2 Templates

Resumo: *Nesta segunda parte sobre a implementação de aplicativos web, vamos discutir como trabalhar com Templates - modelos* `html` *que facilitarão a construção do nosso projeto!*

Introdução

Após concluir a nossa primeira parte, com a apresentação de um servidor web e configuração do nosso sistema para o desenvolvimento, nós já temos um aplicativo web simples, mas funcional, com a seguinte estrutura de arquivo:

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

Vimos que, para executar o aplicativo, foi necessário definirmos `FLASK_APP=bloguvv.py` na nossa sessão de terminal e executamos `flask run`. Isso iniciou um servidor web com o aplicativo, que foi possível abrirmos digitando a URL `http://localhost:5000/` na barra de endereços do navegador.

Agora, vamos continuar trabalhando no mesmo aplicativo e, em particular, aprendendo a como gerar páginas web mais elaboradas que tenham uma estrutura complexa e muitos componentes dinâmicos. Se algo sobre o aplicativo ou o fluxo de trabalho de desenvolvimento até agora não estiver claro, revise o material novamente antes de continuar.

Modo `DEBUG`

Os aplicativos `Flask` podem ser executados opcionalmente no modo de depuração (**debug**). Neste modo, dois módulos muito convenientes do servidor de desenvolvimento chamados `reloader` e `debugger` são habilitados por padrão.

Quando o `reloader` está habilitado, o `Flask` observa todos os arquivos de código-fonte do seu projeto e reinicia automaticamente o servidor quando qualquer um dos arquivos é modificado. Ter

um servidor em execução com o `reloader` habilitado é extremamente útil durante o desenvolvimento, porque toda vez que você modifica e salva um arquivo-fonte, o servidor reinicia automaticamente e pega a alteração.

O `debugger` é uma ferramenta baseada na web que aparece no seu navegador quando seu aplicativo gera uma exceção não tratada. A janela do navegador da web se transforma em um **rastreio de pilha interativo** (*interactive stack trace*) que permite que nós inspecionemos o código-fonte e avaliemos expressões em qualquer lugar na pilha de chamadas.

> **Atenção!**
>
> Nunca habilite o modo de depuração em um servidor de produção. O depurador em particular permite que o cliente solicite execução remota de código, então ele torna seu servidor de produção vulnerável a ataques. Como uma simples medida de proteção, o depurador precisa ser ativado com um PIN, impresso no console pelo comando `flask run`.

Por padrão, o modo de depuração está desabilitado. Para habilitá-lo, defina uma variável de ambiente `FLASK_DEBUG=1` antes de invocar flask run:

```
(venv) $ export FLASK_APP=hello.py
(venv) $ export FLASK_DEBUG=1
(venv) $ flask run
```

Se você estiver usando o Microsoft Windows, utilize `set` ao invés de `export` para definir as variáveis.

O que são *Templates*?

Para começar, queremos que a página inicial do nosso aplicativo tenha um título que dê boas-vindas aos usuários. Por enquanto, vamos ignorar o fato de que o aplicativo ainda não tem o conceito de usuários, pois isso virá depois. Em vez disso, vamos utilizar um usuário simulado, implementando como um dicionário `Python`, da seguinte forma:

```
user = {'username': 'Wanderson'}
```

> **Nota**
>
> Criar objetos simulados é uma técnica útil que permite que você se concentre em uma parte do aplicativo sem ter que se preocupar com outras partes do sistema que ainda não existem.

Por ora, desejamos projetar a página inicial do nosso aplicativo e não queremos que o fato de não ter um sistema de usuário em funcionamento nos distraia. Assim, vamos apenas criar um objeto de usuário para poder continuar.

A função de visualização no aplicativo retorna uma `string` simples (estamos nos referindo ao arquivo `routes.py`). O que desejamos fazer agora é expandir essa `string` retornada em uma página `HTML` completa. Por exemplo:

```
# app/routes.py

from app import app

@app.route('/')
@app.route('/index')
def index():
    user = {'username': 'Wanderson'}
    return '''
<html>
    <head>
        <title>Home Page - Blog UVV</title>
    </head>
    <body>
        <h1>Ola, ''' + user['username'] + '''!</h1>
    </body>
</html>'''
```

Caso precise revisar HTML, recomendo que consulte o material disponível em site W3C para uma breve revisão.

Atualize a função visualização conforme mostrado acima e execute o aplicativo novamente para ver como ele fica no seu navegador.

A solução usada acima para entregar HTML ao navegador não é boa! Por exemplo, considere o quão complexo o código nesta função de visualização se tornará quando você adicionar postagens de blog de usuários, que mudarão constantemente. O aplicativo também terá mais funções de visualização que serão associadas a outras URLs, então imagine se um dia decidirmos alterar o *layout* deste aplicativo e tiver que atualizar o HTML em cada função de visualização. Isso claramente não é uma opção que escalará conforme o aplicativo cresce.

Se pudermos manter a lógica do nosso aplicativo separada do layout ou apresentação de suas páginas da web, as coisas serão muito mais bem organizadas, concorda? Poderíamos até contratar um web designer para criar um site matador enquanto codificamos a lógica do aplicativo em `Python`.

Pois bem, os modelos ajudam a atingir essa separação entre a apresentação e a lógica de negócios. No `Flask`, os modelos são escritos como arquivos separados, armazenados em uma pasta de templates que está dentro do pacote do aplicativo. Depois de certificar-se de que estamos no diretório do `bloguvv`, vamos criar o diretório onde os modelos serão armazenados:

```
(venv) $ mkdir app/templates
```

Abaixo você pode ver seu primeiro template, que é similar em funcionalidade à página HTML retornada pela função de visualização `index()` acima. Agora, vamos escrever este arquivo em `app/templates/index.html`:

```
<!DOCTYPE html>
<html>
    <head>
        <title>{{ title }} - Blog UVV</title>
    </head>
    <body>
```

```
        <h1>Ola, {{ user.username }}!</h1>
    </body>
</html>
```

Esta é uma página HTML curta e padrão. A única coisa interessante nesta página é que há alguns *placeholders* para o conteúdo dinâmico, incluídos em seções `{{ ... }}`. Esses *placeholders* representam as partes da página que são variáveis e só serão conhecidas em tempo de execução.

Agora que a apresentação da página foi descarregada para o modelo HTML, a função de visualização pode ser simplificada:

```
# app/routes.py

from app import app

@app.route('/')
@app.route('/index')
def index():
    user = {'username': 'Wanderson'}
    return render_template('index.html', title='Home', user=user)
```

Isso parece muito melhor, certo? Experimente esta nova versão do aplicativo para ver como o modelo funciona. Depois que a página for carregada no seu navegador, você pode querer visualizar o HTML de origem e compará-lo com o modelo original.

A operação que converte um modelo em uma página HTML completa é chamada de **renderização**. Para renderizar o modelo, tivemos que importar uma função que vem com o `framework Flask` chamada `render_template()`. Esta função pega um nome de arquivo de modelo e uma lista variável de argumentos de modelo e retorna o mesmo modelo, mas com todos os marcadores de posição nele substituídos por valores reais.

A função `render_template()` invoca o mecanismo de template `Jinja` que vem junto com o `framework Flask`. `Jinja` substitui blocos `{{ ... }}` pelos valores correspondentes, dados pelos argumentos fornecidos na chamada `render_template()`.

Declarações Condicionais

Vimos como o Jinja substitui *placeholders* por valores reais durante a renderização, mas esta é apenas uma das muitas operações poderosas que o Jinja suporta em arquivos de modelo. Por exemplo, os modelos também suportam instruções de controle, fornecidas dentro de blocos `{% ... %}`. A próxima versão do modelo `index.html` adiciona uma instrução condicional:

```
<!-- app/templates/index.html -->

<!DOCTYPE html>
<html>
    <head>
        {% if title %}
        <title>{{ title }} - Blog UVV</title>
        {% else %}
        <title>Bem vindo ao Blog UVV!</title>
```

```
        {% endif %}
    </head>
    <body>
        <h1>Ola, {{ user.username }}!</h1>
    </body>
</html>
```

Agora o template está um pouco mais inteligente. Se a função de visualização esquecer de passar um valor para a variável *placeholder* `title`, então, em vez de mostrar um título vazio, o template fornecerá um padrão. Você pode tentar como essa condicional funciona removendo o argumento `title` na chamada `render_template()` da função de visualização.

Loops

O usuário logado provavelmente vai querer ver postagens recentes de usuários conectados na página inicial, então o que vamos fazer agora é estender o aplicativo para dar suporte a isso.

Mais uma vez, vamos contar com o truque prático do objeto simulado para criar alguns usuários e algumas postagens para mostrar:

```
# app/routes.py

from app import app

@app.route('/')
@app.route('/index')
def index():
    user = {'username': 'Wanderson'}
    posts = [
        {
            'author': {'username': 'Joao'},
            'body': 'Belo dia em Vila Velha!'
        },
        {
            'author': {'username': 'Maria'},
            'body': 'Bora para o cinema hoje?'
        }
    ]
    return render_template('index.html', title='Home', user=user, posts=posts)
```

Para representar postagens de usuários, estamos usando uma lista, onde cada elemento é um dicionário que tem campos de autor e corpo. Quando conseguirmos implementar usuários e postagens do blog de verdade, vamos tentar preservar esses nomes de campo o máximo possível, para que todo o trabalho que estamos fazendo para projetar e testar o modelo da página inicial usando esses objetos falsos continue válido ao introduzirmos usuários e postagens reais.

No lado do modelo, temos que resolver um novo problema. A lista de postagens pode ter qualquer número de elementos, cabe à função de visualização decidir quantas postagens serão apresentadas na página. O modelo não pode fazer nenhuma suposição sobre quantas postagens existem,

então ele precisa estar preparado para renderizar quantas postagens a visualização enviar de forma genérica.

Para esse tipo de problema, Jinja oferece uma estrutura de controle for:

```
<!-- app/templates/index.html -->

<!DOCTYPE html>
<html>
    <head>
        {% if title %}
        <title>{{ title }} - Blog UVV</title>
        {% else %}
        <title>Bem vindo ao Blog UVV!</title>
        {% endif %}
    </head>
    <body>
        <h1>Ola, {{ user.username }}!</h1>
        {% for post in posts %}
        <div><p>{{ post.author.username }} disse: <b>{{ post.body }}</b></p></↩
            div>
        {% endfor %}
    </body>
</html>
```

Simples, certo? Experimente esta nova versão do aplicativo e não deixe de brincar com a adição de mais conteúdo à lista de posts para ver como o modelo se adapta e sempre renderiza todos os posts que a função view envia.

Herança de Template

A maioria dos aplicativos web hoje em dia têm uma barra de navegação no topo da página com alguns *links* usados com frequência, como um *link* para ediar seu perfil, para fazer *login*, sair, etc. Podemos facilmente adicionar uma barra de nabegação ao modelo `index.html` com mais HTML, porém conforme o aplicativo cresce, precisamos dessa mesma barra de navegação em outras páginas. E, não queremos realmente ter que manter várias cópias da barra de navegação em muitos modelos HTML. É uma boa prática não se repetir, se possível!

O `Jinja` tem um recurso de herança de modelo que aborda especificamente esse problema. Em essência, o que você pode fazer é mover as partes do *layout* da página que são comuns a todos os modelos para um modelo base, do qual todos os outros modelos são derivados.

Então, o que faremos agora é definir um modelo base chamado `base.html` que incluirá uma barra de navegação simples e também a lógica de título que implementamos anteriormente. Iremos precisar escrever o seguinte modelo no arquivo `app/templates/base.html`:

```
<!-- app/templates/base.html -->

<!DOCTYPE html>
<html>
    <head>
```

```
      {% if title %}
        <title>{{ title }} - Blog UVV</title>
        {% else %}
        <title>Bem vindo ao Blog UVV!</title>
      {% endif %}
    </head>
    <body>
        <div>Blog UVV: <a href="/index">Home</a></div>
        <hr>
        {% block content %}{% endblock %}
    </body>
</html>
```

Neste modelo, utilizamos a declaração de controle de bloco para definir o local onde os modelos derivados podem se inserir. Os blocos recebem um nome exclusivo, que os modelos derivados podem referenciar quando fornecem seu conteúdo.

Com o template `base` no lugar, agora podemos simplificar `index.html` fazendo-o herdar de `base.html`:

```
{% extends "base.html" %}

{% block content %}
    <h1>Hi, {{ user.username }}!</h1>
    {% for post in posts %}
    <div><p>{{ post.author.username }} says: <b>{{ post.body }}</b></p></div>
    {% endfor %}
{% endblock %}
```

Como o modelo `base.html` agora cuidará da estrutura geral da página, removemos todos esses elementos de `index.html` e deixamos apenas a parte `content`. A declaração `extends` estabelece o *link* de herança entre os dois modelos, para que o `Jinja` saiba que quando for solicitado a renderizar `index.html`, ele precisa incorporá-lo dentro de `base.html`. Os dois modelos têm declarações de bloco correspondentes com o nome `content`, e é assim que o `Jinja` sabe como combinar os dois modelos em um.

Agora, quando precisarmos criar páginas adicionais para o aplicativo, podemos criá-las como modelos derivados do mesmo modelo `base.html`, e é assim todas as páginas do aplicativo compartilharão a mesma aparência sem duplicação.