UTILIZAÇÃO DA LINGUAGEM PYTHON COM O MICROFRAMEWORK FLASK PARA APLICAÇÕES WEB

Ezequiel Marques Ramos

Universidade do Estado de Santa Catarina
Desenvolvimento de Aplicações na Web
Professora Débora Cabral Nazário

RESUMO

Python é umas das cinco linguagens de programação mais populares segundo o índice TIOBE em Setembro de 2017. Neste artigo é apresentado uma aplicação WEB utilizando Python com o microframework Flask em conjunto com o banco de dados SQLite. Os tópicos abordados partirão desde um breve histórico, a instalação das ferramentas até a aplicação em si dividida em pequenos passos.

Palavras Chave: Python, Flask, SQLite3, Desenvolvimento Web.

ABSTRACT

Python is one of the five most popular programming languages according to the TIOBE index in September 2017. In this article we present a WEB application using Python with the Flask microframework running with a SQLite3 database. The topics covered will start from a brief history, the installation of the tools until the application itself divided into small steps.

Keywords: Python, Flask, SQLite3, Web Development.

1. INTRODUÇÃO

Este artigo apresentará como construir uma aplicação Web em Python utilizando o microframework Flask. A aplicação final conterá um formulário HTML, para inserção de registros e uma listagem para consulta dos registros inseridos. A parte de back-end receberá parâmetros dinâmicos direto da tela para fazer as inserções. A inserção dos dados será realizada utilizando o banco de dados SQLite3, nativo do Python. Com isto, poderá ser analisado o nível de complexidade do desenvolvimento web Python utilizando a ferramenta Flask.

2. PYTHON

Python é uma linguagem de programação de alto nível, interpretada, orientada a objetos, funcional, de tipagem dinâmica e forte. Python foi lançado em 1991 por Guido van Rossum e foi projetada com a filosofia de enfatizar a importância do esforço do programador sobre o esforço computacional. Prioriza a legibilidade do código, na comunidade de desenvolvedores foi gerado o termo “Pythonico” que descreve um código com fácil leitura e exigir poucas linhas de código se comparado ao mesmo programa em outras linguagens

Devido às suas características, ela é principalmente utilizada para processamento de textos, dados científicos e páginas dinâmicas para a web.

3. FLASK

Flask é um microframework do Python para desenvolvimento, é baseado em outras duas ferramentas Python, Werkzeug que é um ferramenta de WSGI(Web Server Gateway Interface) e Jinja2 que é uma ferramenta para construção de templates HTML em Python.

4. INSTALAÇÃO

Muitas distribuições linux já incluem o Python instalado em seus sistemas operacionais. Porém, caso não possua o python instalado, executar no terminal:

```
$ sudo add-apt-repository
ppa:jonathonf/python-3.6
$ sudo apt-get update
$ sudo apt-get install python3.6
```

Para Windows e Mac utilizar o link para download disponibilizado diretamente no site oficial do Python(https://www.python.org/downloads/).

Tendo o Python instalado, ao executar no terminal:

```
$ python3.6
```

Deverá ser apresentado o console do Python contendo a informação da versão sendo executada. Digite “exit()” para sair do console.

Antes de instalar o Flask, é necessário se certificar que o módulo Pip do Python esta instalado. Tendo o Pip instalado, executado no terminal:

```
$ sudo python3.6 -m pip install flask
```

A instalação deverá ocorrer normalmente e sem erros. Todo o desenvolvimento a seguir considera que o Python e a ferramenta Flask estejam devidamente instalados.

5. DESENVOLVIMENTO APLICAÇÃO

5.1. “Hello World” em Flask

Para começar, criaremos um script simples que inicia um servidor web e retorna uma mensagem de “Hello World” renderizada no HTML. Conforme abaixo:

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello_world():
    return "Hello World!"

app.run(debug=True)
```

O script acima importa a classe `Flask` do módulo `flask`, cria o objeto `app` a partir desta classe(o primeiro argumento é o nome do módulo da aplicação) utiliza se a variável `__name__` quando usamos em uma aplicação de módulo único(como nos exemplo utilizados neste artigo). Utilizamos então o decorador `@app.route` para dizer ao Flask qual URL deve executar a função. É então defina uma função, que pode ter qualquer nome, mas deve ser escrita logo em seguida do decorador. A função retorna a *string* “Hello World” que será apresentada no navegador do usuário. Por fim, é executado o método `run` do objeto instanciado com a aplicação, este método é responsável por iniciar a aplicação, pode receber vários parâmetros, dentre eles o parâmetro `debug` que quando setado como verdadeiro deverá apresentar eventuais mensagens de erro ocorridas em tempo de execução diretamente no navegador para o usuário.

Para executar este programa, basta copiar o script acima, gravar em um arquivo como “hello.py” ou alguma coisa similar. Somente garanta que seu arquivo não seja salvo como “flask.py”, pois isto pode gerar um conflito com o próprio módulo flask. Tendo feito isto, basta abrir a pasta onde foi salvo o arquivo no terminal do seu sistema operacional e executar o comando:

```
$ python3.6 hello.py
 * Running on http://127.0.0.1:5000/
```

A aplicação vai ser executada e estará disponível ao acessar no navegador a partir da URL apresentada no terminal. Caso nenhum parâmetro

seja informado utilizará o padrão servidor 127.0.0.1(local), porta 5000.

5.2. Recebendo Parâmetros

Em aplicações web, é crucial que consigamos enviar parâmetros para que sejam adicionadas variáveis em nosso servidor que poderão ser utilizadas para alguma eventual tratativa. No exemplo a seguir, é apresentado o envio de parâmetros pelo padrão *Path Param*.

```
from flask import Flask
app = Flask(__name__)

@app.route('/sayYourName/<name>')
def sayYourName(name):
    return name

@app.route('/sayYourAge/<int:number>')
def sayYourAge(number):
    return str(number + 3)

app.run(debug=True)
```

A diferença entre esse exemplo e o anterior, é que foi definido na URL da rota uma marcação especial como `<name>`. Este mesmo nome utilizado na marcação é passado então como argumento para a função definida abaixo.

Como é possível perceber, a função "sayYourName" retorna a string recebida por parâmetro, ou seja, todo o conteúdo informado pelo parâmetro "name" deverá ser apresentado no navegador para o usuário. Caso a URL seja acessada como "http://127.0.0.1:5000/sayYourName/Teste", a mensagem de resposta no navegador deverá ser "Teste".

Ainda no mesmo exemplo, é utilizada uma outra função interessante deste tipo de marcação que é, além de definir um nome, definir um tipo específico para recebimento deste parâmetro. Os tipos básicos aceitos nesta marcação são: "string", "int" e "float". No exemplo, a função "sayYourAge" somente será executada se o parâmetro enviado for um número inteiro válido, somará então este número com 3 fará a conversão para *string*(a partir da chamada da função `str`) e retornará o dado para a aplicação que estiver acessando a URL. Caso a URL seja acessada como "http://127.0.0.1:5000/sayYourAge/5", a mensagem de resposta no navegador deverá ser "8".

Outra forma de enviar parametros é pelo padrão *Query Param* e *Payload,* que é apresentada no exemplo abaixo.

```
from flask import Flask, request
import json

app = Flask(__name__)

@app.route('/login', methods=['POST',
'GET'])
def login():
    if request.method == 'GET':
        user= request.args.get('user')
        password=
request.args.get('pass')

    if request.method == 'POST':
        dataDict=
json.loads(request.data)
        user = dataDict["user"]
        password= dataDict["pass"]

    return "user:" + user + " pass:" +
password

app.run(debug=True)
```

Neste exemplo, é adicionado um novo parâmetro, `methods`, no decorador de rota. Este parâmetro recebe uma lista de *strings* indicando todos métodos HTTPs que poderão acessar aquela URL. No exemplo, a URL "/login" só poderá ser acessada caso o método HTTP seja "GET" ou "POST". Os métodos mais comuns disponíveis são: "GET", "HEAD", "POST", "PUT", "DELETE" e "OPTIONS".

Dentro da função "login", é possível perceber que há uma condição a ser executada sobre o valor de uma propriedade de uma classe estática. Como pode se perceber no início do script, a classe `request` foi importada também do módulo `flask`. A propriedade `method` contém o valor do método HTTP utilizado para acessar esta URL.

Caso o método utilizado para acessar esta URL seja "GET" o script irá buscar os dados enviados por *Query Param* utilizando o método `request.args.get`, este método recebe um parâmetro *string* com o nome do parâmetro enviado na URL por *Query Param*. No exemplo, caso o usuário acesse a URL como "http://127.0.0.1:5000/login?user=HelloWorld&pass=123" utilizando o método HTTP "GET", deverá ser apresentado no navegador "user:HelloWorld pass:123".

Caso o método utilizado para acessar esta URL seja "POST" o script irá buscar os dados enviados por *Payload* utilizando o método `request.data`. No exemplo, o script considera que será enviado um objeto JSON com duas propriedades(*user* e *pass*). É feito então um parse do JSON através do módulo `json`, também importado no inicio do script, e armazenado na variável `dataDict`, os dados então poderão ser acessados normalmente.

5.3. Templates

Conforme mencionado anteriormente, o retorno das funções roteadas são diretamente renderizadas no formato HTML para o navegador. Então considerando que estas aplicações irão crescer e queiramos ter um *output* que seja amigável para o usuário, precisaremos utilizar as tags HTML.

Seguindo esta definição, temos um exemplo simples de um *output* HTML abaixo:

```
from flask import Flask
app = Flask(__name__)

@app.route('/sayYourName/<name>')
def sayYourName(name):
    return '''<html>
<head>
    <title>''' + name + '''</title>
</head>
<body>
    <h1>
        Hello, ''' + name + '''
    </h1>
</body>
</html>'''

app.run(debug=True)
```

Este exemplo não mostra nada além do que já foi visto nos exemplos anteriores, porém é perceptível que o código começou a ficar um pouco bagunçado. Considere o quão complexo o código viria a se tornar caso fosse necessário retornar uma página HTML maior e mais complexa, com várias lógicas embutidas.

Este script irá funcionar perfeitamente, porém, conforme é concordado entre a comunidade de desenvolvimento Python, este não é um código "Pythonico" e claramente, não é uma opção escalável. É neste momento então que passamos a utilizar os *Templates*.

A idéia dos templates é deixar a lógica da aplicação separada do *layout*/*output* HTML. Esta separação é interessante também nos casos onde uma pessoa é responsável somente pelo design(web designer) da aplicação enquanto outra pessoa é responsável somente pelo comportamento da aplicação(desenvolvedor back-end).

Os *templates* serão arquivos com extensão ".html" e deverão estar dentro de uma pasta que deverá ser criada no mesmo diretório onde está o script da aplicação, o nome desta pasta deve ser obrigatoriamente "templates".

Criaremos então para nosso exemplo, dois templates, sendo eles:

index.html:

```
{% if user and password %}
<p>
Usuario criado com sucesso.
<br>
Usuario:{{user}} Senha:{{password}}
</p>
{% endif %}

<form action="/index" method="get">
    Usuário:
    <input type="text" name="user">
    <br>
    Senha:
    <input type="text" name="pass">
    <br>
    <br>
    <input type="submit">
</form>
```

list.html:

```
<h1>Users List:</h1>
<table border=1>
    <tr>
        <th>user</th>
        <th>password</th>
    </tr>

    {% for user in users %}
    <tr>
        <td>{{ user.user }}</td>
        <td>{{ user.password }}</td>
    </tr>
    {% endfor %}

</table>
```

Como é possível perceber, os arquivos criados estão escrito basicamente no padrão HTML exceto pela diferença que existem alguns espaços reservados para conteúdos dinâmicos incluídos nas seções com "`{{ }}`" e "`{% %}`".

Antes de continuar com esta explicação, vamos voltar a escrever a aplicação. Então teremos:

```
from flask import Flask,
render_template
app = Flask(__name__)

@app.route('/')
@app.route('/index')
def index():
    user = request.args.get('user')
    password = request.args.get(
'pass')
    if user and password:
        users.append({
            "User":user,
            "Password": password
        })
    return render_template(
'index.html',user=user,password=passwo
rd)

@app.route('/list')
def list():
    return render_template(
'list.html',users=users)

users = []

app.run(debug=True)
```

Para importar os *templates,* é preciso importar uma nova função do flask chamada `render_template`. Como é possível perceber no script, esta função recebe como parâmetro o nome do arquivo de um template e uma lista de variáveis que poderão ser utilizadas como argumentos dentro *template* renderizado.

A função `render_template` invoca o Jinja2, ferramenta para construção de templates que faz parte do framework flask. O Jinja2 então irá substituir os blocos "`{{ }}`" com os valores correspondentes enviados por parâmetro na função `render_template`, renderizando os valores diretamente no HTML. Os blocos "`{% %}`" definem uma declaração de controle(*Control Statement*), dentro destes blocos é possível escrever pequenos scripts com condições, fazendo com que parte do template só seja renderizada se satisfizer a condição, ou estruturas de repetições, fazendo com que parte do template seja renderizada repetidas vezes conforme a regra de repetição. Ambos os casos podem ser consultados nos templates "index.html" e "list.html" escritos anteriormente.

O exemplo contém três rotas definidas, duas delas("/" e "/index") devem executar a função index que caso não receba parâmetros por *Query Param,* deverá retornar para o usuário o template index.html que contém um formulário simples com dois campos(usuário e senha) renderizado. Caso a URL receba os parâmetros "user" e "pass" por *Query Param,* a função então irá adicionar essa informação a uma lista de "users" declarada no escopo global do script e renderizar o mesmo template, porém o mesmo agora irá apresentar uma mensagem informativa avisando que o usuário foi criado com sucesso.

A outra rota é declarada para URL "/list", esta rota deverá renderizar o template list.html passando por parâmetro a lista de usuários, o *template* deverá apresentar uma tabela com uma linha para cada usuário criado utilizando o formulário descrito anteriormente.

5.4. Conexão Banco de Dados SQLite3

Seguindo a mesma ideia do exemplo anterior, vamos agora, ao invés de gravar a lista de usuários na memória do programa através de uma variável, gravar diretamente em um banco de dados, para isto, será utilizado o banco de dados SQLite3, nativo do Python.

Para utilizar o SQLite3, basta importa-lo da mesma forma que é importado o flask. Tendo Importado o módulo `sqlite3`, é necessário executar o método `connect` que conecta ao banco de dados. Esse método recebe um parâmetro que é o nome do arquivo de banco de dados, caso o arquivo não exista ainda, o arquivo é criado automaticamente.

O método `connect` retorna então um objeto de conexão, tendo este objeto, é possível então criar um objeto Cursor e chamar seu método `execute` para executar comandos SQL.

Todos os dados inseridos e *commitados* deverão então ser persistidos no arquivo definido na conexão e estarão disponíveis nas próximas execuções do script.

A seguir, o mesmo exemplo anterior, agora utilizando um banco de dados para armazenamento das informações de usuário:

```
import sqlite3
from flask import Flask, request,
render_template

conn = sqlite3.connect('test.db')
c = conn.cursor()

c.execute("SELECT name FROM
sqlite_master WHERE type='table' AND
NAME='users' ")

if not c.fetchone():
    c.execute("CREATE TABLE users(user
text, password text)")

conn.close()

app = Flask(__name__)

@app.route('/')
@app.route('/index')
def index():

    user = request.args.get('user')
    password= request.args.get('pass')

    conn = sqlite3.connect('test.db')

    if user and password:
        c = conn.cursor()
        c.execute("INSERT INTO users
(user, password)
VALUES(?,?)",(user,password))
        conn.commit()

    conn.close()
    return render_template(
'index.html',user=user,password=passwo
rd)

@app.route('/list')
def list():
    conn = sqlite3.connect('test.db')
    c = conn.cursor()
    c.execute("SELECT * FROM users")
    users = c.fetchall()
    conn.close()

    return render_template(
'list.html',users=users)

app.run(debug=True)
```

O script acima executa logo no início de sua execução uma *query* no banco buscando na tabela "sqlite_master" um registro do tipo "table" e nome "users". Esta tabela é uma tabela nativa dos bancos SQLite, e armazenam informações de metadados(sobre o próprio banco de dados), no caso, essa query é executada para saber se essa tabela já existe no banco. Na linha a seguir, é possível perceber que caso não seja encontrado um registro, é executado uma query para criação da tabela "users" no banco.

As outras diferenças vão aparecer na função "index" que contém o formulário de criação de usuário, onde, ao receber valores por parâmetro, ao invés de ser incluído um novo dado numa lista do Python, é feita a inserção do dado direto no banco de dados. Na função "list" é executado uma seleção de todos os registros da tabela "users", o parâmetro será enviado para o template da mesma forma do exemplo anterior.

6. CONCLUSÃO

Tendo em vista o script gerado para realização da aplicação com todos recursos que ela oferece e a instalação das ferramentas, conclui se que é bastante simples desenvolver uma aplicação web utilizando Python com microframework Flask, tanto para iniciantes quanto, principalmente, desenvolvedores que já tenham tido alguma experiência com desenvolvimento web.

Quanto a sintaxe da ferramenta Flask, se mostra bem simples de utilizar e considerada pela comunidade de desenvolvedores Python a ferramenta mais Pythonica. Isto porque o Flask foi escrito bem depois da maioria das ferramentas com mesmo propósito, e aproveitou para acertar onde outras ferramentas erraram e se aproveitar dos feedbacks da comunidade. Para aplicações pequenas e simples, o Flask resolve o problema com bastante facilidade de desenvolvimento. Para aplicações maiores e mais complexas, a comunidade de desenvolvedores sugere outras ferramentas para Python como o Django, por exemplo. Porém, o Flask também ter soluções bem inteligentes para este tipo de situação.

7. REFERÊNCIAS BIBLIOGRÁFICAS

Welcome to Python.org. Disponível em: https://www.python.org/. Último acesso: 03/10/2017

Welcome | Flask (A Python Microframework). Disponível em: http://flask.pocoo.org/. Último acesso: 03/10/2017

Flask - Full Stack Python. Disponível em: https://www.fullstackpython.com/flask.html. Último acesso: 03/10/2017

Whats is pythonic? | Secret Weblog. Disponível em: https://blog.startifact.com/posts/older/what-is-pythonic.html. Último acesso: 03/10/2017

The Flask Mega-Tutorial, Part I: Hello, World! - miguelgrinberg.com. Disponível em: https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world. Último acesso: 03/10/2017

TIOBE Index | TIOBE - The Software Quality Company. Disponível em: https://www.tiobe.com/tiobe-index/. Último acesso: 03/10/2017

Programação Orientada a Objetos com Python. Disponível em: http://www.devmedia.com.br/introducao-ao-desenvolvimento-web-com-python/6552. Último acesso: 03/10/2017

11.13. sqlite3 — DB-API 2.0 interface for SQLite databases — Python 2.7.14 documentation. Disponível em: https://docs.python.org/2/library/sqlite3.html. Último acesso: 03/10/2017