
DESENVOLVIMENTO
DE APIs E
MICROSSERVIÇOS

9

Desenvolvendo o *front-end* com o Jinja2

Victor Williams Stafusa da Silva

Resumo

Já exploramos diversos recursos do Flask para o desenvolvimento de um back-end completo. Agora, vamos conhecer o Jinja2 e mais alguns recursos do Flask para aprendermos como podemos desenvolver o front-end de uma aplicação web com com HTML de uma forma modular e que permita uma fácil manutenção.

9.1. Objetivos deste capítulo

- Criar e organizar *templates* HTML e processá-los com o Jinja2.
- Utilizar o Jinja2 para criar respostas HTTP em formato HTML dinâmico.
- Servir conteúdo estático com o Jinja2.
- Ler dados de formulários com o Flask.
- Gerenciar *cookies* e sessões no Flask.
- Enviar notificações ao usuário com o uso de mensagens armazenadas na sessão.

9.2. Iniciando com o Jinja2

Já utilizamos o Flask para implementar lógica de programação. No entanto, muitas vezes precisamos também de um *front-end* HTML. E a principal função do Jinja2, que acompanha o Flask é permitir uma criação desses HTMLs de forma simples e limpa.

O Jinja2 funciona com base no conceito de *templates*. *Templates* são *strings* com trechos de código a serem interpretados. É o mesmo conceito das strings formatadas do Python, mas com características mais robustas de formatação. O principal uso dos templates no Jinja2 é a geração de conteúdo em formato HTML, embora possa ser utilizado para outras finalidades também. Vejamos um exemplo com templates. Observe a seguinte estrutura de pastas:

Figura 9.1. Arquivos de um projeto com Flask e Jinja2.

projeto

```
├ 📁 templates
│  └ oi.html
├ 📁 static
│  ├ estilos.css
│  └ favicon.png
└ meuapp.py
```

Fonte: do autor, 2021

No exemplo da Figura 9.1, temos 4 arquivos de exemplo: `meuapp.py`, `oi.html`, `estilos.css` e `favicon.png`. As pastas `templates` e `static` são especiais. Na pasta `static`, colocamos os arquivos que serão enviados ao cliente inalterados pelo Flask (no caso, o `estilos.css` e o `favicon.png`). Na pasta `templates`, colocamos os *templates* que serão processados pelo Jinja2 (no caso o `oi.html`). Neste exemplo, no arquivo `meuapp.py`, temos o seguinte código:

Codificação 9.1. meuapp.py

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

@app.route("/")
@app.route("/bom-dia")
def bom_dia():
    return render_template("oi.html", mensagem = "Bom dia")

@app.route("/boa-tarde")
def boa_tarde():
    return render_template("oi.html", mensagem = "Boa tarde")

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

Fonte: do autor, 2021

Já no arquivo `oi.html`, temos o seguinte código:

Codificação 9.2. oi.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <meta name="viewport"
        content="width=device-width, initial-scale=1.0">
    <title>Saudações</title>
    <link rel="icon" type="image/png"
        href="{{ url_for('static', filename = 'favicon.png') }}" />
    <link rel="stylesheet" type="text/css"
        href="{{ url_for('static', filename = 'estilos.css') }}" />
  </head>
  <body>
    <h1>Olá</h1>
    <p class="msg">{{mensagem}}</p>
  </body>
</html>
```

Fonte: do autor, 2021

Finalmente, o arquivo `estilos.css` contém o seguinte código:

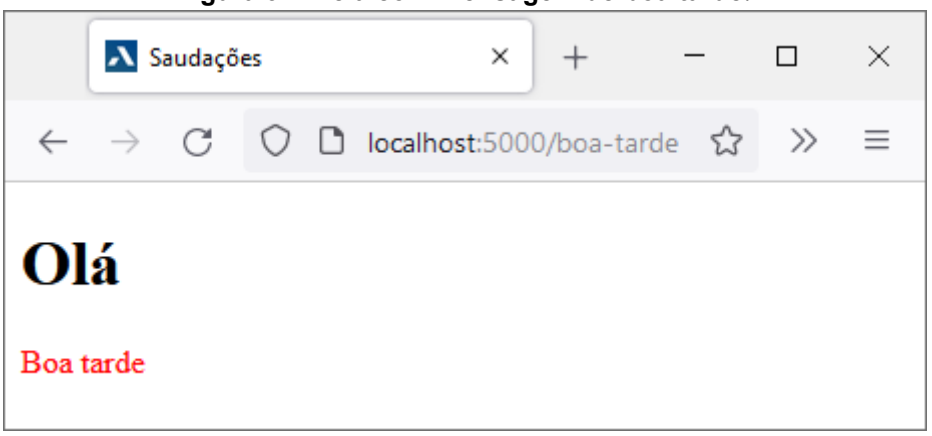
Codificação 9.3. estilos.css

```
.msg {
    color: red;
}
```

Fonte: do autor, 2021

A figura `favicon.png` pode ser qualquer figura PNG que você quiser (usamos o logotipo da Faculdade Impacta para exemplificar aqui). Essa figura será utilizada como ícone da página na barra de abas do navegador. Ao acessar a URL `http://localhost:5000/boa-tarde`, uma tela semelhante a da Figura 9.2 aparecerá.

Figura 9.2. Tela com mensagem de boa tarde.



Fonte: do autor, 2021

Para entendermos o que aconteceu até aqui, vamos ver por partes. Primeiramente, o `render_template` é a função do Flask, que como o nome diz, é responsável por renderizar *templates*. Se necessário, o retorno dessa função pode ser manipulado com o objeto `response` tal como exemplificado na seção 8.8 do capítulo anterior. Para a função `render_template`, qualquer número de parâmetros nomeados pode ser passado para que os mesmos tornem-se disponíveis dentro do *template*. Normalmente, a maioria do conteúdo de um *template* são as partes estáticas, também chamadas de fixas, que são aquelas que nunca mudam. No entanto, os *templates* têm também partes dinâmicas, que são aquelas que estão demarcadas por `{{` e por `}}` ou, como veremos mais adiante, por `{%` e por `%}`. As partes dinâmicas têm seu conteúdo definido apenas em tempo de execução pela função `render_template`, podendo os seus conteúdos variarem dependendo das variáveis do programa.

No exemplo, temos um parâmetro nomeado utilizado na função `render_template` chamado `mensagem`, que pode ser acessado no *template* com a parte dinâmica `{{mensagem}}`. Além disso, um *link* para um arquivo estático (os que estão na pasta `static`) pode ser disponibilizado por meio do uso de `{{ url_for('static', filename = '<nome-do-arquivo>') }}`. A função `url_for` serve para montar URLs para recursos gerenciados pelo Flask / Werkzeug, tal como demonstrado no exemplo.

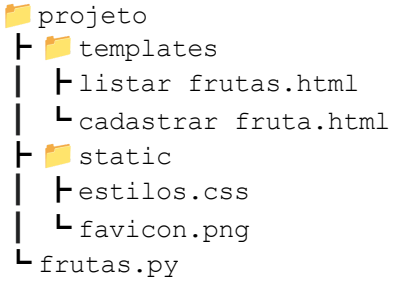
Atenção:

1. Você pode utilizar diversos parâmetros com o `render_template`. Por exemplo, `render_template("exemplo.html", nome = "João", idade = 23)`.
2. Você também pode utilizar o `url_for` para criar dinamicamente um *link* para uma de suas funções decoradas com o `@app.route(...)`. Por exemplo, se você tem uma função `teste` mapeada como `"/exemplo/<nome>/<idade>"`, você pode criar um *link* para ela ao usar `{{ url_for('teste', nome = 'Paulo', idade = 8) }}`.

9.3. Ifs e fors com Jinja2

Consideremos agora, o projeto da Figura 9.3:

Figura 9.3. Arquivos de um segundo projeto com Flask e Jinja2.



Fonte: do Autor, 2021.

Vejamos então o arquivo `frutas.py`:

Codificação 9.4. frutas.py

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

frutas = [
    {"nome": "uva", "cor": "roxa"},
    {"nome": "maçã", "cor": "vermelha"},
    {"nome": "morango", "cor": "vermelha"}
]

@app.route("/frutas")
def listar_frutas():
    return render_template("listar_frutas.html", listagem = frutas)

@app.route("/frutas/novo")
def form_criar_frutas():
    return render_template("cadastrar_fruta.html")

@app.route("/frutas", methods = ["POST"])
def criar_fruta():
    f = request.form
    # Exercício: Validar se os campos do form da requisição estão ok.
    frutas.append({nome: f["nome"], cor: f["cor"]})
    return redirect("/frutas")

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

Fonte: do autor, 2021

Eis o conteúdo do arquivo `listar_frutas.html`:

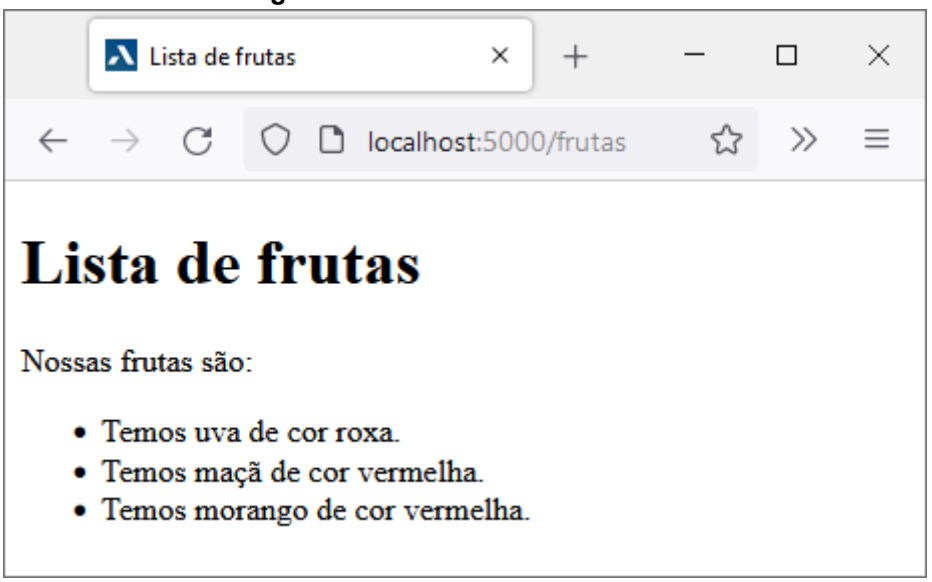
Codificação 9.5. listar_frutas.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <meta name="viewport"
        content="width=device-width, initial-scale=1.0">
    <title>Lista de frutas</title>
    <link rel="icon" type="image/png"
        href="{{ url_for('static', filename = 'favicon.png') }}" />
    <link rel="stylesheet" type="text/css"
        href="{{ url_for('static', filename = 'estilos.css') }}" />
  </head>
  <body>
    <h1>Lista de frutas</h1>
    {% if listagem | length == 0 %}
      <p>Desculpe, não há frutas cadastradas.</p>
    {% else %}
      <p>Nossas frutas são:</p>
      <ul>
        {% for f in listagem %}
          <li>Temos {{f['nome']}} de cor {{f['cor']}}.</li>
        {% endfor %}
      </ul>
    {% endif %}
  </body>
</html>
```

Fonte: do autor, 2021

Ao acessar a URL `http://localhost:5000/frutas`, uma tela semelhante a da Figura 9.4 aparecerá, onde se vê a lista de figuras.

Figura 9.4. Tela com lista de frutas



Fonte: do autor, 2021

Se você trocar a definição da lista de frutas no código por `frutas = []`, verá a mensagem dizendo que não há frutas cadastradas.

Para entender o que aconteceu, primeiramente notamos que a rota `"/frutas"` invoca a função `render_template` passando o parâmetro `listagem = frutas`, e portanto, dentro do *template*, haverá uma variável chamada `listagem` que conterá uma lista do Python.

No template do Jinja2, podemos utilizar um bloco `{% if condição %}` para verificar uma condição. Cada bloco `{% if ... %}` deve ter um bloco `{% endif %}` correspondente. Entre eles, blocos `{% elseif condição %}` e `{% else %}` também são permitidos. Ao utilizarmos `{% if listagem | length == 0 %}`, criamos uma condição que verificará se a lista é vazia e se for, uma mensagem é exibida. Com o bloco `{% else %}`, definimos o que será exibido se a lista não estiver vazia.

Outro bloco que podemos usar é o `{% for variável in expressão %}`, que deve ser terminado com um `{% endfor %}`. Este bloco, permite que uma parte do template possa ser repetida diversas vezes, com cada iteração correspondendo ao item de uma lista. Como a lista que estamos iterando é a listagem, que contém em cada elemento um dicionário representando uma fruta, ao fazer `{% for f in listagem %}` estamos dando o nome de `f` para a fruta de cada iteração. Sendo a variável `f` um dicionário que representa uma fruta, podemos utilizar `{{f['nome']}}` para acessar o seu nome e `{{f['cor']}}` para acessar a sua cor.

9.4. Formulários com Jinja2

Na figura 9.3, há também o arquivo `cadastrar_fruta.html`, que também é referenciado no código de `frutas.py`. Eis o seu conteúdo:

Codificação 9.6. cadastrar_fruta.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <meta name="viewport"
        content="width=device-width, initial-scale=1.0">
    <title>Cadastrar fruta</title>
    <link rel="icon" type="image/png"
        href="{{ url_for('static', filename = 'favicon.png') }}" />
    <link rel="stylesheet" type="text/css"
        href="{{ url_for('static', filename = 'estilos.css') }}" />
  </head>
  <body>
    <h1>Cadastrar fruta</h1>
    <form action="/frutas" method="POST">
      <div>
        <label for="campo_nome">Nome da fruta:</label>
        <input type="text" id="campo_nome" name="nome" />
      </div>
      <div>
        <label for="campo_cor">Cor da fruta:</label>
        <input type="text" id="campo_cor" name="cor" />
```

```
      </div>
      <div><button type="submit">Salvar</button></div>
    </form>
  </body>
</html>
```

Fonte: do autor, 2021

Primeiramente, notamos que este arquivo contém um formulário HTML. O campo `action` deste formulário contém `"/frutas"`, e usa o verbo POST, e portanto, ao ser submetido, enviará a requisição à função `criar_fruta`. Dentro deste formulário, temos dois campos de nomes `nome` e `cor`.

Já na função `criar_fruta`, capturamos o conteúdo do formulário com `request.form`. O seu funcionamento é análogo ao funcionamento do `request.args`, inclusive no que concerne a campos multivalorados. Assim sendo, é possível ler-se os valores dos dados enviados no formulário da requisição dessa forma.

Experimente acessar a URL `http://localhost:5000/frutas/novo` e teste o formulário para cadastrar algumas frutas.

Atenção:

É uma boa prática de programação sempre dar um *redirect* após receber um POST quando se está disponibilizando conteúdo em HTML, forçando então o navegador a fazer um GET assim que receber a resposta do POST. Isso serve para evitar casos de POSTs duplicados, em especial quando o usuário utiliza o botão de voltar de seu navegador. O *redirect* pode ser dado tanto pela função `redirect` do Flask ou por meio de alguma outra forma de retorno que tenha um *status* 3xx.

9.5. Herança de templates

Talvez, você tenha percebido que grande parte dos conteúdos dos *templates* que utilizamos são repetitivos, o que tende a atrapalhar na hora de realizar manutenção neste código, vez que isso implicaria em copiar e colar códigos em diversos *templates*. Para resolver esse problema, o Jinja2 disponibiliza o conceito de herança de *templates*.

Vamos aplicar o conceito de herança de *templates* no código das duas seções anteriores. Nada precisará ser alterado no lado do Flask / Python, sendo todas as mudanças necessárias realizadas apenas nos *templates*. Primeiramente, colocamos o conteúdo repetitivo em um novo *template* que vamos chamar de `modelo.html`:

Codificação 9.7. modelo.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <meta name="viewport"
        content="width=device-width, initial-scale=1.0">
    <title>{% block titulo1 %}{% endblock %}</title>
    <link rel="icon" type="image/png"
        href="{{ url_for('static', filename = 'favicon.png') }}" />
    <link rel="stylesheet" type="text/css"
```

```
        href="{{ url_for('static', filename = 'estilos.css') }}" />
  </head>
  <body>
    <h1>{% block titulo2 %}{% endblock %}</h1>
    <div class="tudo">
      {% block conteudo %}{% endblock %}
    </div>
  </body>
</html>
```

Fonte: do autor, 2021

Observe que esse *template* tem três blocos, que representam as partes dinâmicas dele a serem preenchidos pelo *template* que dele herdar. Cada bloco tem um nome específico e é delimitado por `{% block <nome-do-bloco %}`. Além disso, cada bloco um deve ter um `{% endblock %}` correspondente. Dessa forma, os três blocos que temos, chamam-se respectivamente `titulo1`, `titulo2` e `conteudo`. Além disso, colocamos o bloco `conteudo` dentro de uma *tag* `<div>`. E sim, temos `titulo1` e `titulo2` porque um é o título dentro da *tag* `<title>` e o outro é o da *tag* `<h1>`.

Então, reescrevemos os nossos dois *templates* para herdarem de `modelo.html`. Primeiramente, comecemos com `listar_fruta.html`:

Codificação 9.7. Reescrevendo listar_fruta.html

```
{% extends "modelo.html" %}
{% block titulo1 %}Lista de frutas{% endblock %}
{% block titulo2 %}Lista de frutas{% endblock %}
{% block conteudo %}
  {% if listagem | length == 0 %}
    <p>Desculpe, não há frutas cadastradas.</p>
  {% else %}
    <p>Nossas frutas são:</p>
    <ul>
      {% for f in listagem %}
        <li>Temos {{f['nome']}} de cor {{f['cor']}}.</li>
      {% endfor %}
    </ul>
  {% endif %}
{% endblock %}
```

Fonte: do autor, 2021

Em seguida, é a vez de `cadastrar_fruta.html`:

Codificação 9.8. Reescrevendo cadastrar_fruta.html

```
{% extends "modelo.html" %}
{% block titulo1 %}Cadastrar fruta{% endblock %}
{% block titulo2 %}Cadastrar fruta{% endblock %}
{% block conteudo %}
  <form action="/frutas" method="POST">
    <div>
      <label for="campo_nome">Nome da fruta:</label>
      <input type="text" id="campo_nome" name="nome" />
    </div>
    <div>
      <label for="campo_cor">Cor da fruta:</label>
```

```
      <input type="text" id="campo_cor" name="cor" />
    </div>
    <div><button type="submit">Salvar</button></div>
  </form>
{% endblock %}
```

Fonte: do autor, 2021

Nas *templates* que herdam da *template* principal, primeiramente define-se de qual *template* será realizada a herança. Isso é feito por meio da *tag* `{% extends "nome-da-template-base" %}`. Em seguida, você pode definir o conteúdo de cada bloco, enquanto que a estrutura geral será herdada da *template* `modelo.html`. Dessa forma, o conteúdo repetitivo fica centralizado em uma *template*, ficando nas demais *templates* apenas o conteúdo que lhes é único, resultando em *templates* mais fáceis de entender, mais fáceis de gerenciar e mais fáceis de alterar.

Atenção:

Você pode ter quantas *templates* base quiser para que outras *templates* delas herdem. Você também pode fazer uma *template* A herdar de uma *template* B que por sua vez herda de uma *template* C. No entanto, você não pode fazer uma *template* herdar diretamente de duas ou mais *templates*. O nome da *template* base também não precisa ser fixo, ele pode ser também uma variável informada na função `render_template`.

9.6. Utilizando *cookies* para criar uma tela de login

Um desafio comum no desenvolvimento de aplicações *web* é o de reconhecer um usuário quando o mesmo revisita o *site*. A forma mais utilizada para tal é por meio de *cookies*. Os *cookies* são emitidos pelo servidor (no cabeçalho `Set-Cookie` da resposta HTTP) e reenviados pelo navegador sempre que o site é revisitado (no cabeçalho `Cookie` da requisição HTTP).

Embora seja possível manipular-se os cabeçalhos dos *cookies* diretamente, o Flask já dispõe de funcionalidades que permitem manipular-se os *cookies* sem que o programador precise diretamente lidar com a complexidade e a dificuldade de gerenciar-se esses cabeçalhos corretamente. Embora o conjunto de *cookies* seja um conjunto de chaves e valores de *strings*, manipulá-los diretamente não é fácil, pois há várias outras dificuldades envolvidas tais como tempo de expiração de *cookies*, criptografia, restrições de origem e mais algumas questões de segurança, complexidades essas abstraídas pelo Flask.

Para ler os valores dos *cookies* enviados na requisição HTTP, utilizamos os valores retornados por `request.cookies`. O `request.cookies` funciona de forma análoga ao `request.args` e ao `request.form`. Já para definir o valor de um *cookie* na resposta HTTP, utilizamos o objeto `response` (aquele criado pela função `make_response`) e nele chamamos o método `set_cookie`. Veja um exemplo de um código que utiliza os *cookies* para implementar uma funcionalidade de *login*:

Codificação 9.9. Utilização dos *cookies* para implementar uma funcionalidade de *login*

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

usuarios = [
    {"login": "Maria", "senha": "1234"},
    {"login": "Roberto", "senha": "4321"},
    {"login": "Carlos", "senha": "abcd"},
    {"login": "Paula", "senha": "xyz"}
]

def verificar_login(login, senha):
    for u in usuarios:
        if u["login"] == login and u["senha"] == senha:
            return u
    return None

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

def autenticar_login():
    login = request.cookies.get("login", "")
    senha = request.cookies.get("senha", "")
    return verificar_login(login, senha)

@app.route("/login")
def form_login():
    logado = autenticar_login()
    if logado is not None:
        return redirect("/dashboard")
    return render_template("login.html", err="")

@app.route("/")
@app.route("/dashboard")
def dashboard():
    logado = autenticar_login()
    if logado is None:
        return redirect("/login")
    return render_template("dashboard.html", user = logado)

@app.route("/login", methods = ["POST"])
def fazer_login():
    # Exercício: Validar se os campos do form da requisição estão
    # corretos.
    login = request.form.get("login", "")
    senha = request.form.get("senha", "")
    logado = verificar_login(login, senha)
    if logado is None:
        return render_template("login.html", err="Senha errada"), 302
    resposta = make_response(redirect("/dashboard"))
    resposta.set_cookie("login", login,
        httponly = True, samesite = "Strict")
    resposta.set_cookie("senha", senha,
        httponly = True, samesite = "Strict")
```

```
    return resposta

@app.route("/logout", methods = ["POST"])
def logout():
    t = render_template("login.html", err="Tchau.")
    resposta = make_response(t)
    resposta.set_cookie("login", "",
        httponly = True, samesite = "Strict")
    resposta.set_cookie("senha", "",
        httponly = True, samesite = "Strict")
    return resposta

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

Fonte: do autor, 2021

Para deixar este código completo e executável, precisamos ainda incluir os templates que são referenciados. O primeiro deles é a tela de *login* (`login.html`). Vejamos como é o código de um HTML minimalista para implementar tal funcionalidade:

Codificação 9.10. login.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <meta name="viewport"
        content="width=device-width, initial-scale=1.0">
    <title>Login</title>
  </head>
  <body>
    <div>{{err}}</div>
    <form action="/login" method="POST">
      <div>
        <label for="campo_login">Login:</label>
        <input type="text" id="campo_login" name="login" />
      </div>
      <div>
        <label for="campo_senha">Senha:</label>
        <input type="password" id="campo_senha" name="senha" />
      </div>
      <div><button type="submit">Entrar</button></div>
    </form>
  </body>
</html>
```

Fonte: do autor, 2021

E também, temos a tela do *dashboard* (`dashboard.html`):

Codificação 9.11. dashboard.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <meta name="viewport"
```

```
        content="width=device-width, initial-scale=1.0">
    <title>Dashboard</title>
  </head>
  <body>
    <div>Olá, {{user['login']}}. Seja bem-vindo(a).</div>
    <form action="/logout" method="POST">
      <div><button type="submit">Sair</button></div>
    </form>
  </body>
</html>
```

Fonte: do autor, 2021

Neste código, temos diversas funções:

- `verificar_login` é responsável por consultar o banco de dados para saber se um *login* e uma senha correspondem a algum usuário. Se corresponderem, um objeto representando este usuário é retornado. Caso contrário, `None` é retornado.
- `autenticar_login` lê o login e senha a partir dos *cookies* e verifica se são válidos ao retornar resultado de `verificar_login`. Essa função é utilizada para proteger todas as rotas que precisam de *login* para serem acessadas.
- `form_login` fornece ao navegador o formulário de *login* por meio da função `render_template`. Mas se o usuário já estiver logado, ele será redirecionado à página principal da aplicação (o *dashboard*).
- `dashboard` fornece ao navegador a tela principal da aplicação por meio da função `render_template`. Mas se o usuário não estiver logado, ele será redirecionado à página de *login*.
- `login` realiza o *login* do usuário. Se o *login* for bem sucedido, as informações de login serão salvas nos *cookies* e o usuário será redirecionado à tela principal da aplicação. Se não for bem sucedido, ele será redirecionado de volta à tela de *login* e uma mensagem de erro será exibida.
- `logout` realiza o *logout* do usuário ao limpar o conteúdo de seus *cookies* e redirecioná-lo à página de *login*.

O método `set_cookie` merece atenção especial. O primeiro parâmetro é o nome do *cookie*. O segundo parâmetro é o valor do *cookie*. Os demais parâmetros definem a segurança do *cookie*. O parâmetro `httponly = True` faz com que o *cookie* não seja legível via JavaScript, o que poderia possibilitar que o *cookie* fosse roubado por um terceiro mal-intencionado que tenha algum controle sobre o JavaScript enviado ao navegador do cliente ao utilizar `document.cookie`. O parâmetro `samesite = "Strict"` proíbe que o cookie seja utilizado por requisições que não se originem do próprio *site*. Poderíamos acrescentar também o parâmetro `secure = True` se quisermos fazer com que o *cookie* só seja legível se o protocolo for HTTPS, e não HTTP.

Teste esse código colocando um dos usuários e senhas definidos e efetuando o *login*. Estando logado, teste o botão de *logout*. Sem estar logado, tente acessar a URL do *dashboard* diretamente. Estando logado, tente acessar a URL da tela de *login* diretamente. Finalmente, estando deslogado, tente logar com a senha errada. Se tudo estiver correto, todos esses testes terão o resultado esperado.

9.7. Utilizando a sessão

Entretanto, uma forma mais produtiva de gerenciar o *login* e as informações que são gravadas no navegador do cliente é por meio da sessão (objeto `session`). A sessão permite ao programador colocar dados arbitrários dentro de um *cookie* assinado digitalmente sem ter que se preocupar em como os *cookies* são gerenciados, codificados, lidos ou protegidos. Vamos reescrever o código anterior utilizando a sessão:

Codificação 9.12. Utilizando a sessão

```
from flask import Flask, request, session
from flask import make_response, redirect, render_template
app = Flask(__name__)

# Exercício: Use uma chave verdadeiramente segura lida do env ou de
# algum outro lugar seguro para que ela não apareça diretamente no
# código-fonte.
app.secret_key = "Grande segredo secreto e misterioso"

usuarios = [
    {"login": "Maria", "senha": "1234"},
    {"login": "Roberto", "senha": "4321"},
    {"login": "Carlos", "senha": "abcd"},
    {"login": "Paula", "senha": "xyz"}
]

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

def verificar_login(login, senha):
    for u in usuarios:
        if u["login"] == login and u["senha"] == senha:
            return u
    return None

@app.route("/login")
def form_login():
    if "logado" in session:
        return redirect("/dashboard")
    return render_template("login.html", err="")

@app.route("/")
@app.route("/dashboard")
def dashboard():
    if "logado" not in session:
        return redirect("/login")
    return render_template("dashboard.html", user=session["logado"])

@app.route("/login", methods = ["POST"])
def fazer_login():
    # Exercício: Validar se os campos do form da requisição estão
    # corretos.
    login = request.form.get("login", "")
    senha = request.form.get("senha", "")
    logado = verificar_login(login, senha)
```

```
    if logado is None:
        return render_template("login.html", err="Senha errada"), 302
    session["logado"] = logado
    return redirect("/dashboard")

@app.route("/logout", methods = ["POST"])
def logout():
    session.pop("logado", None)
    return render_template("login.html", err="Tchau.")

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

Fonte: do autor, 2021

O código resultante é visivelmente muito mais simples. Nele, podemos notar que a função `autenticar_login`, deixou de ser necessária, sendo substituída simplesmente por um teste que verifica se uma chave `"logado"` está dentro do `session`. O usuário resultante de `verificar_login` é diretamente adicionado ao `session`, e junto com ele, todas as propriedades e atributos nele existentes, sem que precisemos separar o *login* da senha ou de quaisquer outras coisas que queiramos colocar junto. A chave `"logado"` é deletada na função `logout` por meio do `session.pop("logado", None)`. Um detalhe a se notar é que para que a sessão possa ser devidamente assinada digitalmente, o `app.secret_key` tem que estar definido com uma chave secreta que será utilizada para a realização da criptografia. Idealmente, coloque essa chave secreta em um arquivo de configuração seguro.

Finalmente, teste esse código da mesma forma que você fez com o código que usava *cookies* ao invés de sessão. O comportamento de ambos os códigos deve ser o mesmo.

Atenção:

Você pode colocar praticamente qualquer coisa que você quiser nos *cookies* ou na sessão. As possibilidades vão muito além da funcionalidade de *login*. Por exemplo, uma funcionalidade típica de *sites* de *e-commerce* que também é muitas vezes implementada por meio da sessão é o carrinho de compras, onde os itens escolhidos pelo usuário são armazenados na sessão.

9.8. Notificando o usuário com mensagens na sessão

Uma funcionalidade muito útil para notificar-se o usuário de forma limpa é por meio da função `flash`, que enfileira uma mensagem para ser visualizada pelo usuário dentro da sessão. As mensagens são então lidas (e desenfileiradas) por meio da função `get_flashed_messages` no Jinja2. Vejamos um exemplo simples:

Codificação 9.13. Notificando o usuário com mensagens na sessão

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

@app.route("/visitar/<planeta>")
def visitar_planeta(planeta):
```

```
    return "OK"

@app.route("/planetas")
def listar_planetas_visitados_recentemente():
    return render_template("planetas.html")

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

Fonte: do autor, 2021

E também, temos o arquivo (`planetas.html`):

Codificação 9.14. planetas.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <meta name="viewport"
        content="width=device-width, initial-scale=1.0">
    <title>Planetas visitados</title>
  </head>
  <body>
    {% with planetas = get_flashed_messages() %}
      {% if planetas %}
        <p>Os planetas visitados recentemente são:</p>
        <ul>
          {% for p in planetas %}
            <li>{{p}}.</li>
          {% endfor %}
        </ul>
      {% else %}
# Continua na próxima página.
# Continuação da página anterior.
        <p>Desculpe, nenhum planeta foi visitado recentemente.</p>
      {% endif %}
    {% endwith %}
  </body>
</html>
```

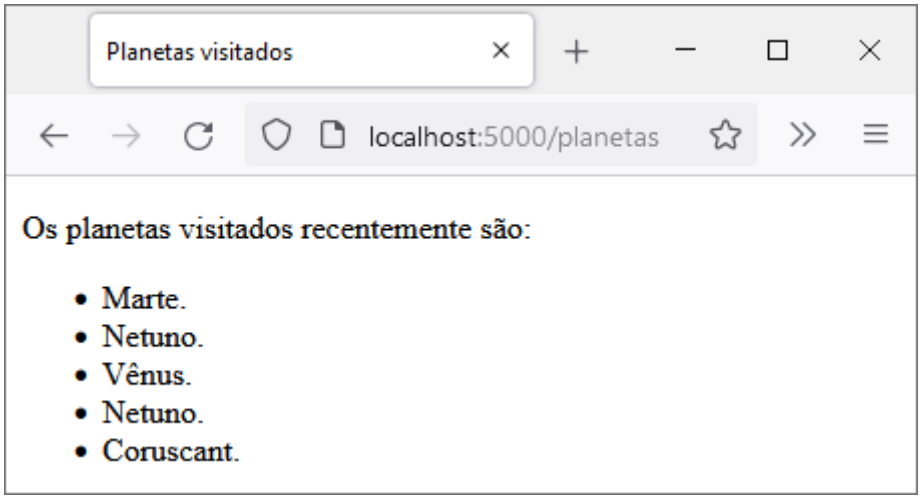
Fonte: do autor, 2021

Executemos esse código, e visitemos as seguintes URLs (nesta ordem):

- `http://localhost:5000/visitar/Marte`
- `http://localhost:5000/visitar/Netuno`
- `http://localhost:5000/visitar/Vênus`
- `http://localhost:5000/visitar/Netuno`
- `http://localhost:5000/visitar/Coruscant`
- `http://localhost:5000/planetas`

O resultado, deve ser semelhante ao da figura 9.5:

Figura 9.5. Tela com lista de planetas visitados.



Fonte: do autor, 2021

No entanto, se revisitarmos a última URL, o resultado será que nenhum planeta foi visitado recentemente. Isso ocorre porque a função `get_flashed_messages` desenfileira as mensagens que estavam guardadas.

Se você tentar executar este processo em navegadores diferentes, visitando planetas diferentes em cada um deles, verá que cada navegador tem a sua sessão independente e portanto serão listados apenas os planetas visitados pelo usuário desde a última visita à rota `"/planetas"`, sem que as visitas executadas por outros usuários interfiram.

Esse mecanismo é muito útil para implementar mensagens e avisos ao usuário, principalmente em processos assíncronos. Combinando com JavaScript, possibilita que mensagens apareçam na tela em formato de *popup* quando uma página é carregada.

Vamos praticar?

Ainda haverão conceitos de modularização e organização de código a serem aprendidos nos capítulos seguintes, mas com o que temos apresentado até aqui, você já deverá ter o suficiente para criar tanto o *back-end* quanto o *front-end* de um projeto com Flask. Esta é uma parte muito importante da sua jornada na Faculdade Impacta, pois é aqui que todas as peças de banco de dados, lógica de programação, desenvolvimento *front-end* e desenvolvimento *back-end* se encontram, permitindo que você já possa começar a desenvolver suas primeiras aplicações completas. Recomendamos então, que pratique com os seguintes exercícios:

- Junte o banco de dados, o *back-end* e o *front-end* e tente implementar uma aplicação completa, embora pequena e simples, que permita a realização de um CRUD de alguma entidade de negócio de um sistema, tal como funcionário, cliente, pedido ou fornecedor. Não esqueça de implementar a funcionalidade de *login* e *logout*, de proteger adequadamente as rotas que necessitam que o usuário esteja logado e de validar todos os dados adequadamente.

- Incremente suas páginas com JavaScript. Você pode colocar os arquivos JavaScript dentro da pasta `static`. e referenciá-los com `url_for`. Ou então, poderá colocar tags `<script>` dentro dos seus arquivos HTML.
- Para abordar tudo o que o Jinja2 e o Flask oferecem, decerto precisaríamos de centenas de páginas de conteúdo, extrapolando muito o propósito deste material. Assim sendo, há diversos recursos interessantes e úteis que deixamos de fora para não engrossar o conteúdo desta unidade excessivamente. Portanto, recomendamos que você pesquise, explore e experimente você mesmo mais profundamente esses recursos.

Referências

Jouravlev, Michael. **Redirect After Post.** 2004. Disponível em: <https://www.theserverside.com/news/1365146/Redirect-After-Post>. Acesso em 25 ago. 2021.

The Pallets Projects. **Security Considerations.** (s.d.) Disponível em: <https://flask.palletsprojects.com/en/2.0.x/security/>. Acesso em 25 ago. 2021.

The Pallets Projects. **Sessions.** (s.d.) Disponível em: <https://flask.palletsprojects.com/en/2.0.x/api/?highlight=session#sessions>. Acesso em 25 ago. 2021.

The Pallets Projects. **Template Designer Documentation.** (s.d.) Disponível em: <https://jinja.palletsprojects.com/en/3.0.x/templates/>. Acesso em 25 ago. 2021.