3 Formulários WEB

Resumo: *Na parte 2, criamos um modelo simples para a página inicial do aplicativo, utilizando objetos falsos como marcadores de posição para coisas que ainda não temos, como usuários e postagens de blog. Agora, abordaremos uma das muitas lacunas que ainda temos neste aplicativo, especificamente como aceitar entradas de usuários por meio de formulários da web. Os formulários da web são um dos blocos de construção mais básicos em qualquer aplicativo da web. Usaremos formulários para permitir que os usuários enviem postagens de blog e também para fazer login no aplicativo. Antes de prosseguir, precisamos nos certificar que o aplicativo do* **Blog UVV** *esteja instalado como o deixamos na parte anterior e de que possamos executá-lo sem erros.*

Introdução ao Flask-WTF

Para manipular os formulários da web neste aplicativo, utilizaremos a extensão `Flask-WTF`, que é um `wrapper`[1] fino em torno do pacote `WTForms` que o integra bem com o `Flask`. Esta é a primeira extensão `Flask` que estamos de fato utilizando, mas não será a última. As extensões são uma parte muito importante do ecossistema `Flask`, pois fornecem soluções para problemas sobre os quais o `Flask` intencionalmente não opina.

As extensões `Flask` são pacotes `Python` regulares que são instalados com `pip` nos ambientes virtuais. Sendo assim, vamos prosseguir e instalar o `Flask-WTF` nos nossos ambientee virtuais:

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

Até agora o nosso aplicativo é muito simples e, por esse motivo, não precisamo nos preocupar com sua configuração. Mas, para qualquer aplicativo, exceto os mais simples, vamos descobrir que o `Flask` (e possivelmente também as extensões do `Flask`) oferece liberdade em como fazer as coisas e nós é quem precisamos tomar algumas decisões para passar ao `framework` uma lista de variáveis de configuração.

Existem vários formatos para o aplicativo especificar opções de configuração. A solução mais básica é definir suas variáveis como chave em `app.config`, que usa um estilo de dicionário para trabalhar com variáveis. Por exemplo, podemos fazer algo assim:

[1] Um `wrapper` é um empacotador, uma camada de abstração destinada a chamar uma ou mais funções. Alguns exemplos são o `SDK` da AWS, o próprio `request` do HTTP, as `APIs REST` do `Android`, entre outros. Sua função é centralizar o manuseio de bibliotecas e classes de determinadas linguagens.

```
app = Flask(__name__)
app.config['SECRET_KEY'] = 'voce-nunca-saberah'

# ... e gradativamente vamos incluindo variaveis de configuracao
```

Embora a sintaxe acima seja suficiente para criar opções de configuração para o `Flask`, uma boa prática é o princípio da separação de *"concentimentos"*. Ou seja, em vez de colocar as nossoas configuração no mesmo lugar onde criamos o nosso aplicativo, podemos utilizar uma estrutura um pouco mais elaborada que nos permite manter a configuração do sistem em um arquivo separado.

Uma solução que realmente podemos aplicar e que é muito extensível é usar uma classe `Python` para armazenar variáveis de configuração. Para manter as coisas bem organizadas, vamos criar a classe de configuração em um módulo `Python` separado. Abaixo, podemos ver a nova classe de configuração para este aplicativo, armazenada em um módulo `config.py` no diretório de nível superior.

```
import os

class Config:
    SECRET_KEY = os.environ.get('SECRET_KEY') or 'voce-nunca-saberah'
```

Bem simples, certo? As configurações são definidas como variáveis de classe dentro da classe `Config`. Como o aplicativo precisa de mais itens de configuração, eles podem ser adicionados a essa classe e, mais tarde, se descobrirmos que será necessário ter mais de um conjunto de configurações, podemos criar `subclasses` dele. Mas, não se preocupe com isso ainda. Aproveite e observe como ficou a distribuição dos nossos arquivos na estrutura da aplicação até agora.

```
bloguvv/
  venv/
  app/
    __init__.py
    routes.py
    templates/
      base.html
      index.html
  bloguvv.py
  config.py
```

A variável de configuração `SECRET_kEY` que adicionamos como o único item de configuração é uma parte importante na maioria dos aplicativos `Flask`. O `Flask` e algumas de suas extensões usam o valor da chave secreta como uma **chave criptográfica**, útil para gerar assinaturas ou *tokens*. A extensão `Flask_WTF` a usa para proteger formulários da web contra um ataque desagradável chamado **Cross-Site Request Forgery**[2] ou **CSRF** (pronucia-se *"seasurf"*). Como o nome indica, a chave secreta deve ser *secreta*, pois a força dos *tokens* e assinaturas gerados com ela depende de nenhuma pessoa fora dos mantededores confiáveis do aplicativo saber disso.

o valor da chave secreta é definido como uma expressão com dois termos, unidos pelo operador `or`. O primeiro termo procura o valor de uma variável de ambiente, também chamada `SECRET_KEY`. O

[2]Trata-se de um ataque baseado em requisição HTTP feita entre sites na tentativa de se passar por um usuário legítimo. Sua fragilidade envolve a captura dos *Cookies* nas máquinas clientes e, eventualmente, o mal uso de *javascrip* na manipulação de formulários web.

segundo termo é apenas uma *string* codificada. Este é um padrão que devemos repetir frequentemente para variáveis de configuração. A ideia é que um valor originado de uma variável de ambiente seja o preferido, mas se o ambiente não definir a variável, então a *string* codificada será utilizada como padrão. Enquanto estivermos desenvolvendo este aplicativo, os requisitos de segurança são baixos. Entretanto, quando este aplicativo for implantado em um servidor de produção é preponderante definirmos um único e difícil valor para a nossa chave de segurança, de modo que ninguém mais saiba.

Agora que temos um arquivo de configuração, precisamos dizer ao `Flask` para lê-lo e aplicá-lo. Isso pode ser feito logo após a instância do aplicativo `Flask` ser criada usando o método `app.config.from_object()`:

```
from flask import Flask
from config import Config

app = Flask(__name__)
app.config.from_object(Config)

from app import routes
```

A maneira como estamos importando a classe `Config` pode parecer confusa no começo, mas se observarmos como a classe `Flask` (maiúsculo "F") é importada do pacote `flask` (minúsculo "f"), nota-se que estamos fazendo o mesmo com a configuração. O minúsculo “config” é o nome do módulo `Python config.py`, e obviamente aquele com o maiúsculo "C"é a classe real.

Como mencionado acima, os itens de configuração podem ser acessados com uma sintaxe de dicionário de `app.config`. Aqui podemos ver uma sessão rápida com o interpretador `Python` onde verificamos qual é o valor da chave secreta:

```
>>> from microblog import app
>>> app.config['SECRET_KEY']
'you-will-never-guess'
```

Hashing

Os programadores usam o *hashing* para transformar os dados de entrada em um valor de tamanho fixo. Esse valor representa os dados de forma exclusiva, e a técnica de *hashing* facilita a transmissão e o armazenamento seguro de várias formas de dados.

O *hashing* protege os dados contra acesso não autorizado e adulteração. É um ingrediente essencial nos casos de uso de integridade e segurança de dados.

Nesta parte vamos explora tudo o que você precisa saber sobre *hashing* em `Python`.

O que é Hashing em `Python`

Hashing converte dados de entrada, como uma *string*, arquivo ou objeto, em uma *string* de bytes de tamanho fixo. O `hash` ou `digest` representa a entrada de uma maneira única e reproduzível.

O *hashing* desempenha um papel significativo na detecção de manipulação de dados e no aumento da segurança. Ele pode calcular um valor de `hash` para um arquivo, mensagem ou outro tipo

de dado. Um aplicativo armazena o `hash` de forma segura para verificar posteriormente que os dados não foram adulterados.

Um dos usos mais comuns do *hashing* na segurança é o armazenamento de senhas. O *hashing* é uma alternativa viável ao armazenamento de senhas de texto simples em um banco de dados. Quando um usuário digita sua senha, o sistema faz o `hash` antes de armazená-la no banco de dados. Se um hacker acessar o banco de dados, ele descobrirá que a senha é difícil de ser roubada.

As funções de *hashing* do Python tornam tudo isso possível. Essas funções matemáticas permitem que um aplicativo manipule dados em valores de `hash`.

Como criar uma função Hashing eficaz

Uma função de hash deve atender aos seguintes critérios para ser eficaz e segura:

- **Determinística**: dada a mesma entrada, a função deve sempre retornar a mesma saída.
- **Eficiente**: deve ser computacionalmente eficiente ao calcular o valor de `hash` de uma determinada entrada.
- **Resistente a colisões**: a função deve minimizar a chance de duas entradas gerarem o mesmo valor de `hash`.
- **Uniforme**: as saídas da função devem ser distribuídas uniformemente no intervalo de valores de `hash` possíveis.
- **Não inversível**: deve ser improvável que um computador calcule o valor de entrada da função com base no valor de `hash`.
- **Não previsível**: prever os resultados da função deve ser um desafio, dado um conjunto de entradas.
- **Sensível a alterações de entrada**: a função deve ser sensível a pequenas diferenças na entrada. Pequenas alterações devem causar uma grande diferença no valor de `hash` resultante.

Casos de uso de Hashing

Quando você tiver uma função de *hashing* adequada com todas essas características, poderá aplicá-la a vários casos de uso. As funções de *hashing* funcionam bem para:

- **Armazenamento de senhas**: o *hashing* é uma das melhores maneiras de armazenar senhas de usuários em sistemas modernos. O `Python` combina vários módulos para fazer `hash` e proteger as senhas antes de armazená-las em um banco de dados.
- **Armazenamento em cache**: o *hashing* armazena a saída de uma função para economizar tempo ao chamá-la posteriormente.
- **Recuperação de dados**: o `Python` usa uma tabela de `hash` com uma estrutura de dados de dicionário integrada para recuperar rapidamente os valores por chave.
- **Assinaturas digitais**: o *hashing* pode verificar a autenticidade das mensagens que têm assinaturas digitais.
- **Verificações de integridade de arquivos**: o *hashing* pode verificar a integridade de um arquivo durante sua transferência e download.

Função de Hashing integrada do Python

A função de hashing integrada do `Python`, `hash()`, retorna um valor inteiro que representa o objeto de entrada. Em seguida, o código usa o valor de `hash` resultante para determinar o local do objeto na tabela de `hash`. Essa tabela de `hash` é uma estrutura de dados que implementa dicionários e conjuntos.

O código abaixo demonstra como a função `hash()` funciona:

```
my_string = "Ola Mundo!"

# Calculate the hash value of the string
hash_value = hash(my_string)

# Print the string and its hash value
print("String: ", my_string)
print("Valor Hash: ", hash_value)
```

cuja saída é:

```
String:  Ola Mundo!
Valor Hash:  -2532871366394334124
```

Observe que ao rodar novamente o código, iremos obter um novo valor para o `hash` da mensagem.

```
String:  Ola Mundo!
Valor Hash:  -6006117206146337754
```

O valor do hash é diferente quando invocado uma segunda vez porque as versões recentes do `Python` (versões 3.3 e posteriores), por padrão, aplicam uma semente de `hash` aleatória para essa função. A semente muda em cada invocação do `Python`. Em uma única instância, os resultados serão idênticos.

Por exemplo, vamos colocar esse código em nosso arquivo `exemplo_02.py`:

```
my_string = "Ola Mundo"

# Calculando 2 valores hash da string
hash_value1 = hash(my_string)
hash_value2 = hash(my_string)

# Imprimindo a string e seus respectivos hashs
print("String: ", my_string)
print("Valor Hash 1: ", hash_value1)
print("Valor Hash 2: ", hash_value2)
```

Obtemos a seguinte saída:

```
String:  Ola Mundo
Valor Hash 1:  5459162928360796302
Valor Hash 2:  5459162928360796302
```

Limitações do Hashing

Embora a função `hash()` do `Python` seja promissora para vários casos de uso, suas limitações a tornam inadequada para fins de segurança. Veja como:

- **Ataques de colisão**: uma colisão ocorre quando duas entradas diferentes produzem o mesmo valor de `hash`. Um invasor pode usar o mesmo método de criação de entrada para contornar medidas de segurança que dependem de valores de `hash` para autenticação ou verificações de integridade de dados.

- **Tamanho de entrada limitado**: como as funções de `hash` produzem uma saída de tamanho fixo, independentemente do tamanho da entrada, uma entrada de tamanho maior do que a saída da função de `hash` pode causar uma colisão.

- **Previsibilidade**: uma função de `hash` deve ser determinística, fornecendo o mesmo resultado sempre que você fornecer a mesma entrada. Os invasores podem tirar proveito desse ponto fraco pré-compilando valores de `hash` para muitas entradas e, em seguida, comparando-os com *hashes* de valores-alvo para encontrar uma correspondência. Esse processo é chamado de **ataque de tabela arco-íris**.

Para evitar ataques e manter seus dados seguros, use algoritmos de `hash` seguros projetados para resistir a essas vulnerabilidades.

Uso do hashlib para hashing seguro em Python

Em vez de usar a função `hash()` nativa do `Python`, utilize `hashlib` para um *hashing* mais seguro. Este módulo do `Python` oferece uma variedade de algoritmos de `hash` para criptografar dados de forma segura. Estes algoritmos incluem **MD5**, **SHA-1** e a família **SHA-2** mais segura, que engloba **SHA-256**, **SHA-384**, **SHA-512**, entre outros.

MD5

O algoritmo criptográfico `md5`, amplamente utilizado, revela um valor de `hash` de 128 bits. Use o código abaixo para gerar um `hash` `md5` usando o construtor da `hashlib`:

```
import hashlib

text = "Ola Mundo!"
hash_object = hashlib.md5(text.encode())
print(hash_object.hexdigest())
```

A saída do código acima será consistente em todas as execuções:

```
973de02327c22cb78f9c1d525fdbd039
```

Observação: o método `hexdigest()` no código acima retorna o `hash` em um formato hexadecimal seguro para qualquer apresentação não binária (como e-mail).

SHA-1

A função de hash SHA-1 protege os dados criando um valor de hash de 160 bits. Use o código abaixo com o construtor sha1 para o hash SHA-1 do módulo hashlib:

```
import hashlib

text = "Ola Mundo!"
hash_object = hashlib.sha1(text.encode())
print(hash_object.hexdigest())
```

A saída do código acima:

```
cbdf6ff8ea93c229d8e019563c538a0838387f58
```

SHA-256

Há várias opções de hash na família SHA-2. O construtor hashlib SHA-256 gera uma versão mais segura dessa família com um valor de hash de 256 bits.

Os programadores costumam usar o SHA-256 para criptografia, como assinaturas digitais ou códigos de autenticação de mensagens. O código abaixo demonstra como gerar um hash SHA-256:

```
import hashlib

text = "Ola Mundo!"
hash_object = hashlib.sha256(text.encode())
print(hash_object.hexdigest())
```

A saída do código acima:

```
af3724df163a19e336eb9f2c0b12fdcf9c206dd353077a7cc6c41755fae5d0b6
```

SHA-384

SHA-384 é um valor de hash de 384 bits. Os programadores costumam usar a função SHA-384 em aplicativos que precisam de mais segurança de dados.

Com base nos exemplos anteriores, você provavelmente pode adivinhar que esta é uma instrução que gerará um hash SHA-384:

```
hash_object = hashlib.sha384(text.encode())
```

SHA-512

O SHA-512 é o membro mais seguro da família SHA-2. Ele gera um valor de hash de 512 bits. Os programadores o utilizam para aplicativos de alto rendimento, como a verificação da integridade dos dados. O código abaixo mostra como gerar um hash SHA-512 com o módulo hashlib no Python:

```
hash_object = hashlib.sha512(text.encode())
```

Como escolher um algoritmo de Hashing

Como esses algoritmos são diferentes, selecione o algoritmo de *hashing* com base no caso de uso e nos requisitos de segurança. Aqui estão algumas etapas que você deve seguir:

- **Entenda o caso de uso**: o caso de uso determina o tipo de algoritmo a ser usado. Por exemplo, ao armazenar dados confidenciais, como senhas, o algoritmo de `hash` deve proteger contra ataques de força bruta.
- **Considere suas necessidades de segurança**: os requisitos de segurança do seu caso de uso dependem do tipo de dados que você pretende armazenar, e eles determinam qual algoritmo escolher. Por exemplo, um algoritmo de *hashing* robusto é mais adequado para armazenar informações altamente sensíveis.
- **Pesquise os algoritmos de hashing disponíveis**: explore cada tipo de *hashing* para entender seus pontos fortes e fracos. Essas informações ajudam você a selecionar a melhor opção para o seu caso de uso.
- **Avalie o algoritmo de hashing selecionado**: depois que você escolher um algoritmo de *hashing*, avalie se ele atende aos seus requisitos de segurança. Esse processo pode envolver testes contra ataques ou vulnerabilidades conhecidas.
- **Implemente e teste o algoritmo de hashing**: por fim, implemente e teste o algoritmo minuciosamente para garantir que ele funcione de forma correta e segura.

Como usar o Hashing para armazenamento de senhas

O *hashing* tem excelente potencial para armazenar senhas, um componente essencial da segurança cibernética.

O ideal é que o aplicativo faça `hash` e armazene as senhas em um banco de dados seguro para evitar acesso não autorizado e violações de dados. No entanto, o `hash` sozinho pode não ser suficiente para proteger as informações. As senhas com `hash` ainda são suscetíveis a ataques de força bruta e de dicionário. Os hackers geralmente usam essas práticas para adivinhar senhas e obter acesso não autorizado às contas.

Uma maneira mais segura de usar *hashing* para o armazenamento de senhas envolve a técnica de **salting**. Salting adiciona *strings* ou caracteres únicos e aleatórios a cada senha antes de transformá-la em `hash`. O `salt` é único para cada senha, e o aplicativo o armazena junto com a senha em `hash` no banco de dados.

Sempre que um usuário faz login, o aplicativo recupera o `salt` do banco de dados, adiciona-o à senha inserida e, em seguida, faz o `hash` do `salt` e da senha combinados.

Se um invasor ganhar acesso ao banco de dados, ele terá que calcular o `hash` para cada senha e cada possível valor de `salt`. Salting torna esses ataques mais complexos, sendo uma técnica útil para desencorajar ataques de dicionário.

O módulo `secrets` do `Python` facilita o *salting*. Esse módulo gera *salts* aleatórios, armazenando senhas de forma segura e gerenciando *tokens* e chaves criptográficas.

O código abaixo usa a biblioteca `hashlib` e o módulo `secrets` para proteger ainda mais as senhas dos usuários:

```
import hashlib
import secrets
```

```
# Gera um salt aleatorio, usando o modulo secrets
salt = secrets.token_hex(16)
print(salt)

# Define a senha do usuario
password = "bolinhas"

# Hash para a senha, usando o salt e o algoritmo SHA-256
hash_object = hashlib.sha256((password + salt).encode())

# Transforma a saida hash em uma representacao hexadecimal
hash_hex = hash_object.hexdigest()

print(hash_hex)
```

As saídas são as seguintes:

```
57bb950e7be8a47f02a0b1583ecc9afa
e492515d1b26203aef1fcb579f9ef3d553df3577ceb0f18bfeaf3d791894669b
```

Como usar o Hashing para verificação de integridade de dados

O *hashing* também ajuda a verificar a integridade dos dados e a proteger os dados transmitidos contra modificações e adulterações. Essa técnica de quatro etapas usa uma função de hash criptográfico para dar ao arquivo um valor de hash exclusivo.

Primeiro, selecione a função de hash apropriada e use para gerar um valor de hash para os dados de entrada. Armazene esse valor de hash e use para comparação quando necessário. Sempre que você precisar verificar a integridade dos dados, o aplicativo gerará o valor de hash dos dados atuais usando a mesma função de hash. Em seguida, o aplicativo compara o novo valor de hash com o valor armazenado para garantir que eles sejam idênticos. Em caso afirmativo, os dados não serão corrompidos.

O valor de hash é exclusivo, e até mesmo uma pequena alteração nos dados de entrada aciona um valor de hash significativamente diferente. Isso facilita a detecção de alterações ou modificações não autorizadas nos dados transmitidos.

As etapas abaixo demonstram o uso de uma função de hash para verificações de integridade de dados.

ETAPA 1: importe o módulo `hashlib`:

```
import hashlib
```

ETAPA 2: use um algoritmo de hash `hashlib`:

```
def generate_hash(file_path):

    # Abra o arquivo em modo binario
    with open(file_path, "rb") as f:
```

```
        # Leia o conteudo do arquivo
        contents = f.read()

        # Gere o hash SHA-256 do conteudo
        hash_object = hashlib.sha256(contents)

        # Retorne a representacao hexadecimal do hash
        return hash_object.hexdigest()
```

ETAPA 3: chame a função e passe o caminho do arquivo:

```
file_path = "path/to/my/file.txt"
hash_value = generate_hash(file_path)
print(hash_value)
```

ETAPA 4: gere *hashes* para o arquivo original e para o arquivo transmitido ou modificado:

```
# Generando hash do arquivo original
original_file_path = "path/to/my/file.txt"
original_file_hash = generate_hash(original_file_path)

# Transmitir ou modificar o arquivo (por exemplo, copiando para uma diferente ↩
    localizao)
transmitted_file_path = "path/to/transmitted/file.txt"

# Gerar o hash do arquivo transmitido
transmitted_file_hash = generate_hash(transmitted_file_path)
```

ETAPA 5: Compare os dos *hashes*:

```
if original_file_hash == transmitted_file_hash:
    print("O arquivo nao foi adulterado")
else:
    print("O arquivo foi adulterado")
```

O *hashing* é inestimável para a integridade de dados e a segurança de senhas. Você aproveita ao máximo uma função de *hashing* quando implementa técnicas de *hashing* seguras, como o uso do módulo `hashlib` e `salting`.

Essas técnicas ajudam a prevenir ataques do tipo *rainbow*, colisões e outras vulnerabilidades de segurança que afetam o *hashing*. Programadores frequentemente utilizam essas técnicas com funções de *hashing* em `Python` para garantir a integridade dos dados de arquivos e armazenar senhas de forma segura.

Formulário de Login do Usuário

A extensão `Flask-WTF` usa classes `Python` para representar formulários da web. Uma classe de formulário simplesmente define os campos dos formulário como variáveis de classe.

Mais uma vez, tendo em mente a separação de consentimentos, utilizaremos um novo módulo `app/forms.py` para armazenar nossas classes de formulários web. Para começar, vamos definir um formulário de login de usuário, que pede ao usuário para inserir um nome de usuário e uma senha. O formulário também incluirá uma caixa de seleção *"lembrar de mim"* e um botão de envio. Nosso código ficaria assim:

```
# app/forms.py

from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, BooleanField, SubmitField
from wtforms.validators import DataRequired

class LoginForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired()])
    password = PasswordField('Password', validators=[DataRequired()])
    remember_me = BooleanField('Remember Me')
    submit = SubmitField('Sign In')
```

A maioria das extensões `Flask` usa uma convenção de nomenclatura `flask_<nome>` para seu símbolo de importação de nível superior. Neste caso, o `Flask-WTF` tem todos os seus símbolos `flask_wtf`. É aqui que a classe base `FlaskForm` é importada no topo de `app/forms.py`.

As quatro classes que representam os tipos de campo que estamos utilizando para este formulário são importadas diretamente do pacote `WTForms`, já que a extensão `Flask-WTF` não fornece versões personalizadas. Para cada campo, um objeto é criado com uma variável de classe na classe `LoginForm`. Cada campo recebe uma descrição ou rótulo com oum primeiro argumento.

O argumento opcional `validators` que vemos em alguns dos campos é usado para anexar comportamentos de validação aos campos. O *validator* `DataRequired` simplesmente verifica se o campo não foi enviado vazio. Há muitos outros validadores disponíveis, alguns dos quais serão usados em outros formulários.

Templates dos Formulários

O próximo passo é adicionar o formulário a um modelo HTML para que ele possa ser renderizado em uma página da web. A boa notícia é que os campos que são definidos na classe `LoginForm` sabem como se renderizar como HTML, então essa tarefa é bem simples. Abaixo, vejamos o modelo de *login* que vamos armazenar no arquivo `app/templates/login.html`.

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

{% block content %}
    <h1>Sign In</h1>
    <form action="" method="post" novalidate>
        {{ form.hidden_tag() }}
        <p>
            {{ form.username.label }}<br>
            {{ form.username(size=32) }}
        </p>
        <p>
            {{ form.password.label }}<br>
```

```
            {{ form.password(size=32) }}
        </p>
        <p>{{ form.remember_me() }} {{ form.remember_me.label }}</p>
        <p>{{ form.submit() }}</p>
    </form>
{% endblock %}
```

Para este modelo, estamos reutilizando o modelo `base.html`, que definimos anteriormente, por meio de herança do modelo `extends`. Na verdade, faremos isso com todos os modelos para garantir um *layout* consistente que inclua uma barra de navegação superior em todas as páginas do aplicativo.

Este modelo espera que um objeto de formulário instanciado da classe `LoginForm` seja fornecido com um argumento, que podemos ver refenciado como `form`. Este argumento será enviado pela função de visualização de login, que ainda não escrevemos!

O elemento HTML `<form>` é usado como um contêiner para o formulário da web. O atributo `action` do formulário é usado para informar ao navegador a URL que deve ser usada ao enviar as informações que o usuário inseriu no formulário. Qando a ação é definida como uma *string* vazia, o formulário é enviado para a URL que está atualmente na barra de endereço, que é a URL que renderizou o formulário na página. O atributo `method` especifica o método de solicitação HTTP que deve ser usado ao enviar o formulário para o servidor. O padrão é enviá-lo com uma solicitação `GET`, mas em quase todos os casos, usar uma solicitação `POST` proporciona uma melhor experiência do usuário porque solicitações desse tipo podem enviar os dados do formulário no corpo da solicitação, enquanto solicitações `GET` adicionam os campos do formulário à URL, desorganizando a barra de endereço do navegador. O atributo `novalidate` é usado para dizer ao navegador da web para não aplicar validação aos campos neste formulário, o que efetivamente deixa essa tarefa para o aplicativo `Flask` em execução no servidor. Usar `novalidate` é totalmente opcional, mas para este primeiro formulário é importante que a gente o defina porque isso nos permitirá executar teste de validação do lado do servidor a seguir.

O argumento do modelo `form.hidden_tag()` gera um campo oculto que inclui um *token* que é usado para proteger o formulário contra ataques CSRF. Tudo o que precisamos fazer para proteger o formulário é incluir este campo oculto e ter a variável `SECRET_KEY` definida na configuração do `Flask`. Se cuidarmos dessas duas coisas, o `Flask-WTF` fará o resto pra gente.

Se você já escreveu formulários HTML da web no passado, pode ter achado estranho que não haja campos HTML neste modelo. Isso ocorre porque os campos do objeto do formulário sabem commo se renderizar com HTML. Tudo o que precisamos fazer é incluir `{{ form.<field_name>.label }}` onde desejamos o rótudo e `{{ form.<field_name>() }}` onde desejamos o campo. Para campos que exigem atributos HTML adicionais, eles podem ser passados como argumentos. Os campos `usename` e `password` neste modelo usam o tamanho como um argumento que será adicionado ao elemento HTML `<input>` com um atributo. É assim que também podemos anexar classes `CSS` ou `IDs` a campos de formulários.

Visualização dos Formulários

O passo final antes de vermos este formulário no navegador é codificar uma nova função de visualização no aplicativo que renderiza o modelo da seção anterior.

Então, vamos escrever uma nova função de visualização mapeada para a URL `/login` que cria um formulário e o passa para o modelo a ser renderizado. Esta função de visualização também pode ir no módulo `app/routes.py`:

```
# app/routes.py
from flask import render_template
from app import app
from app.forms import LoginForm

@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)

@app.route('/login')
def login():
    form = LoginForm()
    return render_template('login.html', title='Sign In', form=form)
```

No código acima, nós importamos a classe `LoginForm` de `forms.py`, instanciamos um objeto dele e o enviamos para o *template*. A sintaxe `form=form` pode parecer estranha, mas está simplesmente passando o objeto de formulário criado na linha acima (e mostrado no lado esquerdo). Isso é tudo o que é necessário para renderizar os campos do formulário.

Para facilitar o acesso ao formulário de `login`, o *template* base pode expandir o elemento `<div>` em `base.html` para incluir um *link* para ele na barra de navegação, ou seja:

```
<!-- 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>
                <a href="/login">Login</a>
```

```
            </div>
        <hr>
        {% block content %}{% endblock %}
    </body>
</html>
```

Podemos agora executar o aplicativo e ver o formulário no seu navegador web. Com o aplicativo em execução, digite `http://localhost:5000/` na barra de endereços do navegador e, em seguida, clique no *link* `Login` na barra de navegação superior para ver o novo formulário de login.

Recebendo Dados do Formulários

Se você tentar pressionar o botão de envio, o navegador exibirá um erro *"Method Not Allowed"*. Isso ocorre porque a função de visualização de `login` na seção anterior fez metade do trabalho até agora. Ela pode exibir o formulário em uma página da web, mas ainda não tem lógica para processar dados enviados pelo usuário. Esta é outra área em que o `Flask-WTF` torna o trabalho realmente fácil. Aqui está uma versão atualizada da função de visualização que aceita e valida os dados enviados pelo usuário:

```
# app/routes.py
from flask import render_template, flash, redirect
from app import app
from app.forms import LoginForm

@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)

@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        flash('Login requested for user {}, remember_me={}'.format(
            form.username.data, form.remember_me.data))
        return redirect('/index')
    return render_template('login.html', title='Sign In', form=form)
```

A primeira novidade nesta versão é o argumento `methods` no decorator de rotas. Isso informa ao `Flask` que esta função aceita solicitações `GET` e `POST`, substituindo o padrão, que é aceitar apenas solicitações `GET`. O protocolo HTTP afirma que as solicitações `GET` são aquelas que retornam informações ao cliente (o navegador da web neste caso). Todas as solicitações `POST` são normalmente usadas quando o navegador envia dados do formulário para o servidor (na realidade, as solicitações `GET` também podem ser usada para esse propósito, mas não é uma política recomendada). O erro *"Method Not Allowed"* que o navegador nos mostrou antes aparece porque o navegador tentou enviar uma solicitação `POST` e o aplicativo não foi configurado para aceitá-la. Ao fornecer o argumento `methods`, você está informando ao `Flask` quais métodos de solicitação devem ser aceitos.

O método `form.validate_on_submit()` faz todo o trabalho de processamento do formulário. Quando o navegador envia a solicitação `GET` para receber a página da web com o formulário, esse método retornará `False`, então, nesse caso, a função pula a instrução `if` e vai diretamente para renderizar o modelo na última linha da função.

Quando o navegador envia a solitiação `POST` como resultado do usuário pressionando o botão de envio, `form.validate_on_submit()` reunirá todos os dados, executará todos os validadores anexados aos campos e, se tudo estiver certo, retornará `True`, indicando que os dados são válidos e podem ser processados pelo aplicativo. Mas se pelo menos um campo falhar na validação, a função retornará `False` e isso fará com que o formulário seja renderizado de volta para o usuário, como no caso da solicitação `GET`. Mais tarde, adicionaremos uma mensagem de erro quando a validação falar.

Quando `form.validate_on_submit()` retorna `True`, a função de visualização de `login` chama duas novas funções, importadas do `Flask`. A função `flash()` é uma maneira útil de mostrar uma mensagem ao usuário. Muitos aplicativos usam essa técnica para informar ao usuário se alguma ação foi bem-sucedida ou não. Nesse caso, usaremos esse mecanismo como uma solução temporária, porque ainda não dispomos de toda a infraestrutura necessária para fazer `login` de usuários de verdade. O melhor que podemos fazer, por enquanto, é mostrar uma mensagem que confirma que o aplicativo recebeu as credenciais.

A segunda nova função usada na função de visualização de `login` é o `redirect()`. Essa função instrui o navegador da web do cliente a navegar automaticamente para uma página diferente, fornecida como um argumento. Essa função de visualização a sua para redirecionar o usuário para a página de índice do aplicativo.

Quando chamamos a função `flash()`, o `Flask` armazena a mensagem, mas as mensagens *flasheadas* não aparecerão magicamente nas páginas da web. Os modelos do aplicativo precisam renderizar essas mensagens *flasheadas* para o *layout* do site. Adicionaremos essas mensagens ao modelo base, para que todos os modelos herdem essa funcionalidade. Este é o método base atualizado:

```
<!-- 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>
                <a href="/login">Login</a>
            </div>
        <hr>

        {% with messages = get_flashed_messages() %}
        {% if messages %}
        <ul>
            {% for message in messages %}
            <li>{{ message }}</li>
            {% endfor %}
        </ul>
        {% endif %}
        {% endwith %}

        {% block content %}{% endblock %}
    </body>
</html>
```

Estamos usando um construtor denominado `with` para atribuir o resultado da chamada `get_flashed_messages()` a uma variável `messages`, tudo no contexto do modelo. A função `get_flashed_messages()` vem do `Flask` e retorna uma lista de todas as mensagens que foram registradas com `flash()` anteriormente. O condicional que se segue verifica se `messages` tem algum conteúdo e, nesse caso, um elemento `<ul>` é renderizado com cada mensagem como um item de lista `<li>`. Esse estilo de renderização não fica ótimo para mensagens de *status*, mas o tópico de estilização do aplicativo da web virá mais tarde.

Uma propriedade interessante dessas mensagens `flashed` é que, uma vez solicitadas pela função `get_flashed_messages()`, elas são removidas da lista de mensagens, então elas aparecem apenas uma vez após a funçao `flash()` ser chamada.

Este é um ótimo momento para testar o aplicativo mais uma vez e assim verificar como ele está funcionando. Certifique-se de tentar enviar o formulário com os campos de nome de usuários ou senha vazios, para o `validador DataRequired` interrompa o processo de envio.

Melhorando a Validação dos Campos

Os validadores que são anexados aos campos do formulário impedem que dados inválidos sejam aceitos no aplicativo. A maneiro como o aplicativo lida com entradas inválidas dos formulário é exibindo novamente o formulário para permitir que o usuário faça as correções necessárias.

Se você tentou enviar dados inválidos, tenho certeza de que notou que, embora os mecanismos de validação funcionem bem, não há nenhuma indicação dada ao usuário de que algo está errado com o formulário. O usuário simplesmente recebe o formulário de volta. A próxima tarefa é melhorar a experiência do usuário adicionando uma mensagem de erro siginificativa ao lado de cada campo que falhou na validação.

Na verdade, os validadores de formulário já geram essas mensagens de erro descritivas, então tudo o que falta é alguma lógica adicional no modelo para renderizá-las. Aqui está o modelo de `login` com mensgens de validação de campo adicionadas nos campos de nome de usuário e senha:

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

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

{% block content %}
    <h1>Sign In</h1>
    <form action="" method="post" novalidate>
        {{ form.hidden_tag() }}
        <p>
            {{ form.username.label }}<br>
            {{ form.username(size=32) }}<br>
            {% for error in form.username.errors %}
            <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <p>
            {{ form.password.label }}<br>
            {{ form.password(size=32) }}<br>
            {% for error in form.password.errors %}
            <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <p>{{ form.remember_me() }} {{ form.remember_me.label }}</p>
        <p>{{ form.submit() }}</p>
    </form>
{% endblock %}
```

A única mudança que fizemos foi adicionar *loops* (`for`) logo após os campos `username` e `password` que renderizam as mensagens de erro adicionadas pelos validadores em vermelho. Como regra geral, quaisquer campos que tenham validadores anexados terão quaisquer mensagens de erros resultantes da validação adicionadas em `form.<field_name>.erros`. Esta será uma lista, porque os campos podem ter vários validadores anexados e mais de um pode etar fornecendo mensagens de erro para exibir ao usuário.

Se você tentar enviar o formulário com um `username` ou `password` vazio, agora você receberá uma bela mensagem de erro em vermelho.

Gerando Links

O formulário de `login` está bem completo agora, mas antes de fechar essa parte, precisamos discutir a maneira correta de incluir *links* em modelos e redirecionamentos. Até agora, vimos algumas instâncias nas quais *links* são definidos. Por exemplo, esta é a barra de navegação atual no modelo base:

```
    <div>
        Microblog:
        <a href="/index">Home</a>
        <a href="/login">Login</a>
    </div>
```

A função de visualização do `login` também define um *link* que é passado para a função `redirect()`:

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        # ...
        return redirect('/index')
    # ...
```

Um problema com a escrita de *links* diretamente em modelos e arquivos de origem é que se um dia decidirmos reorganizar nossos *links* em todo o aplicativos, a princípio isso terá que ser feito também no `back-end`.

Logo, para ter melhor controle sobre esses *links*, o `Flask` fornece uma função chamada `url_for()`, que gera URLs usando seu mapeamento interno de URLs para funções de visualização. Por exemplo, a expressão `url_for('login')` retorna `/login`, e a `url_for('index')` retorna para `/index`. O argumento para `url_for()` é o nome do endpoint, que é o nome da função de visualização.

Você pode estar se perguntando agora por que é melhor usar os nomes das funções em vez de URLs. O fato é que as URLs são muito mais propensas a mudar do que os nomes das funções de visualização, que são completamente internas. Uma razão secundária é que, como iremos aprender mais tarde, algumas URLs têm componentes dinâmicos nelas, então gerar essas URLs manualmente exigiria concatenar vários elementos, o que é tedioso e propenso a erros. A função `url_for()` também é capaz de gerar essas URLs complexas com uma sintaxe muito mais elegante.

Então, de agora em diante, usaremos `url_for()` toda vez que precisarmos gerar uma URL de aplicativo. A barra de navegação no modelo base então se torna:

```
<!-- 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="{{ url_for('index') }}">Home</a>
            <a href="{{ url_for('login') }}">Login</a>
        </div>
        <hr>

        {% with messages = get_flashed_messages() %}
        {% if messages %}
        <ul>
            {% for message in messages %}
            <li>{{ message }}</li>
```

```
            {% endfor %}
        </ul>
        {% endif %}
        {% endwith %}

        {% block content %}{% endblock %}
    </body>
</html>
```

E aqui está a função de visualização `login()` atualizada:

```
# app/routes.py

from flask import render_template, flash, redirect, url_for
from app import app
from app.forms import LoginForm

@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)


@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        flash('Login requested for user {}, remember_me={}'.format(
            form.username.data, form.remember_me.data))
        return redirect(url_for('index'))
    return render_template('login.html',  title='Sign In', form=form)
```

Continue...