1 Introdução a frameworks web

Resumo: *Bem-vindo! Você está prestes a começar uma jornada para aprender como criar aplicativos web. Nesta primeira parte do curso, vamos aprender conceitos básicos da estrutura HTTP e apresentar algumas referências associadas a frameworks como* `Node.js` *e* `Flask`*. Ao final desta introdução, esperamos que você tenha um aplicativo web simples em execução no seu computador!*

Os primeiros passos

Vamos iniciar a jornada de Programação Web Avançada implementando o protocolo HTTP a partir de algumas bibliotecas disponíveis em python. Nosso objetivo é construir um framework que nos auxiliará a melhor entender a dinâmica de um sistema Web.

O Python fornece funcionalidades básicas para implementar o protocolo HTTP no módulo `http.server`. **Esse é um módulo disponível apenas para fins didáticos e/ou quando estamos trabalhando em ambiente de desenvolvimento**. Portanto, ele não é recomendado para implementação em ambiente de produção!

Basicamente, o módulo `http.server` cria e escuta no protocolo HTTP, despachando as requisições para um manipulador. O código para criar e executar o servidor se parece com isso:

```
# my_webserver.py

import http.server as http_server

class MyWebServer():
    def __init__(self, http_handler = http_server.SimpleHTTPRequestHandler) -> ↩
        None:
        self.http_handler = http_handler

    def run(self, host = "localhost", port = 3001):
        httpd = http_server.HTTPServer((host, port), self.http_handler)
        print(f"Servidor Web rodando em {host}:{port}")
        httpd.serve_forever()
```

A classe `http_server.SimpleHTTPRequestHandler` mapeia diretamente a estrutura de diretórios (se necessário), além de fornecer as definições dos **verbos** para as solicitações HTTP - por exemplo, `GET`, `POST`, `PUT`, `DELETE`, etc.

A classe `http_server.HTTPServer` armazena o endereço do servidor como variáveis de instâncias chamadas `server_name` e `server_port`. O servidor é acessado pelo manipulador (*handler*), normalmente por meio da variável de instância definida.

Para manipularmos os métodos HTTP precisamos implementar funções, tais como `do_METHOD`. A seguir, apresentamos o código proposto em que definimos a função `do_GET`.

```
# main.py

from my_webserver import MyWebServer
from http.server import SimpleHTTPRequestHandler

import os

PORT = (os.getenv('PORT') or 3001) # port 3001 by default

class ManuseioHttpRequest(SimpleHTTPRequestHandler):
    def do_GET(self):
        if self.path == "/":
            self.send_response(200) # Send HTTP code
            self.send_header("Content-Type", "text/html; charset=utf-8") # Send↩
                Headers
            self.end_headers() # Fim da definicao do Headers
            self.wfile.write("<p>Ola Mundo!</p>".encode()) # HTTP body

        elif self.path == "/pagina1":
            self.send_response(200) # Send HTTP code
            self.send_header("Content-Type", "text/html; charset=utf-8") # Send↩
                Headers
            self.end_headers() # Fim da definicao do Headers
            self.wfile.write("<p>Essa seria uma rota para uma pagina!</p>".↩
                encode()) # HTTP body

        elif self.path == "/index":
            self.send_response(200) # Send HTTP code
            self.send_header("Content-Type", "text/html; charset=utf-8") # Send↩
                Headers
            self.end_headers() # Fim da definicao do Headers
            res_body = open('index.html',"r").read().format_map({
                    "PORT":PORT
            })
            self.wfile.write(res_body.encode()) # HTTP body

        else:
            self.send_error(418)

app = MyWebServer(ManuseioHttpRequest)
```

```
if __name__ == "__main__":
    app.run()
```

Um modelo de arquivo `.html` também será utilizado nesse exemplo introdutório. Note que neste caso, a classe `SimpleHTTPRequestHandler` irá renderizar o arquivo `index.html`, tal que o nosso servidor simplesmente retorna o conteúdo daquele arquivo. E, caso façamos alguma alteração no interior do arquivo `index.html`, a página será renderizada após ser recarregada no servidor.

```
# index.html

<!DOCTYPE html>
<html>
<head>
    <title>Aula Web Server</title>
</head>

<body>
<h1>Nosso servidor HTTP esta rodando na Porta: {PORT}</h1>
</body>
</html>
```

Embora implementar um manipulador de solicitação HTTP personalizado possa fornecer *insights* valiosos sobre como os servidores da web operam e como estruturas como `Flask`, `Node.js`, `Jinja` ou `Django` funcionam internamente, para a maioria dos propósitos práticos usar um framework já estabelecido e testado geralmente é a melhor escolha.

O `SimpleHTTPRequestHandler` é ideal para fins de teste, prototipagem e aprendizado devido à sua simplicidade e facilidade de uso. Ele permite a configuração rápida de um servidor HTTP básico que pode servir arquivos estáticos e manipular solicitações `GET` simples. Além disso, ele fornece uma plataforma direta para experimentar conceitos de desenvolvimento web sem a complexidade adicional de implementar um manipulador de solicitação personalizado.

No entanto, é importante observar que, embora o `SimpleHTTPRequestHandler` seja adequado para teste e aprendizado, ele não é a solução ideal para um servidor em ambiente de produção. Tais ambientes normalmente exigem recursos mais avançados, como segurança, escalabilidade, otimização de desempenho e suporte para vários métodos HTTP além de apenas solicitações GET. Nesses casos, usar uma estrutura web madura e rica em recursos, que são projetados especificamente para criar aplicativos web robustos, é sempre a abordagem recomendada.

Flask - Parte I: Olá Mundo!

Estamos prestes a começar uma jornada para aprender como criar aplicativos web com Python e o framework Flask. Também veremos nesse curso as estruturas Web a partir do Node.js, porém no contexto de comparação entre propostas de desenvolvimento.

Nessa primeira parte iremos aprender como configurar um projeto Flask. Ao final, teremos um aplicativo web Flask simples em execução no seu computador!

Se você não tiver o Python instalado no seu computador, vá em frente e instale-o agora. Se o seu sistema operacional não fornecer um pacote Python, você pode baixar um instalador do site oficial do Python. Se você estiver usando o Microsoft Windows observe que não há uma versão nativa do Python nesse sistema operacional. Os demais sistemas baseados em Unix (Linux) e Mac OS X (IOS) já contém o Python instalado.

Para garantir que sua instalação do Python esteja funcional, você pode abrir uma janela de terminal e digitar python3 (Linux) ou py (Windows), ou se isso não funcionar, apenas python. Aqui está o que você deve esperar ver:

```
Python 3.12.3 (main, Jan 17 2025, 18:03:48) [GCC 13.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

O interpretador Python agora está esperando em um prompt interativo, onde você pode inserir instruções Python. Para sair do prompt interativo, você pode digitar exit() e pressionar Enter. Nas versões Linux e Mac OS X do Python, você também pode sair do interpretador pressionando Ctrl-D. No Windows, o atalho de saída é Ctrl-Z seguido por Enter.

O próximo passo é instalar o Flask, mas antes de entrar nisso, quero falar sobre as melhores práticas associadas à instalação de pacotes Python.

No Python, pacotes como o Flask estão disponíveis em um repositório público, de onde qualquer um pode baixá-los e instalá-los. O repositório oficial de pacotes Python é chamado PyPI, que significa **Python Package Index** (algumas pessoas também se referem a esse repositório como *"cheese shop"*). Instalar um pacote do PyPI é muito simples, porque o Python vem com uma ferramenta chamada pip que faz esse trabalho.

Para instalar um pacote na sua máquina, você usa o pip da seguinte forma:

```
pip install <package-name>
```

Curiosamente, esse método de instalação de pacotes não funcionará na maioria dos casos. Se seu interpretador Python foi instalado globalmente para todos os usuários do seu computador, é provável que sua conta de usuário regular não tenha permissão para fazer modificações nele, então a única maneira de fazer o comando acima funcionar é executá-lo de uma conta de administrador. Mas mesmo sem essa complicação, considere o que acontece quando você instala um pacote dessa forma. A ferramenta pip vai baixar o pacote do PyPI e, em seguida, adicioná-lo à sua instalação do Python. Desse ponto em diante, todo *script* Python que você tiver em seu sistema terá acesso a esse pacote.

Para resolver o problema de manter diferentes versões de pacotes para diferentes aplicativos, o Python usa o conceito de ambientes virtuais. Um ambiente virtual é uma cópia completa do interpretador Python. Quando você instala pacotes em um ambiente virtual, o interpretador Python de todo o sistema não é afetado, apenas a cópia é. Então a solução para ter liberdade completa para instalar qualquer versão de seus pacotes para cada aplicativo é usar um ambiente virtual diferente

para cada aplicativo. Os ambientes virtuais têm o benefício adicional de serem de propriedade do usuário que os cria, então eles não exigem uma conta de administrador.

Vamos começar criando um diretório onde o projeto ficará. Vou chamar esse diretório de **bloguvv**, já que esse será o nome do meu aplicativo:

```
$ mkdir bloguvv
$ cd bloguvv
```

O suporte para ambientes virtuais está incluído em todas as versões recentes do Python, então tudo o que você precisa fazer para criar um é isto:

```
python3 -m venv venv
```

Com este comando, estamos pedindo ao `Python` para executar o pacote `venv`, que cria um ambiente virtual chamado `venv`. O primeiro `venv` no comando é um argumento para a opção `-m` que é o nome do pacote de ambiente virtual `Python`, e o segundo é o nome do ambiente virtual que usarei para este ambiente específico. Se você achar isso confuso, pode substituir o segundo `venv` por um nome diferente que deseja atribuir ao seu ambiente virtual. Em geral, crio meus ambientes virtuais com o nome `venv` no diretório do projeto, então sempre que entro em um projeto, encontro seu ambiente virtual correspondente.

Observe que em alguns sistemas operacionais você pode precisar usar `python` ou `py` em vez de `python3` no comando acima.

Após a conclusão do comando, você terá um diretório chamado `venv` onde os arquivos do ambiente virtual são armazenados.

Agora você tem que dizer ao sistema que você quer usar esse ambiente virtual, e você faz isso ativando-o. Para ativar seu novo ambiente virtual, você usa o seguinte comando:

```
$ source venv/bin/activate
(venv) $ _
```

Se você estiver usando uma janela de prompt de comando do `Microsoft Windows`, o comando de ativação será um pouco diferente:

```
$ Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser
$ venv\Scripts\activate
(venv) $ _
```

O primeiro comando redefinirá a política de execução para o usuário atual (armazenado em HKEY_CURRENT_USER) em vez da máquina local (HKEY_LOCAL_MACHINE). Isso é útil se você não tiver controle administrativo sobre o computador. `RemoteSigned` é uma política de execução que permite um script rodar na máquina caso ele esteja sendo impedido de ser executado. O parâmetro `CurrentUser` altera a política de execução somente para o usuário em execução.

Obsrve que quando ativamos um ambiente virtual a localização do seu interpretador `Python` é adicionada à variável de ambiente `PATH` na sua sessão de comando atual, que determina onde procurar por arquivos executáveis. Para lembrá-lo de que você ativou o ambiente virtual, o comando de ativação modifica seu prompt de comando para incluir o nome do ambiente.

Quando você ativa um ambiente virtual, a configuração da sua sessão de terminal é modificada para que o interpretador `Python` armazenado dentro dela seja o que é invocado quando você digita

`python`. Além disso, o prompt do terminal é modificado para incluir o nome do ambiente virtual ativado. As alterações feitas na sua sessão de terminal são todas temporárias e privadas para essa sessão, então elas não persistirão quando você fechar a janela do terminal. Se você trabalha com várias janelas de terminal abertas ao mesmo tempo, é perfeitamente aceitável ter diferentes ambientes virtuais ativados em cada uma delas.

Agora que você tem um ambiente virtual criado e ativado, você pode finalmente instalar o `Flask` nele:

```
(venv) $ pip install flask
```

Se quiser confirmar que seu ambiente virtual agora tem o `Flask` instalado, você pode iniciar o interpretador `Python` e importar o `Flask` para ele:

```
>>> import flask
>>> _
```

Se esta declaração não lhe der nenhum erro, você pode se parabenizar, pois o `Flask` está instalado e pronto para ser usado.

Observe que os comandos de instalação acima não especificam qual versão do `Flask` você deseja instalar. O padrão quando nenhuma versão é especificada é instalar a versão mais recente disponível no repositório de pacotes. O nosso curso está sendo projetado para a versão 3 ou superior do `Flask`. O comando acima instalará a versão 3.x mais recente, que deve ser apropriada para a maioria dos usuários.

O `Flask` é um **framework** pequeno para a maioria dos padrões — pequeno o suficiente para ser chamado de "micro-framework" e pequeno o suficiente para que, uma vez que você se familiarize com ele, provavelmente seja capaz de ler e entender todo o seu código-fonte.

Mas ser pequeno não significa que ele faz menos do que outros frameworks. O `Flask` foi projetado como um framework extensível do zero; ele fornece um núcleo sólido com os serviços básicos, enquanto as extensões fornecem o resto. Como você pode escolher os pacotes de extensão que deseja, você acaba com uma pilha enxuta que não tem nenhum inchaço e faz exatamente o que você precisa.

O Flask tem três dependências principais. Os **subsistemas de roteamento**, **depuração** e **Web Server Gateway Interface** (WSGI) que vêm da `Werkzeug`; o suporte aos *Templates* é fornecido pela `Jinja2`; e a integração da linha de comando vem da `Click`. Essas dependências são todas de autoria de **Armin Ronacher**, o autor do Flask.

O `Flask` não tem suporte nativo para acessar bancos de dados, validar formulários da web, autenticar usuários ou outras tarefas de alto nível. Esses e muitos outros serviços essenciais que a maioria dos aplicativos da web precisa estão disponíveis por meio de extensões que se integram aos pacotes principais. Como desenvolvedor, você tem o poder de escolher as extensões que funcionam melhor para seu projeto, ou até mesmo escrever as suas próprias, se você se sentir inclinado a isso. Isso contrasta com uma estrutura maior, onde a maioria das escolhas foram feitas para você e são difíceis ou às vezes impossíveis de mudar.

Prosseguindo, instale o `IPython` (*Interactive Python*) - um shell de comando para computação interativa em diversas linguagens de programação, desenvolvido originalmente para a linguagem de programação `Python`, que oferece introspecção, *rich media*, sintaxe de `shell`, preenchimento automático de tabulações e histórico.

```
(venv) $ pip install ipython
```

Iremos utilizá-lo para efetuar testes durante nosso desenvolvimento.

A construção de um aplicativo web nos existirá uma postura de programação pensada em **pacotes**. Outro pensamento comum no desenvolvimento de `Python` é o conceito de **Factory**. À frente abordaremos esse conceito um pouco mais.

Em `Python`, um subdiretório que inclui um arquivo `__init__.py` é considerado um pacote e pode ser importado. Quando você importa um pacote, o `__init__.py` executa e define quais símbolos o pacote expõe para o mundo externo.

Vamos criar um pacote chamado app, que hospedará o aplicativo. Certifique-se de que você está no diretório do balaio e execute o seguinte comando (ou crie uma pasta no Explore):

```
(venv) $ mkdir app
```

O `__init__.py` para o pacote do aplicativo conterá o seguinte código:

```
# app/__init__.py

from flask import Flask

app = Flask(__name__)

from app import routes
```

O *script* acima cria o objeto do aplicativo como uma instância da classe `Flask` importada do pacote flask. A variável `__name__` passada para a classe `Flask` é uma variável predefinida do `Python`, que é definida como o nome do módulo no qual é usada. O `Flask` usa o local do módulo passado aqui como um ponto de partida quando precisa carregar recursos associados, como arquivos de modelo, que abordaremos mais à frente. Para todos os fins práticos, passar `__name__` quase sempre configurará o `Flask` da maneira correta. O aplicativo então importa o módulo de rotas, que ainda não existe.

Um aspecto que pode parecer confuso a princípio é que há duas entidades chamadas `app`. O pacote do aplicativo é definido pelo diretório do aplicativo e pelo script `__init__.py`, e é referenciado na declaração `from app import routes`. A variável do aplicativo é definida como uma instância da classe `Flask` no *script* `__init__.py`, o que a torna um membro do pacote do aplicativo.

Outra peculiaridade é que o módulo de rotas é importado na parte inferior e não na parte superior do *script*, como sempre é feito. A importação inferior é uma solução alternativa bem conhecida que evita **importações circulares**, um problema comum com aplicativos `Flask`. Você verá que o módulo de rotas precisa importar a variável `app` definida neste *script*, então colocar uma das importações recíprocas na parte inferior evita o erro que resulta das referências mútuas entre esses dois arquivos.

Então o que vai no módulo de rotas? As rotas manipulam as diferentes **URLs** que o aplicativo suporta. No `Flask`, os manipuladores para as rotas do aplicativo são escritos como funções `Python`, chamadas funções de visualização. As funções de visualização são mapeadas para uma ou mais URLs de rota para que o `Flask` saiba qual lógica executar quando um cliente solicita uma determinada URL.

A maneira mais conveniente de definir uma rota em um aplicativo `Flask` é por meio do decorador `app.route` exposto pela instância do aplicativo. O exemplo a seguir mostra como uma rota é declarada usando este decorador. Você precisa escrever em um novo módulo chamado `app/routes.py`:

```
# app/routes.py

from app import app

@app.route('/')
@app.route('/index')
def index():
    return "Ola Mundo!"
```

Funções como `index()` que manipulam URLs de aplicativos são chamadas de funções de visualização. No exemplo apresentado aqui, e´sta função de visualização é bem curta e apenas retorna uma saudação como uma *string*. As duas linhas estranhas `@app.route` acima da função são **decoradores**, um recurso exclusivo da linguagem `Python`. Um decorador modifica a função que o segue. Um padrão comum com decoradores é usá-los para registrar funções como retornos de chamada para certos eventos. Neste caso, o decorador `@app.route` cria uma associação entre a URL fornecida como argumento e a função. Neste exemplo, há dois decoradores, que associam as URLs `/` e `/index` a esta função. Isso significa que quando um navegador da web solicita qualquer uma dessas duas URLs, o `Flask` vai invocar esta função e passar seu valor de retorno de volta para o navegador como uma resposta. Se isso não fizer sentido ainda, fará em breve, quando você executar este aplicativo.

> **Nota**
>
> Incorporar strings de resposta com código HTML em arquivos de origem `Python` leva a um código difícil de manter. Os exemplos nesta unidade fazem isso apenas para introduzir o conceito de respostas. Iremos aprender uma maneira melhor de gerar respostas HTML a seguir.

Para concluir o aplicativo, você precisa ter um *script* `Python` no nível superior que defina a instância do aplicativo `Flask`. Vamos chamar este script de `bloguvv.py` e defini-lo como uma única linha que importa a instância do aplicativo:

```
# bloguvv.py
from app import app
```

Lembra das duas entidades de aplicativo? Aqui você pode ver as duas juntas na mesma frase. A instância do aplicativo `Flask` é chamada `app` e é um membro do pacote de aplicativo. A instrução `from app import app` importa a variável `app` que é um membro do pacote de aplicativo. Se você achar isso confuso, pode renomear o pacote ou a variável para outra coisa.

Só para ter certeza de que está fazendo tudo corretamente, abaixo você pode ver um diagrama da estrutura do projeto até agora:

```
bloguvv/
  venv/
  app/
    __init__.py
    routes.py
  bloguvv.py
```

Acredite ou não, esta primeira versão do aplicativo agora está completa! Antes de executá-lo, no entanto, o `Flask` precisa ser informado sobre como importá-lo, definindo a variável de ambiente `FLASK_APP`:

```
(venv) $ export FLASK_APP=bloguvv.py
```

Se você estiver usando o prompt de comando do `Microsoft Windows`, use `set` em vez de `export` no comando acima.

```
(venv) $ set FLASK_APP=bloguvv.py
```

Você está pronto para ser surpreendido? Você pode executar seu primeiro aplicativo web digitando o comando `flask run`, conforme mostrado abaixo:

```
(venv) $ flask run
 * Serving Flask app 'bloguvv.py' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production ↩
       deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

O que aconteceu aqui? O comando `flask run` procurará uma instância do aplicativo `Flask` no módulo referenciado pela variável de ambiente `FLASK_APP`, que neste caso é `bloguvv.py`. O comando configura um servidor web que é configurado para encaminhar solicitações para este aplicativo.

Após a inicialização do servidor, ele aguardará as conexões do cliente. A saída do `flask run` indica que o servidor está sendo executado no endereço **IP 127.0.0.1**, que é sempre o endereço do seu próprio computador. Este endereço é tão comum que também tem um nome mais simples que você pode ter visto antes: **localhost**. Os servidores de rede escutam conexões em um número de porta específico. Os aplicativos implantados em servidores web de produção geralmente escutam na porta 443, ou às vezes 80 se não implementarem criptografia, mas o acesso a essas portas requer direitos de administração. Como este aplicativo está sendo executado em um ambiente de desenvolvimento, o `Flask` usa a porta 5000. Agora abra seu navegador da web e insira a seguinte URL no campo de endereço:

```
http://localhost:5000/
```

Você vê os mapeamentos de rotas do aplicativo em ação? A primeira URL mapeia para `/`, enquanto a segunda mapeia para `/index`. Ambas as rotas são associadas à única função de visualização no aplicativo, então elas produzem a mesma saída, que é a string que a função retorna. Se você digitar qualquer outra URL, obterá um erro, pois apenas essas duas URLs são reconhecidas pelo aplicativo.

Se tudo deu certo até aqui, parabéns! Concluímos nosso primeiro grande passo para nos tornarmos um desenvolvedor web!