
DESENVOLVIMENTO
DE APIs E
MICROSSERVIÇOS

7

Introdução ao Flask

Victor Williams Stafusa da Silva

Resumo

Chegou a hora de desenvolvermos o lado servidor de aplicações HTTP. Para isso, o Flask é o framework ideal para realizar um desenvolvimento rápido e simples de aplicações que devam aceitar ou responder a requisições HTTP. Nesta parte, você verá como começar com o Flask e com ele desenvolver o lado servidor de suas primeiras aplicações.

7.1. Objetivos deste capítulo

- Entender para que serve o Flask.
- Começar o desenvolvimento de aplicações *server-side* com o Flask.
- Configurar minimamente o servidor com Flask para ter a aplicação em execução.
- Entender a funcionalidade de *hot-deploy* do Flask.
- Definir rotas com o Flask e associá-las a código a ser executado.
- Parametrizar rotas no Flask.
- Definir códigos de status HTTP nas respostas de requisições com o Flask.

7.2. Flask, o que é isso?

Diversos *frameworks web* utilizados em diversas linguagens de programação que objetivam o desenvolvimento *full-stack*, buscam não apenas permitir disponibilizar aplicações que recebam e respondam requisições HTTP, mas sim que ofereçam soluções integradas com desenvolvimento de *front-end*, mapeamento objeto-relacional do banco de dados, mapeamento e ciclo de vida de objetos de *front-end* com o *back-end*, execução de código remoto em ambiente distribuído, entre outros. Embora possa parecer fascinante, tais *frameworks*, na busca de prover soluções completas para o desenvolvimento de aplicações, também podem ser significativamente complexos, difíceis de se aprender e muitas vezes impõem determinadas formas de desenvolvimento às aplicações que não permitem ao desenvolvedor utilizar uma tecnologia diferente no banco de dados, na modelagem de regras de negócio, no *front-end*, ou na forma como as

requisições HTTP são recebidas, processadas e respondidas. Exemplos de *frameworks* assim são o Django, o Spring, o JSF, o Web Forms, o Zend, entre outros.

Já o Flask, é um *microframework* utilizado para poder servir requisições HTTP à funções em Python e a partir delas montar respostas a essas requisições. Por *microframework*, entende-se que o Flask é pequeno e enxuto e objetiva ser um *framework* simples, limitando-se a execução apenas do que se propõe a fazer, que é servir requisições e respostas HTTP e deixando ao desenvolvedor a tarefa de escolher quais serão as outras ferramentas ou *frameworks* utilizados para o desenvolvimento de outros aspectos das aplicações, sem impor restrições a essas escolhas. Isso significa que o Flask não busca ser uma ferramenta de banco de dados, não busca mapear no *back-end* os componentes do *front-end* e nem modelar regras de negócio da aplicação, pois para essas atividades, já existem diversas outras tecnologias e bibliotecas que podem ser utilizadas. Tudo que ele faz é apenas e somente servir requisições e respostas HTTP e associar rotas à funções criadas pelo desenvolvedor da aplicação.

Apesar disso, o Flask também inclui o Jinja2 que é uma ferramenta simples para o processamento de *templates*. *Templates* são como *strings* formatadas, mas possuem algum grau de complexidade a mais, sendo "esqueletos" de um texto a ser produzido (geralmente, mas não necessariamente, em HTML) com partes a serem avaliadas e processadas dinamicamente em tempo de execução. Explicaremos mais sobre ele no capítulo 9.

Internamente, o Flask utiliza uma biblioteca chamada Werkzeug (que significa "ferramenta" em alemão, pronunciado aproximadamente como "vérc-zóig"). O Werkzeug serve para realizar diversas tarefas, tais como roteamento, criação de requisições e respostas HTTP e o processamento delas. Na verdade, o Flask mesmo é apenas uma fina casca por cima do Werkzeug e o que o Flask mesmo faz por si só é apenas possibilitar associar funções do seu código a rotas HTTP de uma forma muito mais simples e direta do que seria por meio do uso direto do Werkzeug. Ou seja, internamente é o Werkzeug quem acaba decidindo qual requisição deverá ser tratada por qual função no Python.

Para instalar o Flask (juntamente com o Jinja2, com o Werkzeug e outras bibliotecas acompanhantes), um comando ***pip install --user flask*** ou ***pip3 install --user flask*** é o suficiente.

7.3. Começando a codificar com o Flask

Bem, neste ponto, você provavelmente deve estar se perguntando como fazer um código em Flask. Comecemos então com o seguinte código:

Codificação 7.1. Iniciando com o Flask

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

@app.route("/")
def bom_dia():
    return "Bom dia"
```

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

Fonte: do autor, 2021

Salve este código como **exemplo.py**. Para executá-lo basta executar o comando ***python exemplo.py***.

Atenção:

Você pode salvar o seu arquivo com praticamente qualquer nome com a extensão ".py". No entanto, não o salve como **flask.py**, pois neste caso, você confundirá a instrução **import** e a sua aplicação não irá funcionar.

Ao executar este programa, a seguinte saída (ou similar) deverá aparecer:

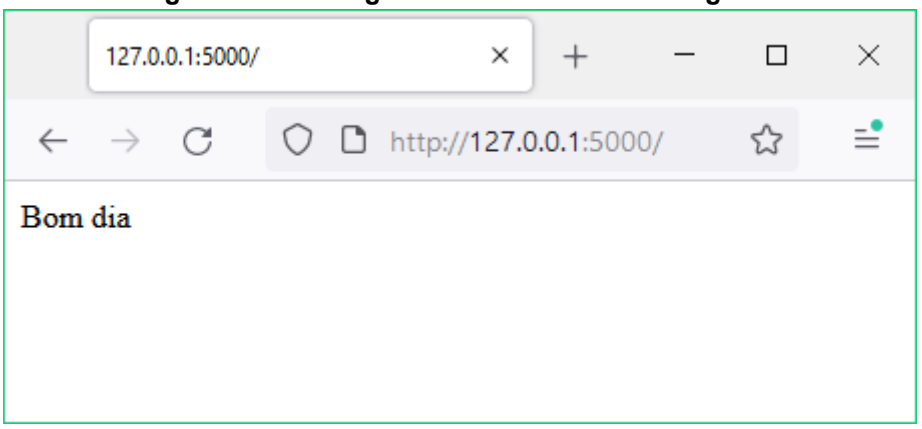
Codificação 7.2. Iniciando com o Flask 2

```
C:\DAM\Capitulo_07>python exemplo.py
 * Serving Flask app "exemplo" (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)
```

Fonte: do autor, 2021

Neste ponto, acesse o navegador na URL **http://127.0.0.1:5000/**. O resultado deverá ser semelhante ao demonstrado na Figura 7.1.

Figura 7.1. Mensagem de "Bom dia" no navegador



Fonte: do autor, 2021

Pois bem, essa é a sua primeira aplicação do lado servidor. Ela está executando no *localhost* (127.0.0.1) e na porta TCP 5000. A realização de uma operação GET na URL **http://127.0.0.1:5000/** acaba por produzir essa resposta "Bom dia".

Vamos ver o que exatamente o código faz. A primeira linha, que contém o código `from flask import Flask` serve para fazer a importação do Flask.

Atenção:

É importante perceber-se que o **flask** (com letras minúsculas) é o nome da biblioteca de onde é feita a importação, enquanto que **Flask** (com a inicial maiúscula) é o nome de uma classe que será importada. É importante não confundir o **flask** com todas as letras minúsculas e o **Flask** com a inicial maiúscula.

A próxima instrução, `app = Flask(__name__)`, serve para instanciar a classe Flask por meio de seu construtor. O parâmetro `__name__` deverá se expandir para `"__main__"`, e serve para indicar qual parte da aplicação está instanciando a classe Flask, sendo que `"__main__"` indica que este é o escopo principal da aplicação, ou seja, o local onde a execução da aplicação foi iniciada. Como nossa aplicação por enquanto é bem pequena, isso não é importante, mas será importante quando tivermos aplicações maiores com diversos módulos. Ainda nesta linha, `app` é apenas o nome da variável que estamos utilizando para armazenar a instância criada, e é por meio dessa variável (e do objeto para o qual ela aponta) que interagimos com o servidor.

O trecho seguinte corresponde à função `bom_dia`, que é a função que queremos expor aos clientes por meio do protocolo HTTP. A função por si só, apenas retorna a *string* `"bom dia"`, mas perceba que nela um decorador foi aplicado, `@app.route("/")`. Esse decorador diz que a função será roteada para a rota "/" (a raiz das rotas do sistema), e o objeto que usamos para efetuar o roteamento é o objeto `app`, que é uma instância da classe Flask.

Finalmente, o `app.run()` serve para fazer com que o objeto `app` (da classe Flask) comece a atender requisições HTTP. Ele é protegido por um bloco `if __name__ == "__main__":`, que embora neste simplório exemplo não faça nenhuma diferença, servirá para evitar que o Flask acidentalmente seja executado de dentro de um arquivo importado quando estivermos trabalhando com aplicações grandes, garantindo que ele só executará a partir do escopo principal da aplicação.

Atenção:

A chamada ao `app.run()` não retornará em condições normais, mantendo a execução da *thread* que a executa travada indefinidamente. Uma *thread* é uma linha de execução do programa, e o Flask criará outras *threads* para atender às requisições recebidas, mas manterá a *thread* que o iniciou travada. Assim sendo, qualquer instrução após o `app.run()`, mesmo que seja um simples `print("Oi")`, não será executada enquanto o servidor não for de alguma forma derrubado.

Assim sendo, não tente inicializar ou configurar alguma coisa na sua aplicação ou executar algum processamento após chamar o `app.run()`. Configure tudo o que for necessário ou antes de chamar esse método ou então em *threads* diferentes da que executarão esse método.

7.4. Configurando o servidor do Flask

A porta padrão do Flask é a porta TCP 5000. Talvez você queira utilizar um outro número de porta. Por exemplo, para executá-lo na porta TCP 8765, use o seguinte nas últimas duas linhas do código:

Codificação 7.3. Configurando o servidor do Flask

```
if __name__ == "__main__":
    app.run(port = 8765)
```

Fonte: do autor, 2021

Na configuração padrão, o Flask estará atendendo apenas às requisições locais do seu computador. Requisições originárias de outros locais da sua rede ou da *internet* serão recusadas. Isso pode ser resolvido com o parâmetro `host = "0.0.0.0"`, que fará o Flask aceitar requisições originárias de qualquer endereço:

Codificação 7.4. Configurando o servidor do Flask 2

```
if __name__ == "__main__":
    app.run(host = "0.0.0.0", port = 8765)
```

Fonte: do autor, 2021

f7.4.1. Configurando o *hot-deploy*

O Flask também possibilita o *hot-deploy*, o que significa automaticamente reiniciar a aplicação sempre que o código for modificado. Essa característica pode ser muito útil para os casos onde você esteja mudando os arquivos com os códigos com frequência (dezenas ou centenas de vezes) e não queira ter o trabalho de derrubar e reiniciar o servidor manualmente sempre que um arquivo for modificado. Ou ainda, serve também para evitar o aborrecimento resultante de ao esquecer de fazer isso, não ver aplicadas as suas últimas alterações realizadas. Para ativar tal funcionalidade, pode-se utilizar a opção `debug = True`:

Codificação 7.5. Configurando o *hot-deploy*

```
if __name__ == "__main__":
    app.run(host = "0.0.0.0", port = 8765, debug = True)
```

Fonte: do autor, 2021

Atenção:

O *hot-deploy* não é 100% confiável. Primeiramente, ao modificar o código, toda a aplicação será derrubada e irá reiniciar automaticamente, o que significa que variáveis de memória terão seus valores perdidos e serão reinicializadas. Além disso, se o código modificado tiver algum erro de compilação ou algum outro problema que impeça a correta execução do Flask, a aplicação não irá subir. Por vezes, outros tipos de efeitos colaterais podem também ser observados.

Assim sendo, embora o *hot-deploy* possa ser muito útil para agilizar o desenvolvimento, ao fazer a validação final da aplicação, sempre execute um teste subindo-a "fria" para se certificar de que nenhum problema foi mascarado pelo *hot-deploy*. Da mesma forma, se após um *hot-deploy*, você suspeitar de algum comportamento estranho da aplicação, tente testar subindo-a "fria".

7.4.2. Configurando o Flask para ambiente de produção

Talvez você tenha notado, no log da aplicação, uma advertência em letras vermelhas "*WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.*", ou seja, que isso não deve ser usado num servidor de produção. O motivo para isso é que o Flask é um *framework* para ser executado numa aplicação servidora WSGI (*Web Service Gateway Interface*), mas não se destina a ser por si só a aplicação servidora WSGI. O WSGI é o padrão que descreve como servidores HTTP e aplicações em Python devem interagir e se intercomunicar, possibilitando que aplicações Python possam ser implantadas em servidores HTTP como o Apache, o nginx e outros. Na falta de um servidor WSGI, para "quebrar o galho" e facilitar o desenvolvimento de aplicações, o Werkzeug possui dentro de si uma aplicação servidora WSGI simples para testes, que é a que estamos utilizando, mas ela é apenas um "quebra-galho" e não foi projetada para ser estável, eficiente, segura e completa, além de não ter uma boa performance, características essas que são as esperadas num ambiente de produção.

Uma das formas de se resolver esse problema (dentre muitas outras existentes) é utilizar o Waitress como servidor WSGI ao alterar as últimas linhas para o seguinte:

Codificação 7.6. Configurando o Flask para ambiente de produção

```
if __name__ == "__main__":
    from waitress import serve
    serve(app, host = "0.0.0.0", port = 8765)
```

Fonte: do autor, 2021

Possivelmente você precisará executar um comando **pip install --user waitress** ou **pip3 install --user waitress** para que isso funcione.

Atenção:

Não é possível utilizar `debug = True` com o Waitress. Afinal de contas, o *debug* é algo que é inerente a um ambiente de desenvolvimento ou de testes, e não de produção. No entanto, uma vez que o Waitress é um servidor de produção, e não um de testes, não faz sentido haver nele a funcionalidade de *debug* ou de *hot-deploy*.

7.5. Configurando as rotas

A nossa aplicação, no momento, não faz muita coisa interessante ainda. Vamos configurá-la para funcionar com diferentes rotas. Por exemplo:

Codificação 7.7. Configurando as rotas

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
@app.route("/bom-dia")
@app.route("/good-morning")
def bom_dia():
```

```
    return "Bom dia"

@app.route("/boa-tarde")
@app.route("/good-afternoon")
def boa_tarde():
    return "Boa tarde"

@app.route("/boa-noite")
@app.route("/good-night")
def boa_noite():
    return "Boa noite"

if __name__ == "__main__":
    app.run(host = "0.0.0.0", port = 8765)
```

Fonte: do autor, 2021

Execute a aplicação. Comece acessando em seu navegador de *internet*, a URL **http://localhost:8765/bom-dia**, e você verá a mensagem de bom dia no navegador. Acesse a URL **http://localhost:8765/boa-tarde**, e você verá a mensagem de boa tarde. Acesse a URL **http://localhost:8765/boa-noite**, e você verá a mensagem de boa noite.

Como você deve ter deduzido ao olhar o código, isso acontece porque cada uma dessas URLs está mapeada para uma função diferente. Assim sendo, o Flask (para ser mais preciso, o Werkzeug), ao receber uma requisição, analisará o caminho (*path*) dela para decidir qual função será chamada. Se a URL tiver o caminho **/bom-dia**, a função **bom_dia** será chamada. Se tiver o caminho **/boa-tarde**, a função **boa_tarde** será chamada. Se tiver o caminho **/boa-noite**, a função **boa_noite** será chamada.

Olhando o código, pode-se ver também que é possível colocar-se várias rotas distintas na mesma função. Dessa forma, **/good-morning**, também servirá para chamar a função **bom_dia**.

Atenção:

1. Não se esqueça de mapear a rota padrão "/" em alguma função. Caso contrário, se o usuário acessar a aplicação apenas com a URL sem nenhum *path*, o resultado será um erro 404.
2. Se o Flask encontrar duas ou mais funções com o mesmo nome para as suas rotas, a aplicação não executará, sendo abortada com um erro.
3. Todas as rotas devem obrigatoriamente iniciar com "/", caso contrário o Flask lançará um erro e recusará mapeá-la.

7.5.1. Rotas com vários elementos

Rotas também podem ter caminhos compostos por vários elementos separados por barras. Por exemplo:

Codificação 7.8. Rotas com vários elementos

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

```
@app.route("/um/caminho/com/varios/elementos")
def um_caminho_longo():
    return "Oi, pessoal!"

if __name__ == "__main__":
    app.run(host = "0.0.0.0", port = 8765)
```

Fonte: do autor, 2021

Acessando-se então no navegador a URL da rota especificada, ou seja, **http://localhost:8765/um/caminho/com/varios/elementos**, uma mensagem onde se lê **Oi pessoal** será exibida. As rotas podem ter qualquer quantidade de elementos separados por barras.

7.5.2. Rotas com um elemento parametrizado

Rotas também podem conter parâmetros de caminho (*path*) da URL, como demonstrado no código de exemplo abaixo:

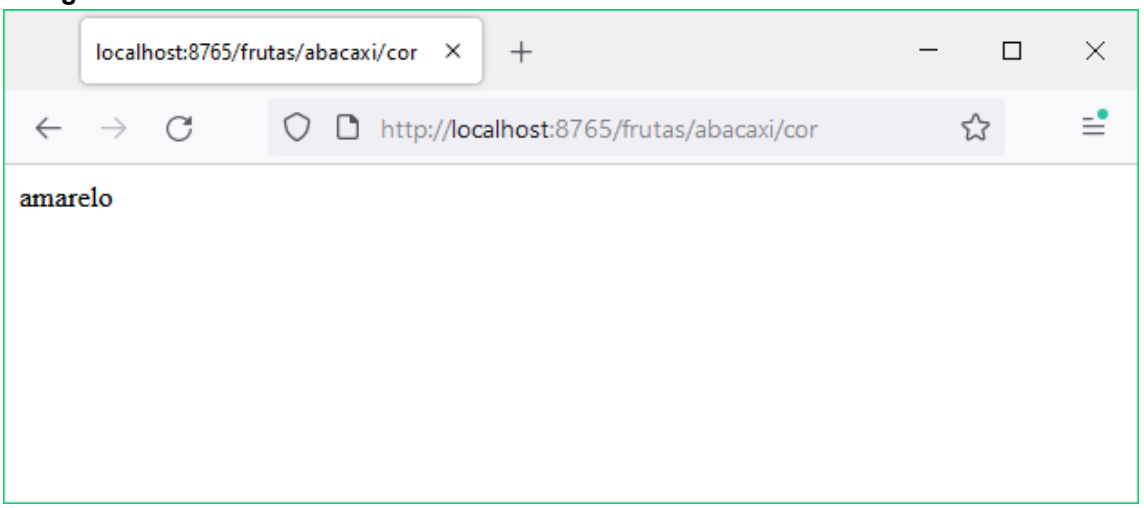
Codificação 7.9. Rotas com um elemento parametrizado

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

cores_frutas = {
    "morango": "vermelho",
    "uva": "roxo",
    "banana": "amarelo",
    "abacaxi": "amarelo",
    "limão": "verde"
}

# Continua na próxima página.
# Continuação da página anterior.

@app.route("/frutas/<nome_fruta>/cor")
def frutas(nome_fruta):
    if nome_fruta in cores_frutas:
        return cores_frutas[nome_fruta]
    return "Não sei"

if __name__ == "__main__":
    app.run(host = "0.0.0.0", port = 8765)
```

Fonte: do autor, 2021

Ao executar-se esse código, e então acessar no navegador a URL **http://localhost:8765/frutas/abacaxi/cor**, uma tela semelhante a da Figura 7.2 deverá aparecer.

Figura 7.2. Cor da fruta "abacaxi" sendo executada pelo programa e exibida no navegador



Fonte: do autor, 2021

Observe, que na execução, o conteúdo da palavra "abacaxi" foi lido a partir da URL. Semelhantemente, "uva", "morango", "limão" ou "banana" poderiam também terem sido utilizados em seu lugar. Para que um valor possa ser lido da URL, é importante que o mesmo seja um dos parâmetros na função a ser roteada. O exato mesmo nome deve ser mostrado também como parte da URL, delimitado por "<" e ">". Assim sendo, a rota `"/frutas/<nome_fruta>/cor"`, contém um parâmetro a ser definido, que é o chamado `nome_fruta`, que é exatamente o nome do parâmetro da função apresentada na definição `def frutas(nome_fruta):`.

7.5.3. Rotas com vários elementos parametrizados

É possível também apresentar-se mais do que um parâmetro na URL. Por exemplo, veja o seguinte código:

Codificação 7.10. Rotas com vários elementos parametrizados

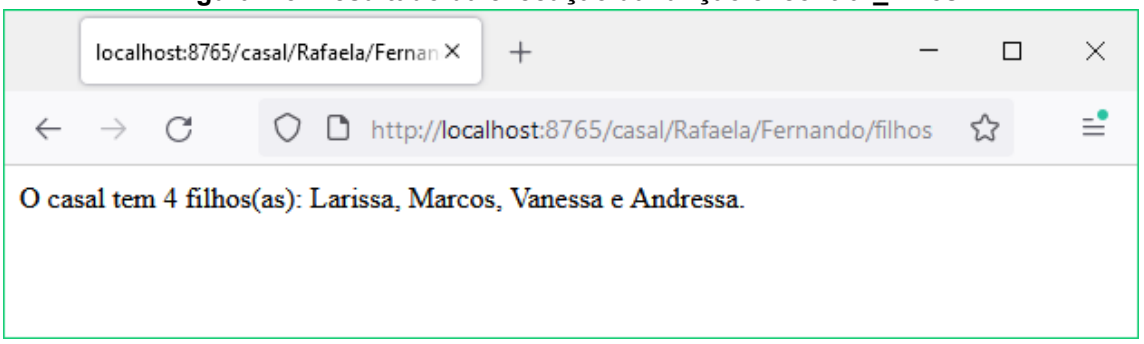
```
from flask import Flask

app = Flask(__name__)

# Continua na próxima página.
# Continuação da página anterior.

casais_e_filhos = {
    "Camila,Paulo": ["Pedro", "Carlos", "Mariana"],
    "Laura,Joaquim": ["Sandra", "Daniela"],
    "Rafaela,Fernando": ["Larissa", "Marcos", "Vanessa", "Andressa"],
    "Maria,André": ["Juliano"],
    "Tatiana,Luís": ["Kelly", "Kelvin", "Karla"],
    "Beatriz,José": ["João", "Marcelo", "Guilherme"],
    "Tamara,Rodolfo": []
}

def filhos(mae, pai):
```

```
    casal = f"{mae},{pai}"
    if casal not in casais_e_filhos: return None
    filhos = casais_e_filhos[casal]
    if len(filhos) == 0: return "O casal não tem filhos."
    if len(filhos) == 1:
        return f"O casal tem 1 filho(a): {filhos[0]}."
    return f"O casal tem {len(filhos)} filhos(as): " \
        + f"{ ', '.join(filhos[:-1])} e {filhos[-1]}."

@app.route("/frutas/<mae>/<pai>/filhos")
def encontrar_filhos(mae, pai):
    f = filhos(mae, pai)
    if f is None: return "Casal não encontrado."
    return f

if __name__ == "__main__":
    app.run(host = "0.0.0.0", port = 8765)
```

Fonte: do autor, 2021

Ao executar-se esse código, e então acessar no navegador a URL **http://localhost:8765/casal/Rafaela/Fernando/filhos**, uma tela semelhante a da Figura 7.3 deverá aparecer.

Figura 7.3. Resultado da execução da função encontrar_filhos



Fonte: do autor, 2021

Na função **encontrar_filhos**, vemos que há dois parâmetros: **mae** e **pai**. Esses são os mesmos dois parâmetros que são declarados na rota. Assim sendo, seja nessa função, ou seja em qualquer outra função com uma rota definida, o Flask consegue associar cada um dos parâmetros com uma parte da URL, e ao receber uma requisição HTTP, irá extrair da URL requisitada, as partes correspondentes aos parâmetros da função como *strings* e então invocará essa função utilizando esses parâmetros.

Atenção:

Se os parâmetros da função não forem exatamente os mesmos que estão declarados na rota, com exatamente os mesmos nomes, um erro será produzido pelo Flask quando ele tentar invocar a função. Entretanto, a ordem dos parâmetros da função não precisa ser a mesma que está declarada na rota.

7.5.4. Rotas com elementos tipados

Você deve ter notado que os elementos parametrizados das rotas até o momento (**<nome_fruta>**, **<mae>** e **<pai>**), sempre são do tipo *string*. Entretanto, podemos defini-los com o tipo *int* ou *float* se quisermos que sejam números. Para isso, basta prefixar os valores na declaração da rota com **int:** ou **float:**. Você também pode prefixar com **string:** se quiser deixar explícito o comportamento padrão (de ser uma *string*). Veja o exemplo a seguir:

Codificação 7.11. Rotas com elementos tipados

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

@app.route("/<float:a>/menos/<float:b>")
def subtrair(a, b):
    return str(a - b)

fib = [0, 1]

def fibonacci(valor):
    while valor >= len(fib):
        fib.append(fib[-1] + fib[-2])
    return fib[valor]

@app.route("/fibonacci/<int:valor>")
def num_fibonacci(valor):
    return f"O {valor}º valor da sequência de " \
        + f"Fibonacci é {fibonacci(valor)}."

if __name__ == "__main__":
    app.run(host = "0.0.0.0", port = 5000)
```

Fonte: do autor, 2021

Execute esse exemplo no navegador e tente acessar as seguintes URLs:

- **http://localhost:5000/4.2/menos/9.18**
- **http://localhost:5000/fibonacci/18**
- **http://localhost:5000/macaco/mais/elefante**
- **http://localhost:5000/fibonacci/3.14**
- **http://localhost:5000/fibonacci/cachorro**

Observe que os valores com **a** e **b** da função **subtrair** são do tipo *float*. Se fossem *strings*, a subtração não seria possível e causaria um erro. Da mesma forma, o tipo de **valor** na função **num_fibonacci** é do tipo *int*. Com isso, as tentativas de se utilizar as funções com os tipos de dados incorretos (as URLs em vermelho) resultam em erros 404, sem que você como programador tenha que se preocupar em codificar alguma lógica de conversão, de validação ou de tratamento específicas para isso.

No entanto, nem tudo está correto e ainda temos problemas:

Atenção:

Rotas com **float:** ou **int:**. não aceitam números negativos por padrão. Para que sejam aceitos, especifique **int(signed = True):** ou **float(signed = True):**.

Por exemplo, verifique a URL **http://localhost:5000/-3.14/menos/-3.14** e observe que o resultado é um erro 404. Para resolver isso, a função **subtrair** ficará da seguinte forma:

Codificação 7.12. Rotas com elementos tipados - Erro 404

```
@app.route("/<float(signed=True):a>/menos/<float(signed=True):b>")
def subtrair(a, b):
   return str(a - b)
```

Fonte: do autor, 2021

Entretanto, isso ainda não soluciona todos os nossos casos:

Atenção:

Rotas com **float:** não aceitam números inteiros sem o ponto decimal que também são aceitos por **int:**. Para resolver isso, ou sempre adicione o ponto decimal e um zero em seguida ou então defina múltiplas rotas na mesma função.

Verifique a URL **http://localhost:5000/3/menos/3** e observe que o resultado também é mais um erro 404. Para resolver mais esse problema, a função **subtrair** deverá sofrer mais uma alteração:

Codificação 7.13. Rotas com elementos tipados - Erro 404 - parte 2

```
@app.route("/<float(signed=True):a>/menos/<float(signed=True):b>")
@app.route("/<int(signed = True):a>/menos/<float(signed = True):b>")
@app.route("/<float(signed = True):a>/menos/<int(signed = True):b>")
@app.route("/<int(signed = True):a>/menos/<int(signed = True):b>")
def subtrair(a, b):
   return str(a - b)
```

Fonte: do autor, 2021

Existem outras formas de se controlar os valores possíveis em a necessidade de codificar regras de validação explícitas tais como:

- Especificar o número exato de dígitos para **int**, com zeros preenchidos à esquerda se necessários, tal como em **<int(fixed_digits = 11):cpf>**.
- Especificar valores mínimos e/ou máximos para o **float** e o **int**, tais como em **<float(min = 0, max = 10):nota>** ou **<int(min = 1, max = 12):mes>** ou **<float(min = 1.0):pagamento>**.
- Especificar o tamanho máximo e/ou mínimo de um *string*, tal como em **<string(minlength = 4):busca>** ou **<string(minlength = 5, maxlength = 200):nome_candidato>**.
- Especificar o tamanho exato de uma *string*, tal como em **<string(length = 12):promocode>** ou **<string(length = 7):placa_do_carro>**.
- Obter diversos componentes intermediários do caminho, mesmo que contenham diversas “/”, tal como **"pessoa/<path:nome_pagina>/editar"**.
- Obter um UUID, tal como **"arquivo/<uuid:codigo>/compartilhar"**.

7.6. Códigos de estados HTTP na resposta da requisição

Até o momento, todas as requisições bem sucedidas retornam o *status* 200 e todos os erros estão sendo tratados internamente pelo Flask / Werkzeug, com regras de validação definidas apenas nas rotas. No entanto, por vezes faz-se necessário o retorno de códigos de *status* específicos, inclusive códigos na faixa 4XX devido a implementação de regras de validação e/ou consistência que não tem como serem definidas nas rotas. A forma mais simples de se realizar tal atividade é acrescentar o código de *status* desejado como um segundo valor de retorno da função onde a rota é aplicada. Para exemplificar, acrescentemos a função **dividir** semelhante à função **menos** da listagem de código anterior:

Codificação 7.14. Códigos de estados HTTP na resposta da requisição

```
@app.route("/<float(signed=True):a>/divide/<float(signed=True):b>")
@app.route("/<int(signed = True):a>/divide/<float(signed = True):b>")
@app.route("/<float(signed = True):a>/divide/<int(signed = True):b>")
@app.route("/<int(signed = True):a>/divide/<int(signed = True):b>")
def dividir(a, b):
    if b == 0: return "Não divida por zero!", 422
    return str(a / b)
```

Fonte: do autor, 2021

Tente então acessar **http://localhost:5000/1/divide/0** no navegador. O resultado será que a mensagem de erro aparecerá e a requisição resultará num código de *status* 422. Para comprovar esse resultado, podemos fazer um outro programa (o cliente) que faz a requisição ao servidor por meio da biblioteca requests (enquanto o servidor estiver executando, obviamente). O código do cliente é o seguinte:

Codificação 7.15. Códigos de estados HTTP na resposta da requisição 2

```
import requests

a = input("Digite o valor de a: ")
b = input("Digite o valor de b: ")
url = f"http://localhost:5000/{a}/divide/{b}"

x = requests.get(url)
if x.status_code != 200:
    print(f"[{x.status_code}] {x.text}")
else:
    print(f"{a} dividido por {b} é igual a {x.text}.")
```

Fonte: do autor, 2021

Veja o resultado de algumas execuções deste pequeno programa:

Codificação 7.16. Resultado de algumas execuções

```
C:\DAM\Capitulo_07>python cliente.py
Digite o valor de a: 4
Digite o valor de b: 5
4 dividido por 5 é igual a 0.8.

C:\DAM\Capitulo_07>python cliente.py
Digite o valor de a: -12.25
Digite o valor de b: 0.25
```

```
-12.25 dividido por 0.25 é -49.0.

C:\DAM\Capitulo_07>python cliente.py
Digite o valor de a: morango
Digite o valor de b: banana
[404] <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>404 Not Found</title>
<h1>Not Found</h1>
<p>The requested URL was not found on the server. If you entered the URL manually please check your spelling and try again.</p>


C:\DAM\Capitulo_07>python cliente.py
Digite o valor de a: 12
Digite o valor de b: 0
[422] Não divida por zero!

C:\DAM\Capitulo_07>
```

Fonte: do autor, 2021

Observe na última execução que pudemos obter o código de *status* que foi definido dentro da função **dividir**, diferentemente do que acontece nas outras três execuções onde o código de *status* foi definido pelo Flask (200 nas primeiras duas e 404 na terceira).

Vamos praticar?

- Tente realizar as funções de soma, multiplicação, raiz quadrada e outras funções matemáticas que você achar interessante, inclusive com regras de validação especiais e retorno de *status* específicos.
- Além da função que devolve os nomes dos filhos de um casal, tente criar uma que devolva qual é o i-ésimo filho e devolva um código de *status* 404 caso um filho inexistente tente ser acessado.
- Tente criar funções de teste para o seu servidor que testem todas as suas rotas utilizando para tal a biblioteca requests.

Referências

Anatoly; Zhong, Yuchen; *et al.* 2019. **Flask at first run: Do not use the development server in a production environment.** Disponível em: <https://stackoverflow.com/a/54381386/540552>. Acesso em 10 jul. 2021.

The Pallets Projects. **Quickstart.** (s.d.) Disponível em: <https://flask.palletsprojects.com/en/2.0.x/quickstart>. Acesso em 10 jul. 2021.

The Pallets Projects. **URL Routing.** (s.d.) Disponível em: <https://werkzeug.palletsprojects.com/en/2.0.x/routing>. Acesso em 10 jul. 2021.

The Pallets Projects. **Welcome to Flask - Flask Documentation.** (s.d.) Disponível em: <https://flask.palletsprojects.com/en/2.0.x>. Acesso em 10 jul. 2021.

The Pallets Projects. **Werkzeug.** (s.d.) Disponível em: <https://werkzeug.palletsprojects.com/en/2.0.x>. Acesso em 10 jul. 2021.