**Tutorial: Criando um Gerenciador de Tarefas com Flask**

**José Alfredo F. Costa – 27-09-24**

**Versão em edição / correção**

**1. Introdução**

* Visão geral do projeto
* Requisitos e ferramentas necessárias

**2. Configuração do Ambiente**

* Instalação do Python
* Criação de um ambiente virtual
* Instalação do Flask e dependências

**3. Estrutura do Projeto**

* Criação da estrutura de diretórios
* Explicação da arquitetura MVC

**4. Configuração do Banco de Dados**

* Escolha do banco de dados (SQLite)
* Configuração do SQLAlchemy

**5. Modelagem de Dados**

* Criação do modelo de Tarefa

**6. Rotas e Views**

* Implementação das rotas CRUD
* Criação das views correspondentes

**7. Templates**

* Criação dos templates HTML
* Uso do Jinja2 para renderização dinâmica

**8. Funcionalidades CRUD**

* Criar tarefa
* Ler tarefas
* Atualizar tarefa
* Excluir tarefa

**9. Estilização com CSS**

* Adição de estilos básicos
* Uso de um framework CSS (opcional)

**10. Testes**

* Escrita de testes unitários
* Testes de integração

**11. Deployment**

* Preparação para deploy
* Opções de hospedagem (ex: Heroku, PythonAnywhere)

**12. Conclusão e Próximos Passos**

* Recap do que foi aprendido
* Sugestões para melhorias e recursos adicionais

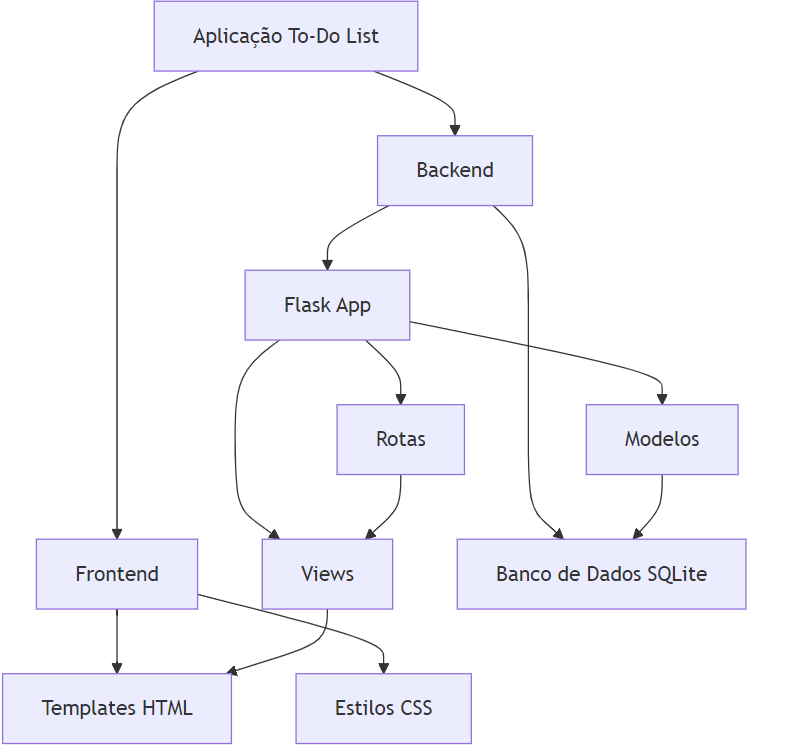


Fig. 1 - Diagrama da estrutura da aplicação usando Mermaid[[1]](#footnote-0)

Este diagrama mostra a estrutura básica da nossa aplicação, incluindo a separação entre frontend e backend, e como os diferentes componentes se relacionam.

Agora, vamos detalhar a estrutura de diretórios do projeto.

**todo\_app/**

│

├── app/

│ ├── \_\_init\_\_.py

│ ├── models.py

│ ├── routes.py

│ └── templates/

│ ├── base.html

│ ├── index.html

│ ├── create.html

│ ├── edit.html

│ └── delete.html

│

├── static/

│ └── css/

│ └── style.css

│

├── tests/

│ ├── \_\_init\_\_.py

│ ├── test\_models.py

│ └── test\_routes.py

│

├── config.py

├── run.py

├── requirements.txt

└── README.md

Esta estrutura de diretórios segue as melhores práticas para um projeto Flask, separando claramente os diferentes componentes da aplicação.

Com esta estrutura e visão geral, estamos prontos para começar o desenvolvimento da nossa aplicação de gerenciamento de tarefas.

**Capítulo 1: Introdução**

**Visão Geral do Projeto**

Neste tutorial, vamos criar um Gerenciador de Tarefas (To-Do List) usando o framework Flask em Python. Esta aplicação web permitirá aos usuários:

1. Adicionar novas tarefas
2. Visualizar uma lista de todas as tarefas
3. Marcar tarefas como concluídas
4. Editar tarefas existentes
5. Excluir tarefas

O projeto servirá como uma excelente introdução ao desenvolvimento web com Flask, cobrindo conceitos fundamentais como rotas, templates, interações com banco de dados e CRUD (Create, Read, Update, Delete) operations.

**Requisitos e Ferramentas Necessárias**

Para seguir este tutorial, você precisará das seguintes ferramentas e conhecimentos:

1. **Python**: Versão 3.7 ou superior. Nosso código será escrito em Python.
2. **Flask**: O framework web que usaremos para criar nossa aplicação.
3. **SQLAlchemy**: Para interações com o banco de dados.
4. **HTML/CSS**: Conhecimento básico para criar os templates e estilizar nossa aplicação.
5. **Git**: (Opcional) Para controle de versão.
6. **Editor de código**: Como VSCode, PyCharm, ou Sublime Text.

Conhecimentos prévios que serão úteis:

1. Familiaridade básica com Python
2. Entendimento básico de HTML e CSS
3. Conceitos fundamentais de desenvolvimento web (HTTP, requisições GET/POST)

Não se preocupe se você não dominar todos esses conceitos - explicaremos cada passo detalhadamente ao longo do tutorial.

# Capítulo 2: Configuração do Ambiente

## Instalação do Python

1. Visite o site oficial do Python (<https://www.python.org/downloads/>).
2. Baixe a versão mais recente do Python 3 para seu sistema operacional.
3. Execute o instalador e siga as instruções na tela.
4. Verifique a instalação abrindo um terminal e digitando:

python --version

## Criação de um Ambiente Virtual

Os ambientes virtuais são uma prática recomendada para isolar as dependências do projeto. Siga estes passos:

1. Abra um terminal e navegue até o diretório onde você deseja criar seu projeto.
2. Crie um novo diretório para o projeto e entre nele:

mkdir todo\_app

cd todo\_app

1. Crie um ambiente virtual:

python -m venv venv

1. Ative o ambiente virtual:
2. No Windows:

venv\Scripts\activate

1. No macOS e Linux:

source venv/bin/activate

## Instalação do Flask e Dependências

Com o ambiente virtual ativado, instale o Flask e outras dependências necessárias:

pip install flask flask-sqlalchemy

Após a instalação, crie um arquivo requirements.txt para rastrear as dependências:

pip freeze > requirements.txt

O conteúdo do seu requirements.txt deve ser similar a:

click==8.0.3

Flask==2.0.2

Flask-SQLAlchemy==2.5.1

itsdangerous==2.0.1

Jinja2==3.0.3

MarkupSafe==2.0.1

SQLAlchemy==1.4.29

Werkzeug==2.0.2

**Capítulo 3: Estrutura do Projeto**

**Criação da Estrutura de Diretórios**

Vamos criar a estrutura de diretórios para nosso projeto. Esta estrutura seguirá as melhores práticas para aplicações Flask:

todo\_app/

│

├── app/

│ ├── \_\_init\_\_.py

│ ├── models.py

│ ├── routes.py

│ └── templates/

│ ├── base.html

│ ├── index.html

│ ├── create.html

│ ├── edit.html

│ └── delete.html

│

├── static/

│ └── css/

│ └── style.css

│

├── tests/

│ ├── \_\_init\_\_.py

│ ├── test\_models.py

│ └── test\_routes.py

│

├── config.py

├── run.py

├── requirements.txt

└── README.md

Crie esta estrutura de diretórios usando os comandos do seu sistema operacional ou o explorador de arquivos.

**Explicação da Arquitetura MVC**

Nossa aplicação seguirá o padrão de arquitetura Model-View-Controller (MVC). Vamos entender cada componente:

1. **Model (app/models.py)**:
2. Define a estrutura dos dados
3. Lida com a lógica de negócios
4. Interage com o banco de dados
5. **View (app/templates/\*)**:
6. Responsável pela apresentação
7. Renderiza os dados para o usuário
8. Usa HTML com template engine Jinja2
9. **Controller (app/routes.py)**:
10. Gerencia o fluxo de dados entre Model e View
11. Define as rotas da aplicação
12. Processa as requisições do usuário

Vamos criar o arquivo app/\_\_init\_\_.py para inicializar nossa aplicação Flask:

from flask import Flask

from flask\_sqlalchemy import SQLAlchemy

# Inicializa a aplicação Flask

app = Flask(\_\_name\_\_)

# Configura o banco de dados

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///todo.db'

app.config['SQLALCHEMY\_TRACK\_MODIFICATIONS'] = False

# Inicializa o objeto SQLAlchemy

db = SQLAlchemy(app)

# Importa as rotas (será criado posteriormente)

from app import routes

Este arquivo inicializa nossa aplicação Flask e configura o banco de dados SQLite. As rotas serão importadas depois que criarmos o arquivo routes.py.

Para visualizar melhor como os componentes da nossa aplicação se relacionam, vamos criar um diagrama usando Mermaid:

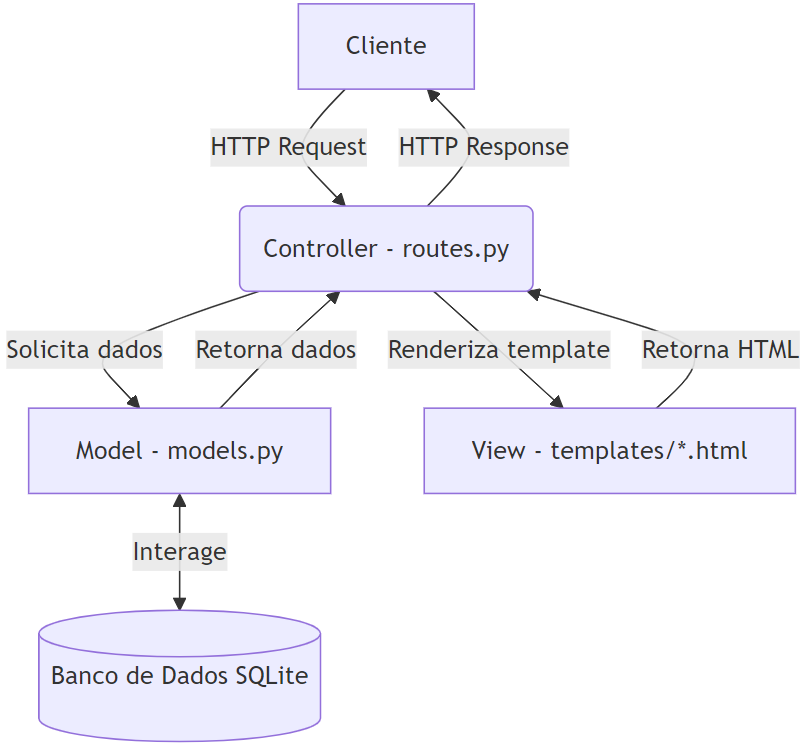


Fig. 2 - Diagrama ilustra como os componentes do padrão MVC interagem em nossa aplicação Flask[[2]](#footnote-1)

**Capítulo 4: Configuração do Banco de Dados**

**Escolha do Banco de Dados**

Para nossa aplicação de gerenciamento de tarefas, usaremos o SQLite como banco de dados. SQLite é uma escolha excelente para aplicações pequenas e médias, pois:

1. É embutido e não requer um servidor separado
2. É fácil de configurar e usar
3. Não requer configuração adicional para desenvolvimento local

**Configuração do SQLAlchemy**

Já iniciamos a configuração do SQLAlchemy no arquivo app/\_\_init\_\_.py. Vamos revisar e entender cada linha:

from flask\_sqlalchemy import SQLAlchemy

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///todo.db'

app.config['SQLALCHEMY\_TRACK\_MODIFICATIONS'] = False

db = SQLAlchemy(app)

1. SQLALCHEMY\_DATABASE\_URI: Define o caminho para o arquivo do banco de dados SQLite.
2. SQLALCHEMY\_TRACK\_MODIFICATIONS: Desativa o sistema de notificação de modificações do SQLAlchemy para melhor performance.
3. db = SQLAlchemy(app): Cria uma instância do SQLAlchemy vinculada à nossa aplicação Flask.

**Criação do Banco de Dados**

Para criar o banco de dados, adicionaremos um script no arquivo run.py. Isso garantirá que o banco de dados seja criado quando executarmos nossa aplicação pela primeira vez.

Agora, vamos criar o conteúdo do arquivo run.py:

from app import app, db

if \_\_name\_\_ == '\_\_main\_\_':

    with app.app\_context():

        db.create\_all()

    app.run(debug=True)

Este script cria o banco de dados (se não existir) e inicia o servidor de desenvolvimento Flask.

**Capítulo 5: Modelagem de Dados**

**Criação do Modelo de Tarefa**

Vamos criar nosso modelo de Tarefa no arquivo app/models.py. Este modelo representará a estrutura de uma tarefa em nosso aplicativo.

Aqui está o código para o modelo de Tarefa:

from app import db

from datetime import datetime

class Task(db.Model):

id = db.Column(db.Integer, primary\_key=True)

title = db.Column(db.String(100), nullable=False)

description = db.Column(db.String(200))

created\_at = db.Column(db.DateTime, default=datetime.utcnow)

completed = db.Column(db.Boolean, default=False)

def \_\_repr\_\_(self):

return f'<Task {self.id}: {self.title}>'

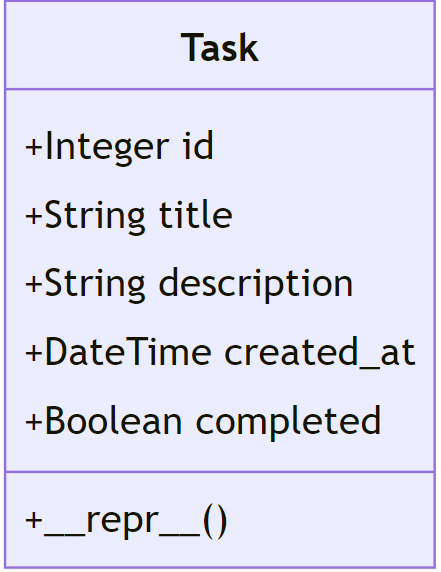
Vamos entender cada campo do nosso modelo:

1. id: Um identificador único para cada tarefa.
2. title: O título da tarefa (obrigatório).
3. description: Uma descrição opcional da tarefa.
4. created\_at: A data e hora de criação da tarefa.
5. completed: Um booleano indicando se a tarefa foi concluída.

O método \_\_repr\_\_ fornece uma representação string do objeto Task, útil para debugging.

**Diagrama do Modelo de Dados**

Para visualizar melhor nosso modelo de dados, vamos criar um diagrama usando Mermaid[[3]](#footnote-2):



Este diagrama ilustra a estrutura do nosso modelo de Tarefa, mostrando todos os campos e seus tipos.

**Capítulo 6: Rotas e Views**

**Implementação das Rotas CRUD**

Vamos criar as rotas para nossa aplicação no arquivo app/routes.py. Implementaremos as operações CRUD (Create, Read, Update, Delete) para nossas tarefas.

Aqui está o código para nossas rotas:

from flask import render\_template, request, redirect, url\_for

from app import app, db

from app.models import Task

@app.route('/')

def index():

tasks = Task.query.all()

return render\_template('index.html', tasks=tasks)

@app.route('/create', methods=['GET', 'POST'])

def create():

if request.method == 'POST':

title = request.form['title']

description = request.form['description']

new\_task = Task(title=title, description=description)

db.session.add(new\_task)

db.session.commit()

return redirect(url\_for('index'))

return render\_template('create.html')

@app.route('/update/<int:id>', methods=['GET', 'POST'])

def update(id):

task = Task.query.get\_or\_404(id)

if request.method == 'POST':

task.title = request.form['title']

task.description = request.form['description']

task.completed = 'completed' in request.form

db.session.commit()

return redirect(url\_for('index'))

return render\_template('edit.html', task=task)

@app.route('/delete/<int:id>')

def delete(id):

task = Task.query.get\_or\_404(id)

db.session.delete(task)

db.session.commit()

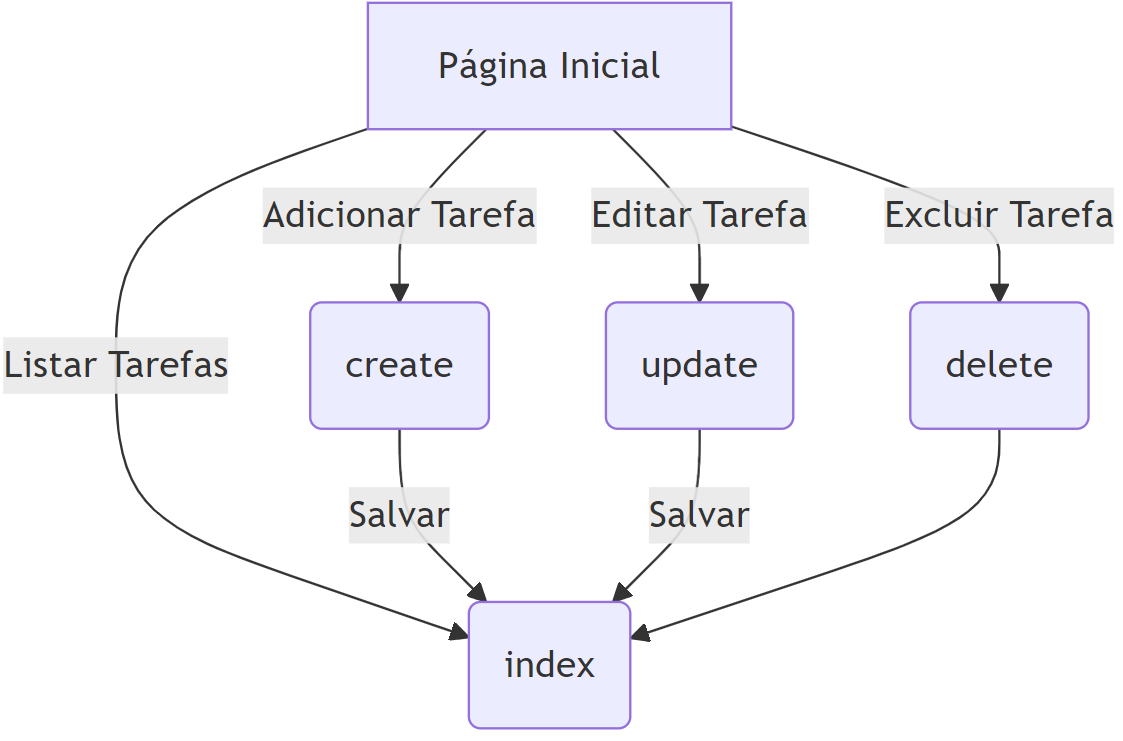
return redirect(url\_for('index'))

Vamos entender cada rota:

1. index(): Exibe todas as tarefas.
2. create(): Permite criar uma nova tarefa.
3. update(): Permite editar uma tarefa existente.
4. delete(): Permite excluir uma tarefa.

**Diagrama de Fluxo das Rotas**

Para visualizar melhor como nossas rotas se conectam, vamos criar um diagrama de fluxo:



Este diagrama ilustra como as diferentes rotas se conectam em nossa aplicação

# Capítulo 7: Templates

## Criação dos Templates HTML

Neste capítulo, criaremos os templates HTML para nossa aplicação. Usaremos o Jinja2, o motor de templates padrão do Flask, para renderizar conteúdo dinâmico.

### Template Base (base.html)

Primeiro, vamos criar um template base que será estendido por outros templates:

<!DOCTYPE html>

<html lang="pt-br">

<head>

<meta charset="UTF-8">

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

<title>{% block title %}Gerenciador de Tarefas{% endblock %}</title>

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

</head>

<body>

<header>

<h1>Gerenciador de Tarefas</h1>

<nav>

<ul>

<li><a href="{{ url\_for('index') }}">Início</a></li>

<li><a href="{{ url\_for('create') }}">Nova Tarefa</a></li>

</ul>

</nav>

</header>

<main>

{% block content %}

{% endblock %}

</main>

<footer>

<p>&copy; 2024 Gerenciador de Tarefas</p>

</footer>

</body>

</html>

### Template da Página Inicial (index.html)

Agora, vamos criar o template para a página inicial:

{% extends "base.html" %}

{% block content %}

<h2>Lista de Tarefas</h2>

{% if tasks %}

<ul class="task-list">

{% for task in tasks %}

<li>

<h3>{{ task.title }}</h3>

<p>{{ task.description }}</p>

<p>Criada em: {{ task.created\_at.strftime('%d/%m/%Y %H:%M') }}</p>

<p>Status: {{ 'Concluída' if task.completed else 'Pendente' }}</p>

<a href="{{ url\_for('update', id=task.id) }}">Editar</a>

<a href="{{ url\_for('delete', id=task.id) }}" onclick="return confirm('Tem certeza que deseja excluir esta tarefa?');">Excluir</a>

</li>

{% endfor %}

</ul>

{% else %}

<p>Nenhuma tarefa cadastrada.</p>

{% endif %}

{% endblock %}

### Template de Criação de Tarefa (create.html)

Aqui está o template para criar uma nova tarefa:

{% extends "base.html" %}

{% block title %}Nova Tarefa{% endblock %}

{% block content %}

<h2>Nova Tarefa</h2>

<form method="POST">

<div>

<label for="title">Título:</label>

<input type="text" id="title" name="title" required>

</div>

<div>

<label for="description">Descrição:</label>

<textarea id="description" name="description"></textarea>

</div>

<button type="submit">Criar Tarefa</button>

</form>

{% endblock %}

### Template de Edição de Tarefa (edit.html)

Por fim, o template para editar uma tarefa existente:

{% extends "base.html" %}

{% block title %}Editar Tarefa{% endblock %}

{% block content %}

<h2>Editar Tarefa</h2>

<form method="POST">

<div>

<label for="title">Título:</label>

<input type="text" id="title" name="title" value="{{ task.title }}" required>

</div>

<div>

<label for="description">Descrição:</label>

<textarea id="description" name="description">{{ task.description }}</textarea>

</div>

<div>

<label for="completed">Concluída:</label>

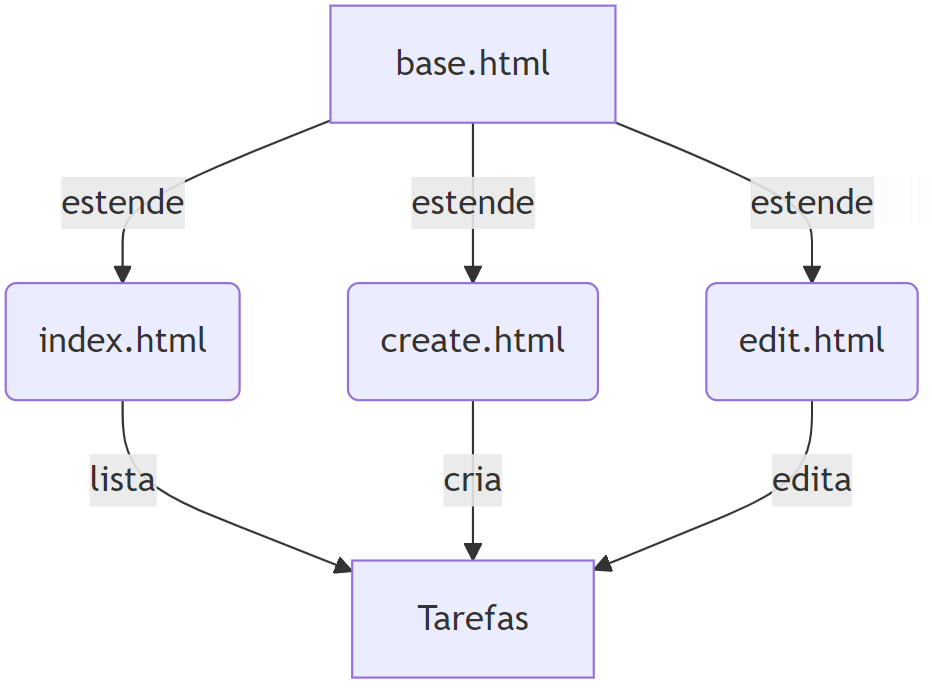
<input type="checkbox" id="completed" name="completed" {% if task.completed %}checked{% endif %}>

</div>

<button type="submit">Atualizar Tarefa</button>

</form>

{% endblock %}



Este diagrama mostra como os templates se relacionam entre si e com as tarefas.

**Capítulo 8: Funcionalidades CRUD**

Neste capítulo, vamos implementar as funcionalidades CRUD (Create, Read, Update, Delete) em nossa aplicação.

**Create (Criar Tarefa)**

A funcionalidade de criação de tarefas já está implementada na rota create(). Vamos revisá-la:

@app.route('/create', methods=['GET', 'POST'])

def create():

if request.method == 'POST':

title = request.form['title']

description = request.form['description']

new\_task = Task(title=title, description=description)

db.session.add(new\_task)

db.session.commit()

return redirect(url\_for('index'))

return render\_template('create.html')

**Read (Ler Tarefas)**

A leitura de tarefas é realizada na rota index():

@app.route('/')

def index():

tasks = Task.query.all()

return render\_template('index.html', tasks=tasks)

**Update (Atualizar Tarefa)**

A atualização de tarefas é feita na rota update():

@app.route('/update/<int:id>', methods=['GET', 'POST'])

def update(id):

task = Task.query.get\_or\_404(id)

if request.method == 'POST':

task.title = request.form['title']

task.description = request.form['description']

task.completed = 'completed' in request.form

db.session.commit()

return redirect(url\_for('index'))

return render\_template('edit.html', task=task)

**Delete (Excluir Tarefa)**

A exclusão de tarefas é realizada na rota delete():

@app.route('/delete/<int:id>')

def delete(id):

task = Task.query.get\_or\_404(id)

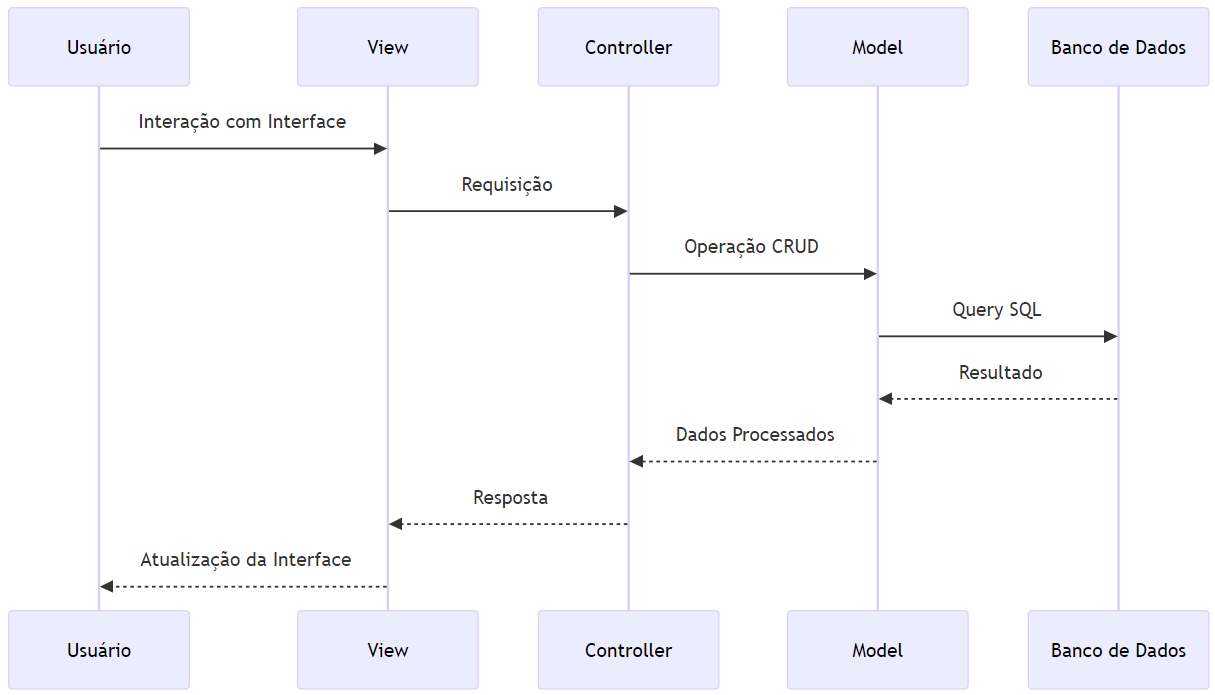
db.session.delete(task)

db.session.commit()

return redirect(url\_for('index'))

**Fluxo de Dados CRUD**

Para entender melhor como os dados fluem em nossas operações CRUD, vamos criar um diagrama:



Este diagrama ilustra como os dados fluem entre as diferentes camadas da nossa aplicação durante as operações CRUD[[4]](#footnote-3).

# Capítulo 9: Estilização com CSS

Neste capítulo, vamos adicionar estilos à nossa aplicação para melhorar sua aparência e usabilidade.

## Criação do Arquivo CSS

Crie um arquivo chamado style.css no diretório static/css/. Aqui está um exemplo de CSS para nossa aplicação:

css

body {

font-family: Arial, sans-serif;

line-height: 1.6;

margin: 0;

padding: 0;

background-color: #f4f4f4;

}

header {

background-color: #333;

color: #fff;

text-align: center;

padding: 1rem;

}

nav ul {

padding: 0;

list-style-type: none;

}

nav ul li {

display: inline;

margin-right: 10px;

}

nav ul li a {

color: #fff;

text-decoration: none;

}

main {

padding: 2rem;

}

.task-list {

list-style-type: none;

padding: 0;

}

.task-list li {

background-color: #fff;

margin-bottom: 1rem;

padding: 1rem;

border-radius: 5px;

box-shadow: 0 2px 5px rgba(0,0,0,0.1);

}

form div {

margin-bottom: 1rem;

}

label {

display: block;

margin-bottom: 0.5rem;

}

input[type="text"], textarea {

width: 100%;

padding: 0.5rem;

border: 1px solid #ddd;

border-radius: 3px;

}

button {

background-color: #333;

color: #fff;

padding: 0.5rem 1rem;

border: none;

border-radius: 3px;

cursor: pointer;

}

button:hover {

background-color: #555;

}

footer {

text-align: center;

padding: 1rem;

background-color: #333;

color: #fff;

position: fixed;

bottom: 0;

width: 100%;

}

## Aplicação dos Estilos

Os estilos serão automaticamente aplicados à nossa aplicação, pois já incluímos o link para o arquivo CSS no nosso template base (base.html):

html

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

## Responsividade

Para tornar nossa aplicação responsiva, podemos adicionar media queries ao nosso CSS. Por exemplo:

css

@media (max-width: 600px) {

header {

padding: 0.5rem;

}

main {

padding: 1rem;

}

.task-list li {

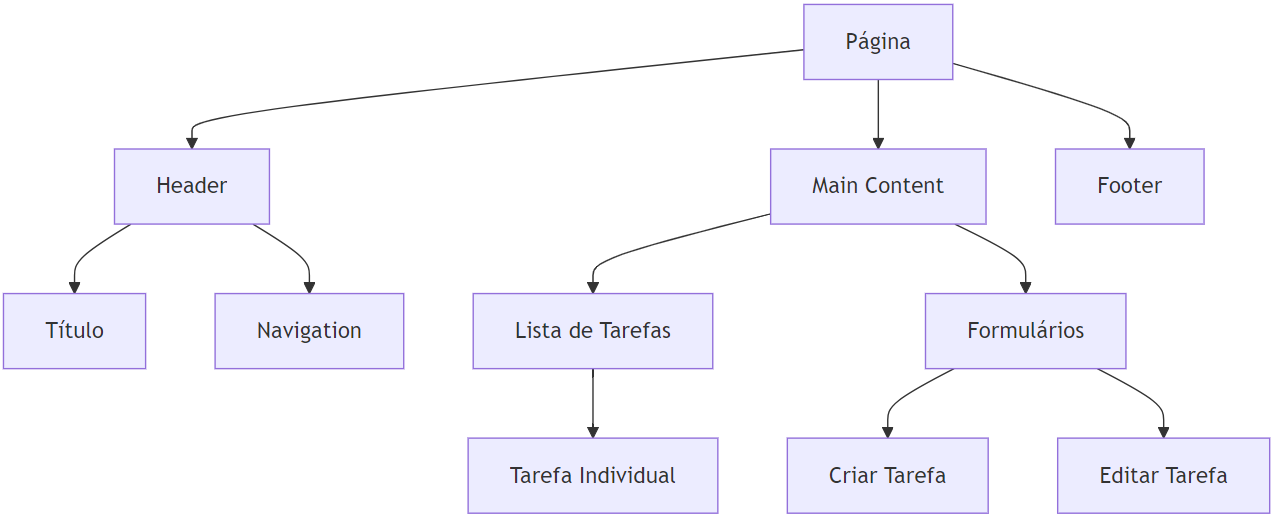
padding: 0.5rem;

}

}

Isso ajustará o layout para telas menores, como smartphones.

Para visualizar melhor como os estilos afetam nossa aplicação, vamos criar um diagrama mostrando a estrutura visual básica:



Este diagrama ilustra a estrutura visual básica da nossa aplicação após a aplicação dos estilos CSS[[5]](#footnote-4).

**Capítulo 10: Testes**

Neste capítulo, vamos implementar testes para nossa aplicação de gerenciamento de tarefas. Usaremos o módulo unittest do Python em conjunto com o cliente de teste do Flask para criar nossos testes.

**Configuração do Ambiente de Teste**

Primeiro, vamos criar um arquivo tests.py na raiz do nosso projeto. Neste arquivo, vamos importar as dependências necessárias e configurar nossa classe de teste:

import unittest

from app import app, db

from app.models import Task

class TodoAppTestCase(unittest.TestCase):

def setUp(self):

app.config['TESTING'] = True

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///test.db'

self.client = app.test\_client()

with app.app\_context():

db.create\_all()

def tearDown(self):

with app.app\_context():

db.session.remove()

db.drop\_all()

**Testes Unitários**

Agora, vamos adicionar alguns testes unitários para nossa aplicação:

def test\_index(self):

response = self.client.get('/')

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

self.assertIn(b'Gerenciador de Tarefas', response.data)

def test\_create\_task(self):

response = self.client.post('/create', data=dict(

title='Teste de Tarefa',

description='Descrição de teste'

), follow\_redirects=True)

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

self.assertIn(b'Teste de Tarefa', response.data)

def test\_update\_task(self):

task = Task(title='Tarefa Original', description='Descrição original')

with app.app\_context():

db.session.add(task)

db.session.commit()

task\_id = task.id

response = self.client.post(f'/update/{task\_id}', data=dict(

title='Tarefa Atualizada',

description='Descrição atualizada'

), follow\_redirects=True)

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

self.assertIn(b'Tarefa Atualizada', response.data)

def test\_delete\_task(self):

task = Task(title='Tarefa para Deletar', description='Será deletada')

with app.app\_context():

db.session.add(task)

db.session.commit()

task\_id = task.id

response = self.client.get(f'/delete/{task\_id}', follow\_redirects=True)

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

self.assertNotIn(b'Tarefa para Deletar', response.data)

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

**Executando os Testes**

Para executar os testes, basta rodar o seguinte comando no terminal:

python tests.py

**Cobertura de Testes**

Para medir a cobertura de nossos testes, podemos usar a biblioteca coverage. Primeiro, instale-a:

pip install coverage

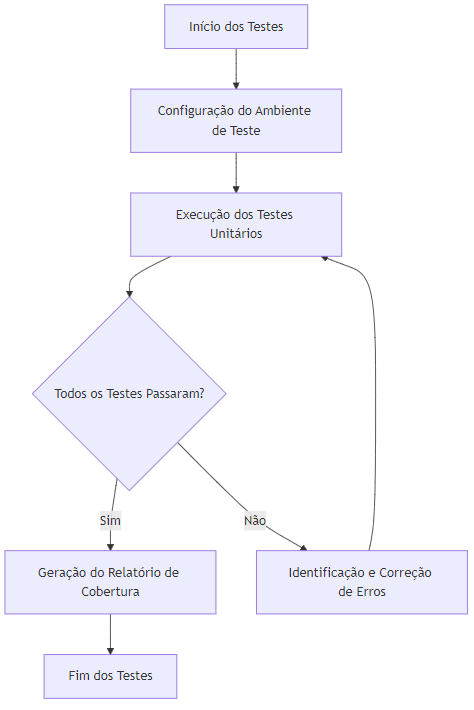
Em seguida, execute os testes com cobertura:

coverage run tests.py

Para ver o relatório de cobertura:

coverage report -m

Agora, vamos criar um diagrama para visualizar o fluxo de execução dos testes[[6]](#footnote-5):



**Capítulo 11: Deployment**

Neste capítulo, vamos preparar nossa aplicação para deployment e implantá-la na plataforma Heroku.

**Preparação para Deployment**

1. Crie um arquivo Procfile na raiz do projeto com o seguinte conteúdo:

web: gunicorn run:app

1. Instale o Gunicorn:

pip install gunicorn

1. Crie um arquivo requirements.txt com todas as dependências:

pip freeze > requirements.txt

1. Certifique-se de que o arquivo .gitignore inclua:

venv/

\*.pyc

\_\_pycache\_\_/

instance/

.pytest\_cache/

.coverage

htmlcov/

dist/

build/

\*.egg-info/

**Deployment no Heroku**

1. Instale o Heroku CLI e faça login:

heroku login

1. Crie um novo app no Heroku:

heroku create nome-do-seu-app

1. Adicione o banco de dados PostgreSQL:

heroku addons:create heroku-postgresql:hobby-dev

1. Configure as variáveis de ambiente:

heroku config:set FLASK\_APP=run.py

heroku config:set FLASK\_ENV=production

1. Faça deploy da aplicação:

git push heroku main

1. Execute as migrações do banco de dados:

heroku run python

>>> from app import db

>>> db.create\_all()

>>> exit()

1. Abra a aplicação no navegador:

heroku open

**Monitoramento e Logs**

Para monitorar sua aplicação e ver os logs:

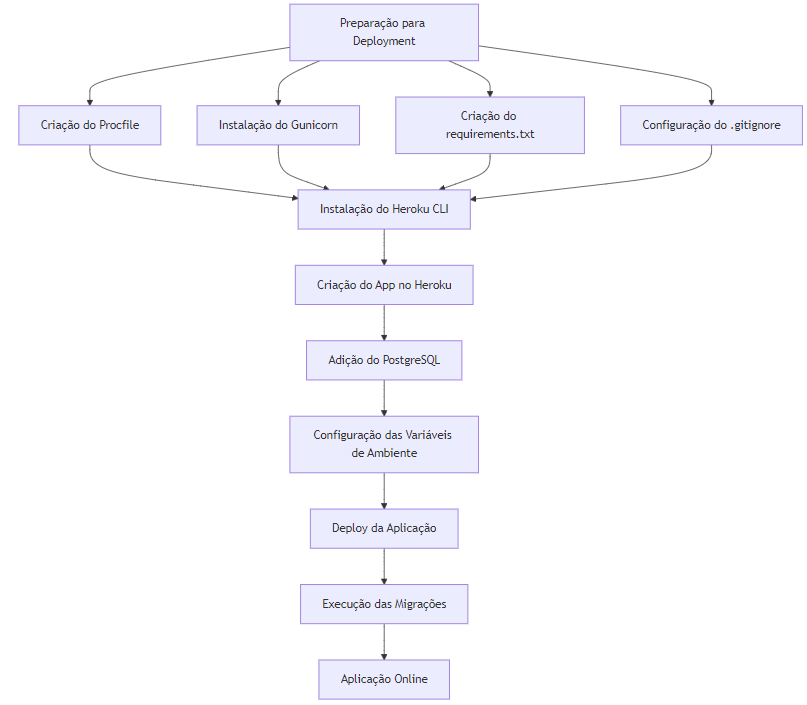
heroku logs --tail

**Scaling**

Para escalar sua aplicação:

heroku ps:scale web=1

Agora, vamos criar um diagrama para visualizar o processo de deployment[[7]](#footnote-6):



# Capítulo 12: Conclusão e Próximos Passos

## Recap do que foi Aprendido

Neste tutorial, exploramos o desenvolvimento completo de uma aplicação web de gerenciamento de tarefas usando Flask. Aprendemos sobre:

1. Configuração do ambiente de desenvolvimento Flask
2. Estruturação de um projeto Flask
3. Modelagem de dados com SQLAlchemy
4. Criação de rotas e views
5. Desenvolvimento de templates com Jinja2
6. Implementação de operações CRUD
7. Estilização da aplicação com CSS
8. Escrita e execução de testes unitários
9. Processo de deployment no Heroku

## Próximos Passos e Melhorias Possíveis

Para continuar melhorando nossa aplicação, podemos considerar os seguintes passos:

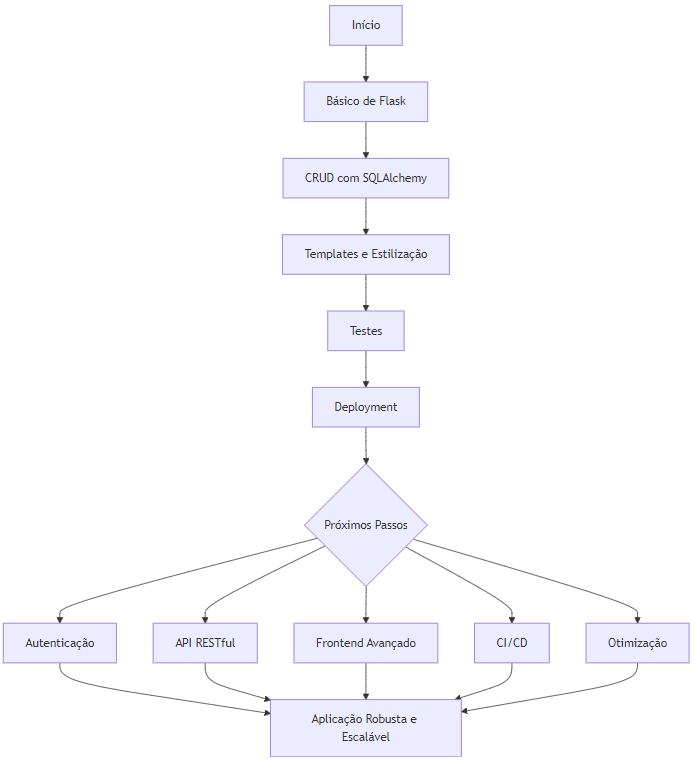
1. **Autenticação de Usuários**: Implementar um sistema de login para que cada usuário tenha suas próprias tarefas.
2. **Categorização de Tarefas**: Adicionar a capacidade de categorizar tarefas (por exemplo, trabalho, pessoal, urgente).
3. **Datas de Vencimento**: Incluir datas de vencimento para as tarefas e implementar notificações.
4. **API RESTful**: Criar uma API para permitir integração com outras aplicações ou desenvolvimento de um frontend separado.
5. **Melhorias na UI/UX**: Implementar uma interface de usuário mais sofisticada, possivelmente usando um framework JavaScript como React ou Vue.js.
6. **Testes de Integração e E2E**: Adicionar testes de integração e end-to-end para aumentar a confiabilidade da aplicação.
7. **Continuous Integration/Continuous Deployment (CI/CD)**: Configurar um pipeline de CI/CD para automatizar testes e deployment.
8. **Monitoramento e Logging**: Implementar um sistema robusto de monitoramento e logging para facilitar a manutenção e resolução de problemas.
9. **Otimização de Performance**: Analisar e otimizar o desempenho da aplicação, possivelmente implementando caching.
10. **Internacionalização**: Adicionar suporte para múltiplos idiomas.

## Recursos Adicionais para Aprendizado

1. Documentação oficial do Flask: <https://flask.palletsprojects.com/>
2. Flask Mega-Tutorial por Miguel Grinberg: <https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world>
3. Curso de Flask na Pluralsight: <https://www.pluralsight.com/courses/flask-getting-started>
4. Livro "Flask Web Development" por Miguel Grinberg

Lembre-se, o desenvolvimento de software é uma jornada contínua de aprendizado e melhoria. Continue explorando, experimentando e construindo!

Para concluir, vamos criar um diagrama que ilustra o caminho de aprendizado e as possíveis direções futuras[[8]](#footnote-7):



Este tutorial forneceu uma base sólida para o desenvolvimento web com Python e Flask, abordando desde os conceitos básicos até técnicas mais avançadas de teste e deployment.

## Sumarizando:

## Vamos começar com os códigos principais:

Estrutura do Projeto Flask

todo\_app/

│

├── app/

│ ├── \_\_init\_\_.py

│ ├── models.py

│ ├── routes.py

│ └── templates/

│ ├── base.html

│ ├── index.html

│ ├── create.html

│ └── edit.html

│

├── static/

│ └── css/

│ └── style.css

│

├── tests/

│ ├── \_\_init\_\_.py

│ ├── test\_models.py

│ └── test\_routes.py

│

├── config.py

├── run.py

├── requirements.txt

└── README.md

Este é o esqueleto do projeto. Ele define a estrutura de diretórios e arquivos que serão usados em toda a aplicação.

Agora, vamos para o código principal da aplicação:

Inicialização da Aplicação Flask

from flask import Flask

from flask\_sqlalchemy import SQLAlchemy

# Inicializa a aplicação Flask

app = Flask(\_\_name\_\_)

# Configura o banco de dados

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///todo.db'

app.config['SQLALCHEMY\_TRACK\_MODIFICATIONS'] = False

# Inicializa o objeto SQLAlchemy

db = SQLAlchemy(app)

# Importa as rotas

from app import routes

Este código fica no arquivo app/\_\_init\_\_.py. Ele inicializa a aplicação Flask, configura o banco de dados SQLite e importa as rotas.

Agora, vamos para o modelo de dados:

Modelo de Tarefa

from app import db

from datetime import datetime

class Task(db.Model):

    id = db.Column(db.Integer, primary\_key=True)

    title = db.Column(db.String(100), nullable=False)

    description = db.Column(db.String(200))

    created\_at = db.Column(db.DateTime, default=datetime.utcnow)

    completed = db.Column(db.Boolean, default=False)

    def \_\_repr\_\_(self):

        return f'<Task {self.id}: {self.title}>'

Este código fica no arquivo app/models.py. Ele define a estrutura da tabela de tarefas no banco de dados.

Agora, as rotas da aplicação:

Rotas da Aplicação

from flask import render\_template, request, redirect, url\_for

from app import app, db

from app.models import Task

@app.route('/')

def index():

    tasks = Task.query.all()

    return render\_template('index.html', tasks=tasks)

@app.route('/create', methods=['GET', 'POST'])

def create():

    if request.method == 'POST':

        title = request.form['title']

        description = request.form['description']

        new\_task = Task(title=title, description=description)

        db.session.add(new\_task)

        db.session.commit()

        return redirect(url\_for('index'))

    return render\_template('create.html')

@app.route('/update/<int:id>', methods=['GET', 'POST'])

def update(id):

    task = Task.query.get\_or\_404(id)

    if request.method == 'POST':

        task.title = request.form['title']

        task.description = request.form['description']

        task.completed = 'completed' in request.form

        db.session.commit()

        return redirect(url\_for('index'))

    return render\_template('edit.html', task=task)

@app.route('/delete/<int:id>')

def delete(id):

    task = Task.query.get\_or\_404(id)

    db.session.delete(task)

    db.session.commit()

    return redirect(url\_for('index'))

Este código fica no arquivo app/routes.py. Ele define as rotas da aplicação e a lógica para criar, ler, atualizar e deletar tarefas.

Agora, vamos para os templates HTML:

Template Base HTML

<!DOCTYPE html>

<html lang="pt-br">

<head>

    <meta charset="UTF-8">

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

    <title>{% block title %}Gerenciador de Tarefas{% endblock %}</title>

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

</head>

<body>

    <header>

        <h1>Gerenciador de Tarefas</h1>

        <nav>

            <ul>

                <li><a href="{{ url\_for('index') }}">Início</a></li>

                <li><a href="{{ url\_for('create') }}">Nova Tarefa</a></li>

            </ul>

        </nav>

    </header>

    <main>

        {% block content %}

        {% endblock %}

    </main>

    <footer>

        <p>&copy; 2024 Gerenciador de Tarefas</p>

    </footer>

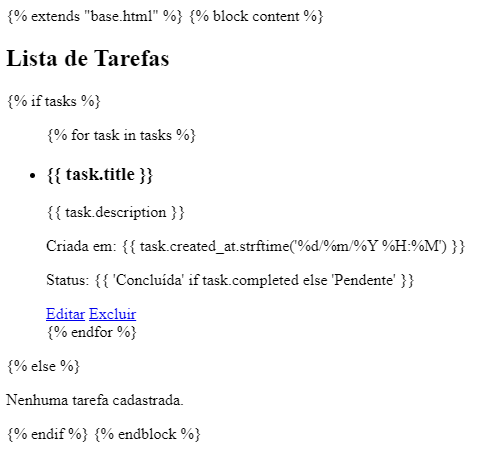
</body>

</html>

Este é o template base (templates/base.html) que será estendido por outros templates.



Template da Página Inicial



{% extends "base.html" %}

{% block content %}

<h2>Lista de Tarefas</h2>

{% if tasks %}

    <ul class="task-list">

    {% for task in tasks %}

        <li>

            <h3>{{ task.title }}</h3>

            <p>{{ task.description }}</p>

            <p>Criada em: {{ task.created\_at.strftime('%d/%m/%Y %H:%M') }}</p>

            <p>Status: {{ 'Concluída' if task.completed else 'Pendente' }}</p>

            <a href="{{ url\_for('update', id=task.id) }}">Editar</a>

            <a href="{{ url\_for('delete', id=task.id) }}" onclick="return confirm('Tem certeza que deseja excluir esta tarefa?');">Excluir</a>

        </li>

    {% endfor %}

    </ul>

{% else %}

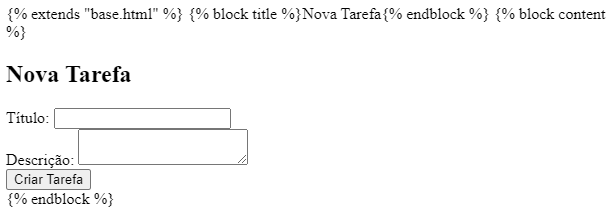
    <p>Nenhuma tarefa cadastrada.</p>

{% endif %}

{% endblock %}

Este é o template da página inicial (templates/index.html), que lista todas as tarefas.

Template de Criação de Tarefa



{% extends "base.html" %}

{% block title %}Nova Tarefa{% endblock %}

{% block content %}

<h2>Nova Tarefa</h2>

<form method="POST">

    <div>

        <label for="title">Título:</label>

        <input type="text" id="title" name="title" required>

    </div>

    <div>

        <label for="description">Descrição:</label>

        <textarea id="description" name="description"></textarea>

    </div>

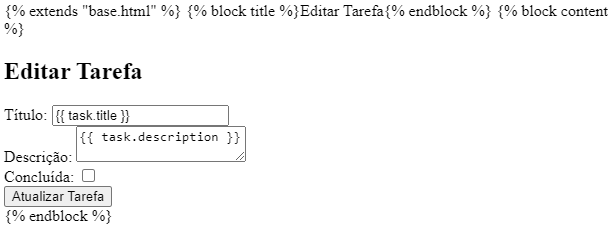
    <button type="submit">Criar Tarefa</button>

</form>

{% endblock %}

Este é o template para criar uma nova tarefa (templates/create.html).

Template de Edição de Tarefa



{% extends "base.html" %}

{% block title %}Editar Tarefa{% endblock %}

{% block content %}

<h2>Editar Tarefa</h2>

<form method="POST">

    <div>

        <label for="title">Título:</label>

        <input type="text" id="title" name="title" value="{{ task.title }}" required>

    </div>

    <div>

        <label for="description">Descrição:</label>

        <textarea id="description" name="description">{{ task.description }}</textarea>

    </div>

    <div>

        <label for="completed">Concluída:</label>

        <input type="checkbox" id="completed" name="completed" {% if task.completed %}checked{% endif %}>

    </div>

    <button type="submit">Atualizar Tarefa</button>

</form>

{% endblock %}

Este é o template para editar uma tarefa existente (templates/edit.html).

Agora, vamos para o CSS:

Estilos CSS da Aplicação

body {

    font-family: Arial, sans-serif;

    line-height: 1.6;

    margin: 0;

    padding: 0;

    background-color: #f4f4f4;

}

header {

    background-color: #333;

    color: #fff;

    text-align: center;

    padding: 1rem;

}

nav ul {

    padding: 0;

    list-style-type: none;

}

nav ul li {

    display: inline;

    margin-right: 10px;

}

nav ul li a {

    color: #fff;

    text-decoration: none;

}

main {

    padding: 2rem;

}

.task-list {

    list-style-type: none;

    padding: 0;

}

.task-list li {

    background-color: #fff;

    margin-bottom: 1rem;

    padding: 1rem;

    border-radius: 5px;

    box-shadow: 0 2px 5px rgba(0,0,0,0.1);

}

form div {

    margin-bottom: 1rem;

}

label {

    display: block;

    margin-bottom: 0.5rem;

}

input[type="text"], textarea {

    width: 100%;

    padding: 0.5rem;

    border: 1px solid #ddd;

    border-radius: 3px;

}

button {

    background-color: #333;

    color: #fff;

    padding: 0.5rem 1rem;

    border: none;

    border-radius: 3px;

    cursor: pointer;

}

button:hover {

    background-color: #555;

}

footer {

    text-align: center;

    padding: 1rem;

    background-color: #333;

    color: #fff;

    position: fixed;

    bottom: 0;

    width: 100%;

}

@media (max-width: 600px) {

    header {

        padding: 0.5rem;

    }

    main {

        padding: 1rem;

    }

    .task-list li {

        padding: 0.5rem;

    }

}

Este CSS fica no arquivo static/css/style.css e define o estilo visual da aplicação.

Agora, para iniciar o projeto e chegar até o deploy, siga estes passos:

1. Instale o Python (versão 3.7 ou superior) em seu sistema.
2. Crie um novo diretório para o projeto e navegue até ele no terminal.
3. Crie um ambiente virtual:

python -m venv venv

1. Ative o ambiente virtual:
   * No Windows: venv\Scripts\activate
   * No macOS/Linux: source venv/bin/activate
2. Instale as dependências:

pip install flask flask-sqlalchemy gunicorn

1. Crie a estrutura de diretórios e arquivos conforme mostrado no primeiro artefato.
2. Copie os códigos dos artefatos para seus respectivos arquivos.
3. Execute o aplicativo localmente para testes:

python run.py

1. Crie um arquivo Procfile na raiz do projeto com o seguinte conteúdo:

web: gunicorn run:app

1. Crie um arquivo requirements.txt:

pip freeze > requirements.txt

1. Inicialize um repositório Git e faça o commit de seus arquivos.
2. Crie uma conta no Heroku (se ainda não tiver uma).
3. Instale o Heroku CLI e faça login:

heroku login

1. Crie um novo app no Heroku:

heroku create nome-do-seu-app

1. Configure as variáveis de ambiente no Heroku:

heroku config:set FLASK\_APP=run.py

heroku config:set FLASK\_ENV=production

1. Faça o deploy da aplicação:

git push heroku main

1. Execute as migrações do banco de dados:

heroku run python

>>> from app import db

>>> db.create\_all()

>>> exit()

1. Abra a aplicação no navegador:

heroku open

Seguindo esses passos, você terá sua aplicação Flask de gerenciamento de tarefas funcionando localmente e implantada no Heroku.

Lembre-se de que este é um projeto básico e pode ser expandido com mais funcionalidades, melhorias de segurança e otimizações de desempenho conforme necessário.

1. <https://mermaid.live/view#pako:eNpVUc1ygjAQfpXMnsGBgpZw6IyKqB09tDA9NHjIkKiMQBgSprWOT9NDH8QXawp0WnLK95Nvs7sXSAXj4MOhptURxUFSIn2mZFrlWUpvX7dPgWJhBgJtMql2yDQf0IyEtSgVL9mut7f0nMxoevpjZy0bkJgXVU4Vl2gVbzcDcUEWUmW5kGgeRb0yb5WQhDmVJzStqp4PW35JnoWicsCtyFY3oVMG7Jq8ZPxNDlIf9RfLVCDGUUCZLhs9bTLFe8-q83Rg2YV0YN21AgYUvC5oxvTELj9SAurIC56Ar6-M72mTqwSS8qqttFEiOpcp-KpuuAG1aA5H8Pc0lxo1FdMzCTKqJ1_8WipavgrxH4J_gXfwHQuP7u6died6Y9tzbdeAM_gmHlkYe67lTizHdhwbXw34aAPskWW7Y2-CMbbdCdavDOAsU6Ledgtv9379BmKVlMY> [↑](#footnote-ref-0)
2. <https://mermaid.live/view#pako:eNpdkltLwzAUx79KOE-bdHW9ka6I4FZBwYFuwwfXPYTluAXSpKYputt3N91F1Lwk5-R_fjmX7GCpOUIGK8OqNZnlhSJu3c1HUqCyuCC93u3-YTZ7JhP8aLC2ezLsjLSyRkuJhvSI0Y3F2q823VPs8Bgy1VIshWWEM67rPRnNx-4h6fRlu7f6xUk_OuonaLVRP_Lhb9YEFUcjtoxYLCvJLO5JPn8V-OlwF1d9feWvbSnP1PwP9WE2fvoHPddUV1rVjnd3SeamvXx0pRu2cv77eWfI1FITjiRvcyPTlydhsbsAD0o0JRPctW_Xhhdg11hiAZk7cnxnjbQFFOrgpKyxerpRS8isadAD17XVGrJ3JmtnNRV3JeSCuTGUF0nF1JvWv03IdvAFWUgjP00iGvVTGsQBjWMPNpD1Qp-mg4gGSZhE4aAfUHrwYHtEBH4_iJOU9qMoHiRhGoYeIBeuO-PT_I_f4PANxxOj5Q> [↑](#footnote-ref-1)
3. <https://mermaid.live/view#pako:eNplkcFqhDAQhl8lzKmlImnUVXMse-mhp91TESQks25oTCRG6FZ890Z3C0Lnkn--fxhmJjNIpxA4SCPG8ahF50XfWBJjI-Qsxi8y38kaL-82YIeeaLWDp-C17UjQweB_rHCUXg9BO7szjyLgWfdIpMcoVSvCzn1zzqCwRLp-MBjtnde2Hgfftk_Pd7hAAj36XmgVN9mGbSBcsccGeJQKL2IyoYHGrqViCu50sxJ48BMm4N3UXYFfhBljNg0qTvO4xF_JIOync_sU-AzfwOs8zTNGizLLD0XBWJbADThLXw9VXeSsqlhB8yiXBH62BjSNPKdZVZasqDJK6wRQ6eD8x-Mj1mf5BcToepQ> [↑](#footnote-ref-2)
4. <https://mermaid.live/view#pako:eNplkk1u2zAQha9CzFoxYtqSZS4CxNImQIU2DuRFoM1AGicEJFLlD1rH8GGKLnoQX6yUWbcFzBVnvvcwjz9HaHVHIMDSV0-qpVLim8GhUSysEY2TrRxROVYztKy2_vzDSH2LdxPeSfp2i4oJFVo5o_uezK2gmgRVyNHfsnIzwQ2qVrOOWImdto2Kuvru4WEn2JNyZPD86_xTs1YPsd5jS1G1C6pCsG04oLTyIougCKAS7PP4111s6zKyKrByI9izJ3NgL8-fYrvc3EXTlqzvXQhzlcchl3jsi9EtWRujxkkxaXCN2jq8BgvdWrBH57GXH38ydPjvAJDAQGZA2YUXOk6uBtw7DdSACNuO9hhSNNCoU5Cid_rloFoQznhKwGj_9g5ij70NlR87dNfnvUrCFb9q_X8J4gjfQSz4bMXXy1XOs_kizddpAofQXc6y9ZIvF5zzdMWz7JTAx8V_P1uv8nma3Wd5OufzLF8kQJ102lTxf12-2ek3w_XJgw> [↑](#footnote-ref-3)
5. <https://mermaid.live/view#pako:eNpNkU1uwjAQRq8SzTqgBBKnyaJSCb9tqSqVVR0WI2yCpcRGxkGliMNwgJ6Ci9WB0MYrz_eeRpqZI6wU45BArnG7cRbDTDr2PdH3yzkXEpdOp_M4oFOOjOtlA-sspXMU0kmVNFyaNhnSsVLmzx7U2YguLj-mKlQ7HNM33IscjVCyydM6n9BXsTPoMO4sUPM17tp0atvrsiouZy3UnUxqMqM33ZlJJvaCVVg0eFrjZ5pqgbrp2SYvdMSE-UfgQsl1iYLZxRxrMQOz4SXPILFfZp2qMBlk8mRVrIz6OMgVJEZX3AWtqnwDyRqLna2qLUPDhwLtgsu7skX5qVS7hOQIX5B0wi4JSBxGQRT0iOcT4sIBkl7c7fsk8iI_6hHyEAfhyYXvawevG4dB4PVjLw5J7Pd9F7gdRun57a7X855-AVe3lOk> [↑](#footnote-ref-4)
6. <https://mermaid.live/view#pako:eNpNksFugkAQhl9lM2c0ggjKoY0CGg9tmtZeCh5WGHQTYc2yJFr0YZoeeuqpj8CLdQFt2cvuv_nmz7-zU0LEYwQHtoIedmTlhRlRaxoss-o7YpzEPCcrzCXma9Lr3ZFZ4PIsYdtC0Oqr-qwBMk03DDOJJMaWXbcus6bCDfwjRsWNvtmR14zJ6kMwnl9xt8G9csVr6h98onlOBU3vLy3n1dz5haVn4gcL7AR5xj2V1Y-ocyNx-QaFVEHX3bpHxZ7JPFjGKjJLWHQtr3kh8OqFxBfiL9m8fUgr_EYsgjlLu90BDVIUKWWx6mZZoyHIHaYYgqOOMSa02MsQwuyiUFpI_nLKInCkKFADwYvtDpyE7nOlikNMJXqMql9Jb8iBZm-cdyU4JRzBGU70vmmZhmXo-mikNg1O4Ohm3zAsc2xZumHrljE2Lxq8NwaD_mQ4sdStPR4MbNuYDDXAmEkuHtphaGbi8gu6M7K5> [↑](#footnote-ref-5)
7. <https://mermaid.live/view#pako:eNplkt9u2jAUxl_F8sWuaBSS0EAuJtGEAi3VOnXaxRIuvOQQjpbYmWNPUMTDTLvYE-wJeLEZE0RRLVnysX7fdz7_2dFcFEAjWkrWrMmXJOPEjHH6LKFhkh3-Hv4IclyRBJpKbGvgaklubj6SuzSW2AGFIM9S5CusYNk5WCZO57xVrLpgU80xF5JfYcm1lYSfGiUcW7WO2qgrdpLGgq-w1PIicEpUWHIhz93vyAcSm5mYObGy-3dJZiDFD03ixbwT3Vtwep1l3DSEn-EOnFpwlo4LvBxftKqU8PJ50UEzC83fpWUt-cokHn7_AmxJAWRcf0dz0nP0udU9pKfrNryJUGF-knfMg2Ue08kGcn3xfcLStvkHbQc-WnCRvrEgn3iFHJa0R2uQNcPCvP7uiGdUrc2lZzQyywJWTFcqoxnfG5RpJV62PKeRkhp6VApdrmm0YlVrKt0UTEGCzPSvz0jD-Dch3pY02tENjbwgdNzb4XAw6AfDwHddv0e3NPL7jh8Enhf6o4Hb9269cN-jr9bBdUb-yGz0w6HrhqE3MgooUAn5dPq99hPv_wN0xvE5> [↑](#footnote-ref-6)
8. <https://mermaid.live/view#pako:eNpdks1u4jAQx1_FmsOeAksKIR-HlUICbVqqZYG9rMPBm5hirR1HsVOVIh6m2kNPe-oj5MXqNVRa1tJInvn_ZmzP-ACFLClE8NCQeofWaV4hs2KcVd2fgskN6vW-oAmedC-KFRKVFM04Ub82J25i5QQny-8pKqRAq2_zmBc7KvZnIrFEitdU1JxoqhBFU6UZZ8-ke-1-yzOXWm5qOGWgc3BqgzOc0prLvaCVPgszK1wfFk339sSEVGhBlJLqeJKvrXyD41abHFZcnHQSMxwvMrScrtbbll8ot3jWyMoklih-JFX3SsrL1DucZJ-T9CI2x181E_-96QZ9QpmxW2N3xuYWvcdxzT_uhJbyZ6s0sV0pCO9eHinfgAOCNoKw0kzm8LdYDto0leYQmW1Jt6TlOoe8OhqUtFqu9lUBkW5a6kAj24cdRFvClfHaujRtTxkxExYfSE2qH1L-60J0gCeIfK8_DPxg4I7cwPfdMHRgD1HP6195wzD0vMAfjMaDq-HRgWdbwO0b1gvGYRi6o3EYeK4DtGRaNvenj2X_1_EdA0DAcw> [↑](#footnote-ref-7)