('Confirme a Senha', validators=[DataRequired(), EqualTo('senha')])
22     submit = SubmitField('Salvar')
23
24     def validate_username(self, username):
25         if User.query.filter_by(username=username.data).first():
26             raise ValidationError('Username já utilizado, favor utilize outro!!!')
27
28     def save(self):
29         senha = bcrypt.generate_password_hash(self.senha.data)
30         user = User(
31             email = self.email.data,
32             username = self.username.data,
33             nome = self.nome.data,
34             sobrenome = self.sobrenome.data,
35             password = senha,
36         )
37         db.session.add(user)
38         db.session.commit()
39
40     def get_user(self):
41         self.save()
42         return User.query.filter_by(username=self.username.data).first()
```

Adicionamos a função para salvar o usuário no banco de dados e uma função para retornar o usuário cadastrado

Para salvar a senha do usuário criptografamos usando o bcrypt para segurança

Agora vamos criar a rota

```
2 from flask_login import login_required, login_user, logout_user
```

```
18 @app.route('/novo_usuario', methods=['GET', 'POST'])
19 def newUser():
20     form = UserForm()
21     if form.validate_on_submit():
22         user = form.get_user()
23         login_user(user, remember=True)
24         return redirect(url_for('homepage'))
25     return render_template('novo_usuario.html', form=form)
```

No arquivo nav.html vamos incluir a rota para cadastro

```
<a href="{{ url_for('newUser')}}" class="btn btn-warning col-2">Cadastrar</a>
```

Agora vamos criar o novo_usuario.html

```
1  {% extends 'base.html' %}
2
3  {% block content %}
4      <h1>Cadastro de Usuário</h1>
5
6      <form method="post" class="form mt-2 p-4 border shadow rounded">
7          {{form.csrf_token}}
8
9          {{form.username.label()}}
10         {{form.username(class='form-control')}}
11
12         {{form.nome.label()}}
13         {{form.nome(class='form-control')}}
14
15         {{form.sobrenome.label()}}
16         {{form.sobrenome(class='form-control')}}
17
18         {{form.email.label()}}
19         {{form.email(class='form-control')}}
20
21         {{form.senha.label()}}
22         {{form.senha(class='form-control')}}
23
24         {{form.senha_confirmacao.label()}}
25         {{form.senha_confirmacao(class='form-control')}}
26
27         <br><br>
28         <div class="row justify-content-end">
29             <div class="col-4">
30                 {{form.submit(class='btn btn-primary col-12')}}
31             </div>
32         </div>
33     </form>
34 {% endblock %}
35
36
```



The screenshot displays the Flask web application interface. At the top, there is a navigation bar with five buttons: "Home" (green), "Lista Filme" (cyan), "Novo Filme" (blue), "Calcular Valor 50" (black), and "Cadastrar" (yellow). Below the navigation bar, the title "Cadastro de Usuário" is shown in blue. The main content area features a registration form with the following fields: "Username", "Nome", "Sobrenome", "E-Mail", "Senha", and "Confirme a Senha". Each field has a corresponding label above it. A blue "Salvar" button is located at the bottom right of the form.

Criando rota para Login

Nosso próximo passo, é definir a rota de login, vamos criar o form de login primeiro

```
9 class LoginForm(FlaskForm):
10     username = StringField('Username', validators=[DataRequired()])
11     senha = PasswordField('Senha', validators=[DataRequired()])
12     submit = SubmitField('Login')
13
14     def login(self):
15         user = User.query.filter_by(username=self.username.data).first()
16         if user:
17             if bcrypt.check_password_hash(user.password, self.senha.data):
18                 return user
19             else:
20                 raise Exception('Senha Incorreta')
21         else:
22             raise Exception('Usuário não encontrado')
```

Agora vamos definir a rota

```
28 @app.route('/login')
29 def login():
30     form = LoginForm()
31     if form.validate_on_submit():
32         user = form.login()
33         login_user(user, remember=True)
34         return redirect(url_for('homepage'))
35     return render_template('login.html')
36
```

Vamos criar o login.html

```
1 <!DOCTYPE html>
2 <html lang="pt-br">
3 <head>
4     <meta charset="UTF-8">
5     <title>{% block title %}Primeiro Projeto Flask{% endblock %}</title>
6     <link
7         href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/css/bootstrap.min.css"
8         rel="stylesheet"
9         integrity="sha384-EVSTQN3/azprG1Anm3QDgpJLIm9Nao0Yz1ztcQTWfspd3yD65VohhpuuCOmLASjC"
10        crossorigin="anonymous"
11    >
12    <link rel="stylesheet" href="{{ url_for('static', filename='css/login.css') }}">
13 </head>
14 <body class="text-blue1">
15
16     <body class="text-center">
17         <form class="form-signin" method="POST">
18             {{form.csrf_token}}
19
20             
21             <h1 class="h3 mb-3 font-weight-normal">Faça login</h1>
22
23             {{form.username.label()}}
24             {{form.username(class='form-control', autofocus=True)}}
25
26             {{form.senha.label()}}
27             {{form.senha(class='form-control', autofocus=True)}}
28
29         </div>
30         {{form.submit(class="btn btn-lg btn-primary btn-block col-12")}}
31         <p class="mt-5 mb-3 text-muted">&copy; 2023</p>
32     </form>
33 </body>
34 </html>
35
```

Vamos criar os CSS para essa pagina

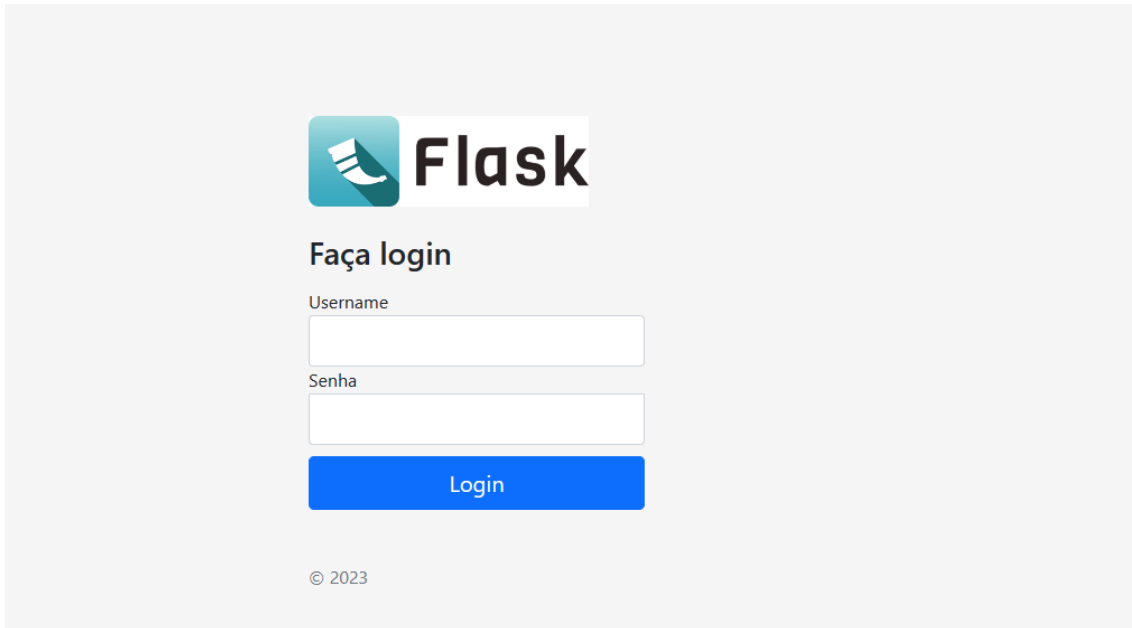
login.css


```
1  html, body { height: 100%; }
2
3  body {
4      display: -ms-flexbox;
5      display: flex;
6      -ms-flex-align: center;
7      align-items: center;
8      padding-top: 40px;
9      padding-bottom: 40px;
10     background-color: #f5f5f5;
11 }
12 .form-signin {
13     width: 100%;
14     max-width: 330px;
15     padding: 15px;
16     margin: auto;
17 }
18 .form-signin .checkbox { font-weight: 400; }
19
20 .form-signin .form-control {
21     position: relative;
22     box-sizing: border-box;
23     height: auto;
24     padding: 10px;
25     font-size: 16px;
26 }
27
28 .form-signin .form-control:focus { z-index: 2; }
29
30 .form-signin input[type="email"] {
31     margin-bottom: -1px;
32     border-bottom-right-radius: 0;
33     border-bottom-left-radius: 0;
34 }
35 .form-signin input[type="password"] {
36     margin-bottom: 10px;
37     border-top-left-radius: 0;
38     border-top-right-radius: 0;
39 }
```

E por fim incluir a rota de login no arquivo nav.html

```
<a href="{{ url_for('login')}}" class="btn btn-secondary col-2">Login</a>
```

Também vamos remover a rota de calculo que usamos para aprender usar parâmetros nas rotas

A screenshot of a Flask web application's login page. At the top, there is a Flask logo and the word "Flask" in a large, bold, black font. Below this, the text "Faça login" (Log in) is displayed. There are two input fields: "Username" and "Senha" (Password). Below the "Senha" field is a blue button labeled "Login". At the bottom of the form, there is a small copyright notice: "© 2023".

 **Flask**

Faça login

Username

Senha

Login

© 2023

Criando Rota para Logout

Por último vamos implementar a rota de logout

```
38 @app.route('/logout')
39 def logout():
40     logout_user()
41     return redirect(url_for('homepage'))
42
```

Vamos adicionar a rota de logout no arquivo nav.html

```
<a href="{{ url_for('logout')}}" class="btn btn-danger col-2">Sair</a>
```

Estrutura de Decisão if no HTML com Flask

Uma outra estrutura muito importante é a tomada de decisão, podemos fazer isso no HTML utilizando o flask, usando o seguinte bloco

```
{% if CONDIÇÃO %}
```

Bloco se verdadeiro

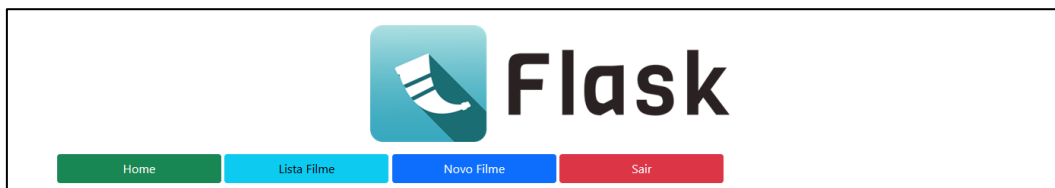
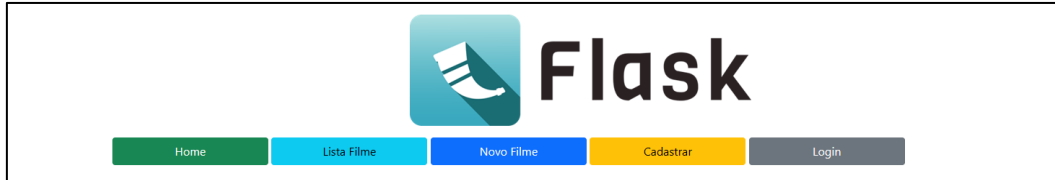
```
{% else %}
```

Bloco se falso

```
{% endif %}
```

Vamos implementar essa estrutura no arquivo nav, o que queremos fazer é, se o usuário não estiver logado, vamos deixar visível a opção para cadastro e login, se estiver logado essas opções não estarão disponíveis, mas ficará disponível a opção sair

```
11
12     {% if current_user.is_authenticated %}
13         <a href="{{ url_for('logout')}}" class="btn btn-danger col-2">Sair</a>
14     {% else %}
15         <a href="{{ url_for('newUser')}}" class="btn btn-warning col-2">Cadastrar</a>
16         <a href="{{ url_for('login')}}" class="btn btn-secondary col-2">Login</a>
17     {% endif %}
18
```



Permissão de Acesso para Usuários Logados

No flask também existe um decorator que nos permite controlar o acesso a determinadas áreas de nosso aplicativo para usuários que estejam logados, vamos incluir essa funcionalidade na rota de cadastro de filmes, ou seja, apenas usuários que estejam logados poderão cadastrar novos filmes.

Para isso no arquivo de rotas, devemos importar a função `login_required` da ferramenta `flask_login`

```
2 from flask_login import login_required
```

Na rota que queremos adicionar essa funcionalidade devemos incluir após o `app.route`

```
59 @app.route('/filmes/novo', methods=['GET', 'POST'])
60 @login_required
61 def novoFilme():
62     form = FilmeForm()
63     if form.validate_on_submit():
64         form.save()
65         return redirect(url_for('homepage'))
66
67     return render_template('filme_novo.html', form=form)
```

Agora, um usuário que não esteja logado no sistema tentar entrar na página de cadastro, será redirecionado para tela de login, assim enquanto não logar no sistema, não poderá acessar a página com essa funcionalidade.

Vincular registros a outras tabelas (FK)

Agora vamos aprender em como criar relações entre tabelas, para isso vamos incluir uma nova funcionalidade em nosso projeto, que será adicionar comentários nos filmes.

No models vamos adicionar a seguinte função

```
30 class FilmeComentario(db.Model):
31     id = db.Column(db.Integer, primary_key=True)
32     data_criacao = db.Column(db.DateTime, nullable=True, default=datetime.utcnow())
33     comentario = db.Column(db.String, nullable=True)
34     id_filme = db.Column(db.Integer, db.ForeignKey('filmes.id'), nullable=True)
35     id_user = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=True)
36
```

Adicionamos o tipo `db.ForeignKey`, que serve para informar que esse campo é um FK, vinculando essa tabela com a tabela Filmes, no parâmetro passamos o nome da tabela todo em minúsculo seguido do campo que estamos vinculando.

Também vamos incluir o campo comentários no Model Filmes e Users

```
20 class Filmes(db.Model):
21     id = db.Column(db.Integer, primary_key=True)
22     data_criacao = db.Column(db.DateTime, nullable=True, default=datetime.utcnow())
23     titulo = db.Column(db.String, nullable=True)
24     ano = db.Column(db.Integer, nullable=True)
25     resumo = db.Column(db.String, nullable=True)
26     comentarios = db.relationship("FilmeComentario", backref='filme', lazy=True)
27
```

```
11 class User(db.Model, UserMixin):
12     id = db.Column(db.Integer, primary_key=True)
13     email = db.Column(db.String(100), unique=True)
14     username = db.Column(db.String(100))
15     nome = db.Column(db.String(100))
16     sobrenome = db.Column(db.String(100))
17     password = db.Column(db.String(100))
18     comentarios_filmes = db.relationship("FilmeComentario", backref='user', lazy=True)
```

O campo “comentarios” serve para informar que temos uma relação com a tabela “FilmeComentario”

Lembrando que sempre que realizarmos qualquer alteração no models devemos realizar os procedimentos de migração

```
flask db migrate
```

```
flask db upgrade
```

Agora que já preparamos nosso banco de dados, precisamos criar o formulário para criar o comentário, para isso vamos adicionar a seguinte função no arquivo forms

```
76 class FilmeComentarioForm(FlaskForm):
77     comentario = TextAreaField('Comentário', validators=[DataRequired()])
78     submit = SubmitField('Salvar')
79
80     def save(self, id_filme, id_user):
81         comentario = FilmeComentario(
82             comentario = self.comentario.data,
83             id_filme = id_filme,
84             id_user = id_user
85         )
86
87         db.session.add(comentario)
88         db.session.commit()
89
```

Nesse form, o único campo que será editado será o campo comentário, para a função save, informamos qual o id do filme e id do usuário, assim criando os vínculos necessários

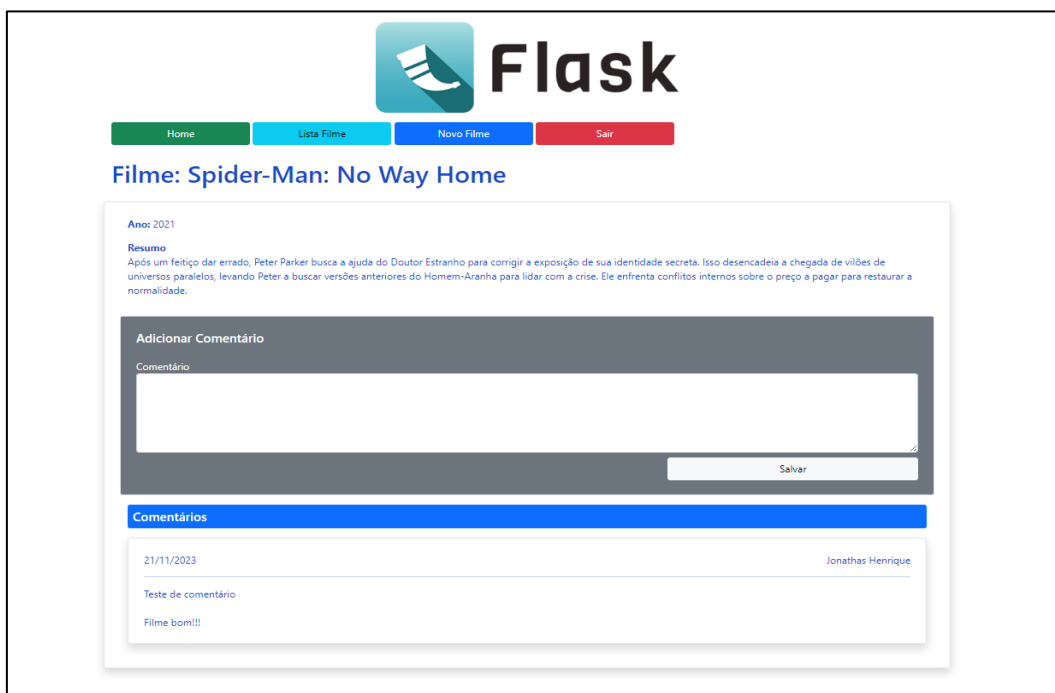
Agora precisamos criar a rota para adicionar o comentário, nesse caso, não vamos criar uma rota específica, mas vamos editar a rota filme_detail, incluindo o formulário para esse campo.

Como precisamos passar o id do usuário para salvar o comentário, precisamos verificar se o usuário está logado para adicionar o comentário.

```
77 @app.route('/filme/<int:id>', methods=['GET', 'POST'])
78 def detailFilme(id):
79     filme = Filmes.query.get(id)
80     form = FilmeComentarioForm()
81     if form.validate_on_submit() and current_user.is_authenticated:
82         form.save(filme.id, current_user.id)
83
84     return render_template('filme_detail.html', object=filme, form=form)
85
```

Também precisamos alterar o arquivo film_detail.html

```
20
21 {% if current_user.is_authenticated %}
22 <form method="post" class="form col-12 mt-3 border bg-secondary text-white p-4 rounded">
23     <h5>Adicionar Comentário</h5>
24     {{form.csrf_token}}
25
26     {{form.comentario.label(class='mt-3')}} <br>
27     {{form.comentario(class='form-control', rows=5)}}
28
29     <div class="row justify-content-end mt-2">
30         <div class="col-4">
31             {{form.submit(class='btn btn-light col-12')}}
32         </div>
33     </div>
34 </form>
35 {% endif %}
36
37 {% if object.comentarios %}
38 <div class="col-12 mt-3">
39     <h5 class="bg-primary text-white p-2 rounded">Comentários</h5>
40     <p>
41         {% for c in object.comentarios %}
42         <div class="col-12 border rounded mt-2 shadow p-4">
43             <div class="row justify-content-between">
44                 <div class="col-4">{{c.data_criacao.strftime('%d/%m/%Y')}}</div>
45                 <div class="col-4 text-end">{{c.user.nome.title()}} {{c.user.sobrenome.title()}}</div>
46             </div>
47             <hr>
48             {% for r in c.comentario.split('\n') %}
49                 {{r}} <br>
50             {% endfor %}
51         </div>
52         {% endfor %}
53     </p>
54 </div>
55 {% endif %}
56
```



The screenshot shows the Flask application interface. At the top, there's a navigation bar with buttons for Home, Lista Filme, Novo Filme, and Sair. Below the navigation bar, the title 'Filme: Spider-Man: No Way Home' is displayed. The main content area shows the movie details for the year 2021, including a synopsis. Below the synopsis, there's a section titled 'Adicionar Comentário' with a text input field and a 'Salvar' button. Underneath, there's a section titled 'Comentários' which lists the comments. The first comment is from 'Jonathas Henrique' dated '21/11/2023', with the text 'Teste de comentário' and 'Filme bom!!!'.

Trabalhando com Tipo de dados de arquivos

Para trabalharmos com arquivos no Flask, devemos primeiro alterar algumas configurações de nosso app, vamos editar o arquivo `__init__.py` da pasta app

```
16 app.config['UPLOAD_FOLDER'] = r'static/data'
```

Também vamos editar a função Filmes do arquivo `models.py`

```
21 class Filmes(db.Model):
22     id = db.Column(db.Integer, primary_key=True)
23     data_criacao = db.Column(db.DateTime, nullable=True, default=datetime.utcnow())
24     titulo = db.Column(db.String, nullable=True)
25     ano = db.Column(db.Integer, nullable=True)
26     resumo = db.Column(db.String, nullable=True)
27     comentarios = db.relationship("FilmeComentario", backref='filme', lazy=True)
28     imagem = db.Column(db.String, default='default.png')
29
```

Incluimos o campo imagem, esse campo é do tipo texto, pois o que vamos gravar no banco de dados não é a imagem em si, e sim apenas o seu nome, como default, vamos deixar o nome da imagem padrão, que iremos incluir no próximo passo.

Lembrando que sempre que realizarmos qualquer alteração no models devemos realizar os procedimentos de migração

```
flask db migrate
```

```
flask db upgrade
```

Agora vamos preparar as pastas e o arquivo padrão, dentro da pasta static vamos criar a pasta data, dentro desta nova pasta vamos criar a pasta filmes e por fim dentro da pasta filmes vamos incluir uma imagem com o nome 'default.png' que servirá para deixar quando não ter uma imagem definida.

Nosso próximo passo agora é alterar o arquivo `forms.py`

Para garantir que o arquivo contenha um nome seguro, vamos importar a seguinte ferramenta:

```
4 from werkzeug.utils import secure_filename
```

Também importaremos a biblioteca os para trabalharmos com os arquivos e pastas

```
4 import os
```

E por fim importaremos a variável do nosso app para trabalharmos com a configuração da pasta que definimos anteriormente

```
7 from app import db, bcrypt, app
```

Agora iremos alterar a classe FilmeForm

```
57 class FilmeForm(FlaskForm):
58     titulo = StringField('Titulo', validators=[DataRequired()])
59     ano = StringField('Ano', validators=[DataRequired()])
60     resumo = TextAreaField('Resumo', validators=[DataRequired()])
61     imagem = FileField('Imagem', validators=[DataRequired()])
62     submit = SubmitField('Salvar')
63
64     def validate_titulo(self, titulo):
65         if Filmes.query.filter_by(titulo=titulo.data).first():
66             raise ValidationError('Este Filme já foi incluído!!!')
67
68     def save(self):
69         arquivo = self.imagem.data
70         nome_seguro = secure_filename(arquivo.filename)
71         caminho = os.path.join(
72             os.path.abspath(os.path.dirname(__file__)), # Recupera o caminho absoluto onde o arquivo routes.py está armazenado
73             app.config['UPLOAD_FOLDER'], # Define a pasta de Upload que criamos
74             'filmes', # define a pasta de Filmes que criamos para organizar nossos arquivos
75             nome_seguro # define o nome seguro para o caminho
76         )
77         filme = Filmes(
78             titulo = self.titulo.data,
79             ano = self.ano.data,
80             resumo = self.resumo.data,
81             imagem=nome_seguro
82         )
83         arquivo.save(caminho)
84         db.session.add(filme)
85         db.session.commit()
```

Foi incluído o campo imagem, do tipo FileField e também alterado a função save da classe, agora ao criar um filme precisamos definir onde será salvo a imagem e qual nome iremos salvar.

Para o nome, estamos usando a função secure_filename para modificar o nome do arquivo para um nome seguro para o servidor.

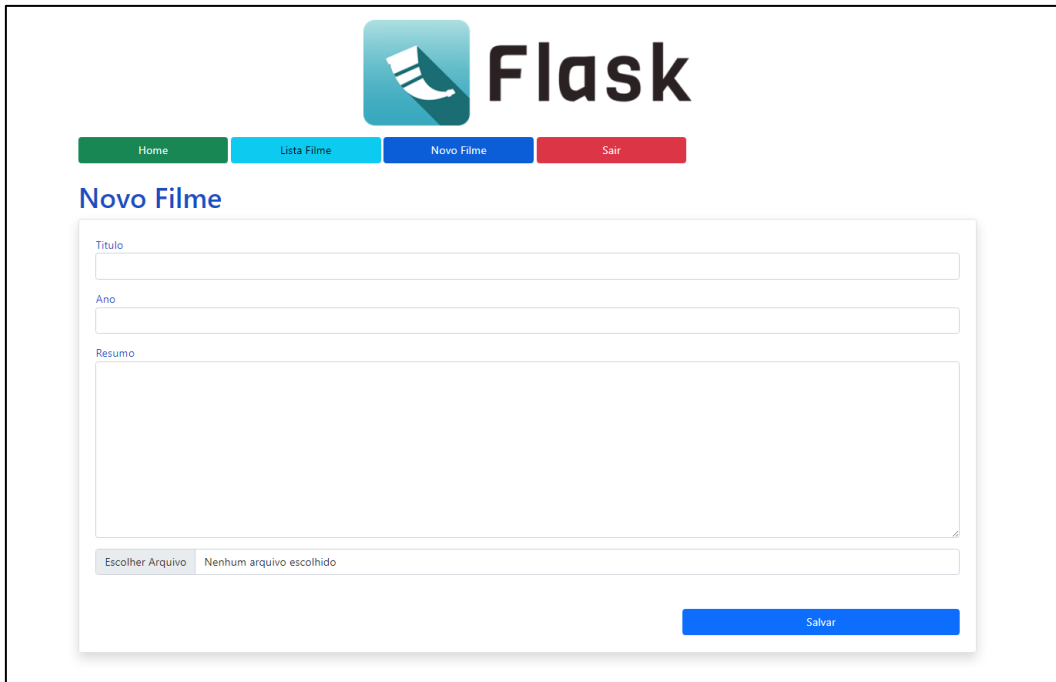
O último passo agora é alterar o HTML do cadastro e o HTML para apresentação, vamos começar com o do cadastro, ou seja o arquivo 'filme_novo.html'.

```
3 {% block content %}
4     <h1>Novo Filme</h1>
5
6     <form method="post" class="form mt-2 p-4 border shadow rounded" enctype="multipart/form-data">
7         {{form.csrf_token}}
8
9         {{form.titulo.label()}}
10        {{form.titulo(class='form-control')}}
11
12        {{form.ano.label(class='mt-3')}}
13        {{form.ano(class='form-control')}}
14
15        {{form.resumo.label(class='mt-3')}}
16        {{form.resumo(class='form-control', rows=10)}}
17
18        {{form.imagem(class='form-control mt-3')}}
19
20        <br><br>
21        <div class="row justify-content-end">
22            <div class="col-4">
23                {{form.submit(class='btn btn-primary col-12')}}
24            </div>
25        </div>
26
27    </form>
28 {% endblock %}
```

Onde foi adicionado o parâmetro 'enctype="multipart/form-data"' no form e incluído a linha 18.

Agora vamos alterar o HTML para exibição, ou seja o arquivo 'filme_detail.html', onde vamos adicionar as seguintes linhas de código para exibir as imagens.

```
21 <div class="col-12 mt-3 text-center">
22 |   
23 </div>
```



The screenshot displays the Flask application's user interface. At the top, there's a navigation bar with four buttons: 'Home' (green), 'Lista Filme' (cyan), 'Novo Filme' (blue), and 'Sair' (red). Below the navigation bar, the 'Novo Filme' section is active, showing a form for adding a new movie. The form includes three input fields: 'Titulo' (Title), 'Ano' (Year), and 'Resumo' (Summary). Below these fields is a file upload section with a button labeled 'Escolher Arquivo' and a text area showing 'Nenhum arquivo escolhido'. At the bottom right of the form is a blue 'Salvar' (Save) button.

Trabalhando com Variável de Ambiente

Variável de ambiente é uma forma segura de manter informações sigilosas, tais como SECRET_KEYS, dados de acesso a banco de dados, entre outras informações.

Essas variáveis só existem no servidor que você cria, sendo assim, pode estar compartilhando o código do APP, sem a preocupação de estar disponibilizando dados sensíveis, para trabalharmos com essas informações vamos utilizar a biblioteca “os” que já vem instalado com o Python

Para definir uma variável de ambiente, podemos definir em nosso Sistema Operacional e posteriormente no servidor, ou também podemos utilizar a biblioteca dotenv, assim podendo criar a variável de sistema em um arquivo chamado ‘.env’, é claro que esse arquivo não deve ser compartilhado, ele irá criar a variável de ambiente para o seu projeto, posteriormente devemos criar as mesmas variáveis dentro do nosso servidor.

Vamos criar as variáveis de sistema para as variáveis que criamos no arquivo local_settings.py e vamos estar excluindo esse arquivo.

Arquivo .env

```
1
2 SECRET_KEY = '99aaf492fd9dc8dd9b2e9c23301c55aa'
3
4 DATABASE_URL = 'sqlite:///database.db'
5
```

Vamos instalar a biblioteca dotenv

```
pip install python-dotenv
```

Por fim vamos atualizar o arquivo __init__.py do app

```
7 import os
8 from dotenv import load_dotenv
9 load_dotenv('.env')
10
11 app = Flask(__name__)
12
13 app.config['SECRET_KEY'] = os.getenv('SECRET_KEY')
14
15 app.config['SQLALCHEMY_DATABASE_URI'] = os.getenv('DATABASE_URL')
16 app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
17
18 app.config['UPLOAD_FOLDER'] = r'static/data'
```

GitHub

O GitHub é uma plataforma de hospedagem de código-fonte baseada em nuvem, conhecida por facilitar o controle de versão e a colaboração entre desenvolvedores. Seu propósito principal é oferecer um ambiente para o armazenamento, gerenciamento e compartilhamento de projetos usando o sistema de controle de versão Git.

- **Controle de Versão Colaborativo:** Permite que várias pessoas trabalhem em um mesmo projeto, controlando e registrando alterações ao longo do tempo. Isso facilita o acompanhamento das mudanças, identificação de responsáveis e reversão a versões anteriores se necessário.
- **Facilitação da Colaboração:** Oferece ferramentas para revisão de código, discussões, gerenciamento de problemas (issues) e solicitações de pull. Isso promove uma colaboração mais eficaz entre os membros da equipe.
- **Hospedagem e Distribuição de Projetos:** Fornece um espaço centralizado na nuvem para hospedar projetos, tornando-os acessíveis a qualquer pessoa com permissões adequadas. Isso é especialmente útil para o compartilhamento de código-fonte aberto.
- **Integração com Ferramentas de Desenvolvimento:** Oferece integração com diversas ferramentas e serviços, como integração contínua (CI), gestão de tarefas, deploy automatizado e mais, facilitando a automação de processos no ciclo de vida do desenvolvimento.

Em resumo, o GitHub se destaca como uma plataforma essencial para desenvolvedores, fornecendo recursos fundamentais para o desenvolvimento de software colaborativo, gerenciamento eficiente de projetos e compartilhamento de código-fonte.

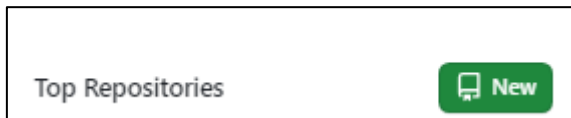
Instalação

Acesse o site <https://git-scm.com/downloads> e realize o download do seu Sistema Operacional, instale o arquivo de acordo com as instruções.

Acesse o site <https://github.com/> crie uma conta

Criando Repositório

Acesse sua conta no github, e clique em New nas opções de repositório




Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Required fields are marked with an asterisk ().*

Owner *


jmiente

Repository name *

flask_teste

✔ flask_teste is available.

Great repository names are short and memorable. Need inspiration? How about [symmetrical-waddle](#) ?

Description (optional)

☒
Public

☐
Private

Anyone on the internet can see this repository. You choose who can commit.

You choose who can see and commit to this repository.

Initialize this repository with:

☐ Add a README file

This is where you can write a long description for your project. [Learn more about READMEs.](#)

Add .gitignore

.gitignore template: None

Choose which files not to track from a list of templates. [Learn more about ignoring files.](#)

Choose a license

License: None

A license tells others what they can and can't do with your code. [Learn more about licenses.](#)

① You are creating a public repository in your personal account.

Create repository

Adicione um nome e altere apenas para public ou private, não faça nenhuma outra alteração, pois queremos criar um repositório em branco

Após criar irá aparecer as instruções em como preparar o repositório

Quick setup — if you've done this kind of thing before

Set up in Desktop or **HTTPS** SSH

Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

...or create a new repository on the command line

```
echo "# flask_teste" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/jmiente/flask_teste.git
git push -u origin main
```

...or push an existing repository from the command line

```
git remote add origin https://github.com/jmiente/flask_teste.git
git branch -M main
git push -u origin main
```

Preparando Repositório e Arquivo .gitignore

Crie um arquivo chamado .gitignore na raiz do projeto, esse arquivo servirá para definirmos o que não irá ser realizado upload no github, vamos definir as seguintes configurações

```
2  __pycache__
3  *.pyc
4  .venv
5  .env
6  instance
```

Após criar o arquivo gitignore, vamos no terminal vamos digitar as instruções acima

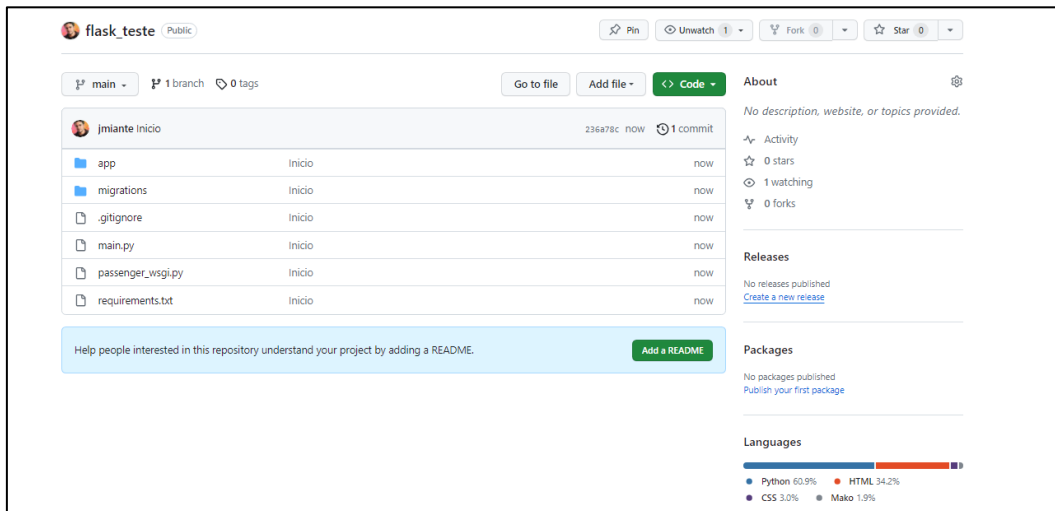
1. Configure o usuário do git (apenas uma vez)
`git config --global user.email seuemail@example.com`
`git config --global user.name "seu nome"`
2. Inicia o repositório no desktop (apenas a primeira vez)
`git init`
3. Adiciona os arquivos para envio de lote
`git add .`
4. Cria o Commit do envio
`git commit -m "First Commit"`
5. Configura o Branch (apenas a primeira vez)
`git branch -m main`
6. Configure qual o repositório
`git remote add origin https://github.com/SEU_USUÁRIO/SEU_REPOSITORIO.git`

7. Realize o upload dos arquivos

`git push -u origin main`

Na primeira vez, pode ser que solicite o login, irá abrir uma caixa de opções, selecione o login pelo navegador e realize o login normalmente.

Ao atualizar a página do seu repositório no git, irá aparecer a seguinte tela



Aqui estará seu projeto.

Atualizando Repositório

Sempre que realizar qualquer alteração no seu projeto, deverá atualizar o github, para isso segue sempre os três passos abaixo:

1. Adicione as alterações em um lote
`git add`.
2. Crie o commit do lote
`git commit -m "Descrição da atualização"`
3. Realize o upload dos arquivos
`git push`

Isso irá garantir que sempre o seu repositório no git estará com a última versão do seu código.

Deploy

O deploy é o processo de disponibilizar uma aplicação ou sistema para uso em um ambiente operacional, como um servidor web, para que os usuários finais possam interagir com ele. Resumidamente, é a etapa em que o código-fonte de uma aplicação é implementado e executado em um ambiente de produção, pronto para ser utilizado pelos usuários. Este processo envolve a transferência dos arquivos e recursos necessários, configurando tudo para funcionar corretamente no ambiente em questão. O deploy é crucial para transformar o código de desenvolvimento em um produto final acessível aos usuários.

Render.com (Opção Gratuita)

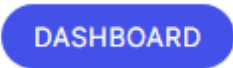
Antes de realizar o Deploy no Render, devemos instalar o biblioteca gunicorn, vamos instalar a partir do pip, depois vamos atualizar o requirements.txt utilizando o comando:

```
pip freeze > requirements.txt
```

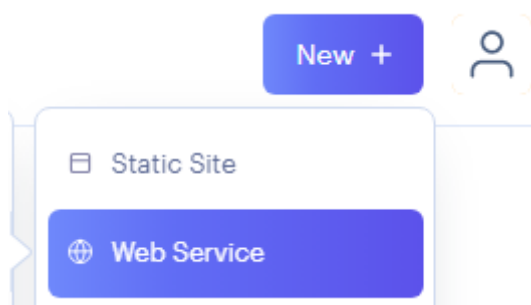
Após isso devemos atualizar nosso repositório do GIT

Agora estamos preparados para realizarmos Deploy no Render, então devemos seguir os seguintes passos:

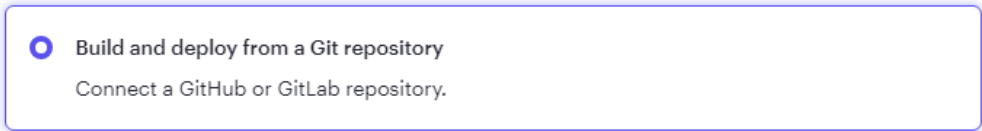
1. Acesse o site <https://render.com/> e crie uma conta (Recomendado criar a conta a partir da conta do GitHub)
2. Acesse o Dashboard

A blue pill-shaped button with the word "DASHBOARD" in white capital letters.

3. Clique em NEW e em WEB SERVICE



4. Crie o Projeto a partir do GitHub

A screenshot of the "Build and deploy from a Git repository" option in the Render.com interface. It features a blue radio button, the text "Build and deploy from a Git repository", and a subtext "Connect a GitHub or GitLab repository." enclosed in a light blue rounded rectangle.

5. Selecione o Repositório do Git e clique em CONNECT



6. Devemos adicionar o Nome do Aplicativo, esse nome deve ser único

7. Selecione a Região

8. Mantenha com a opção Python 3

9. No Start Command, devemos deixar o seguinte comando:

Onde estamos informando que usaremos o gunicorn para renderizar nosso app, utilizaremos o aplicativo app que criamos em nosso projeto seguido da variável app

10. Selecione o Plano que deseja, sendo a primeira opção gratuita

Instance Type	RAM	CPU	Price
<input checked="" type="radio"/> Free	512 MB	0.1 CPU	\$0 / month
<input type="radio"/> Starter	512 MB	0.5 CPU	\$7 / month
<input type="radio"/> Standard	2 GB	1 CPU	\$25 / month

11. Clique em Create Web Service



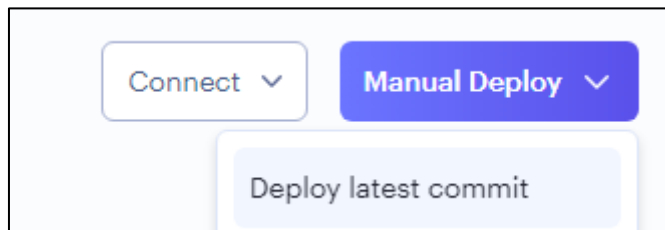
Com isso o aplicativo começara a ser realizado o deploy

12. Como o Render pode ter conflito de versão das bibliotecas, a partir do próximo passo, devemos estar repetindo caso os passos abaixo (somente se acontecer o erro abaixo)

```
ERROR: Could not find a version that satisfies the requirement blinker==1.7.0 (from -r requirements.txt (line 3))
ERROR: No matching distribution found for blinker==1.7.0 (from -r requirements.txt (line 3))
WARNING: You are using pip version 20.1.1; however, version 23.3.1 is available.
You should consider upgrading via the '/opt/render/project/src/.venv/bin/python -m pip install --upgrade pip' command.
--> Build failed 🚫
```

Nesse caso a biblioteca blinker não foi possível recuperar a versão 1.7.0, para resolvermos isso, no nosso projeto no arquivo requirements.txt, vamos apagar a versão dessa biblioteca, recomendo apagar a versão somente das bibliotecas que tiverem incompatibilidade, para isso vamos repetir o processo abaixo até o deploy ser completado

1. Verifique qual biblioteca retornou erro (no caso acima foi a blinker)
2. No Arquivo requirements.txt remova os dados da versão, mantendo apenas o nome da biblioteca
3. Atualize o repositório no github
4. No Render, dentro do projeto, clique em Manual Deploy e em Deploy latest commit

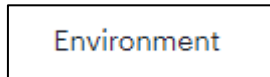


Após passar dos erros de incompatibilidade, se seu projeto utiliza variável de ambiente, retornará o erro que não foi encontrado esses valores, vamos criar a partir do próximo passo

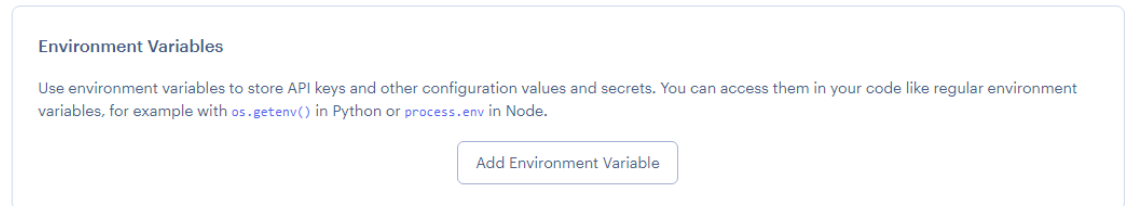
```
Error: Either 'SQLALCHEMY_DATABASE_URI' or 'SQLALCHEMY_BINDS' must be set.
```

13. Para criar variáveis de ambiente no servido, devemos seguir os passos abaixo:

1. Clique em Environment

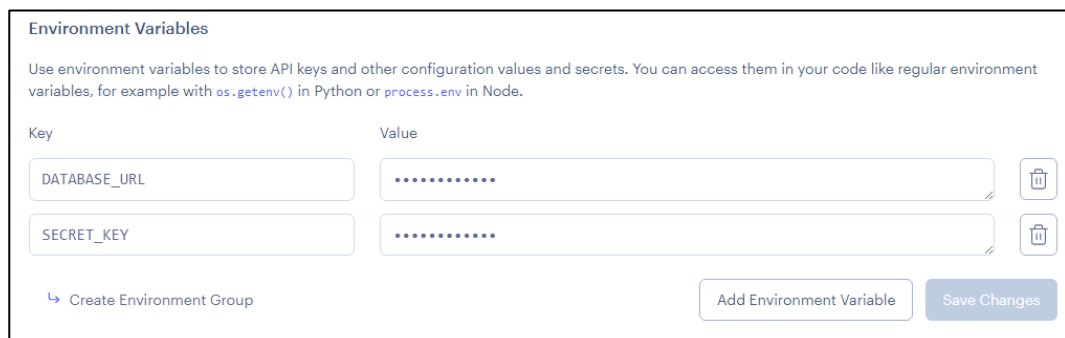


2. Clique em ADD ENVIRONMENT VARIABLE

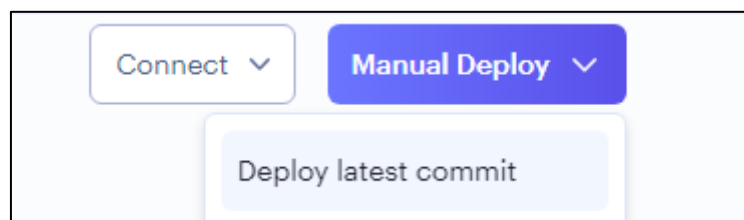


3. Adicione o nome da variável e o seu valor

Repita o processo para todas as variáveis necessárias



Por fim, realize o processo de realizar o deploy manual



Seu Site estará disponível no link abaixo do nome do projeto



Um problema que temos no Render é que ele não aceita o SQLite, então devemos incluir o banco de dados PostgreSQL, que inclusive ele tem na plataforma, porém precisamos adaptar nosso projeto para o Postgres, vamos abaixo para o passo a passo:

1. Na criptografia do projeto, devemos adaptar para gravar no formato UTF-8, para isso no forms na classe LoginForm e UserForm devemos realizar as seguintes alterações:

```
19 |         if bcrypt.check_password_hash(user.password.encode('utf-8'), self.senha.data):
```

```
41 |         senha = bcrypt.generate_password_hash(self.senha.data.encode('utf-8'))
```

2. Instalar a biblioteca pycpg2 e incluir manualmente no arquivo requirements.txt, vamos incluir manualmente todas novas bibliotecas pois no deploy precisamos adaptar a compatibilidade das versões.
3. Atualizar o repositório GIT
4. Criar o Banco de Dados no Render, vamos seguir os passos abaixo

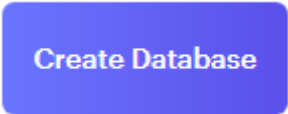
- 4.1. Clique em new e em PostgreSQL

A blue rectangular button with a white icon of a database cylinder and the text 'PostgreSQL'.

- 4.2. Defina um nome para o banco de dados

- 4.3. Selecione o Plano (sendo o primeiro gratuito)

- 4.4. Clique em Create Database

A blue rectangular button with the text 'Create Database' in white.

4.5. Aguarde o status alterar de Creating para Available

Status

Creating

Status

Available

4.6. Em connections você poderá copiar a URL de acesso para o Banco de Dados, para acessar do seu computador, deverá copiar a URL, External Database URL, ou seja, a sua Variável de ambiente para conectar o banco de dados Render da sua máquina será essa URL, para a Variável direto no servidor Render, deverá ser a URL, Internal Database URL

Internal Database URL

External Database URL

4.6. Após copiar os dados da External Database, abra seu arquivo local e altere a variável de ambiente de conexão com o banco de dados

```
4 DATABASE_URL = 'postgres://jm_flask_teste_user:kl
```

4.7. Altere o escrito postgres para postgresql

```
4 DATABASE_URL = 'postgresql://jm_flask_teste_user:kl
```

4.8. Rode as migrate do banco de dados para construir, esse processo deve ser rodado na primeira vez que conectar ao banco de dados e sempre que houver alterações nos modelos

```
flask db upgrade
```

4.9. Altere a URL da variável de ambiente do servidor também, utilizando o Internal Database URL, lembrando de alterar o escrito postgres para postgresql

4.10. Realize o processo de Deploy manual novamente

Com isso sua aplicação estará pronta e realizada o Deploy

```
==> Build successful 🎉
==> Deploying...
==> Using Node version 14.17.0 (default)
==> Docs on specifying a Node version: https://render.com/docs/node-version
==> Running 'gunicorn app:app'
[2023-11-22 20:41:49 +0000] [41] [INFO] Starting gunicorn 21.2.0
[2023-11-22 20:41:49 +0000] [41] [INFO] Listening at: http://0.0.0.0:10000 (41)
[2023-11-22 20:41:49 +0000] [41] [INFO] Using worker: sync
[2023-11-22 20:41:49 +0000] [44] [INFO] Booting worker with pid: 44
==> Detected service running on port 10000
==> Docs on specifying a port: https://render.com/docs/web-services#port-detection
```

Podendo ser acessado por qualquer pessoa pelo link do seu projeto



*** Um outro problema que temos com a opção gratuita é na atualização de nosso APP, os arquivos estáticos acabam sendo resetados para o padrão, então se você trabalha com UPLOAD de arquivos acaba perdendo esses arquivos, na opção paga, existe como configurar a pasta que não será apagada nas atualizações, assim mantendo os arquivos.

Utilizando CPanel

Python version	3.9.18
Raiz do aplicativo <small>It is a physical address to your application on a server that corresponds with its URI. Upload your application files here.</small>	site/teste
URL do aplicativo <small>É um link HTTP/HTTPS para seu aplicativo</small>	teste.com.br
Arquivo de inicialização do aplicativo	passenger_wsgi.py
Application Entry point <small>Setup wsgi callable object for your application</small>	application
Passenger log file <small>You can define the path along with the filename (e.g. /home/controla/logs/passenger.log)</small>	/home/controla/

- **Python Version:** Selecione a versão o Python
- **Raiz do Aplicativo:** Digite o caminho para a pasta dos arquivos do projeto
- **URL do Aplicativo:** Domínio registrado do APP
- **Arquivo de Inicialização:** Manter sempre passar o `passenger_wsgi.py`
- **App Entry Point:** Manter sempre `application`
- **Passenger log file:** Deixar em branco

No arquivo `__init__.py` do app, adicione a linha abaixo antes das importação das rotas

```
22 application = app
```

Na raiz do aplicativo crie o arquivo `passenger_wsgi.py`

```
1
2 from app import application
3
```

Apague todos os dados da pasta criada no servidor e suba os dados do seu projeto, entre no CPanel novamente e instale as bibliotecas, adicionando o `requirements.txt`

Configuration files	<div>▶ Run Pip Install</div>
	<div>requirements.txt</div> <div>⊕ Add</div>
Execute python script <small>You can also enter a command to execute (e.g. /path/to/manage.py migrate). Note, only python scripts allowed.</small>	<div>Enter the path to the script file</div> <div>▶ Run Script</div>

Digite o `requirements.txt` e clique em Add, posteriormente clique em RUN PIP INSTALL