



# Flask

## Web Development

---

DEVELOPING WEB APPLICATIONS WITH PYTHON

Miguel Grinberg

# Flask Web Development

Take full creative control of your web applications with Flask, the Python-based microframework. With this hands-on book, you'll learn Flask from the ground up by developing a complete social blogging application step by step. Author Miguel Grinberg walks you through the framework's core functionality, and shows you how to extend applications with advanced web techniques such as database migration and web service communication.

Rather than impose development guidelines as other frameworks do, Flask leaves the business of extensions up to you. If you have Python experience, this book shows you how to take advantage of that creative freedom.

- Learn Flask's basic application structure and write an example app
- Work with must-have components—templates, databases, web forms, and email support
- Use packages and modules to structure a large application that scales
- Implement user authentication, roles, and profiles
- Build a blogging feature by reusing templates, paginating item lists, and working with rich text
- Use a Flask-based RESTful API to expose app functionality to smartphones, tablets, and other third-party clients
- Learn how to run unit tests and enhance application performance
- Explore options for deploying your web app to a production server

---

**Miguel Grinberg** has over 25 years of experience as a software engineer, and currently works in the video broadcast industry. He blogs about a variety of topics, including web development, robotics, and photography.

---

WEB DEVELOPMENT / PYTHON

US \$39.99

CAN \$41.99

ISBN: 978-1-449-37262-0



9 781449 372620



Twitter: @oreillymedia  
facebook.com/oreilly

---

# Desenvolvimento Web Flask

**Miguel Grinberg**

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo



**Desenvolvimento Web**

**Flask** por Miguel Grinberg

Copyright © 2014 Miguel Grinberg. Todos os direitos reservados.

Impresso nos Estados Unidos da América.

Publicado por O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Os livros da O'Reilly podem ser adquiridos para uso educacional, comercial ou promocional de vendas. Edições online também estão disponíveis para a maioria dos títulos (<http://my.safaribooksonline.com>). Para mais informações, entre em contato com nosso departamento de vendas corporativo/institucional: 800-998-9938 ou [corporate@oreilly.com](mailto:corporate@oreilly.com).

**Editores:** Meghan Blanchette e Rachel Roumeliotis

**Designer de capa:** Randy Comer

**Editor de Produção:** Nicole Shelby

**Designer de Intérios:** David Futato

**Editor de texto :** Nancy Kotary

**Ilustrador:** Rebecca Demarest

**Revisor:** Charles Roumeliotis

Maio de 2014: Primeira edição

**Histórico de revisões para a primeira edição:**

25-04-2014: Primeira versão

Veja <http://oreilly.com/catalog/errata.csp?isbn=9781449372620> para detalhes do lançamento.

Nutshell Handbook, o logotipo do Nutshell Handbook e o logotipo da O'Reilly são marcas registradas da O'Reilly Media, Inc. *Flask Web Development*, a imagem de um Mastim dos Pireneus e a imagem comercial relacionada são marcas registradas da O'Reilly Media, Inc.

Muitas das designações usadas por fabricantes e vendedores para distinguir seus produtos são reivindicadas como marcas registradas. Onde essas designações aparecem neste livro, e a O'Reilly Media, Inc. estava ciente de uma reivindicação de marca registrada, as designações foram impressas em maiúsculas ou iniciais.

Embora todas as precauções tenham sido tomadas na preparação deste livro, o editor e os autores não assumem responsabilidade por erros ou omissões, ou por danos resultantes do uso das informações aqui contidas.

ISBN: 978-1-449-37262-0

[LSI]

*Para Alícia.*



---

# Índice

Prefácio .....	XI
----------------	----

---

## Parte I. Introdução ao Flask

<b>1. Instalação .....</b>	<b>3</b>
Usando ambientes virtuais	4
Instalando pacotes Python com pip	6
<b>2. Estrutura Básica do Aplicativo.....</b>	<b>7</b>
Inicialização	7
Rotas e funções de visualização	8
Inicialização do servidor	9
Um Aplicativo Completo	9
O ciclo de solicitação-resposta	12
Contextos de Aplicação e Solicitação	12
Solicitar Despacho	14
Solicitar ganchos	14
Respostas	15
Extensões de Frasco	16
Opções de linha de comando com Flask-Script	17
<b>3. Modelos.....</b>	<b>21</b>
O mecanismo de modelo Jinja2	22
Modelos de renderização	22
Variáveis	23
Estruturas de controle	24
Integração do Twitter Bootstrap com o Flask-Bootstrap	26
Páginas de erro personalizadas	29
Links	31

---

Arquivos estáticos	32
Localização de datas e horas com Flask-Moment	33
<b>4. Formulários Web .....</b>	<b>37</b>
Proteção contra falsificação de solicitação entre sites (CSRF)	37
Aulas de formulário	38
Renderização HTML de formulários	40
Manipulação de formulários em funções de visualização	41
Redirecionamentos e sessões de usuário	44
Mensagem piscando	46
<b>5. Bancos de dados.....</b>	<b>49</b>
Bancos de dados	49
SQL Bancos de dados	50
NoSQL SQL ou NoSQL?	51
Python Database Frameworks	51
Gerenciamento de Banco de Dados com Flask-SQLAlchemy	52
Relacionamentos de Definição de Modelo Operações de Banco	54
de Dados Criando as Tabelas Inserindo Linhas Modificando	56
Linhas Excluindo Linhas Consultando Linhas	57
Uso de banco de dados em funções de exibição	58
Integração com o Python Shell	58
Migrações de banco de dados com Flask-Migrate	60
Criando um repositório de migração	60
Criando um script de migração	62
Atualizando o banco de dados	63
Uso de banco de dados em funções de exibição	64
Integração com o Python Shell	64
Migrações de banco de dados com Flask-Migrate	65
Criando um repositório de migração	65
Criando um script de migração	66
Atualizando o banco de dados	67
<b>6. E-mail.....</b>	<b>69</b>
Suporte por e-mail com Flask-Mail	69
Enviando e-mail do Python Shell	70
Integrando e-mails com o aplicativo	71
Envio de e-mail assíncrono	72
<b>7. Grande Estrutura de Aplicação.....</b>	<b>75</b>
Pacote de aplicativos	75
de opções de configuração da	76
estrutura do projeto	78

Usando uma fábrica de aplicativos	78
Implementando a funcionalidade do aplicativo em um blueprint	79
Iniciar script	81
Arquivo de Requisitos	82
Testes de unidade	83
Configuração do banco de dados	85

---

## **Parte II. Exemplo: um aplicativo de blog social**

<b>8. Autenticação do usuário.....</b>	<b>89</b>
Extensões de autenticação para segurança de	89
senha Flask Hashing de senhas com Werkzeug	90
Criando um blueprint de autenticação	90
Autenticação de usuário com Flask-Login	92
Preparando o modelo de usuário para logins	94
Protegendo rotas Adicionando um formulário de	94
login Assinando usuários Conectando usuários	95
Desconectando usuários Testando logins Novo	96
registro de usuário Adicionando um formulário de	97
registro de usuário Cadastrando novo Comercial	99
	99
	100
	100
	102
Confirmação da conta	103
Gerando Tokens de Confirmação com seu perigoso	103
Envio de e-mails de confirmação	105
Gerenciamento de contas	109
<b>9. Funções do usuário.....</b>	<b>111</b>
Representação de Banco de Dados de	111
Funções Atribuição de Função Verificação de	113
Função	114
<b>10. Perfis de Usuário.....</b>	<b>119</b>
Informação do Perfil	119
Página de perfil do usuário	120
Editor de perfil	122
Editor de perfil de nível de usuário	122
Editor de perfil de nível de administrador	124

Avatares de usuário	127
<b>11. Postagens de blogs. . . . .</b>	<b>131</b>
Envio de postagem de blog e exibição de	131
postagens de blog em páginas de perfil Paginação	134
de listas longas de postagem de blog Criação de	135
dados falsos de postagem de blog	135
Renderização de dados em páginas	137
Adicionando um widget de paginação	138
Postagens Rich-Text com Markdown e Flask-PageDown	141
Usando Flask-PageDown	141
Manipulando Rich Text no Servidor	143
Links permanentes para postagens do blog	145
Editor de postagem do blog	146
<b>12. Seguidores. . . . .</b>	<b>149</b>
Relacionamentos de banco de dados revisitados	149
Relacionamentos muitos-para-muitos	150
Relacionamentos autorreferenciais	151
Relações avançadas de muitos para muitos	152
Seguidores na página de perfil	155
Consultar postagens seguidas usando uma associação de banco de dados	158
Mostrar postagens seguidas na página inicial	160
<b>13. Comentários do usuário. . . . .</b>	<b>165</b>
Representação do banco de dados de comentários	165
Envio de comentários e moderação de	167
comentários de exibição	169
<b>14. Interfaces de Programação de Aplicativos . . . . .</b>	<b>175</b>
Introdução aos recursos	175
REST são tudo Métodos de	176
solicitação Corpos de solicitação	177
e resposta Controle de versão	177
RESTful Web Services com Flask	178
Criando um esquema de API Manipulação	179
de erros Autenticação de usuário com	179
Flask-HTTAuth Autenticação baseada	180
em token Serializando recursos de e para JSON	181
Implementando endpoints de recursos	184
	186
	188

Paginação de grandes coleções de recursos	191
Testando Web Services com HTTPie	192

---

## Parte III. A última milha

<b>15. Teste.....</b>	<b>197</b>
Obtendo Relatórios de Cobertura de Código	197
O Cliente de Teste Flask Testando Aplicativos	200
Web Testando Serviços Web Testes de	200
ponta a ponta com Selenium Vale a pena?	204
	205
	209
<b>16. Desempenho.....</b>	<b>211</b>
Criação de perfil de código-fonte de desempenho	211
lento do banco de dados em log	213
<b>17. Implantação.....</b>	<b>215</b>
Fluxo de trabalho de	215
implantação Registro de erros durante a	216
implantação da nuvem de produção A plataforma	217
Heroku Preparando o aplicativo Testando com	218
Foreman Habilitando HTTP seguro com Flask-	218
SSLify Implantando com git push Revisando	222
logs Implantando uma atualização de configuração	223
de servidor de hospedagem tradicional Importando	225
variáveis de ambiente Configurando o registro	226
	227
	227
	227
	228
	228
<b>18. Recursos Adicionais.....</b>	<b>231</b>
Usando um ambiente de desenvolvimento integrado (IDE)	231
Encontrando extensões do Flask	232
Envolvendo-se com o Flask	232
<b>Índice.....</b>	<b>235</b>



---

## Prefácio

O Flask se destaca de outros frameworks porque permite que os desenvolvedores assumam o comando e tenham total controle criativo de seus aplicativos. Talvez você já tenha ouvido a frase “combatendo a estrutura” antes. Isso acontece com a maioria dos frameworks quando você decide resolver um problema com uma solução que não é oficial. Pode ser que você queira usar um mecanismo de banco de dados diferente ou talvez um método diferente de autenticação de usuários.

Desviar-se do caminho definido pelos desenvolvedores do framework lhe dará muitas dores de cabeça.

Frasco não é assim. Você gosta de bancos de dados relacionais? Excelente. Flask suporta todos eles. Talvez você prefira um banco de dados NoSQL? Nenhum problema. Flask também funciona com eles. Quer usar seu próprio mecanismo de banco de dados caseiro? Não precisa de um banco de dados? Continua tudo bem. Com Flask você pode escolher os componentes de sua aplicação ou até mesmo escrever o seu próprio se for isso que você quer. Sem perguntas!

A chave para essa liberdade é que o Flask foi projetado desde o início para ser estendido. Ele vem com um núcleo robusto que inclui a funcionalidade básica de que todos os aplicativos da Web precisam e espera que o restante seja fornecido por algumas das muitas extensões de terceiros no ecossistema e, é claro, por você.

Neste livro apresento meu workflow para desenvolvimento de aplicações web com Flask. Eu não afirmo ter a única maneira verdadeira de construir aplicativos com essa estrutura. Você deve tomar minhas escolhas como recomendações e não como evangelho.

A maioria dos livros de desenvolvimento de software fornece exemplos de código pequenos e focados que demonstram os diferentes recursos da tecnologia de destino isoladamente, deixando o código de “cola” necessário para transformar esses diferentes recursos em aplicativos totalmente funcionais a serem descobertos pelo leitor. . Eu adoto uma abordagem completamente diferente. Todos os exemplos que apresento fazem parte de uma única aplicação que começa muito simples e é expandida a cada capítulo sucessivo. Este aplicativo começa a vida com apenas algumas linhas de código e termina como um aplicativo de blog e rede social bem caracterizado.

## Para quem é este livro

Você deve ter algum nível de experiência em codificação Python para aproveitar ao máximo este livro. Embora o livro não assuma nenhum conhecimento prévio do Flask, supõe-se que conceitos Python como pacotes, módulos, funções, decoradores e programação orientada a objetos sejam bem compreendidos. Alguma familiaridade com exceções e problemas de diagnóstico de rastreamentos de pilha será muito útil.

Ao trabalhar com os exemplos deste livro, você passará muito tempo na linha de comando. Você deve se sentir confortável usando a linha de comando do seu sistema operacional.

Os aplicativos da Web modernos não podem evitar o uso de HTML, CSS e JavaScript. O aplicativo de exemplo que é desenvolvido ao longo do livro obviamente faz uso delas, mas o livro em si não entra em muitos detalhes sobre essas tecnologias e como elas são usadas. Algum grau de familiaridade com essas linguagens é recomendado se você pretende desenvolver aplicativos completos sem a ajuda de um desenvolvedor versado em técnicas do lado do cliente.

Lancei o aplicativo complementar deste livro como código aberto no GitHub. Embora o GitHub possibilite o download de aplicativos como arquivos ZIP ou TAR regulares, recomendo fortemente que você instale um cliente Git e se familiarize com o controle de versão do código-fonte, pelo menos com os comandos básicos para clonar e conferir as diferentes versões do aplicativo diretamente do repositório. A pequena lista de comandos necessários é mostrada em “[Como trabalhar com o código de exemplo](#)” na página xiii. Você também vai querer usar o controle ~~de projeto~~ para se esforçar como uma desculpa para aprender Git!

Finalmente, este livro não é uma referência completa e exaustiva sobre o framework Flask. A maioria dos recursos é coberta, mas você deve complementar este livro com a documentação [oficial do Flask](#).

## Como este livro está organizado

Este livro está dividido em três partes:

A Parte I, [Introdução ao Flask](#), explora os fundamentos do desenvolvimento de aplicações web com o framework Flask e algumas de suas extensões:

- [O Capítulo 1](#) descreve a instalação e configuração do framework Flask.
- [O Capítulo 2](#) mergulha direto no Flask com um aplicativo básico.
- [O Capítulo 3](#) apresenta o uso de modelos em aplicativos Flask.
- [O Capítulo 4](#) apresenta formulários da web.
- [O Capítulo 5](#) apresenta os bancos de dados.

- O Capítulo 6 apresenta o suporte por e-mail. • O

Capítulo 7 apresenta uma estrutura de aplicativos apropriada para aplicativos de médio e grande porte.

**Parte II, Exemplo:** Um aplicativo de blog social, cria o Flasky, o aplicativo de blog e rede social de código aberto que desenvolvi para este livro:

- O Capítulo 8 implementa um sistema de autenticação de usuário. • O

Capítulo 9 implementa as funções e permissões do usuário. • O Capítulo

10 implementa páginas de perfil de usuário. • O Capítulo 11 cria a

interface de blog. • O Capítulo 12 implementa seguidores. • O Capítulo

13 implementa comentários de usuários para postagens de blog. • O

Capítulo 14 implementa uma Interface de Programação de Aplicativo (API).

A Parte III, *The Last Mile*, descreve algumas tarefas importantes não diretamente relacionadas à codificação de aplicativos que precisam ser consideradas antes de publicar um aplicativo:

- O Capítulo 15 descreve em detalhes diferentes estratégias de teste de unidade. •

O Capítulo 16 apresenta uma visão geral das técnicas de análise de desempenho. • O

Capítulo 17 descreve as opções de implantação para aplicativos Flask, tanto tradicionais  
e baseado em nuvem.

- O Capítulo 18 lista recursos adicionais.

## Como trabalhar com o código de exemplo

Os exemplos de código apresentados neste livro estão disponíveis no GitHub em <https://github.com/miguelgrinberg/flasky>.

O histórico de commits neste repositório foi cuidadosamente criado para corresponder à ordem em que os conceitos são apresentados no livro. A maneira recomendada de trabalhar com o código é verificar os commits começando pelo mais antigo, então avançar pela lista de commits conforme você avança no livro. Como alternativa, o GitHub também permite que você baixe cada commit como um arquivo ZIP ou TAR.

Se você decidir usar o Git para trabalhar com o código-fonte, precisará instalar o cliente Git, que pode ser baixado em <http://git-scm.com>. O comando a seguir baixa o código de exemplo usando o Git:

```
$ git clone https://github.com/miguelgrinberg/flasky.git
```

O comando `git clone` instala o código-fonte do GitHub em uma pasta `flasky` que é criada no diretório atual. Esta pasta não contém apenas código-fonte; uma cópia do repositório Git com todo o histórico de alterações feitas no aplicativo também está incluída.

No primeiro capítulo, você será solicitado a *conferir* a versão inicial do aplicativo e, em seguida, nos locais apropriados, será instruído a avançar no histórico.

O comando Git que permite percorrer o histórico de alterações é o `git checkout`. Aqui está um exemplo:

```
$ git checkout 1a
```

O 1a referenciado no comando é um *tag*, um ponto nomeado no histórico do projeto.

Este repositório é marcado de acordo com os capítulos do livro, então a tag 1a usada no exemplo define os arquivos do aplicativo para a versão inicial usada no [Capítulo 1](#). A maioria dos capítulos tem mais de uma tag associada a eles, então, por exemplo, tags 5a, 5b e assim por diante são versões incrementais apresentadas no [Capítulo 5](#).

Além de verificar os arquivos de origem de uma versão do aplicativo, pode ser necessário realizar algumas configurações. Por exemplo, em alguns casos, você precisará instalar pacotes Python adicionais ou aplicar atualizações no banco de dados. Você será informado quando estes são necessários.

Você normalmente não modificará os arquivos de origem do aplicativo, mas se o fizer, o Git não permitirá que você faça check-out de uma revisão diferente, pois isso faria com que suas alterações locais fossem perdidas. Antes de fazer o check-out de uma revisão diferente, você precisará reverter os arquivos para o estado original. A maneira mais fácil de fazer isso é com o comando `git reset` :

```
$ git reset --hard
```

Este comando destruirá suas alterações locais, então você deve salvar qualquer coisa que não queira perder antes de usar este comando.

De tempos em tempos, você pode querer atualizar seu repositório local do GitHub, onde correções de bugs e melhorias podem ter sido aplicadas. Os comandos que fazem isso são:

```
$ git fetch -all $ git  
fetch --tags $ git reset  
--hard origin/master
```

Os comandos `git fetch` são usados para atualizar o histórico de commits e as tags em seu repositório local a partir do repositório remoto no GitHub, mas nada disso afeta os arquivos de origem reais, que são atualizados com o comando `git reset` a seguir. Mais uma vez, esteja ciente de que sempre que o `git reset` for usado, você perderá todas as alterações locais que fez.

Outra operação útil é visualizar todas as diferenças entre duas versões do aplicativo. Isso pode ser muito útil para entender uma mudança em detalhes. Do comando

linha, o comando git diff pode fazer isso. Por exemplo, para ver a diferença entre as revisões 2a e 2b, use:

```
$ git diff 2a 2b
```

As diferenças são mostradas como um *patch*, que não é um formato muito intuitivo para revisar as alterações se você não estiver acostumado a trabalhar com arquivos de patch. Você pode achar que as comparações gráficas mostradas pelo GitHub são muito mais fáceis de ler. Por exemplo, as diferenças entre as revisões 2a e 2b podem ser visualizadas no GitHub em <https://github.com/miguelgrinberg/flasky/compare/2a...2b>

## Usando exemplos de código

Este livro está aqui para ajudá-lo a fazer o seu trabalho. Em geral, se um código de exemplo for oferecido com este livro, você poderá usá-lo em seus programas e documentação. Você não precisa entrar em contato conosco para obter permissão, a menos que esteja reproduzindo uma parte significativa do código. Por exemplo, escrever um programa que usa vários pedaços de código deste livro não requer permissão. Vender ou distribuir um CD-ROM de exemplos dos livros da O'Reilly requer permissão. Responder a uma pergunta citando este livro e citando um código de exemplo não requer permissão. Incorporar uma quantidade significativa de código de exemplo deste livro na documentação do seu produto requer permissão.

Agradecemos, mas não exigimos, atribuição. Uma atribuição geralmente inclui o título, autor, editora e ISBN. Por exemplo: “*Flask Web Development* por Miguel Grinberg (O'Reilly). Copyright 2014 Miguel Grinberg, 978-1-449-3726-2.”

Se você achar que o uso de exemplos de código está fora do uso justo ou da permissão dada acima, sinta-se à vontade para nos contatar em [permissions@oreilly.com](mailto:permissions@oreilly.com).

## Convenções utilizadas neste livro

As seguintes convenções tipográficas são usadas neste livro:

### *Itálico*

Indica novos termos, URLs, endereços de e-mail, nomes de arquivo e extensões de arquivo.

### Largura constante

Usada para listagens de programas, bem como dentro de parágrafos para fazer referência a elementos de programa, como nomes de variáveis ou funções, bancos de dados, tipos de dados, variáveis de ambiente, instruções e palavras-chave.

### **Largura constante negrito**

Mostra comandos ou outro texto que deve ser digitado literalmente pelo usuário.

### Largura constante itálico

Mostra o texto que deve ser substituído por valores fornecidos pelo usuário ou por valores determinados pelo contexto.



Este elemento significa uma dica ou sugestão.



Este elemento significa uma nota geral.



Este elemento indica um aviso ou cuidado.

## Livros on-line do Safari®



*Livros on-line do Safari* é uma biblioteca digital sob demanda que oferece conteúdo especializado em formato de livro e vídeo dos principais autores do mundo em tecnologia e negócios.

Profissionais de tecnologia, desenvolvedores de software, web designers e profissionais de negócios e criativos usam o Safari Books Online como seu principal recurso para pesquisa, solução de problemas, aprendizado e treinamento de certificação.

O Safari Books Online oferece uma variedade de **combinações de produtos** e programas de preços para **organizações, agências governamentais, e indivíduos**. Os assinantes têm acesso a milhares de livros, vídeos de treinamento e manuscritos de pré-publicação em um banco de dados totalmente pesquisável de editoras como O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology e dezenas de **outros**. Para obter mais informações sobre o Safari Books Online, visite-nos [online](#).

### Como entrar em contato conosco

Envie comentários e perguntas sobre este livro à editora:

O'Reilly Media, Inc.  
1005 Gravenstein Highway North  
Sebastopol, CA 95472 800-998-9938  
(nos Estados Unidos ou Canadá) 707-829-0515  
(internacional ou local) 707-829-0104 (fax)

Temos uma página web para este livro, onde listamos erratas, exemplos e qualquer informação adicional. Você pode acessar esta página em <http://www.bit.ly/flask-web-dev>.

Para comentar ou fazer perguntas técnicas sobre este livro, envie um e-mail para [bookquests@oreilly.com](mailto:bookquests@oreilly.com).

Para obter mais informações sobre nossos livros, cursos, conferências e notícias, consulte nosso site em <http://www.oreilly.com>.

Encontre-nos no Facebook: <http://facebook.com/oreilly>

Siga-nos no Twitter: <http://twitter.com/oreillymedia>

Assista-nos no YouTube: <http://www.youtube.com/oreillymedia>

## Agradecimentos

Eu não poderia ter escrito este livro sozinho. Recebi muita ajuda da família, colegas de trabalho, velhos amigos e novos amigos que fiz ao longo do caminho.

Gostaria de agradecer a Brendan Kohler por sua detalhada revisão técnica e por sua ajuda em dar forma ao capítulo sobre Interfaces de Programação de Aplicativos. Também estou em dívida com David Baumgold, Todd Brunhoff, Cecil Rock e Matthew Hugues, que revisaram o manuscrito em diferentes estágios de conclusão e me deram conselhos muito úteis sobre o que cobrir e como organizar o material.

Escrever os exemplos de código para este livro foi um esforço considerável. Agradeço a ajuda de Daniel Hofmann, que fez uma revisão completa do código do aplicativo e apontou várias melhorias. Também sou grato ao meu filho adolescente, Dylan Grinberg, que suspendeu seu vício em Minecraft por alguns fins de semana e me ajudou a testar o código em várias plataformas.

O'Reilly tem um programa maravilhoso chamado Early Release que permite que leitores impacientes tenham acesso a livros enquanto eles estão sendo escritos. Alguns dos meus leitores do Early Release se esforçaram e se envolveram em conversas úteis sobre sua experiência de trabalho com o livro, levando a melhorias significativas. Eu gostaria de reconhecer

Sundeep Gupta, Dan Caron, Brian Wisti e Cody Scott em particular pelas contribuições que fizeram a este livro.

A equipe da O'Reilly Media sempre esteve lá para mim. Acima de tudo, gostaria de reconhecer minha maravilhosa editora, Meghan Blanchette, por seu apoio, conselhos e assistência desde o primeiro dia em que nos conhecemos. Meg tornou a experiência de escrever meu primeiro livro memorável.

Para concluir, gostaria de agradecer imensamente à incrível comunidade Flask.

PARTE I

---

# Introdução ao Flask



## CAPÍTULO 1

# Instalação

**Frasco** é uma estrutura pequena pela maioria dos padrões, pequena o suficiente para ser chamada de “microestrutura”. É pequeno o suficiente para que, quando você se familiarizar com ele, provavelmente será capaz de ler e entender todo o seu código-fonte.

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

Flask tem duas dependências principais. Os subsistemas de roteamento, depuração e Web Server Gateway Interface (WSGI) vêm de [Werkzeug](#), enquanto o suporte ao modelo é fornecido pelo [Jinja2](#). Werkzeug e Jinja2 são de autoria do desenvolvedor principal do Flask.

Não há suporte nativo no Flask para acessar bancos de dados, validar formulários da web, autenticar usuários ou outras tarefas de alto nível. Esses e muitos outros serviços importantes que a maioria dos aplicativos da Web precisam estão disponíveis por meio de extensões que se integram aos pacotes principais. Como desenvolvedor, você tem o poder de escolher as extensões que funcionam melhor para o seu projeto ou até mesmo escrever as suas próprias, se quiser. Isso contrasta com uma estrutura maior, onde a maioria das escolhas foi feita para você e é difícil ou às vezes impossível de mudar.

Neste capítulo, você aprenderá como instalar o Flask. O único requisito que você precisa é de um computador com o Python instalado.



Os exemplos de código neste livro foram verificados para funcionar com Python 2.7 e Python 3.3, portanto, é altamente recomendável usar uma dessas duas versões.

## Usando ambientes virtuais

A maneira mais conveniente de instalar o Flask é usar um ambiente virtual. Um ambiente virtual é uma cópia privada do interpretador Python no qual você pode instalar pacotes de forma privada, sem afetar o interpretador Python global instalado em seu sistema.

Os ambientes virtuais são muito úteis porque evitam a confusão de pacotes e conflitos de versão no interpretador Python do sistema. A criação de um ambiente virtual para cada aplicativo garante que os aplicativos tenham acesso apenas aos pacotes que usam, enquanto o interpretador global permanece organizado e limpo e serve apenas como uma fonte a partir da qual mais ambientes virtuais podem ser criados. Como benefício adicional, os ambientes virtuais não exigem direitos de administrador.

Os ambientes virtuais são criados com o utilitário `virtualenv` de terceiros. Para verificar se você o tem instalado em seu sistema, digite o seguinte comando:

```
$ virtualenv --version
```

Se você receber um erro, será necessário instalar o utilitário.



O Python 3.3 adiciona suporte nativo de ambientes virtuais por meio do módulo `venv` e do comando `pyvenv`. `pyvenv` pode ser usado em vez de `virtualenv`, mas observe que ambientes virtuais criados com `pyvenv` no Python 3.3 não incluem `pip`, que precisa ser instalado manualmente. Essa limitação foi removida no Python 3.4, onde o `pyvenv` pode ser usado como uma substituição completa do `virtualenv`.

A maioria das distribuições Linux fornece um pacote para `virtualenv`. Por exemplo, os usuários do Ubuntu podem instalá-lo com este comando:

```
$ sudo apt-get install python-virtualenv
```

Se você estiver usando o Mac OS X, poderá instalar o `virtualenv` usando `easy_install`:

```
$ sudo easy_install virtualenv
```

Se você estiver usando o Microsoft Windows ou qualquer sistema operacional que não forneça um pacote `virtualenv` oficial, terá um procedimento de instalação um pouco mais complicado.

Usando seu navegador da web, navegue até <https://bitbucket.org/pypa/setuptools>, a casa do instalador `setuptools`. Nessa página, procure um link para baixar o script do instalador.

Este é um script chamado `ez_setup.py`. Salve este arquivo em uma pasta temporária em seu computador e execute os seguintes comandos nessa pasta:

```
$ python ez_setup.py $  
easy_install virtualenv
```



Os comandos anteriores devem ser emitidos a partir de uma conta com direitos de administrador. No Microsoft Windows, inicie a janela do prompt de comando usando a opção “Executar como administrador”. Em sistemas baseados em Unix, os dois comandos de instalação devem ser precedidos por sudo ou executados como usuário root. Uma vez instalado, o utilitário virtualenv pode ser invocado a partir de contas normais.

Agora você precisa criar a pasta que hospedará o código de exemplo, que está disponível em um repositório do GitHub. Conforme discutido na [página “Como trabalhar com o código de exemplo”](#) em [xiii](#), a maneira mais conveniente de fazer isso é verificar o código diretamente do GitHub usando um cliente Git. Os comandos a seguir baixam o código de exemplo do GitHub e inicializam a pasta do aplicativo para a versão “1a”, a versão inicial do aplicativo:

```
$ git clone https://github.com/miguelgrinberg/flasky.git $ cd flasky $ git
checkout 1a
```

A próxima etapa é criar o ambiente virtual Python dentro da pasta *flasky* usando o comando virtualenv. Este comando tem um único argumento obrigatório: o nome do ambiente virtual. Uma pasta com o nome escolhido será criada no diretório atual e todos os arquivos associados ao ambiente virtual estarão dentro dela. Uma convenção de nomenclatura comumente usada para ambientes virtuais é chamá-los de *venv*:

```
$ virtualenv venv
Novo executável python em venv/bin/python2.7
Também criando executável em venv/bin/python
Instalando setuptools.....concluído.
Instalando o pip.....concluído.
```

Agora você tem uma pasta *venv* dentro da pasta *flasky* com um ambiente virtual totalmente novo que contém um interpretador Python privado. Para começar a usar o ambiente virtual, é preciso “ativá-lo”. Se estiver usando uma linha de comando bash (usuários de Linux e Mac OS X), você pode ativar o ambiente virtual com este comando:

```
$ source venv/bin/activate
```

Se você estiver usando o Microsoft Windows, o comando de ativação é:

```
$ venv\Scripts\ativar
```

Quando um ambiente virtual é ativado, a localização de seu interpretador Python é adicionada ao PATH, mas essa alteração não é permanente; ela afeta apenas sua sessão de comando atual. Para lembrá-lo de que você ativou um ambiente virtual, o comando de ativação modifica o prompt de comando para incluir o nome do ambiente:

```
(venv) $
```

Quando você terminar de trabalhar com o ambiente virtual e quiser retornar ao interpretador global do Python, digite deactivate no prompt de comando.

## Instalando pacotes Python com pip

A maioria dos pacotes Python são instalados com o utilitário *pip*, que o virtualenv adiciona automaticamente a todos os ambientes virtuais na criação. Quando um ambiente virtual é ativado, o local do utilitário pip é adicionado ao PATH.



Se você criou o ambiente virtual com pyvenv no Python 3.3, o pip deve ser instalado manualmente. As instruções de instalação estão disponíveis no [site da pip](#). No Python 3.4, o pyvenv instala o pip automaticamente.

Para instalar o Flask no ambiente virtual, use o seguinte comando:

```
(venv) $ pip install frasco
```

Com este comando, o Flask e suas dependências são instalados no ambiente virtual. Você pode verificar se o Flask foi instalado corretamente iniciando o interpretador Python e tentando importá-lo:

```
(venv) $ python >>>
import frasco
>>>
```

Se nenhum erro aparecer, você pode se parabenizar: você está pronto para o próximo capítulo, onde escreverá sua primeira aplicação web.

---

## CAPÍTULO 2

# Estrutura básica do aplicativo

Neste capítulo, você aprenderá sobre as diferentes partes de um aplicativo Flask. Você também escreverá e executará seu primeiro aplicativo Web Flask.

## Inicialização

Todos os aplicativos Flask devem criar uma *instância do aplicativo*. O servidor web passa todas as solicitações que recebe dos clientes para este objeto para manipulação, usando um protocolo chamado Web Server Gateway Interface (WSGI). A instância do aplicativo é um objeto da classe Flask, geralmente criado da seguinte forma:

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

O único argumento necessário para o construtor da classe Flask é o nome do módulo ou pacote principal do aplicativo. Para a maioria dos aplicativos, a variável `__name__` do Python é o valor correto.



O argumento `name` que é passado para o construtor da aplicação Flask é uma fonte de confusão entre os novos desenvolvedores do Flask. O Flask usa esse argumento para determinar o caminho raiz do aplicativo para que posteriormente possa localizar arquivos de recursos relativos ao local do aplicativo.

Mais tarde, você verá exemplos mais complexos de inicialização de aplicativos, mas para aplicativos simples, isso é tudo o que é necessário.

## Rotas e funções de visualização

Clientes como navegadores da Web enviam *solicitações* para o servidor da Web, que por sua vez as envia para a instância do aplicativo Flask. A instância do aplicativo precisa saber qual código precisa ser executado para cada URL solicitada, portanto, mantém um mapeamento de URLs para funções do Python. A associação entre uma URL e a função que a trata é chamada de *rota*.

A maneira mais conveniente de definir uma rota em um aplicativo Flask é por meio do decorador `app.route` exposto pela instância do aplicativo, que registra a função decorada como uma rota. O exemplo a seguir mostra como uma rota é declarada usando este decorador:

```
@app.route('/')
def index(): return
    '<h1>Hello World!</h1>'
```



Decoradores são um recurso padrão da linguagem Python; eles podem modificar o comportamento de uma função de diferentes maneiras. Um padrão comum é usar decoradores para registrar funções como manipuladores para um evento.

O exemplo anterior registra a função `index()` como o manipulador da URL raiz do aplicativo. Se este aplicativo fosse implantado em um servidor associado ao nome de domínio `www.example.com`, navegar para `http://www.example.com` em seu navegador acionaria `index()` para ser executado no servidor. O valor de retorno dessa função, chamado de *resposta*, é o que o cliente recebe. Se o cliente for um navegador da Web, a resposta será o documento exibido ao usuário.

Funções como `index()` são chamadas de *funções de visualização*. Uma resposta retornada por uma função de visualização pode ser uma string simples com conteúdo HTML, mas também pode assumir formas mais complexas, como você verá mais adiante.



Strings de resposta embutidas no código Python levam a um código difícil de manter, e isso é feito aqui apenas para introduzir o conceito de respostas. Você aprenderá a maneira correta de gerar respostas no [Capítulo 3](#).

Se você prestar atenção em como alguns URLs de serviços que você usa todos os dias são formados, você notará que muitos têm seções variáveis. Por exemplo, a URL da sua página de perfil do Facebook é `http://www.facebook.com/<seu-nome>`, então seu nome de usuário faz parte dela. O Flask suporta esses tipos de URLs usando uma sintaxe especial no decorador de rotas .

O exemplo a seguir define uma rota que tem um componente de nome dinâmico:

```
@app.route('/user/<name>') def
user(name): return '<h1>Olá,
%sl</h1>' % name
```

A parte entre colchetes angulares é a parte dinâmica, portanto, quaisquer URLs que correspondam às partes estáticas serão mapeadas para essa rota. Quando a função de visualização é invocada, o Flask envia o componente dinâmico como um argumento. Na função de visualização de exemplo anterior, esse argumento é usado para gerar uma saudação personalizada como resposta.

Os componentes dinâmicos nas rotas são strings por padrão, mas também podem ser definidos com um tipo. Por exemplo, a rota /user/<int:id> corresponderia apenas a URLs que tivessem um número inteiro no segmento dinâmico id . Flask suporta os tipos int, float e path para rotas. O tipo de caminho também representa uma string, mas não considera barras como separadores e, em vez disso, as considera parte do componente dinâmico.

## Inicialização do servidor

A instância do aplicativo tem um método run que inicia o servidor web de desenvolvimento integrado do Flask:

```
if __name__ == '__main__':
    app.run(debug=True)
```

O idioma Python \_\_name\_\_ == '\_\_main\_\_' é usado aqui para garantir que o servidor web de desenvolvimento seja iniciado somente quando o script for executado diretamente. Quando o script é importado por outro script, supõe-se que o script pai iniciará um servidor diferente, portanto, a chamada app.run() é ignorada.

Depois que o servidor é inicializado, ele entra em um loop que aguarda as solicitações e as atende. Esse loop continua até que o aplicativo seja interrompido, por exemplo, pressionando Ctrl-C.

Existem vários argumentos de opção que podem ser dados a app.run() para configurar o modo de operação do servidor web. Durante o desenvolvimento, é conveniente habilitar o modo de depuração, que entre outras coisas ativa o *depurador* e o *recarregador*. Isso é feito passando o argumento debug definido como True.



O servidor web fornecido pelo Flask não se destina ao uso em produção. Você aprenderá sobre servidores web de produção no [Capítulo 17](#).

## Um Aplicativo Completo

Nas seções anteriores, você aprendeu sobre as diferentes partes de um aplicativo Web Flask e agora é hora de escrever um. Todo o script do aplicativo *hello.py* não é nada mais

do que as três partes descritas anteriormente combinadas em um único arquivo. A aplicação é mostrada no [Exemplo 2-1](#).

*Exemplo 2-1. hello.py: Um aplicativo Flask completo*

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

@app.route('/')
def index():
    return '<h1>Hello World!</h1>'

if __name__ == '__main__':
    app.run(debug=True)
```



Se você clonou o repositório Git do aplicativo no GitHub, agora você pode executar git checkout 2a para verificar esta versão do aplicativo.

Para executar o aplicativo, certifique-se de que o ambiente virtual que você criou anteriormente esteja ativado e tenha o Flask instalado. Agora abra seu navegador da web e digite `http://127.0.0.1:5000/` na barra de endereço. A [Figura 2-1](#) mostra o navegador da web após a conexão com o aplicativo.



*Figura 2-1. hello.py aplicativo Flask*

Em seguida, inicie o aplicativo com o seguinte comando:

```
(venv) $ python hello.py *
Executando em http://127.0.0.1:5000/
Reiniciando com reloader
```

Se você digitar qualquer outro URL, o aplicativo não saberá como lidar com isso e retornará um código de erro 404 ao navegador — o erro familiar que você recebe quando navega para uma página da Web que não existe.

A versão aprimorada do aplicativo mostrada no [Exemplo 2-2](#) adiciona uma segunda rota que é dinâmica. Ao visitar este URL, você recebe uma saudação personalizada.

*Exemplo 2-2. hello.py: aplicativo Flask com rota dinâmica*

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

@app.route('/')
def index():
    return '<h1>Hello World!</h1>'

@app.route('/user/<name>')
def user(name):
    return '<h1>Olá, %s!</h1>' % name

if __name__ == '__main__':
    app.run(debug=True)
```



Se você clonou o repositório Git do aplicativo no GitHub, agora você pode executar git checkout 2b para verificar esta versão do aplicativo.

Para testar a rota dinâmica, verifique se o servidor está em execução e navegue até `http://localhost:5000/user/Dave`. O aplicativo responderá com uma saudação personalizada, gerada usando o argumento `name dynamic`. Tente nomes diferentes para ver como a função de visualização sempre gera a resposta com base no nome fornecido. Um exemplo é mostrado na [Figura 2-2](#).

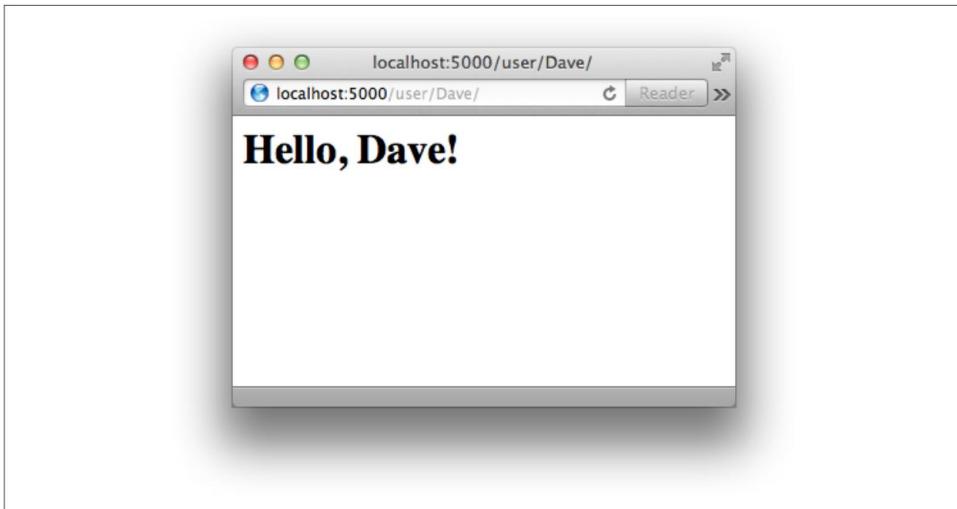


Figura 2-2. Rota Dinâmica

## O ciclo de solicitação-resposta

Agora que você já jogou com um aplicativo básico do Flask, talvez queira saber mais sobre como o Flask faz sua mágica. As seções a seguir descrevem alguns dos aspectos de design da estrutura.

### Contextos de Aplicação e Solicitação

Quando o Flask recebe uma solicitação de um cliente, ele precisa disponibilizar alguns objetos para a função de visualização que irá lidar com isso. Um bom exemplo é o *objeto request*, que encapsula a requisição HTTP enviada pelo cliente.

A maneira óbvia pela qual o Flask poderia dar acesso a uma função de visualização ao objeto de solicitação é enviando-o como um argumento, mas isso exigiria que cada função de visualização no aplicativo tivesse um argumento extra. As coisas ficam mais complicadas se você considerar que o objeto de solicitação não é o único objeto que as funções de visualização podem precisar acessar para atender a uma solicitação.

Para evitar a confusão de funções de visualização com muitos argumentos que podem ou não ser necessários, o Flask usa *contextos* para tornar temporariamente certos objetos globalmente acessíveis. Graças aos contextos, funções de visualização como a seguinte podem ser escritas:

do pedido de importação de frasco

```
@app.route('/') def  
index():
```

```
user_agent = request.headers.get('User-Agent') return
'<p>Seu navegador é %s</p>' % user_agent
```

Observe como nesta visualização a solicitação de função é usada como se fosse uma variável global. Na realidade, request não pode ser uma variável global se você considerar que em um servidor multithread as threads estão trabalhando em diferentes requisições de clientes diferentes ao mesmo tempo, então cada thread precisa ver um objeto diferente na requisição. Os contextos permitem que o Flask torne certas variáveis globalmente acessíveis a uma thread sem interferir nas outras threads.



Uma thread é a menor sequência de instruções que pode ser gerenciada independentemente. É comum que um processo tenha múltiplas threads ativas, algumas vezes compartilhando recursos como memória ou handles de arquivo. Servidores web multithread iniciam um pool de threads e selecionam um thread do pool para lidar com cada solicitação recebida.

Existem dois contextos no Flask: o contexto do *aplicativo* e o contexto da *solicitação*. A Tabela 2-1 mostra as variáveis expostas por cada um desses contextos.

Tabela 2-1. Globais de contexto do frasco

Nome variável	Contexto	Descrição
current_app	Contexto do aplicativo	A instância do aplicativo para o aplicativo ativo.
g	Contexto do aplicativo	Um objeto que o aplicativo pode usar para armazenamento temporário durante o manuseio de um pedido. Essa variável é redefinida a cada solicitação.
solicitação	Solicitar contexto	O objeto de solicitação, que encapsula o conteúdo de uma solicitação HTTP enviada pelo cliente.
sessão	Solicitar contexto	A sessão do usuário, um dicionário que o aplicativo pode usar para armazenar valores que são "lembados" entre as solicitações.

O Flask ativa (ou *envia por push*) o aplicativo e os contextos de solicitação antes de despachar uma solicitação e os remove quando a solicitação é tratada. Quando o contexto do aplicativo é enviado, as variáveis current\_app e g ficam disponíveis para o encadeamento; da mesma forma, quando o contexto de solicitação é enviado, a solicitação e a sessão também ficam disponíveis. Se qualquer uma dessas variáveis for acessada sem um aplicativo ativo ou contexto de solicitação, um erro será gerado. As quatro variáveis de contexto serão revisitadas em detalhes em capítulos posteriores, então não se preocupe se você ainda não entender por que elas são úteis.

A seguinte sessão de shell do Python demonstra como o contexto do aplicativo funciona:

```
>>> from hello import app
>>> from flask import current_app >>>
current_app.name Traceback (última
chamada mais recente):
...
RuntimeError: trabalhando fora do contexto do aplicativo
```

```
>>> app_ctx = app.app_context() >>>
app_ctx.push() >>> current_app.name
'hello' >>> app_ctx.pop()
```

Neste exemplo, `current_app.name` falha quando não há contexto de aplicativo ativo, mas se torna válido quando um contexto é enviado por push. Observe como um contexto de aplicativo é obtido invocando `app.app_context()` na instância do aplicativo.

### Despacho de solicitação

Quando o aplicativo recebe uma solicitação de um cliente, ele precisa descobrir qual função de visualização invocar para atendê-lo. Para esta tarefa, o Flask procura a URL fornecida na solicitação no *mapa de URLs do aplicativo*, que contém um mapeamento de URLs para as funções de visualização que as tratam. O Flask constrói este mapa usando os decoradores `app.route` ou a versão não decorada equivalente `app.add_url_rule()`.

Para ver como é o mapa de URL em um aplicativo Flask, você pode inspecionar o mapa criado para `hello.py` no shell do Python. Para este teste, certifique-se de que seu ambiente virtual esteja ativado:

```
(venv) $ python
>>> from hello import app >>>
app.url_map Map([<Rule
'/' (HEAD, OPTIONS, GET) -> index>, <Rule '/static/
<filename>' (HEAD , OPTIONS, GET) -> static>, <Rule '/user/
<name>' (HEAD, OPTIONS, GET) -> user>])
```

As rotas / e /user/<name> foram definidas pelos decoradores `app.route` na aplicação. A rota `/static/<filename>` é uma rota especial adicionada pelo Flask para dar acesso a arquivos estáticos. Você aprenderá mais sobre arquivos estáticos no [Capítulo 3](#).

Os elementos HEAD, OPTIONS, GET mostrados no mapa de URL são os *métodos de solicitação* que são tratados pela rota. O Flask anexa métodos a cada rota para que diferentes métodos de solicitação enviados para a mesma URL possam ser tratados por diferentes funções de visualização. Os métodos HEAD e OPTIONS são gerenciados automaticamente pelo Flask, então na prática pode-se dizer que nesta aplicação as três rotas no mapa de URL estão anexadas ao método GET . Você aprenderá a especificar diferentes métodos de solicitação para rotas no [Capítulo 4](#).

### Ganchos de

**solicitação** Às vezes é útil executar o código antes ou depois de cada solicitação ser processada. Por exemplo, no início de cada solicitação pode ser necessário criar uma conexão com o banco de dados ou autenticar o usuário que faz a solicitação. Em vez de duplicar o código que faz isso em cada função de visualização, o Flask oferece a opção de registrar funções comuns a serem invocadas antes ou depois de uma solicitação ser despachada para uma função de visualização.

Ganchos de solicitação são implementados como decoradores. Estes são os quatro ganchos suportados pelo Flask:

- `before_first_request`: Registra uma função para ser executada antes que a primeira solicitação seja manipulado.
- `before_request`: Registra uma função para ser executada antes de cada solicitação.
- `after_request`: Registre uma função para ser executada após cada solicitação, se não ocorreram exceções não tratadas.
- `teardown_request`: Registra uma função para ser executada após cada solicitação, mesmo que não tratada ocorreram exceções.

Um padrão comum para compartilhar dados entre funções de gancho de solicitação e funções de visualização é usar o contexto `g` global. Por exemplo, um manipulador `before_request` pode carregar o usuário conectado do banco de dados e armazená-lo em `g.user`. Mais tarde, quando a função de visualização é invocada, ela pode acessar o usuário de lá.

Exemplos de ganchos de solicitação serão mostrados em capítulos futuros, portanto, não se preocupe se isso ainda não fizer sentido.

## Respostas

Quando o Flask invoca uma função de visualização, ele espera que seu valor de retorno seja a resposta à solicitação. Na maioria dos casos, a resposta é uma string simples que é enviada de volta ao cliente como uma página HTML.

Mas o protocolo HTTP requer mais do que uma string como resposta a uma solicitação. Uma parte muito importante da resposta HTTP é o *código de status*, que o Flask por padrão define como 200, o código que indica que a solicitação foi realizada com sucesso.

Quando uma função de visualização precisa responder com um código de status diferente, ela pode adicionar o código numérico como um segundo valor de retorno após o texto de resposta. Por exemplo, a seguinte função de visualização retorna um código de status 400, o código para um erro de solicitação inválida:

```
@app.route('/')
def index(): return
    '<h1>Bad Request</h1>', 400
```

As respostas retornadas por funções de visualização também podem receber um terceiro argumento, um dicionário de cabeçalhos que são adicionados à resposta HTTP. Isso raramente é necessário, mas você verá um exemplo no [Capítulo 14](#).

Em vez de retornar um, dois ou três valores como uma tupla, as funções de visualização do Flask têm a opção de retornar um objeto `Response`. A função `make_response()` recebe um, dois ou três argumentos, os mesmos valores que podem ser retornados de uma função de visualização, e retorna um objeto `Response`. Às vezes é útil realizar essa conversão dentro do

view e, em seguida, use os métodos do objeto de resposta para configurar ainda mais a resposta. O exemplo a seguir cria um objeto de resposta e define um cookie nele:

#### da importação do frasco make\_response

```
@app.route('/') def
index(): response
    = make_response('<h1>Este documento carrega um cookie!</h1>')
    response.set_cookie('answer', '42') return response
```

Existe um tipo especial de resposta chamado *redirecionamento*. Esta resposta não inclui um documento de página; ele apenas fornece ao navegador um novo URL para carregar uma nova página.

Redirecionamentos são comumente usados com formulários da web, como você aprenderá no [Capítulo 4](#).

Um redirecionamento é normalmente indicado com um código de status de resposta 302 e o URL para redirecionar fornecido em um cabeçalho de localização . Uma resposta de redirecionamento pode ser gerada usando um retorno de três valores, ou também com um objeto Response , mas devido ao seu uso frequente, o Flask fornece uma função auxiliar redirect() que cria esta resposta:

```
do redirecionamento de importação do frasco

@app.route('/') def
index():
    return redirect('http://www.example.com')
```

Outra resposta especial é emitida com a função abort , que é usada para tratamento de erros. O exemplo a seguir retorna o código de status 404 se o argumento id dynamic fornecido na URL não representar um usuário válido:

#### do frasco importação abortar

```
@app.route('/user/<id>') def
get_user(id): user = load_user(id)
    se não for user: abort(404)
    return '<h1>Hello, %s</
            h1>' % user .nome
```

Observe que abort não retorna o controle de volta para a função que a chama, mas devolve o controle ao servidor da Web levantando uma exceção.

## Extensões de Frasco

O frasco foi projetado para ser estendido. Ele intencionalmente fica fora de áreas de funcionalidade importante, como banco de dados e autenticação de usuário, dando a você a liberdade de selecionar os pacotes que melhor se adequam ao seu aplicativo ou escrever o seu próprio, se desejar.

Existe uma grande variedade de extensões para muitos propósitos diferentes que foram criadas pela comunidade e, se isso não for suficiente, qualquer pacote ou biblioteca padrão do Python também pode ser usado. Para dar uma ideia de como uma extensão é incorporada em um aplicativo, a seção a seguir adiciona uma extensão ao *hello.py* que aprimora o aplicativo com argumentos de linha de comando.

### Opções de linha de comando com Flask-Script O

servidor web de desenvolvimento do Flask suporta várias opções de configuração de inicialização, mas a única maneira de especificá-las é passando-as como argumentos para a chamada `app.run()` no script. Isso não é muito conveniente; a maneira ideal de passar opções de configuração é por meio de argumentos de linha de comando.

Flask-Script é uma extensão para Flask que adiciona um analisador de linha de comando ao seu aplicativo Flask. Ele vem empacotado com um conjunto de opções de uso geral e também suporta comandos personalizados.

A extensão é instalada com pip:

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

O [Exemplo 2-3](#) mostra as alterações necessárias para adicionar análise de linha de comando ao aplicativo *hello.py*.

#### *Exemplo 2-3. hello.py: Usando Flask-Script*

```
from flask.ext.script import Manager  
manager = Manager(app)  
  
# ...  
  
if __name__ == '__main__':  
    manager.run()
```

Extensões desenvolvidas especificamente para Flask estão expostas no espaço de nomes `flask.ext`. Flask-Script exporta uma classe chamada `Manager`, que é importada de `flask.ext.script`.

O método de inicialização desta extensão é comum a muitas extensões: uma instância da classe principal é inicializada passando a instância da aplicação como argumento para o construtor. O objeto criado é então usado conforme apropriado para cada extensão. Nesse caso, a inicialização do servidor é roteada por meio de `manager.run()`, onde a linha de comando é analisada.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar git checkout 2c para verificar esta versão do aplicativo.

Com essas alterações, o aplicativo adquire um conjunto básico de opções de linha de comando. A execução de `hello.py` agora mostra uma mensagem de uso:

```
$ python hello.py uso:  
hello.py [-h] {shell, runserver} ...
```

argumentos posicionais:  
 {shell, runserver} shell

executar servidor	Executa um shell Python dentro do contexto do aplicativo Flask.
	Executa o servidor de desenvolvimento Flask, ou seja, <code>app.run()</code>

argumentos opcionais:  
`-h, --help` mostre esta mensagem de ajuda e saia

O comando `shell` é usado para iniciar uma sessão de shell Python no contexto do aplicativo. Você pode usar esta sessão para executar tarefas ou testes de manutenção ou para depurar problemas.

O comando `runserver`, como o próprio nome indica, inicia o servidor web. A execução de `python hello.py runserver` inicia o servidor web no modo de depuração, mas há muito mais opções disponíveis:

```
(venv) $ python hello.py runserver --help uso: hello.py  
runserver [-h] [-t HOST] [-p PORT] [--threaded]  
                 [--processes PROCESSES] [--passthrough-errors] [-d] [-r]
```

Executa o servidor de desenvolvimento Flask, ou seja, `app.run()`

argumentos opcionais:  
`-h, --help -t HOST, --host` mostre esta mensagem de ajuda e saia  
`HOST -p PORT, --port`  
`PORT --threaded --processes PROCESSES`  
`--passthrough-errors -d, --no-debug -r, --no-debug` sem recarga

O argumento `--host` é uma opção útil porque informa ao servidor web qual interface de rede escutar para conexões de clientes. Por padrão, o servidor web de desenvolvimento do Flask escuta conexões no host local, portanto, somente conexões originadas de dentro do computador que executa o servidor são aceitas. O comando a seguir faz com que o servidor web escute conexões na interface de rede pública, permitindo que outros computadores na rede também se conectem:

```
(venv) $ python hello.py runserver --host 0.0.0.0 * Executando  
em http://0.0.0.0:5000/* Reiniciando com reloader
```

O servidor web deve agora estar acessível a partir de qualquer computador na rede em <http://abcd:5000>, onde “abcd” é o endereço IP externo do computador que executa o servidor.

Este capítulo introduziu o conceito de respostas às solicitações, mas há muito mais a dizer sobre as respostas. O Flask fornece um suporte muito bom para gerar respostas usando *modelos*, e esse é um tópico tão importante que o próximo capítulo é dedicado a ele.



---

## CAPÍTULO 3

# Modelos

A chave para escrever aplicativos que são fáceis de manter é escrever um código limpo e bem estruturado. Os exemplos que você viu até agora são muito simples para demonstrar isso, mas as funções de visualização do Flask têm dois propósitos completamente independentes disfarçados como um, o que cria um problema.

A tarefa óbvia de uma função de visualização é gerar uma resposta a uma solicitação, como você viu nos exemplos mostrados no [Capítulo 2](#). Para as solicitações mais simples, isso é suficiente, mas em geral uma solicitação aciona uma mudança no estado da aplicação , e a função de visualização também é onde essa alteração é gerada.

Por exemplo, considere um usuário que está registrando uma nova conta em um site. O usuário digita um endereço de e-mail e uma senha em um formulário da web e clica no botão Enviar. No servidor, uma solicitação que inclui os dados do usuário chega e o Flask a envia para a função de visualização que trata das solicitações de registro. Essa função de visualização precisa conversar com o banco de dados para adicionar o novo usuário e, em seguida, gerar uma resposta para enviar de volta ao navegador. Esses dois tipos de tarefas são formalmente chamados de *lógica de negócios* e *lógica de apresentação*, respectivamente.

A mistura de lógica de negócios e apresentação leva a um código difícil de entender e manter. Imagine ter que construir o código HTML para uma tabela grande concatenando os dados obtidos do banco de dados com os literais de string HTML necessários. Mover a lógica de apresentação para *modelos* ajuda a melhorar a capacidade de manutenção do aplicativo.

Um modelo é um arquivo que contém o texto de uma resposta, com variáveis de espaço reservado para as partes dinâmicas que serão conhecidas apenas no contexto de uma solicitação. O processo que substitui as variáveis por valores reais e retorna uma string de resposta final é chamado de *renderização*. Para a tarefa de renderização de modelos, o Flask usa um poderoso mecanismo de modelo chamado *Jinja2*.

## O mecanismo de modelo Ninja2

Em sua forma mais simples, um modelo Ninja2 é um arquivo que contém o texto de uma resposta. O [Exemplo 3-1](#) mostra um modelo Ninja2 que corresponde à resposta da função de visualização index() do [Exemplo 2-1](#).

*Exemplo 3-1. templates/index.html: modelo Ninja2*

```
<h1>Olá, mundol</h1>
```

A resposta retornada pela função view user() do [Exemplo 2-2](#) tem um componente dinâmico, que é representado por uma variável. O [Exemplo 3-2](#) mostra o modelo que implementa essa resposta.

*Exemplo 3-2. templates/user.html: modelo Ninja2*

```
<h1>Olá, {{ name }}</h1>
```

### Renderizando modelos Por

padrão, o Flask procura modelos em uma subpasta de *modelos* localizada dentro da pasta do aplicativo. Para a próxima versão do *hello.py*, você precisa armazenar os modelos definidos anteriormente em uma nova pasta de *modelos* como *index.html* e *user.html*.

As funções de visualização no aplicativo precisam ser modificadas para renderizar esses modelos.

O [Exemplo 3-3](#) mostra essas mudanças.

*Exemplo 3-3. hello.py: renderizando um modelo*

```
do frasco import Flask, render_template

# ...

@app.route('/index') def
index():
    return render_template('index.html')

@app.route('/user/<name>') def
user(name): return
    render_template('user.html', name=name)
```

A função render\_template fornecida pelo Flask integra o mecanismo de modelo Ninja2 com o aplicativo. Esta função toma o nome do arquivo do template como seu primeiro argumento. Quaisquer argumentos adicionais são pares de chave/valor que representam valores reais para variáveis referenciadas no modelo. Neste exemplo, o segundo modelo está recebendo uma variável de nome .

Argumentos de palavras-chave como name=name no exemplo anterior são bastante comuns, mas podem parecer confusos e difíceis de entender se você não estiver acostumado a eles. O “nome” no

lado esquerdo representa o nome do argumento, que é usado no espaço reservado escrito no modelo. O “nome” do lado direito é uma variável no escopo atual que fornece o valor para o argumento de mesmo nome.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar git checkout 3a para verificar esta versão do aplicativo.

## Variáveis

A construção {{ name }} usada no modelo mostrado no [Exemplo 3-2](#) faz referência a uma variável, um espaço reservado especial que informa ao mecanismo de modelo que o valor que entra nesse local deve ser obtido dos dados fornecidos no momento em que o modelo é renderizado .

Jinja2 reconhece variáveis de qualquer tipo, mesmo tipos complexos como listas, dicionários e objetos. A seguir estão mais alguns exemplos de variáveis usadas em templates:

```
<p>Um valor de um dicionário: {{ mydict['key'] }}.</p> <p>Um valor de uma lista: {{ mylist[3] }}.</p> <p>A valor de uma lista, com um índice variável: {{ mylist[myintvar] }}.</p> <p>Um valor do método de um objeto: {{ myobj.somemethod() }}.</p>
```

As variáveis podem ser modificadas com *filtros*, que são adicionados após o nome da variável com uma barra vertical como separador. Por exemplo, o modelo a seguir mostra a variável de nome em maiúscula:

```
Olá, {{ nome|maiúsculo}}
```

A [Tabela 3-1](#) lista alguns dos filtros comumente usados que vêm com o Jinja2.

*Tabela 3-1. Filtros de variáveis Jinja2*

Nome do filtro	Descrição
seguro	Renderiza o valor sem aplicar escape
capitalizar	Converte o primeiro caractere do valor em maiúsculas e o restante em minúsculas
mais baixo	Converte o valor em caracteres minúsculos
superior	Converte o valor em caracteres maiúsculos
título	Capitaliza cada palavra no valor
apagar	Remove os espaços em branco à esquerda e à direita do valor
striptags	Remove quaisquer tags HTML do valor antes de renderizar

O filtro seguro é interessante destacar. Por padrão, Jinja2 *escapa* de todas as variáveis para fins de segurança. Por exemplo, se uma variável for definida com o valor '`<h1>Hello</h1>`', Jinja2

irá renderizar a string como '<h1>Hello</h1>', o que fará com que o elemento h1 seja exibido e não interpretado pelo navegador. Muitas vezes é necessário exibir o código HTML armazenado em variáveis, e para esses casos é utilizado o filtro seguro .



Nunca use o filtro seguro em valores que não são confiáveis, como texto inserido por usuários em formulários da web.

A lista completa de filtros pode ser obtida na [documentação oficial do Jinja2](#).

## Estruturas de controle

Jinja2 oferece várias estruturas de controle que podem ser usadas para alterar o fluxo do template. Esta seção apresenta alguns dos mais úteis com exemplos simples.

O exemplo a seguir mostra como as instruções condicionais podem ser inseridas em um modelo:

```
{% se usuário %}
    Olá, {{ usuário }}! {%
senão %}
    Olá estranho! {%
fim
se %}
```

Outra necessidade comum em templates é renderizar uma lista de elementos. Este exemplo mostra como isso pode ser feito com um loop for :

```
<ul>
{%
para comentários %
<li>{{ comentar }}</li> {%
endfor
%} </ul>
```

Jinja2 também suporta *macros*, que são semelhantes a funções no código Python. Por exemplo:

```
{%
macro render_comment(comment) %
<li>{{ comment }}</li> {%
endmacro %}
```

```
<ul>
{%
para comentários %
{{ render_comment(comment) }} {%
endfor %} </ul>
```

Para tornar as macros mais reutilizáveis, elas podem ser armazenadas em arquivos independentes que são importados de todos os modelos que precisam delas:

```
{%
import 'macros.html' como macros %
<ul>
```

```
{% para comentário nos comentários
    %} {{ macros.render_comment(comment) }}
{%- endfor %} </ul>
```

Partes do código do modelo que precisam ser repetidas em vários locais podem ser armazenadas em um arquivo separado e *incluídas* em todos os modelos para evitar repetições:

```
{% include 'common.html' %}
```

Ainda outra maneira poderosa de reutilizar é por meio de herança de modelo, que é semelhante à herança de classe no código Python. Primeiro, um template base é criado com o nome *base.html*:

```
<html>
<head>
    {% block head %}
        <title>{% block title %}{% endblock %} - Meu aplicativo</title> {% endblock %}
    %} </head> <body> {% block body % } {% endblock %} </body> </html>
```

Aqui as tags de bloco definem elementos que um template derivado pode alterar. Neste exemplo, existem blocos chamados head, title e body; note que o título está contido por head. O exemplo a seguir é um modelo derivado do modelo base:

```
{% extends "base.html" %} {%
    block title %}Index{% endblock %} {% block
    head %} {{ super() }} <style> </style> {% endblock %}
    {% block body %} <h1>Olá,
    Mundo!</h1> {% endblock %}
```

A diretiva `extends` declara que este template deriva de *base.html*. Esta diretiva é seguida por novas definições para os três blocos definidos no template base, que são inseridos nos locais apropriados. Observe que a nova definição do bloco `head`, que não está vazio no template base, usa `super()` para reter o conteúdo original.

O uso no mundo real de todas as estruturas de controle apresentadas nesta seção será mostrado posteriormente, para que você tenha a oportunidade de ver como elas funcionam.

## Integração do Twitter Bootstrap com o Flask-Bootstrap

**Bootstrap** é uma estrutura de código aberto do Twitter que fornece componentes de interface do usuário para criar páginas da Web limpas e atraentes que são compatíveis com todos os navegadores modernos.

Bootstrap é uma estrutura do lado do cliente, portanto, o servidor não está diretamente envolvido com ela. Tudo o que o servidor precisa fazer é fornecer respostas HTML que façam referência às folhas de estilo em cascata (CSS) do Bootstrap e arquivos JavaScript e instanciar os componentes desejados por meio de código HTML, CSS e JavaScript. O lugar ideal para fazer tudo isso é nos templates.

A maneira óbvia de integrar o Bootstrap com o aplicativo é fazer todas as alterações necessárias nos modelos. Uma abordagem mais simples é usar uma extensão do Flask chamada Flask Bootstrap para simplificar o esforço de integração. Flask-Bootstrap pode ser instalado com pip:

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

As extensões do Flask geralmente são inicializadas ao mesmo tempo em que a instância do aplicativo é criada. O [Exemplo 3-4](#) mostra a inicialização do Flask-Bootstrap.

*Exemplo 3-4. hello.py: inicialização do Flask-Bootstrap*

```
from flask.ext.bootstrap import Bootstrap #
bootstrap = Bootstrap(app)
```

Como o Flask-Script no [Capítulo 2](#), o Flask-Bootstrap é importado do espaço de nomes flask.ext e inicializado passando a instância do aplicativo no construtor.

Depois que o Flask-Bootstrap é inicializado, um modelo base que inclui todos os arquivos do Bootstrap fica disponível para o aplicativo. Este template tira vantagem da herança de template do Jinja2; o aplicativo estende um modelo base que possui a estrutura geral da página incluindo os elementos que importam o Bootstrap. O [Exemplo 3-5](#) mostra uma nova versão de *user.html* como um modelo derivado.

*Exemplo 3-5. templates/user.html: Modelo que usa Flask-Bootstrap*

```
{% estende "bootstrap/base.html" %}

{% block title %}Flasky{% endblock %}

{% block navbar %}
<div class="navbar navbar-inverse" role="navigation"> <div
class="container"> <div class="navbar-header">

    <button type="button" class="navbar-toggle" data-
    toggle="collapse" data-target=".navbar-collapse"> <span class="sr-
    only">Alternar navegação <span class="icon-bar"> <span
    class="icon-bar"> <span class="icon-bar">
```

```

</button>
<a class="navbar-brand" href="/">Flasky</a> </div>
<div class="navbar-collapse colapso">

<ul class="nav navbar-nav">
<li><a href="/">Início</a></li> </ul>
</div> </div> </div> {% endblock %}

{%
block content %}
<div class="container">
<div class="page-header">
<h1>Olá, {{ name }}!</h1> </div>
</div> {% endblock %}

```

A diretiva `extend` Jinja2 implementa a herança de template referenciando `bootstrap/base.html` do Flask-Bootstrap. O modelo base do Flask-Bootstrap fornece uma página da Web de esqueleto que inclui todos os arquivos CSS e JavaScript do Bootstrap.

Modelos base definem *blocos* que podem ser substituídos por modelos derivados. As diretivas `block` e `endblock` definem blocos de conteúdo que são adicionados ao template base.

O template `user.html` acima define três blocos chamados `title`, `navbar` e `content`.

Esses são todos os blocos que o modelo base exporta para definir os modelos derivados. O bloco de título é direto; seu conteúdo aparecerá entre as tags `<title>` no cabeçalho do documento HTML renderizado. A barra de navegação e os blocos de conteúdo são reservados para a barra de navegação da página e o conteúdo principal.

Neste template, o bloco `navbar` define uma barra de navegação simples usando componentes Bootstrap. O bloco de conteúdo tem um contêiner `<div>` com um cabeçalho de página dentro. A linha de saudação que estava na versão anterior do modelo agora está dentro do cabeçalho da página. A [Figura 3-1](#) mostra a aparência do aplicativo com essas alterações.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar `git checkout 3b` para verificar esta versão do aplicativo.

A [documentação oficial do Bootstrap](#) é um ótimo recurso de aprendizado cheio de exemplos prontos para copiar/colar.

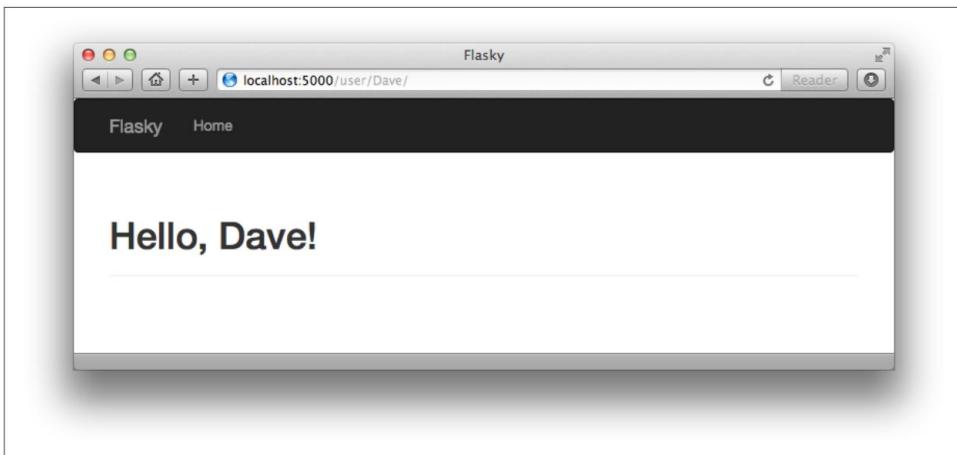


Figura 3-1. Modelos do Twitter Bootstrap

O template `base.html` do Flask-Bootstrap define vários outros blocos que podem ser usados em modelos derivados. A [Tabela 3-2](#) mostra a lista completa de blocos disponíveis.

*Tabela 3-2. Blocos de template base do Flask-Bootstrap*

Nome do bloco	Descrição
doc	Todo o documento HTML
html_attribs	Atributos dentro da tag <html>
html	O conteúdo da tag <html>
cabeça	O conteúdo da tag <head>
título	O conteúdo da tag <title>
metas	A lista de tags <meta>
estilos	Definições de folha de estilo em cascata
body_attribs	Atributos dentro da tag <body>
corpo	O conteúdo da tag <body>
barra de navegação	Barra de navegação definida pelo usuário
conteúdo	Conteúdo de página definido pelo usuário
roteiros	Declarações JavaScript na parte inferior do documento

Muitos dos blocos na [Tabela 3-2](#) são usados pelo próprio Flask-Bootstrap, portanto, substituindo-os diretamente causaria problemas. Por exemplo, os blocos de estilos e scripts são onde os arquivos Bootstrap são declarados. Se o aplicativo precisar adicionar seu próprio conteúdo a um bloco que já tem algum conteúdo, então a função `super()` do Jinja2 deve ser usada. Por exemplo,

é assim que o bloco de scripts precisaria ser escrito no modelo derivado para adicionar um novo arquivo JavaScript ao documento:

```
{% block scripts %}
{{ super() }} <script
type="text/javascript" src="my-script.js"></script> {% endblock %}
```

## Páginas de erro personalizadas

Quando você insere uma rota inválida na barra de endereços do seu navegador, você recebe uma página de erro de código 404. A página de erro agora é muito simples e pouco atraente e não tem consistência com a página que usa o Bootstrap.

O Flask permite que um aplicativo defina páginas de erro personalizadas que podem ser baseadas em modelos, como rotas regulares. Os dois códigos de erro mais comuns são 404, acionados quando o cliente solicita uma página ou rota desconhecida, e 500, acionados quando há uma exceção não tratada. O Exemplo 3-6 mostra como fornecer manipuladores personalizados para esses dois erros.

*Exemplo 3-6. hello.py: páginas de erro personalizadas*

```
@app.errorhandler(404)
def page_not_found(e):
    return render_template('404.html'), 404

@app.errorhandler(500)
def internal_server_error(e): return
    render_template('500.html'), 500
```

Os manipuladores de erros retornam uma resposta, como funções de visualização. Eles também retornam o código de status numérico que corresponde ao erro.

Os modelos referenciados nos manipuladores de erros precisam ser gravados. Esses templates devem seguir o mesmo layout das páginas normais, então neste caso eles terão uma barra de navegação e um cabeçalho de página que mostra a mensagem de erro.

A maneira direta de escrever esses modelos é copiar `templates/user.html` para `templates/404.html` e `templates/500.html` e, em seguida, alterar o elemento de cabeçalho da página nesses dois novos arquivos para a mensagem de erro apropriada, mas isso gerará uma muita duplicação.

A herança de template do Jinja2 pode ajudar nisso. Da mesma forma que o Flask-Bootstrap fornece um template base com o layout básico da página, a aplicação pode definir seu próprio template base com um layout de página mais completo que inclui a barra de navegação e deixa o conteúdo da página a ser definido em derivados. modelos. O Exemplo 3-7 mostra `templates/base.html`, um novo template que herda de `bootstrap/base.html` e define a barra de navegação, mas é ele próprio um template base para outros templates como `templates/user.html`, `templates/404.html` e `templates/500.html`.

*Exemplo 3-7. templates/base.html: modelo de aplicativo base com barra de navegação*

```
{% estende "bootstrap/base.html" %}

{% block title %}Flasky{% endblock %}

{% block navbar %}
<div class="navbar navbar-inverse" role="navigation"> <div
class="container"> <div class="navbar-header">

    <button type="button" class="navbar-toggle" data-
    toggle="collapse" data-target=".navbar-collapse"> <span class="sr-
    only">Alternar navegação <span class="icon-bar"> <span
    class="icon-bar"> <span class="icon-bar"> </button> <a
    class="navbar -brand" href="/">Flasky</a> </div> <div
    class="navbar-collapse colapso">

        <ul class="nav navbar-nav">
            <li><a href="/">Início</a></li> </ul>
        </div> </div> </div> {% endblock %}

{% block content %}
<div class="container"> {%
    block page_content %}{% endblock %} </div>
{% endblock %}
```

No bloco de conteúdo deste template há apenas um elemento container `<div>` que envolve um novo bloco vazio chamado `page_content`, que os templates derivados podem definir.

Os modelos do aplicativo agora herdarão desse modelo em vez de diretamente do Flask-Bootstrap. O Exemplo 3-8 mostra como é simples construir uma página de erro 404 de código personalizado que herda de `templates/base.html`.

*Exemplo 3-8. templates/404.html: página de erro de código 404 personalizado usando herança de modelo atitude*

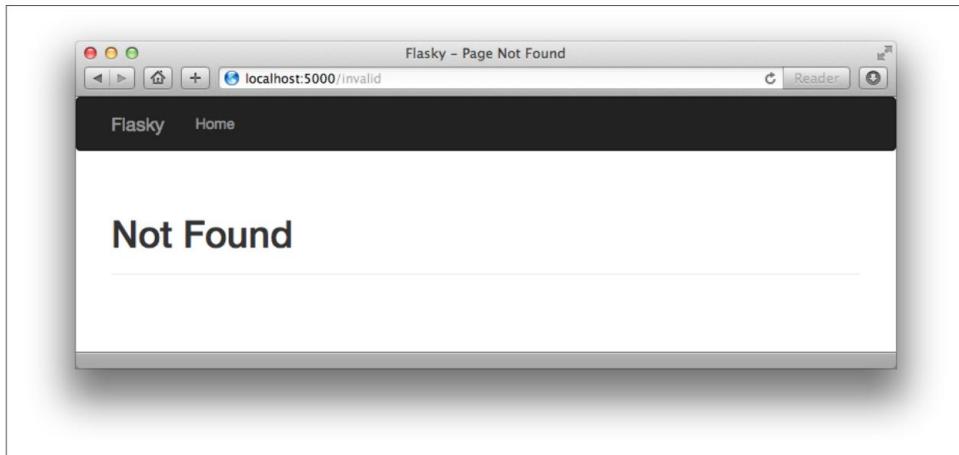
```
{% estende "base.html" %}

{% block title %}Flasky - Página não encontrada{% endblock %}

{% block page_content %}
<div class="page-header">
    <h1>Não encontrado</h1>
```

```
</div>
{%
  bloco final %}
```

A **Figura 3-2** mostra a aparência da página de erro no navegador.



*Figura 3-2. Página de erro de código 404 personalizado*

O template `templates/user.html` agora pode ser simplificado tornando-o herdado do template base, conforme mostrado no **Exemplo 3-9**.

*Exemplo 3-9. templates/user.html: modelo de página simplificado usando herança de modelo*

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

{% block title %}Flasky{% endblock %}

{% block page_content %}
<div class="page-header">
  <h1>Olá, {{ name }}!</h1>
</div>
{% endblock %}
```



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar `git checkout 3c` para verificar esta versão do aplicativo.

## Links

Qualquer aplicativo que tenha mais de uma rota precisará invariavelmente incluir links que conectem as diferentes páginas, como em uma barra de navegação.

Escrever as URLs como links diretamente no template é trivial para rotas simples, mas para rotas dinâmicas com porções variáveis pode ser mais complicado construir as URLs diretamente no template. Além disso, as URLs escritas explicitamente criam uma dependência indesejada nas rotas definidas no código. Se as rotas forem reorganizadas, os links nos modelos podem quebrar.

Para evitar esses problemas, o Flask fornece a função auxiliar `url_for()`, que gera URLs a partir das informações armazenadas no mapa de URLs do aplicativo.

Em seu uso mais simples, essa função usa o nome da função de visualização (ou nome do *terminal* para rotas definidas com `app.add_url_rule()`) como seu único argumento e retorna sua URL.

Por exemplo, na versão atual do `hello.py` a chamada `url_for('index')` retornaria `/`. Chamar `url_for('index', _external=True)` retornaria uma URL absoluta, que neste exemplo é `http://localhost:5000/`.



As URLs relativas são suficientes ao gerar links que conectam as diferentes rotas do aplicativo. URLs absolutos são necessários apenas para links que serão usados fora do navegador da web, como ao enviar links por e-mail.

URLs dinâmicas podem ser geradas com `url_for()` passando as partes dinâmicas como argumentos de palavras-chave. Por exemplo, `url_for('user', name='john', _external=True)` retornaria `http://localhost:5000/user/john`.

Os argumentos de palavras-chave enviados para `url_for()` não se limitam a argumentos usados por rotas dinâmicas. A função adicionará quaisquer argumentos extras à string de consulta. Por exemplo, `url_for('index', page=2)` retornaria `/?page=2`.

## Arquivos estáticos

Os aplicativos da Web não são feitos apenas de código e modelos Python. A maioria dos aplicativos também usa arquivos estáticos, como imagens, arquivos de origem JavaScript e CSS que são referenciados no código HTML.

Você deve se lembrar que quando o mapa de URL do aplicativo `hello.py` foi inspecionado no [Capítulo 2](#), uma entrada estática apareceu nele. Isso ocorre porque as referências a arquivos estáticos são tratadas como uma rota especial definida como `/static/<filename>`. Por exemplo, uma chamada para `url_for('static', filename='css/styles.css', _external=True)` retornaria `http://localhost:5000/static/css/styles.css`.

Em sua configuração padrão, o Flask procura arquivos estáticos em um subdiretório chamado `static` localizado na pasta raiz do aplicativo. Os arquivos podem ser organizados em subdiretórios dentro desta pasta, se desejado. Quando o servidor recebe a URL do exemplo anterior, ele

gera uma resposta que inclui o conteúdo de um arquivo no sistema de arquivos localizado em `static/css/styles.css`.

O Exemplo 3-10 mostra como o aplicativo pode incluir um ícone `favicon.ico` no modelo base para que os navegadores sejam exibidos na barra de endereços.

*Exemplo 3-10. templates/base.html: definição de favicon*

```
{% block head %}
{{ super() }} <link
rel="shortcut icon" href="{{ url_for('static', filename = 'favicon.ico') }}"
type="image/x-icon">
<link rel="icon" href="{{ url_for('static', filename = 'favicon.ico') }}" type="image/x-icon"> { %
bloco final %}
```

A declaração do ícone é inserida no final do bloco de cabeçalho . Observe como `super()` é usado para preservar o conteúdo original do bloco definido nos templates base.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar `git checkout 3d` para verificar esta versão do aplicativo.

## Localização de datas e horas com Flask-Moment

O manuseio de datas e horas em um aplicativo da Web não é um problema trivial quando os usuários trabalham em diferentes partes do mundo.

O servidor precisa de unidades de tempo uniformes que são independentes da localização de cada usuário, portanto, normalmente é usado o Tempo Universal Coordenado (UTC). Para os usuários, no entanto, ver horários expressos em UTC pode ser confuso, pois os usuários sempre esperam ver datas e horários apresentados em seu horário local e formatados de acordo com os costumes locais de sua região.

Uma solução elegante que permite que o servidor trabalhe exclusivamente em UTC é enviar essas unidades de tempo para o navegador da web, onde são convertidas para a hora local e renderizadas.

Os navegadores da Web podem fazer um trabalho muito melhor nessa tarefa porque têm acesso às configurações de fuso horário e localidade no computador do usuário.

Existe uma excelente biblioteca de código aberto do lado do cliente escrita em JavaScript que renderiza datas e horas no navegador chamada `moment.js`. Flask-Moment é uma extensão para aplicativos Flask que integra `moment.js` em templates Jinja2. O Flask-Moment está instalado com pip:

```
(venv) $ pip install frasco-momento
```

A extensão é inicializada conforme mostrado no Exemplo 3-11.

*Exemplo 3-11. hello.py: Initialize o Flask-Moment*

```
from flask.ext.moment import Moment
moment = Moment(app)
```

Flask-Moment depende de *jquery.js* além de *moment.js*. Essas duas bibliotecas precisam ser incluídas em algum lugar no documento HTML - diretamente, nesse caso, você pode escolher quais versões usar, ou por meio das funções auxiliares fornecidas pela extensão, que fazem referência a versões testadas dessas bibliotecas de um Content Delivery. Rede (CDN). Como o Bootstrap já inclui *jquery.js*, apenas *moment.js* precisa ser adicionado neste caso. O [Exemplo 3-12](#) mostra como esta biblioteca é carregada nos scripts do template base.

*Exemplo 3-12. templates/base.html: Importar biblioteca moment.js*

```
{% block scripts %}
{{ super() }}
{{ moment.include_moment() }} {%
endblock %}
```

Para trabalhar com timestamps, o Flask-Moment disponibiliza uma classe de momento para os modelos. O exemplo do [Exemplo 3-13](#) passa uma variável chamada `current_time` para o template para renderização.

*Exemplo 3-13. hello.py: adicione uma variável de data e hora*

**de datetime importação datetime**

```
@app.route('/')
def index():
    return render_template('index.html',
                           current_time=datetime.utcnow())
```

O [Exemplo 3-14](#) mostra como `current_time` é renderizado no template.

*Exemplo 3-14. templates/index.html: renderização de carimbo de data/hora com Flask-Moment*

```
<p>A data e hora locais são {{ moment(current_time).format('LLL') }}.</p> <p>Isso foi
{{ moment(current_time).fromNow(refresh=True) }}</p>
```

O formato `format('LLL')` renderiza a data e a hora de acordo com o fuso horário e as configurações de localidade no computador cliente. O argumento determina o estilo de renderização, de 'L' a 'LLLL' para diferentes níveis de verbosidade. A função `format()` também pode aceitar especificadores de formato personalizados.

O estilo de renderização `fromNow()` mostrado na segunda linha renderiza um timestamp relativo e o atualiza automaticamente com o passar do tempo. Inicialmente, esse carimbo de data/hora será mostrado como "alguns segundos atrás", mas a opção de atualização o manterá atualizado com o passar do tempo, portanto, se você deixar a página aberta por alguns minutos, verá o texto mudando para "um minuto atrás", depois "2 minutos atrás" e assim por diante.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar git checkout 3e para verificar esta versão do aplicativo.

Flask-Moment implementa os métodos `format()`, `fromNow()`, `fromTime()`, `calendar()`, `valueOf()` e `unix()` de `moment.js`. Consulte a [documentação](#) para conhecer todas as opções de formatação oferecidas.



O Flask-Moment assume que os timestamps manipulados pelo aplicativo do lado do servidor são objetos de data e hora “ingênuos” expressos em UTC. Consulte a documentação para o `datetime` pacote na biblioteca padrão para obter informações sobre objetos de data e hora ingênuos e cientes.

Os timestamps renderizados pelo Flask-Moment podem ser localizados em vários idiomas. Um idioma pode ser selecionado no modelo passando o código do idioma para a função `lang()`:

```
 {{ moment.long('it') }}
```

Com todas as técnicas discutidas neste capítulo, você deve ser capaz de construir páginas da Web modernas e fáceis de usar para seu aplicativo. O próximo capítulo aborda um aspecto dos templates ainda não discutidos: como interagir com o usuário através de formulários web.



## CAPÍTULO 4

### Formulários da Web

O objeto de solicitação, apresentado no [Capítulo 2](#), expõe todas as informações enviadas pelo cliente com uma solicitação. Em particular, `request.form` fornece acesso aos dados de formulário enviados em solicitações POST .

Embora o suporte fornecido no objeto de solicitação do Flask seja suficiente para o manuseio de formulários da web, existem várias tarefas que podem se tornar tediosas e repetitivas. Dois bons exemplos são a geração de código HTML para formulários e a validação dos dados do formulário submetido.

O [Frasco-WTF](#) extensão torna o trabalho com formulários web uma experiência muito mais agradável. Esta extensão é um wrapper de integração do Flask em torno do [WTForms independente](#) de estrutura pacote.

Flask-WTF e suas dependências podem ser instaladas com pip:

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

#### Proteção contra falsificação de solicitação entre sites (CSRF)

Por padrão, o Flask-WTF protege todos os formulários contra ataques Cross-Site Request Forgery (CSRF). Um ataque CSRF ocorre quando um site malicioso envia solicitações para um site diferente no qual a vítima está logada.

Para implementar a proteção CSRF, o Flask-WTF precisa que o aplicativo configure uma chave de criptografia. Flask-WTF usa essa chave para gerar tokens criptografados que são usados para verificar a autenticidade de solicitações com dados de formulário. O [Exemplo 4-1](#) mostra como configurar uma chave de criptografia.

*Exemplo 4-1. hello.py: configuração Flask-WTF*

```
app = Flask(__name__)
app.config['SECRET_KEY'] = 'string difícil de adivinhar'
```

O dicionário `app.config` é um local de uso geral para armazenar variáveis de configuração usadas pela estrutura, pelas extensões ou pelo próprio aplicativo. Os valores de configuração podem ser adicionados ao objeto `app.config` usando a sintaxe de dicionário padrão. O objeto de configuração também possui métodos para importar valores de configuração de arquivos ou do ambiente.

A variável de configuração `SECRET_KEY` é usada como chave de criptografia de uso geral pelo Flask e várias extensões de terceiros. Como o próprio nome indica, a força da criptografia depende do valor dessa variável ser secreta. Escolha uma chave secreta diferente em cada aplicativo que você criar e certifique-se de que essa string não seja conhecida por ninguém.



Para maior segurança, a chave secreta deve ser armazenada em uma variável de ambiente em vez de ser incorporada ao código. Esta técnica é descrita no [Capítulo 7](#).

## Aulas de formulário

Ao usar Flask-WTF, cada formulário da web é representado por uma classe que herda da classe `Form`. A classe define a lista de campos no formulário, cada um representado por um objeto. Cada objeto de campo pode ter um ou mais *validadores* anexados; validadores são funções que verificam se a entrada enviada pelo usuário é válida.

[O Exemplo 4-2](#) mostra um formulário web simples que possui um campo de texto e um botão de envio.

```
Exemplo 4-2. hello.py: definição de classe de formulário
de flask.ext.wtf import Formulário
de wtforms import StringField, SubmitField de
wtforms.validators import Obrigatório

class NameForm(Form):
    name = StringField('Qual é o seu nome?', validators=[Required()])
    submit = SubmitField('Submit')
```

Os campos no formulário são definidos como variáveis de classe e cada variável de classe recebe um objeto associado ao tipo de campo. No exemplo anterior, o formulário `NameForm` tem um campo de texto chamado `name` e um botão de envio chamado `enviar`. A classe `StringField` representa um elemento `<input>` com um atributo `type="text"`. A classe `SubmitField` representa um elemento `<input>` com um atributo `type="submit"`. O primeiro argumento para os construtores de campo é o rótulo que será usado ao renderizar o formulário para HTML.

O argumento opcional `validators` incluído no construtor `StringField` define uma lista de validadores que serão aplicados aos dados enviados pelo usuário antes de serem aceitos. O validador `Required()` garante que o campo não seja enviado vazio.



A classe base Form é definida pela extensão Flask-WTF, então é importado de flask.ext.wtf. Os campos e validadores, no entanto, são importados diretamente do pacote WTForms.

A lista de campos HTML padrão suportados por WTForms é mostrada na [Tabela 4-1](#).

*Tabela 4-1. Campos HTML padrão do WTForms*

Tipo de campo	Descrição
StringField	Campo de texto
TextAreaField	Campo de texto de várias linhas
Campo de senha	Campo de texto de senha
Campo oculto	Campo de texto oculto
DataField	Campo de texto que aceita um valor datetime.date em um determinado formato
Campo DataHora	Campo de texto que aceita um valor datetime.datetime em um determinado formato
Campo inteiro	Campo de texto que aceita um valor inteiro
Campo Decimal	Campo de texto que aceita um valor decimal.Decimal
Campo Flutuante	Campo de texto que aceita um valor de ponto flutuante
BooleanField	Caixa de seleção com valores True e False
Campo de Rádio	Lista de botões de rádio
Selecionar campo	Lista suspensa de opções
SelectMultipleField	Lista suspensa de opções com seleção múltipla
Campo de arquivo	Campo de upload de arquivo
EnviarCampo	Botão de envio de formulário
Campo de formulário	Incorporar um formulário como um campo em um formulário de contêiner
Lista de Campos	Lista de campos de um determinado tipo

A lista de validadores integrados do WTForms é mostrada na [Tabela 4-2](#).

Tabela 4-2. Validadores WTForms

Validador	Descrição
E-mail	Valida um endereço de e-mail
EqualTo	Compara os valores de dois campos; útil ao solicitar que uma senha seja digitada duas vezes para confirmação
IPAddress	Valida um endereço de rede IPv4
Comprimento	Valida o comprimento da string inserida
NumberRange	Valida se o valor inserido está dentro de um intervalo numérico
Opcional	Permite entrada vazia no campo, pulando validadores adicionais
Requerido	Valida se o campo contém dados
Regexp	Valida a entrada em relação a uma expressão regular
URL	Valida um URL
Qualquer	Valida que a entrada é um de uma lista de valores possíveis
Nenhum	Valida que a entrada não é uma lista de valores possíveis

## Renderização HTML de formulários

Campos de formulário são callables que, quando invocados, a partir de um template são renderizados para HTML. Supondo que a função view passe uma instância NameForm para o modelo como um argumento chamado formulário, o modelo pode gerar um formulário HTML simples da seguinte forma:

```
<form method="POST">
    {{ form.name.label }} {{ form.name() }}
    {{ form.submit() }}
</form>
```

Claro, o resultado é extremamente nu. Para melhorar a aparência do formulário, quaisquer argumentos enviados nas chamadas que renderizam os campos são convertidos em atributos HTML para o campo; então, por exemplo, você pode fornecer o id do campo ou atributos de classe e, em seguida, definir estilos CSS:

```
<form method="POST">
    {{ form.name.label }} {{ form.name(id='my-text-field') }}
    {{ form.submit() }}
</form>
```

Mas mesmo com atributos HTML, o esforço necessário para renderizar um formulário dessa maneira é significativo, por isso é melhor aproveitar o próprio conjunto de estilos de formulário do Bootstrap sempre que possível. O Flask-Bootstrap fornece uma função auxiliar de alto nível que renderiza um formulário Flask WTF inteiro usando os estilos de formulário predefinidos do Bootstrap, todos com uma única chamada. Usando o Flask Bootstrap, o formulário anterior pode ser renderizado da seguinte forma:

```
{% import "bootstrap/wtf.html" como wtf %}
{{ wtf.quick_form(form) }}
```

A diretiva de importação funciona da mesma forma que os scripts Python regulares e permite que os elementos do template sejam importados e usados em muitos templates. O arquivo `bootstrap/wtf.html` importado define funções auxiliares que renderizam formulários Flask-WTF usando Bootstrap. A função `wtf.quick_form()` pega um objeto de formulário Flask-WTF e o renderiza usando estilos Bootstrap padrão. O modelo completo para `hello.py` é mostrado no [Exemplo 4-3](#).

*Exemplo 4-3. templates/index.html: Usando Flask-WTF e Flask-Bootstrap para renderizar um formulário*

```
{% extends "base.html" %} {%
import "bootstrap/wtf.html" como wtf %}

{% block title %}Flasky{% endblock %}

{% block page_content %}
<div class="page-header">
    <h1>Olá, {{ name }}{{ else %}Estranho{{ endif %}}!</h1> </div>
{{ wtf.quick_form(form) }} {% endblock %}
```

A área de conteúdo do modelo agora tem duas seções. A primeira seção é um cabeçalho de página que mostra uma saudação. Aqui um modelo condicional é usado. As condicionais em Jinja2 têm o formato `{% if variable %}...{% else %}...{% endif %}`. Se a condição for avaliada como True, então o que aparece entre as diretivas if e else é renderizado para o template. Se a condição for avaliada como False, o que estiver entre o else e o endif será renderizado. O modelo de exemplo renderizará a string “Hello, Stranger!” quando o argumento do modelo de nome é indefinido. A segunda seção do conteúdo renderiza o objeto NameForm usando a função `wtf.quick_form()`.

## Manipulação de formulários em funções de visualização

Na nova versão do `hello.py`, a função de visualização `index()` estará renderizando o formulário e também recebendo seus dados. O [Exemplo 4-4](#) mostra a função de visualização `index()` atualizada.

*Exemplo 4-4. hello.py: métodos de rota*

```
@app.route('/', métodos=['GET', 'POST']) def
index():
    nome = Nenhum
    form = NameForm()
    if form.validate_on_submit(): nome
        = form.name.data
        form.name.data =
    return render_template('index.html', form=form, nome=nome)
```

O argumento de métodos adicionado ao decorador `app.route` diz ao Flask para registrar a função de visualização como um manipulador para solicitações GET e POST no mapa de URL. Quando os métodos não são fornecidos, a função de visualização é registrada para lidar apenas com solicitações GET .

A adição de POST à lista de métodos é necessária porque os envios de formulários são tratados de forma muito mais conveniente como solicitações POST . É possível enviar um formulário como uma solicitação GET , mas como as solicitações GET não têm corpo, os dados são anexados à URL como uma string de consulta e ficam visíveis na barra de endereços do navegador. Por esse e vários outros motivos, os envios de formulários são quase universalmente feitos como solicitações POST .

A variável de nome local é usada para armazenar o nome recebido do formulário quando disponível; quando o nome não é conhecido, a variável é inicializada como None. A função `view` cria uma instância da classe

`NameForm` mostrada anteriormente para representar o formulário. O método `validate_on_submit()` do formulário retorna True quando o formulário foi enviado e os dados foram aceitos por todos os validadores de campo. Em todos os outros casos, `validate_on_submit()` retorna False. O valor de retorno desse método serve efetivamente para decidir se o formulário precisa ser renderizado ou processado.

Quando um usuário navega para o aplicativo pela primeira vez, o servidor receberá uma solicitação GET sem dados de formulário, portanto, `validate_on_submit()` retornará False. O corpo da instrução if será ignorado e a requisição será tratada renderizando o template, que obtém o objeto de formulário e a variável name definida como None como argumentos. Os usuários agora verão o formulário exibido no navegador.

Quando o formulário é enviado pelo usuário, o servidor recebe uma solicitação POST com os dados. A chamada para `validate_on_submit()` invoca o validador `Required()` anexo ao campo name. Se o nome não estiver vazio, o validador o aceitará e `validate_on_submit()` retornará True. Agora o nome inserido pelo usuário está acessível como atributo de dados do campo. Dentro do corpo da instrução if , esse nome é atribuído à variável de nome local e o campo de formulário é limpo definindo esse atributo de dados como uma string vazia. A chamada `render_template()` na última linha renderiza o template, mas desta vez o argumento name contém o nome do formulário, então a saudação será personalizada.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar `git checkout 4a` para verificar esta versão do aplicativo.

A [Figura 4-1](#) mostra a aparência do formulário na janela do navegador quando um usuário entra inicialmente no site. Quando o usuário envia um nome, o aplicativo responde com um

saudações. O formulário ainda aparece abaixo dele, para que o usuário possa enviá-lo com um novo nome, se desejar. A [Figura 4-2](#) mostra a aplicação neste estado.

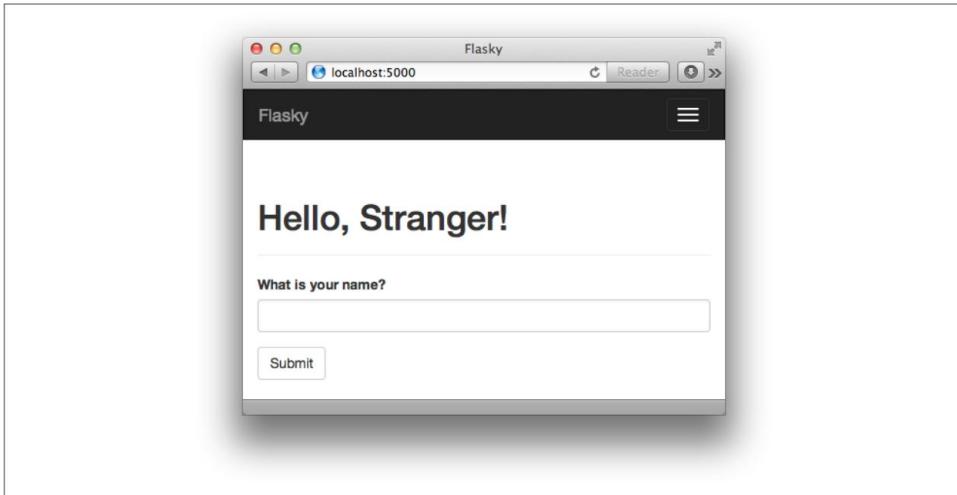


Figura 4-1. Formulário web Flask-WTF

Se o usuário enviar o formulário com um nome vazio, o validador Required() detecta o erro, conforme mostrado na [Figura 4-3](#). Observe quanta funcionalidade está sendo fornecida automaticamente. Este é um ótimo exemplo do poder que extensões bem projetadas como Flask-WTF e Flask-Bootstrap podem dar ao seu aplicativo.

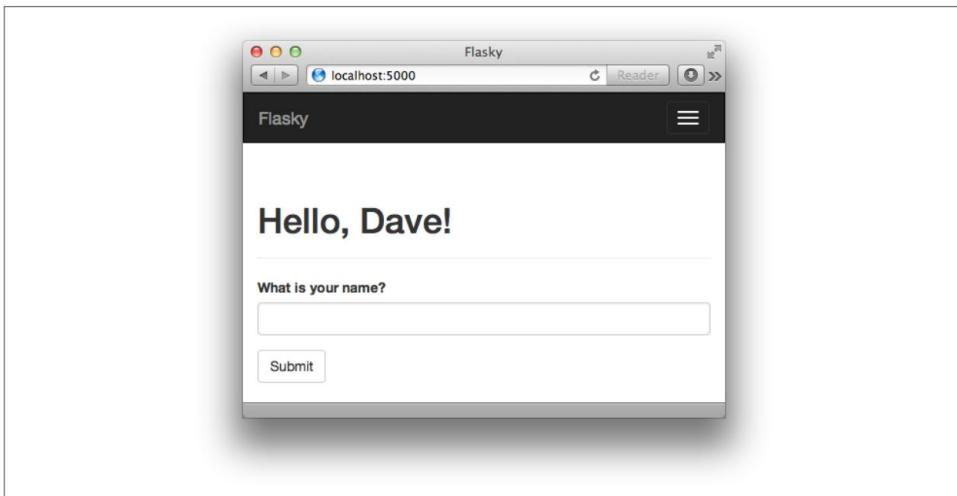


Figura 4-2. Formulário da Web após o envio

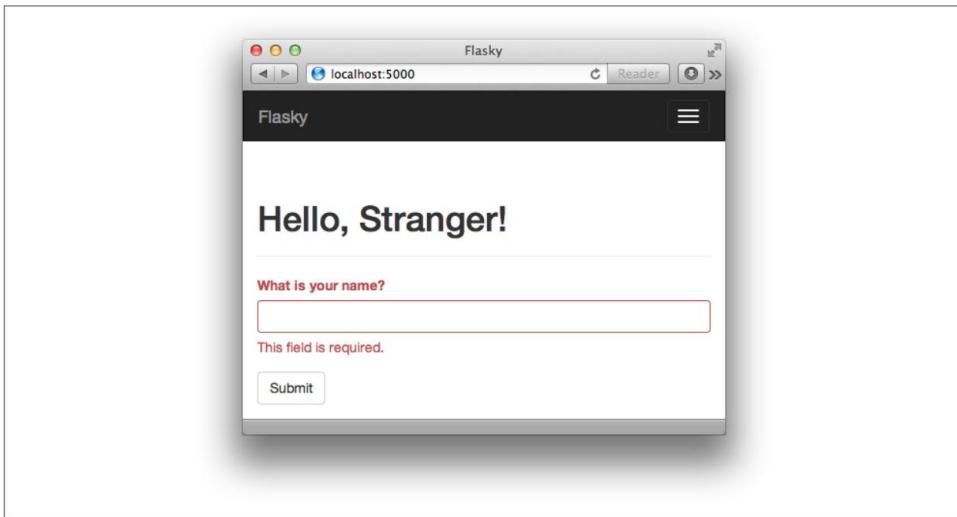


Figura 4-3. Formulário da Web após falha do validador

### Redirecionamentos e sessões de usuário

A última versão do *hello.py* tem um problema de usabilidade. Se você digitar seu nome e enviá-lo e clicar no botão atualizar no seu navegador, provavelmente receberá um aviso obscuro que solicita confirmação antes de enviar o formulário novamente. Isso acontece porque os navegadores repetem a última solicitação que enviaram quando são solicitados a atualizar a página. Quando a última solicitação enviada for uma solicitação POST com dados de formulário, uma atualização causaria um envio de formulário duplicado, o que em quase todos os casos não é a ação desejada.

Muitos usuários não entendem o aviso do navegador. Por esse motivo, é considerado uma boa prática para aplicativos da Web nunca deixar uma solicitação POST como a última solicitação enviada pelo navegador.

Essa prática pode ser alcançada respondendo a solicitações POST com um *redirecionamento* em vez de uma resposta normal. Um redirecionamento é um tipo especial de resposta que possui uma URL em vez de uma string com código HTML. Quando o navegador recebe essa resposta, ele emite uma solicitação GET para a URL de redirecionamento, e essa é a página exibida. A página pode demorar mais alguns milissegundos para carregar por causa da segunda solicitação que deve ser enviada ao servidor, mas fora isso, o usuário não verá nenhuma diferença. Agora, a última solicitação é um GET, portanto, o comando de atualização funciona conforme o esperado. Esse truque é conhecido como *padrão Post/Redirect/Get*.

Mas essa abordagem traz um segundo problema. Quando a aplicação trata da solicitação POST, ela tem acesso ao nome inserido pelo usuário em `form.name.data`, mas assim que essa solicitação termina, os dados do formulário são perdidos. Como a solicitação POST é tratada com um

redirecionamento, o aplicativo precisa armazenar o nome para que a solicitação redirecionada possa tê-lo usado para criar a resposta real.

Os aplicativos podem “lembra” coisas de uma solicitação para outra armazenando-as na *sessão do usuário*, armazenamento privado que está disponível para cada cliente conectado. A sessão do usuário foi apresentada no [Capítulo 2](#) como uma das variáveis associadas ao contexto da solicitação.

É chamado de sessão e é acessado como um dicionário Python padrão.



Por padrão, as sessões do usuário são armazenadas em cookies do lado do cliente que são assinados criptograficamente usando a SECRET\_KEY configurada. Qualquer adulteração do conteúdo do cookie tornaria a assinatura inválida, invalidando assim a sessão.

O [Exemplo 4-5](#) mostra uma nova versão da função view index() que implementa redirecionamentos e sessões de usuário.

*Exemplo 4-5. hello.py: redirecionamentos e sessões de usuário*

```
from flask import Flask, render_template, session, redirect, url_for

@app.route('/', métodos=['GET', 'POST']) def
index(): form = NameForm() if
form.validate_on_submit(): session['name'] =
form.name.data return redirect(url_for('index'))
return render_template('index.html',
form=form, name=session.get('name'))
```

Na versão anterior do aplicativo, uma variável de nome local era utilizada para armazenar o nome inserido pelo usuário no formulário. Essa variável agora é colocada na sessão do usuário como session['name'] para que seja lembrada além da solicitação.

As solicitações que vêm com dados de formulário válidos agora terminarão com uma chamada para redirect(), uma função auxiliar que gera a resposta de redirecionamento HTTP. A função redirect() usa a URL para redirecionar como um argumento. A URL de redirecionamento usada neste caso é a URL raiz, então a resposta poderia ter sido escrita de forma mais concisa como redirect('/'), mas em vez disso a função geradora de URL do Flask url\_for() é usada. O uso de url\_for() para gerar URLs é encorajado porque esta função gera URLs usando o mapa de URLs, então as URLs são compatíveis com as rotas definidas e quaisquer alterações feitas nos nomes das rotas estarão automaticamente disponíveis ao usar esta função.

O primeiro e único argumento obrigatório para url\_for() é o nome do *terminal*, o nome interno que cada rota possui. Por padrão, o ponto final de uma rota é o nome da função de visualização anexada a ela. Neste exemplo, a função de visualização que manipula a URL raiz é index(), portanto, o nome dado a url\_for() é index.

A última mudança está na função `render_template()`, que agora obtém o argumento `name` diretamente da sessão usando `session.get('name')`. Assim como acontece com os dicionários regulares, usar `get()` para solicitar uma chave de dicionário evita uma exceção para chaves que não são encontradas, porque `get()` retorna um valor padrão de `None` para uma chave ausente.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar `git checkout 4b` para verificar esta versão do aplicativo.

Com esta versão do aplicativo, você pode ver que atualizar a página em seu navegador resulta no comportamento esperado.

## Mensagem piscando

Às vezes, é útil fornecer ao usuário uma atualização de status após a conclusão de uma solicitação. Pode ser uma mensagem de confirmação, um aviso ou um erro. Um exemplo típico é quando você envia um formulário de login para um site com um erro e o servidor responde renderizando o formulário de login novamente com uma mensagem acima informando que seu nome de usuário ou senha são inválidos.

Flask inclui essa funcionalidade como um recurso principal. O [Exemplo 4-6](#) mostra como a função `flash()` pode ser usada para esse propósito.

*Exemplo 4-6. hello.py: mensagens em flash*

```
from flask import Flask, render_template, session, redirect, url_for, flash

@app.route('/', métodos=['GET', 'POST']) def
index(): form = NameForm() if
form.validate_on_submit():

    old_name = session.get('name') se
    old_name não for None e old_name != form.name.data: flash('Parece
        que você mudou seu nome!') session['name'] = form.name.data
    form.name.data = return redirect(url_for('index'))

return render_template('index.html',
    formulário = formulário, nome = sessão.get('nome'))
```

Neste exemplo, cada vez que um nome é enviado, ele é comparado com o nome armazenado na sessão do usuário, que teria sido colocado lá durante um envio anterior do mesmo formulário. Se os dois nomes forem diferentes, a função `flash()` é invocada com uma mensagem a ser exibida na próxima resposta enviada de volta ao cliente.

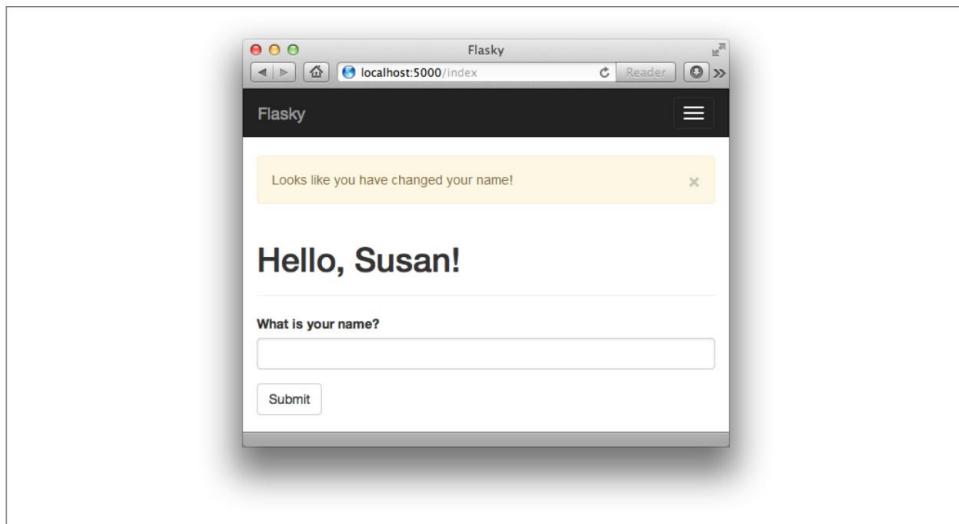
Chamar `flash()` não é suficiente para exibir as mensagens; os modelos usados pelo aplicativo precisam renderizar essas mensagens. O melhor lugar para renderizar mensagens em flash é o template base, pois isso habilitará essas mensagens em todas as páginas. O Flask disponibiliza uma função `get_flashed_messages()` para templates para recuperar as mensagens e renderizá-las, conforme mostrado no [Exemplo 4-7](#).

*Exemplo 4-7. templates/base.html: renderização de mensagens Flash*

```
{% block content %}
<div class="container">
    {% para mensagem em get_flashed_messages() %}
    <div class="alert alert-warning">
        <button type="button" class="close" data-dismiss="alert">&times;</button> {{ message }}
    </div> {% endfor %}

    {% block page_content %}{% endblock %} <
    div> {% endblock %}
```

Neste exemplo, as mensagens são renderizadas usando os estilos CSS de alerta do Bootstrap para mensagens de aviso (um é mostrado na [Figura 4-4](#)).



*Figura 4-4. Mensagem exibida*

Um loop é usado porque pode haver várias mensagens enfileiradas para exibição, uma para cada vez que `flash()` foi chamado no ciclo de solicitação anterior. Mensagens recuperadas

from get\_flashed\_messages() não será retornado na próxima vez que esta função for chamada, então as mensagens em flash aparecem apenas uma vez e são então descartadas.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar git checkout 4c para verificar esta versão do aplicativo.

Ser capaz de aceitar dados do usuário por meio de formulários da web é um recurso exigido pela maioria dos aplicativos, assim como a capacidade de armazenar esses dados em armazenamento permanente. O uso de bancos de dados com o Flask é o tópico do próximo capítulo.

## CAPÍTULO 5

### Bancos de dados

Um *banco de dados* armazena os dados do aplicativo de forma organizada. O aplicativo então emite *consultas* para recuperar partes específicas conforme necessário. Os bancos de dados mais utilizados para aplicações web são aqueles baseados no modelo *relacional*, também chamados de bancos de dados SQL em referência à Linguagem de Consulta Estruturada que utilizam. Mas, nos últimos anos, bancos de dados *orientados a documentos e valores-chave*, informalmente conhecidos como bancos de dados NoSQL, tornaram-se alternativas populares.

#### Bancos de dados SQL

Bancos de dados relacionais armazenam dados em *tabelas*, que modelam as diferentes entidades no domínio da aplicação. Por exemplo, um banco de dados para um aplicativo de gerenciamento de pedidos provavelmente terá tabelas de clientes, produtos e pedidos .

Uma tabela tem um número fixo de *colunas* e um número variável de *linhas*. As colunas definem os atributos de dados da entidade representada pela tabela. Por exemplo, uma tabela de clientes terá colunas como nome, endereço, telefone e assim por diante. Cada linha em uma tabela define um elemento de dados real que consiste em valores para todas as colunas.

As tabelas têm uma coluna especial chamada *chave primária*, que contém um identificador exclusivo para cada linha armazenada na tabela. As tabelas também podem ter colunas chamadas *de chaves estrangeiras*, que fazem referência à chave primária de outra linha da mesma ou de outra tabela. Esses links entre linhas são chamados de *relacionamentos* e são a base do modelo de banco de dados relacional.

A **Figura 5-1** mostra um diagrama de um banco de dados simples com duas tabelas que armazenam usuários e funções de usuário. A linha que conecta as duas tabelas representa uma relação entre as tabelas.

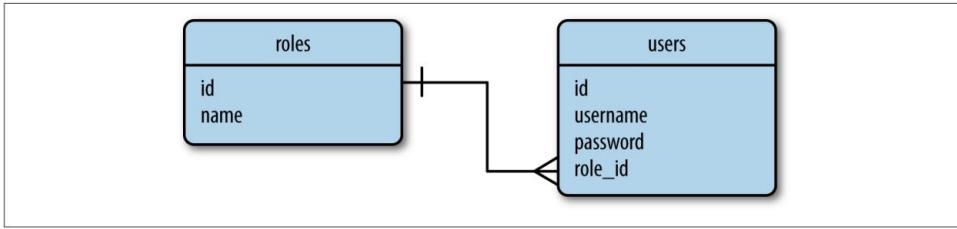


Figura 5-1. Exemplo de banco de dados relacional

Nesse diagrama de banco de dados, a tabela de funções armazena a lista de todas as funções de usuário possíveis, cada uma identificada por um valor de id exclusivo — a chave primária da tabela. A tabela de usuários contém a lista de usuários, cada um com seu próprio ID exclusivo . Além das chaves primárias id , a tabela de funções tem uma coluna de nome e a tabela de usuários tem colunas de nome de usuário e senha . A coluna role\_id na tabela users é uma chave estrangeira que faz referência ao id de uma função e, dessa forma, é estabelecida a função atribuída a cada usuário.

Conforme visto no exemplo, os bancos de dados relacionais armazenam dados de forma eficiente e evitam duplicação. Renomear uma função de usuário neste banco de dados é simples porque os nomes das funções existem em um único local. Imediatamente após a alteração de um nome de função na tabela de funções , todos os usuários que tiverem um role\_id que faça referência à função alterada verão a atualização.

Por outro lado, ter os dados divididos em várias tabelas pode ser uma complicação. Produzir uma lista de usuários com suas funções apresenta um pequeno problema, porque usuários e funções de usuário precisam ser lidos de duas tabelas e *unidos* antes que possam ser apresentados juntos. Os mecanismos de banco de dados relacionais fornecem suporte para realizar operações de junção entre tabelas quando necessário.

## Bancos de dados NoSQL

Os bancos de dados que não seguem o modelo relacional descrito na seção anterior são chamados coletivamente de bancos de dados *NoSQL* . Uma organização comum para bancos de dados NoSQL usa *coleções* em vez de tabelas e *documentos* em vez de registros. Os bancos de dados NoSQL são projetados de forma a dificultar as junções, portanto, a maioria deles não oferece suporte a essa operação. Para um banco de dados NoSQL estruturado como na Figura 5-1, listar os usuários com suas funções exige que o próprio aplicativo execute a operação de junção lendo o campo role\_id de cada usuário e, em seguida, pesquisando a tabela de funções por ele.

Um projeto mais apropriado para um banco de dados NoSQL é mostrado na Figura 5-2. Este é o resultado da aplicação de uma operação chamada *desnormalização*, que reduz o número de tabelas em detrimento da duplicação de dados.

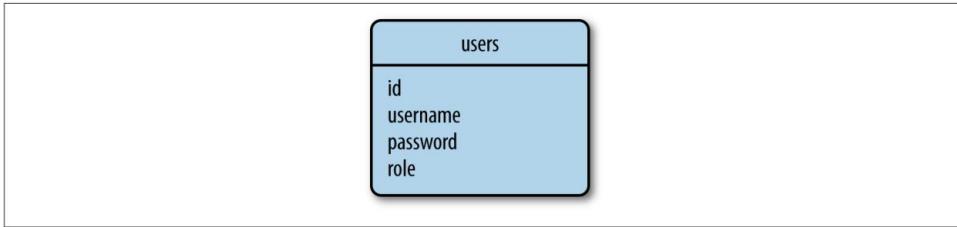


Figura 5-2. Exemplo de banco de dados NoSQL

Um banco de dados com essa estrutura tem o nome da função explicitamente armazenado com cada usuário.

Renomear uma função pode se tornar uma operação cara que pode exigir a atualização de um grande número de documentos.

Mas nem tudo são más notícias com bancos de dados NoSQL. Ter os dados duplicados permite consultas mais rápidas. Listar usuários e suas funções é simples porque não são necessárias junções.

## SQL ou NoSQL?

Os bancos de dados SQL são excelentes no armazenamento de dados estruturados de forma eficiente e compacta.

Esses bancos de dados fazem um grande esforço para preservar a consistência. Os bancos de dados NoSQL relaxam alguns dos requisitos de consistência e, como resultado, às vezes podem obter uma vantagem de desempenho.

Uma análise completa e comparação de tipos de banco de dados está fora do escopo deste livro. Para aplicativos de pequeno e médio porte, os bancos de dados SQL e NoSQL são perfeitamente capazes e têm desempenho praticamente equivalente.

## Estruturas de banco de dados Python

O Python possui pacotes para a maioria dos mecanismos de banco de dados, tanto de código aberto quanto comercial.

O Flask não impõe restrições sobre quais pacotes de banco de dados podem ser usados, portanto, você pode trabalhar com MySQL, Postgres, SQLite, Redis, MongoDB ou CouchDB, se algum deles for o seu favorito.

Como se essas não fossem opções suficientes, também existem vários pacotes de camada de abstração de banco de dados, como SQLAlchemy ou MongoEngine, que permitem trabalhar em um nível mais alto com objetos Python regulares em vez de entidades de banco de dados, como tabelas, documentos ou linguagens de consulta .

Há vários fatores a serem avaliados ao escolher uma estrutura de banco de dados: *Facilidade de uso* Ao comparar mecanismos de banco de dados simples versus camadas de abstração de banco de dados, o segundo grupo claramente vence. Camadas de abstração, também chamadas de mapeadores relacionais de objeto (ORMs) ou mapeadores de documento de objeto (ODMs), fornecem conversão transparente de operações orientadas a objetos de alto nível em instruções de banco de dados de baixo nível.

### *Desempenho As*

conversões que os ORMs e ODMs precisam fazer para traduzir do domínio do objeto para o domínio do banco de dados têm uma sobrecarga. Na maioria dos casos, a penalidade de desempenho é insignificante, mas nem sempre pode ser. Em geral, o ganho de produtividade obtido com ORMs e ODMs supera em muito uma degradação mínima de desempenho, então este não é um argumento válido para descartar completamente ORMs e ODMs. O que faz sentido é escolher uma camada de abstração de banco de dados que forneça acesso opcional ao banco de dados subjacente no caso de operações específicas precisarem ser otimizadas implementando-as diretamente como instruções de banco de dados nativas.

### *Portabilidade*

As opções de banco de dados disponíveis em suas plataformas de desenvolvimento e produção devem ser consideradas. Por exemplo, se você planeja hospedar seu aplicativo em uma plataforma de nuvem, deve descobrir quais opções de banco de dados esse serviço oferece.

Outro aspecto de portabilidade se aplica a ORMs e ODMs. Embora alguns desses frameworks forneçam uma camada de abstração para um único mecanismo de banco de dados, outros abstraem ainda mais e fornecem uma escolha de mecanismos de banco de dados – todos acessíveis com a mesma interface orientada a objetos. O melhor exemplo disso é o SQLAlchemy ORM, que suporta uma lista de mecanismos de banco de dados relacionais, incluindo os populares MySQL, Postgres e SQLite.

### *Integração do Flask*

Escolher um framework que tenha integração com o Flask não é absolutamente necessário, mas evitá-la que você tenha que escrever o código de integração por conta própria. A integração do Flask pode simplificar a configuração e a operação, portanto, deve-se preferir usar um pacote projetado especificamente como uma extensão do Flask.

Com base nesses objetivos, o framework de banco de dados escolhido para os exemplos deste livro será [Flask-SQLAlchemy](#), o wrapper de extensão Flask para [SQLAlchemy](#).

## **Gerenciamento de banco de dados com Flask-SQLAlchemy**

Flask-SQLAlchemy é uma extensão Flask que simplifica o uso de SQLAlchemy dentro de aplicativos Flask. SQLAlchemy é uma poderosa estrutura de banco de dados relacional que suporta vários backends de banco de dados. Ele oferece um ORM de alto nível e acesso de baixo nível à funcionalidade SQL nativa do banco de dados.

Como a maioria das outras extensões, Flask-SQLAlchemy é instalado com pip:

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

No Flask-SQLAlchemy, um banco de dados é especificado como uma URL. A [Tabela 5-1](#) lista o formato de URLs de banco de dados para os três mecanismos de banco de dados mais populares.

*Tabela 5-1. URLs de banco de dados Flask-SQLAlchemy*

URL do mecanismo de banco de dados	
MySQL	mysql://username:password@hostname/database
Postgres	postgresql://username:password@hostname/
SQLite (Unix)	databasesqlite:///absolute/path/to/database
SQLite (Windows)	sqlite:///c:/absolute/path/to/database

Nessas URLs, *hostname* refere-se ao servidor que hospeda o serviço MySQL, que pode ser *localhost* ou um servidor remoto. Servidores de banco de dados podem hospedar vários bancos de dados, então banco de dados indica o nome do banco de dados a ser usado. Para bancos de dados que precisam de autenticação, nome de usuário e senha são as credenciais do usuário do banco de dados.



Os bancos de dados SQLite não têm um servidor, portanto, o nome do host, o nome de usuário e a senha são omitidos e o banco de dados é o nome do arquivo em disco.

A URL do banco de dados do aplicativo deve ser configurada como a chave SQLALCHEMY\_DATABASE\_URI no objeto de configuração do Flask. Outra opção útil é a chave de configuração SQLALCHEMY\_COMMIT\_ON\_TEARDOWN, que pode ser definida como True para habilitar confirmações automáticas de alterações no banco de dados ao final de cada solicitação. Consulte a documentação do Flask-SQLAlchemy para obter informações sobre outras opções de configuração.

O Exemplo 5-1 mostra como inicializar e configurar um banco de dados SQLite simples.

*Exemplo 5-1. hello.py: configuração do banco de dados*

```
de flask.ext.sqlalchemy importar SQLAlchemy

basedir = os.path.abspath (os.path.dirname (__ arquivo__))

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] =\ 'sqlite:///
+ os.path.join(basedir, 'data.sqlite')
app.config['SQLALCHEMY_COMMIT_ON_TEARDOWN'] = Verdadeiro

db = SQLAlchemy(aplicativo)
```

O objeto db instanciado da classe SQLAlchemy representa o banco de dados e fornece acesso a todas as funcionalidades do Flask-SQLAlchemy.

## Definição do modelo

O termo *modelo* é usado para se referir às entidades persistentes usadas pelo aplicativo. No contexto de um ORM, um modelo é tipicamente uma classe Python com atributos que correspondem ao colunas de uma tabela de banco de dados correspondente.

A instância de banco de dados do Flask-SQLAlchemy também fornece uma classe base para modelos como um conjunto de classes e funções auxiliares que são usadas para definir sua estrutura. Os papéis e as tabelas de usuários da [Figura 5-1](#) podem ser definidas como modelos de Função e Usuário , conforme mostrado em [Exemplo 5-2](#).

*Exemplo 5-2. hello.py: definição de modelo de função e usuário*

```
class Role(db.Model):
    __tablename__ = 'funções'
    id = db.Column(db.Integer, primary_key=True)
    nome = db.Column(db.String(64), unique=True)

    def __repr__(self):
        return '<Função %r>' % self.name

class Usuário(db.Model):
    __tablename__ = 'usuários'
    id = db.Column(db.Integer, primary_key=True)
    nome de usuário = db.Column(db.String(64), unique=True, index=True)

    def __repr__(self):
        return '<Usuário %r>' % self.username
```

A variável de classe `__tablename__` define o nome da tabela no banco de dados. Flask SQLAlchemy atribui um nome de tabela padrão se `__tablename__` for omitido, mas esses nomes não seguem a convenção de usar plurais para nomes de tabelas, então é melhor tabelas de nomes explicitamente. As variáveis de classe restantes são os atributos do modelo, definidos como instâncias da classe `db.Column` .

O primeiro argumento dado ao construtor `db.Column` é o tipo da coluna do banco de dados e atributo de modelo. A [Tabela 5-2](#) lista alguns dos tipos de coluna disponíveis, juntamente com o tipo Python usado no modelo.

*Tabela 5-2. Tipos de coluna SQLAlchemy mais comuns*

Nome do tipo	Tipo de Python	Descrição
inteiro	<code>int</code>	Inteiro regular, normalmente 32 bits
<code>SmallInteger</code>	<code>int</code>	Inteiro de curto alcance, normalmente 16 bits
<code>BigInteger</code>	inteiro ou longo	Número inteiro de precisão ilimitada
<code>Floating</code>	<code>flutuador</code>	Número de ponto flutuante
Numérico	<code>decimal.Decimal</code>	Número de ponto fixo

Nome do tipo	Tipo de Python	Descrição
Fragmento	<code>str</code>	Cadeia de comprimento variável
Texto	<code>str</code>	Cadeia de comprimento variável, otimizada para comprimento grande ou não vinculado
Unicode	código único	Cadeia Unicode de comprimento variável
UnicodeText	<code>unicode</code>	Cadeia Unicode de comprimento variável, otimizada para comprimento grande ou não vinculado
booleano	<code>bool</code>	Valor booleano
Encontro	<code>datahora.data</code>	Valor da data
Tempo	<code>datahora.hora</code>	Valor do tempo
	<code>DateTime</code> <code>datetime.datetime</code>	Valor de data e hora
Intervalo	<code>datetime.timedelta</code>	Intervalo de tempo
Enum	<code>Str</code>	Lista de valores de string
PickleType	Qualquer objeto Python	Serialização automática do Pickle
LargeBinary	<code>str</code>	Blob binário

Os argumentos restantes para db.Column especificam opções de configuração para cada atributo. A [Tabela 5-3](#) lista algumas das opções disponíveis.

*Tabela 5-3. Opções de coluna SQLAlchemy mais comuns*

Nome da opção	Descrição
<code>primary_key</code>	Se definido como True, a coluna é a chave primária da tabela.
<code>exclusivo</code>	Se definido como Verdadeiro, não permite valores duplicados para esta coluna.
<code>índice</code>	Se definido como True, crie um índice para esta coluna, para que as consultas sejam mais eficientes.
<code>anulável</code>	Se definido como True, permite valores vazios para esta coluna. Se definido como False, a coluna não permitirá null valores.
<code>predefinição</code>	Defina um valor padrão para a coluna.



Flask-SQLAlchemy requer que todos os modelos definam uma chave *primária* (`primary_key`), que normalmente é chamado de `id`.

Embora não seja estritamente necessário, os dois modelos incluem um método `__repr__()` para dê a eles uma representação de string legível que pode ser usada para depuração e teste propósitos.

## Relacionamentos

Bancos de dados relacionais estabelecem conexões entre linhas em diferentes tabelas por meio do uso de relacionamentos. O diagrama relacional na [Figura 5-1](#) expressa um relacionamento simples entre usuários e suas funções. Esta é uma relação *um-para-muitos* de funções para usuários, porque uma função pertence a muitos usuários e os usuários têm apenas uma função.

O [Exemplo 5-3](#) mostra como o relacionamento um-para-muitos na [Figura 5-1](#) é representado nas classes de modelo.

*Exemplo 5-3. hello.py: relacionamentos*

```
class Role(db.Model): #
    ...
    usuários = db.relationship('Usuário', backref='função')

class User(db.Model): #
    role_id =
        db.Column(db.Integer, db.ForeignKey('roles.id'))
```

Conforme visto na [Figura 5-1](#), um relacionamento conecta duas linhas por meio do usuário de uma chave estrangeira. A coluna `role_id` adicionada ao modelo `User` é definida como uma chave estrangeira e isso estabelece o relacionamento. O argumento '`roles.id`' para `db.ForeignKey()` especifica que a coluna deve ser interpretada como tendo valores de id de linhas na tabela `roles`.

O atributo `users` adicionado ao model `Role` representa a visão orientada a objetos do relacionamento. Dada uma instância da classe `Role`, o atributo `users` retornará a lista de usuários associados a essa função. O primeiro argumento para `db.relationship()` indica qual modelo está do outro lado do relacionamento. Este modelo pode ser fornecido como uma string se a classe ainda não estiver definida.

O argumento `backref` para `db.relationship()` define a direção inversa do relacionamento adicionando um atributo `role` ao modelo `User`. Esse atributo pode ser usado em vez de `role_id` para acessar o modelo de função como um objeto em vez de uma chave estrangeira.

Na maioria dos casos, `db.relationship()` pode localizar a chave estrangeira do relacionamento por conta própria, mas às vezes não pode determinar qual coluna usar como chave estrangeira. Por exemplo, se o modelo `User` tivesse duas ou mais colunas definidas como chaves estrangeiras de função, SQLAlchemy não saberia qual das duas usar. Sempre que a configuração da chave estrangeira for ambígua, argumentos adicionais para `db.relationship()` precisam ser fornecidos. A [Tabela 5-4](#) lista algumas das opções de configuração comuns que podem ser usadas para definir um relacionamento.

*Tabela 5-4. Opções comuns de relacionamento SQLAlchemy*

Nome da opção	Descrição
backref	Adicione uma referência inversa no outro modelo no relacionamento.
primaryjoin	Especifique explicitamente a condição de junção entre os dois modelos. Isso é necessário apenas para relacionamentos ambíguos.
preguiçoso	Especifique como os itens relacionados devem ser carregados. Os valores possíveis são select (os itens são carregados sob demanda na primeira vez que são acessados), imediato (os itens são carregados quando o objeto de origem é carregado), unidos (os itens são carregados imediatamente, mas como uma junção), subconsulta (os itens são carregados imediatamente , mas como uma subconsulta), noload (os itens nunca são carregados) e dinâmico (em vez de carregar os itens, a consulta que pode carregá-los é fornecida).
Lista de usuários	Se definido como False, use um escalar em vez de uma lista.
ordenar por	Especifique a ordenação usada para os itens no relacionamento.
secundário	Especifique o nome da tabela de associação a ser usada em relacionamentos muitos para muitos.
secondaryjoin	Especifique a condição de junção secundária para relacionamentos muitos-para-muitos quando SQLAlchemy não puder determiná-lo por conta própria.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar git checkout 5a para verificar esta versão do aplicativo.

Existem outros tipos de relacionamento além do um-para-muitos. O relacionamento *um-para-um* pode ser expresso como o um-para-muitos descrito anteriormente, mas com a opção uselist definida como False dentro da definição db.relationship() para que o lado “muitos” se torne um lado “um”. O relacionamento *muitos-para-um* também pode ser expresso como um-para-muitos se as tabelas forem invertidas, ou pode ser expresso com a chave estrangeira e a definição db.relationship() ambas no lado “muitos”. O tipo de relacionamento mais complexo, o *muitos-para-muitos*, requer uma tabela adicional chamada *tabela de associação*. Você aprenderá sobre relacionamentos muitos-para-muitos no [Capítulo 12](#).

## Operações de banco de dados

Os modelos agora estão totalmente configurados de acordo com o diagrama de banco de dados da [Figura 5-1](#) e prontos para serem usados. A melhor maneira de aprender a trabalhar com esses modelos é em um shell Python. As seções a seguir o guiarão pelas operações de banco de dados mais comuns.

## Criando as Tabelas

A primeira coisa a fazer é instruir o Flask-SQLAlchemy a criar um banco de dados baseado nas classes de modelo. A função `db.create_all()` faz isso:

```
(venv) $ python hello.py shell >>>
from hello import db >>> db.create_all()
```

Se você verificar o diretório da aplicação, verá agora um novo arquivo chamado `data.sqlite`, o nome que foi dado ao banco de dados SQLite na configuração. A função `db.create_all()` não recriará ou atualizará uma tabela de banco de dados se ela já existir no banco de dados. Isso pode ser inconveniente quando os modelos são modificados e o

as alterações precisam ser aplicadas a um banco de dados existente. A solução de força bruta para atualizar as tabelas de banco de dados existentes é remover primeiro as tabelas antigas:

```
>>> db.drop_all()
>>> db.create_all()
```

Infelizmente, esse método tem o efeito colateral indesejado de destruir todos os dados do banco de dados antigo. Uma solução melhor para o problema de atualização de bancos de dados é apresentada perto do final do capítulo.

## Inserindo Linhas

O exemplo a seguir cria algumas funções e usuários:

```
>>> from hello import Role, User >>>
admin_role = Role(name='Admin') >>>
mod_role = Role(name='Moderator') >>>
user_role = Role(name='User') >>> user_john =
= User(username='john', role=admin_role) >>> user_susan =
User(username='susan', role=user_role) >>> user_david =
User(username='david', role=user_role)
```

Os construtores de modelos aceitam valores iniciais para os atributos de modelo como argumentos de palavra-chave. Observe que até mesmo o atributo `role` pode ser usado, mesmo que não seja uma coluna de banco de dados real, mas uma representação de alto nível do relacionamento um-para-muitos. O atributo `id` desses novos objetos não é definido explicitamente: as chaves primárias são gerenciadas pelo Flask-SQLAlchemy. Os objetos existem apenas no lado do Python até agora; eles ainda não foram gravados no banco de dados. Por causa disso, seu valor de `id` ainda não foi atribuído:

```
>>> print(admin_role.id)
Nenhum
>>> print(mod_role.id)
Nenhum >>> print(user_role.id)
Nenhum
```

As alterações no banco de dados são gerenciadas por meio de uma sessão *de banco de dados*, que o Flask SQLAlchemy fornece como db.session. Para preparar objetos para serem gravados no banco de dados, eles devem ser adicionados à sessão:

```
>>> db.session.add(admin_role)
>>> db.session.add(mod_role) >>>
db.session.add(user_role) >>>
db.session.add(user_john) >>>
db.session .add(user_susan) >>>
db.session.add(user_david)
```

Ou, mais concisamente:

```
>>> db.session.add_all([admin_role, mod_role, user_role, user_john,
...                     user_susan, user_david])
```

Para gravar os objetos no banco de dados, a sessão precisa ser confirmada chamando seu método *commit()*:

```
>>> db.session.commit()
```

Verifique os atributos de id novamente; agora estão definidos:

```
>>> print(admin_role.id) 1
>>> print(mod_role.id)
2
>>> print(user_role.id) 3
```



A sessão de banco de dados db.session não está relacionada ao objeto de sessão Flask discutido no [Capítulo 4](#). As sessões de banco de dados também são chamadas de *transações*.

As sessões de banco de dados são extremamente úteis para manter o banco de dados consistente. A operação de confirmação grava todos os objetos que foram adicionados à sessão atomicamente. Se ocorrer um erro enquanto a sessão estiver sendo gravada, toda a sessão será descartada. Se você sempre confirmar as alterações relacionadas em uma sessão, é garantido que você evitará inconsistências no banco de dados devido a atualizações parciais.



Uma sessão de banco de dados também pode ser *revertida*. Se db.session.rollback() for chamado, todos os objetos que foram adicionados à sessão do banco de dados serão restaurados para o estado em que estão no banco de dados.

## Modificando linhas

O método `add()` da sessão do banco de dados também pode ser usado para atualizar modelos. Continuando na mesma sessão de shell, o exemplo a seguir renomeia a função "Admin" para "Administrator":

```
>>> admin_role.name = 'Administrator' >>>
db.session.add(admin_role) >>>
db.session.commit()
```

## Excluindo linhas

A sessão do banco de dados também possui um método `delete()`. O exemplo a seguir exclui a função "Moderador" do banco de dados:

```
>>> db.session.delete(mod_role) >>>
db.session.commit()
```

Observe que exclusões, como inserções e atualizações, são executadas somente quando a sessão do banco de dados é confirmada.

## Consultando linhas

Flask-SQLAlchemy disponibiliza um objeto de consulta em cada classe de modelo. A consulta mais básica para um modelo é aquela que retorna todo o conteúdo da tabela correspondente:

```
>>> Role.query.all()
[<Função u'Administrador'>, <Função u'Usuário'>]
>>> User.query.all()
[<Usuário u'john'>, <Usuário u'susan'>, <Usuário u'david'>]
```

Um objeto de consulta pode ser configurado para emitir pesquisas de banco de dados mais específicas por meio do uso de *filtros*. O exemplo a seguir encontra todos os usuários que receberam a função "Usuário" :

```
>>> User.query.filter_by(role=user_role).all()
[<Usuário u'susan'>, <Usuário u'david'>]
```

Também é possível inspecionar a consulta SQL nativa que SQLAlchemy gera para uma determinada consulta convertendo o objeto de consulta em uma string:

```
>>> str(User.query.filter_by(role=user_role))
'SELECT users.id AS users_id, users.username AS users_username,
users.role_id AS users_role_id FROM users WHERE :param_1 = users.role_id'
```

Se você sair da sessão do shell, os objetos criados no exemplo anterior deixarão de existir como objetos Python, mas continuarão a existir como linhas em suas respectivas tabelas de banco de dados. Se você iniciar uma nova sessão de shell, precisará recriar objetos Python de suas linhas de banco de dados. O exemplo a seguir emite uma consulta que carrega a função de usuário com o nome "Usuário":

```
>>> user_role = Role.query.filter_by(name='User').first()
```

Filtros como filter\_by() são invocados em um objeto de consulta e retornam um novo inquerir. Vários filtros podem ser chamados em sequência até que a consulta seja configurada conforme necessário.

A [Tabela 5-5](#) mostra alguns dos filtros mais comuns disponíveis para consultas. O completo list está na [documentação do SQLAlchemy](#).

*Tabela 5-5. Filtros comuns de consulta SQLAlchemy*

Opção	Descrição
filter() filter_by()	Retorna uma nova consulta que adiciona um filtro adicional à consulta original
Retorna uma nova consulta que adiciona um filtro de igualdade adicional à consulta original	
limit() offset()	Retorna uma nova consulta que limita o número de resultados da consulta original ao número fornecido
order_by()	Retorna uma nova consulta que aplica um deslocamento na lista de resultados da consulta original
Retorna uma nova consulta que classifica os resultados da consulta original de acordo com os critérios fornecidos	
group_by()	Retorna uma nova consulta que agrupa os resultados da consulta original de acordo com os critérios fornecidos

Depois que os filtros desejados forem aplicados à consulta, uma chamada para all() fará com que o consulta para executar e retornar os resultados como uma lista, mas existem outras maneiras de acionar a execução de uma consulta além de all(). A [Tabela 5-6](#) mostra outros métodos de execução de consultas.

*Tabela 5-6. Executores de consulta SQLAlchemy mais comuns*

Opção	Descrição
tudo()	Retorna todos os resultados de uma consulta como uma lista
primeiro()	Retorna o primeiro resultado de uma consulta ou Nenhum se não houver resultados
first_or_404()	Retorna o primeiro resultado de uma consulta ou aborta a solicitação e envia um erro 404 como resposta se houver não são resultados
get()	Retorna a linha que corresponde à chave primária fornecida ou Nenhum se nenhuma linha correspondente for encontrada
get_or_404()	Retorna a linha que corresponde à chave primária fornecida. Se a chave não for encontrada, ele aborta a solicitação e envia um erro 404 como resposta
contar()	Retorna a contagem de resultados da consulta
paginar()	Retorna um objeto Pagination que contém o intervalo de resultados especificado

Os relacionamentos funcionam de maneira semelhante às consultas. O exemplo a seguir consulta a relação um para muitos entre funções e usuários de ambas as extremidades:

```
>>> usuários = user_role.users
>>> usuários
[<Usuário u'susan'>, <Usuário u'david'>]
>>> users[0].role
<Função u'Usuário'>
```

A consulta `user_role.users` aqui tem um pequeno problema. A consulta implícita executada quando a expressão `user_role.users` é emitida internamente chama `all()` para retornar a lista de usuários. Como o objeto de consulta está oculto, não é possível refiná-lo com filtros de consulta adicionais. Neste exemplo específico, pode ter sido útil solicitar que a lista de usuários seja retornada em ordem alfabética. No [Exemplo 5-4](#), a configuração do relacionamento é modificada com um argumento `lazy = 'dynamic'` para solicitar que a consulta não seja executada automaticamente.

*Exemplo 5-4. app/models.py: relacionamentos dinâmicos*

```
class Role(db.Model):
    users =
        db.relationship('User', backref='role', lazy='dynamic') #
    ...
```

Com o relacionamento configurado desta forma, `user_role.users` retorna uma consulta que ainda não foi executada, então filtros podem ser adicionados a ela:

```
>>> user_role.users.order_by(User.username).all()
[<Usuário u'david', <Usuário u'susan'>]
>>> user_role.users.count()
2
```

## Uso de banco de dados em funções de exibição

As operações de banco de dados descritas nas seções anteriores podem ser usadas diretamente dentro das funções de visualização. O [Exemplo 5-5](#) mostra uma nova versão da rota da página inicial que registra os nomes inseridos pelos usuários no banco de dados.

*Exemplo 5-5. hello.py: uso do banco de dados em funções de visualização*

```
@app.route('/', métodos=['GET', 'POST']) def
index(): form = NameForm() if
form.validate_on_submit():

    user = User.query.filter_by(username=form.name.data).first() se o usuário
    for Nenhum: user = User(username = form.name.data) db.session.add(user)
        session['known'] = False else: session['known'] = True session['name']
        = form.name.data form.name.data = return redirect(url_for('index'))

    ""

return render_template('index.html',
    formulário = formulário, nome = session.get('name'),
    known = session.get('known', False))
```

Nesta versão modificada do aplicativo, cada vez que um nome é enviado, o aplicativo o verifica no banco de dados usando o filtro de consulta `filter_by()`. Uma variável conhecida é escrita na sessão do usuário para que após o redirecionamento a informação possa ser enviada para o template, onde é utilizada para customizar a saudação. Observe que, para que o aplicativo funcione, as tabelas do banco de dados devem ser criadas em um shell Python, conforme mostrado anteriormente.

A nova versão do modelo associado é mostrada no [Exemplo 5-6](#). Este modelo usa o argumento conhecido para adicionar uma segunda linha à saudação que é diferente para conhecido e Novos usuários.

*Exemplo 5-6. templates/index.html*

```
{% extends "base.html" %} {%
import "bootstrap/wtf.html" como wtf %}

{% block title %}Flasky{% endblock %}

{% block page_content %}
<div class="page-header">
<h1>Olá, {{ name }}!</h1> {% if not
known %}
<p>Prazer em conhecê- lo!</p> {% else %}
<p>Feliz em vê-lo novamente!
</p> {% endif %}
</div> {{ wtf.quick_form(form) }} {% endblock %}
```



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar `git checkout 5b` para verificar esta versão do aplicativo.

## Integração com o Python Shell

Ter que importar a instância do banco de dados e os modelos toda vez que uma sessão de shell é iniciada é um trabalho tedioso. Para evitar a repetição constante dessas importações, o comando shell do Flask-Script pode ser configurado para importar automaticamente determinados objetos.

Para adicionar objetos à lista de importação, o comando shell precisa ser registrado com uma função de retorno de chamada `make_context`. Isso é mostrado no [Exemplo 5-7](#).

*Exemplo 5-7. hello.py: Adicionando um contexto de shell*

```
de flask.ext.script import Shell

def make_shell_context():
```

```
    return dict(app=app, db=db, User=User, Role=Role)
manager.add_command("shell", Shell(make_context=make_shell_context))
```

A função `make_shell_context()` registra as instâncias do aplicativo e do banco de dados e os modelos para que sejam importados automaticamente para o shell:

```
$ python hello.py shell >>>
app <Flask 'app'> >>> db
<SQLAlchemy engine='sqlite:///home/flasky/flasky/data.sqlite'>
>>> Usuário

<class 'app.User'>
```



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar `git checkout 5c` para verificar esta versão do aplicativo.

## Migrações de banco de dados com Flask-Migrate

À medida que avança no desenvolvimento de um aplicativo, você descobrirá que seus modelos de banco de dados precisam ser alterados e, quando isso acontece, o banco de dados também precisa ser atualizado.

O Flask-SQLAlchemy cria tabelas de banco de dados a partir de modelos apenas quando eles ainda não existem, então a única maneira de fazê-lo atualizar as tabelas é destruir as tabelas antigas primeiro, mas é claro que isso faz com que todos os dados no banco de dados sejam perdidos.

Uma solução melhor é usar uma estrutura de *migração de banco de dados*. Da mesma forma que as ferramentas de controle de versão de código-fonte acompanham as alterações nos arquivos de código-fonte, uma estrutura de migração de banco de dados acompanha as alterações em um *esquema de banco de dados* e, em seguida, as alterações incrementais podem ser aplicadas ao banco de dados.

O desenvolvedor líder do SQLAlchemy escreveu um framework de migração chamado [Alembic](#), mas em vez de usar o Alembic diretamente, os aplicativos Flask podem usar o [Flask Migrate](#) extensão, um wrapper Alembic leve que se integra ao Flask-Script para fornecer todas as operações por meio de comandos do Flask-Script.

## Criando um repositório de migração

Para começar, o Flask-Migrate deve estar instalado no ambiente virtual:

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

O Exemplo 5-8 mostra como a extensão é inicializada.

*Exemplo 5-8. hello.py: configuração Flask-Migrate*

```
from flask.ext.migrate import Migrate, MigrateCommand
# ...
migrate = Migrate(app, db)
manager.add_command('db', MigrateCommand)
```

Para expor os comandos de migração do banco de dados, o Flask-Migrate expõe uma classe MigrateCommand que está anexada ao objeto gerenciador do Flask-Script. Neste exemplo, o comando é anexado usando db.

Antes que as migrações de banco de dados possam ser mantidas, é necessário criar um repositório de migração com o subcomando init :

```
(venv) $ python hello.py db init Criando
diretório /home/flask/flasky/migrations...done Criando diretório /home/
flask/flasky/migrations/versions...done Gerando /home/flask/flasky/migrations/
alembic.ini...done Gerando /home/flask/flasky/migrations/env.py...done Gerando /
/home/flask/flasky/migrations/env.pyc...done Gerando /home/flask/flasky/migrations /
README...done Gerando /home/flask/flasky/migrations/script.py.mako...done Edite
as configurações de configuração/conexão/log em '/home/flask/flasky/migrations/
alembic.ini' antes de continuar .
```

Este comando cria uma pasta de *migrações* , onde todos os scripts de migração serão armazenados.



Os arquivos em um repositório de migração de banco de dados devem sempre ser adicionados ao controle de versão junto com o restante do aplicativo.

**Criando um Script de Migração No**

Alembic, uma migração de banco de dados é representada por um *script de migração*. Este script tem duas funções chamadas upgrade() e downgrade(). A função upgrade() aplica as alterações do banco de dados que fazem parte da migração e a função downgrade() as remove. Ao ter a capacidade de adicionar e remover as alterações, o Alembic pode reconfigurar um banco de dados para qualquer ponto do histórico de alterações.

As migrações do Alembic podem ser criadas manualmente ou automaticamente usando os comandos de revisão e migração , respectivamente. Uma migração manual cria um script esqueleto de migração com funções upgrade() e downgrade() vazias que precisam ser implementadas pelo desenvolvedor usando diretivas expostas pelo objeto Operations do Alembic. Uma migração automática, por outro lado, gera o código para o upgrade() e

`downgrade()` procurando diferenças entre as definições do modelo e o estado atual do banco de dados.



As migrações automáticas nem sempre são precisas e podem perder alguns detalhes. Os scripts de migração gerados automaticamente devem sempre ser revisados.

O subcomando `migrate` cria um script de migração automática:

```
(venv) $ python hello.py db migrate -m "migração inicial"
INFO [alembic.migration] Contexto impl SQLiteImpl.
INFO [alembic.migration] Assumirá DDL não transacional.
INFO [alembic.autogenerate] Detectado 'funções' da tabela adicionada
INFO [alembic.autogenerate] Detectada a adição da tabela 'users'
INFO [alembic.autogenerate.compare] Detectado índice adicionado
'ix_users_username' em ['username']
Gerando /home/flask/flasky/migrations/versions/1bc
594146bb5_initial_migration.py...done
```



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar `git checkout 5d` para verificar esta versão do aplicativo.

Observe que você não precisa gerar as migrações para este aplicativo, pois todos os scripts de migração estão incluídos no repositório.

## Atualizando o banco de dados

Depois que um script de migração for revisado e aceito, ele poderá ser aplicado ao banco de dados usando o comando `db upgrade` :

```
(venv) $ python hello.py db upgrade
INFO [alembic.migration] Contexto impl SQLiteImpl.
INFO [alembic.migration] Assumirá DDL não transacional.
INFO [alembic.migration] Executando atualização Nenhuma -> 1bc594146bb5, migração inicial
```

Para uma primeira migração, isso é efetivamente equivalente a chamar `db.create_all()`, mas em migrações sucessivas o comando `upgrade` aplica atualizações nas tabelas sem afetar seu conteúdo.



Se você clonou o repositório Git do aplicativo no GitHub, exclua seu arquivo de banco de dados `data.sqlite` e execute o comando `upgrade` do Flask-Migrate para regenerar o banco de dados através do framework de migração.

O tópico de design e uso de banco de dados é muito importante; livros inteiros foram escritos sobre o assunto. Você deve considerar este capítulo como uma visão geral; tópicos mais avançados serão discutidos em capítulos posteriores. O próximo capítulo é dedicado ao envio de e-mail.



---

## CAPÍTULO 6

# E-mail

Muitos tipos de aplicativos precisam notificar os usuários quando determinados eventos ocorrem, e o método usual de comunicação é o e-mail. Embora o pacote *smtplib* da biblioteca padrão do Python possa ser usado para enviar e-mail dentro de um aplicativo Flask, a extensão Flask-Mail envolve o *smtplib* e o integra perfeitamente ao Flask.

## Suporte por e-mail com Flask-Mail

Flask-Mail é instalado com pip:

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

A extensão se conecta a um servidor SMTP (Simple Mail Transfer Protocol) e passa e-mails para ele para entrega. Se nenhuma configuração for fornecida, o Flask-Mail se conectará ao *localhost* na porta 25 e enviará e-mail sem autenticação. A [Tabela 6-1](#) mostra a lista de chaves de configuração que podem ser usadas para configurar o servidor SMTP.

*Tabela 6-1. Chaves de configuração do servidor SMTP do Flask-Mail*

Chave	Descrição padrão
MAIL_HOSTNAME	localhost Nome do host ou endereço IP do servidor de e-mail
MAIL_PORT	25 Porta do servidor de e-mail
MAIL_USE_TLS	False Ativar segurança TLS (Transport Layer Security)
MAIL_USE_SSL	False Ativar segurança SSL (Secure Sockets Layer)
MAIL_USERNAME	Nenhum Nome de usuário da conta de e-mail
MAIL_PASSWORD	Nenhum Senha da conta de e-mail

Durante o desenvolvimento, pode ser mais conveniente conectar-se a um servidor SMTP externo.

Como exemplo, o [Exemplo 6-1](#) mostra como configurar o aplicativo para enviar e-mail por meio de uma conta do Google Gmail.

*Exemplo 6-1. hello.py: configuração do Flask-Mail para Gmail*

```
import os
# ...
app.config['MAIL_SERVER'] = 'smtp.googlemail.com'
app.config['MAIL_PORT'] = 587 app.config['MAIL_USE_TLS']
= Verdadeiro app.config['MAIL_USERNAME'] = os.
environ.get('MAIL_USERNAME') app.config['MAIL_PASSWORD'] =
os.environ.get('MAIL_PASSWORD')
```



Nunca escreva credenciais de conta diretamente em seus scripts, principalmente se você planeja lançar seu trabalho como código aberto. Para proteger as informações de sua conta, faça com que seu script importe informações confidenciais do ambiente.

O Flask-Mail é inicializado conforme mostrado no [Exemplo 6-2](#).

*Exemplo 6-2. hello.py: Inicialização do Flask-Mail*

```
from flask.ext.mail import Mail mail
= Mail(app)
```

As duas variáveis de ambiente que contêm o nome de usuário e a senha do servidor de e-mail precisam ser definidas no ambiente. Se você estiver no Linux ou Mac OS X usando bash, você pode definir essas variáveis da seguinte forma:

```
(venv) $ export MAIL_USERNAME=< nome de usuário do
Gmail> (venv) $ export MAIL_PASSWORD=< senha do Gmail>
```

Para usuários do Microsoft Windows, as variáveis de ambiente são definidas da seguinte forma:

```
(venv) $ set MAIL_USERNAME=< nome de usuário do
Gmail> (venv) $ set MAIL_PASSWORD=< senha do Gmail>
```

## Enviando e-mail do Python Shell

Para testar a configuração, você pode iniciar uma sessão de shell e enviar um e-mail de teste:

```
(venv) $ python hello.py shell >>>
from flask.ext.mail import Message >>> from
hello import mail >>> msg = Message('test
subject', sender='you@example.com', destinatários =['você@example.com'])
...
>>> msg.body = 'text body' >>>
msg.html = '<b>HTML</b> body' >>>
com app.app_context(): mail.send(msg)
...
...
```

Observe que a função send() do Flask-Mail usa current\_app, portanto, ela precisa ser executada com um contexto de aplicativo ativado.

### **Integrando e-mails com o aplicativo**

Para evitar ter que criar mensagens de e-mail manualmente todas as vezes, é uma boa ideia abstrair as partes comuns da funcionalidade de envio de e-mail do aplicativo em uma função. Como um benefício adicional, essa função pode renderizar corpos de e-mail de modelos Jinja2 para ter mais flexibilidade. A implementação é mostrada no [Exemplo 6-3](#).

*Exemplo 6-3. hello.py: suporte por e-mail*

**da mensagem de importação** do `frasco.ext.mail`

```
app.config['FLASKY_MAIL_SUBJECT_PREFIX'] = '[Flasky]'  
app.config['FLASKY_MAIL_SENDER'] = 'Admin Flasky <flasky@example.com>'  
  
def send_email(to, subject, template, **kwargs): msg =  
    Message(app.config['FLASKY_MAIL_SUBJECT_PREFIX'] + subject,  
            sender=app.config['FLASKY_MAIL_SENDER'], destinatários=[to])  
    msg.body = render_template (template + '.txt', **kwargs) msg.html = render_template  
(template + '.html', **kwargs ) mail.send(msg)
```

A função se baseia em duas chaves de configuração específicas do aplicativo que definem uma string de prefixo para o assunto e o endereço que será usado como remetente. A função send\_email recebe o endereço de destino, uma linha de assunto, um modelo para o corpo do email e uma lista de argumentos de palavras-chave. O nome do modelo deve ser fornecido sem a extensão, para que duas versões do modelo possam ser usadas para os corpos de texto simples e rich text. Os argumentos de palavra-chave passados pelo chamador são fornecidos às chamadas render\_template() para que possam ser usados pelos modelos que geram o corpo do email.

A função de visualização index() pode ser facilmente expandida para enviar um e-mail ao administrador sempre que um novo nome for recebido com o formulário. O [Exemplo 6-4](#) mostra essa mudança.

*Exemplo 6-4. hello.py: exemplo de e-mail*

```
# ...  
app.config['FLASKY_ADMIN'] = os.environ.get('FLASKY_ADMIN') #  
@app.route('/', métodos=['GET', 'POST']) def index(): form = NameForm()  
se form.validate_on_submit():  
  
    user = User.query.filter_by(username=form.name.data).first() se o usuário  
    for Nenhum: user = User(username=form.name.data) db.session.add(user)  
    session['known'] = False se app.config['FLASKY_ADMIN']:
```

```

    send_email(app.config['FLASKY_ADMIN'], 'Novo usuário', 'mail/
        new_user', usuário=usuário)
senão:
    session['known'] = True
    session['name'] = form.name.data
    form.name.data = return
    redirect(url_for('index')) return
render_template('index.html', form=form, name =sessão.get('nome'),
    known=session.get('known', False))

```

O destinatário do email é fornecido na variável de ambiente FLASKY\_ADMIN carregada em uma variável de configuração com o mesmo nome durante a inicialização. Dois arquivos de modelo precisam ser criados para as versões de texto e HTML do email. Esses arquivos são armazenados em um

subpasta *mail* dentro dos *templates* para mantê-los separados dos templates normais. Os templates de email esperam que o usuário seja dado como um argumento de template, então a chamada para `send_email()` o inclui como um argumento de palavra-chave.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar `git checkout 6a` para verificar esta versão do aplicativo.

Além das variáveis de ambiente MAIL\_USERNAME e MAIL\_PASSWORD descritas anteriormente, esta versão do aplicativo precisa da variável de ambiente FLASKY\_ADMIN .

Para usuários de Linux e Mac OS X, o comando para iniciar o aplicativo é:

```
(venv) $ export FLASKY_ADMIN=<seu-endereço de e-mail>
```

Para usuários do Microsoft Windows, este é o comando equivalente:

```
(venv) $ set FLASKY_ADMIN=< nome de usuário do Gmail>
```

Com essas variáveis de ambiente definidas, você pode testar o aplicativo e receber um e-mail toda vez que inserir um novo nome no formulário.

## Envio de e-mail assíncrono

Se você enviou alguns e-mails de teste, provavelmente notou que a função `mail.send()` é bloqueada por alguns segundos enquanto o e-mail é enviado, fazendo com que o navegador não responda durante esse período. Para evitar atrasos desnecessários durante o tratamento da solicitação, a função de envio de e-mail pode ser movida para um encadeamento em segundo plano. O Exemplo 6-5 mostra essa mudança.

*Exemplo 6-5. hello.py: suporte por e-mail assíncrono*

**do encadeamento de importação Thread**

```
def send_async_email(app, msg): com
    app.app_context():
```

```
mail.send(msg)

def send_email(to, subject, template, **kwargs): msg =
    Message(app.config['FLASKY_MAIL_SUBJECT_PREFIX'] + subject,
            sender=app.config['FLASKY_MAIL_SENDER'], destinatários=[to])
    msg.body = render_template (template + '.txt', **kwargs) msg.html = render_template
    (template + '.html', **kwargs ) thr = Thread(target=send_async_email, args=[app, msg])
    thr.start() return thr
```

Esta implementação destaca um problema interessante. Muitas extensões do Flask operam sob a suposição de que existem aplicativos ativos e contextos de solicitação. A função `send()` do Flask-Mail usa `current_app`, portanto, requer que o contexto do aplicativo esteja ativo. Mas quando a função `mail.send()` é executada em uma thread diferente, o contexto do aplicativo precisa ser criado artificialmente usando `app.app_context()`.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar `git checkout 6b` para verificar esta versão do aplicativo.

Se você executar o aplicativo agora, perceberá que ele é muito mais responsivo, mas lembre-se de que, para aplicativos que enviam um grande volume de e-mail, ter um trabalho dedicado ao envio de e-mail é mais apropriado do que iniciar um novo thread para cada e-mail . Por exemplo, a execução da função `send_async_email()` pode ser enviada para um [Celery](#) fila de tarefas.

Este capítulo completa a visão geral dos recursos que são obrigatórios para a maioria dos aplicativos da web. O problema agora é que o script `hello.py` está começando a ficar grande e isso dificulta o trabalho. No próximo capítulo, você aprenderá como estruturar um aplicativo maior.



---

## CAPÍTULO 7

# Estrutura de aplicativo grande

Embora ter pequenos aplicativos da Web armazenados em um único script possa ser muito conveniente, essa abordagem não é bem dimensionada. À medida que o aplicativo cresce em complexidade, trabalhar com um único arquivo de origem grande se torna problemático.

Ao contrário da maioria dos outros frameworks web, o Flask não impõe uma organização específica para grandes projetos; a maneira de estruturar o aplicativo é deixada inteiramente para o desenvolvedor. Neste capítulo, é apresentada uma forma possível de organizar uma grande aplicação em pacotes e módulos. Essa estrutura será usada nos demais exemplos do livro.

## Estrutura do projeto

O [Exemplo 7-1](#) mostra o layout básico de um aplicativo Flask.

*Exemplo 7-1. Estrutura básica do aplicativo Flask de vários arquivos*

```
|-flasky |-  
  app/ |-  
    templates/ |-  
    static/ |-main/  
    |__init__.py |-  
      errors.py |-  
      forms.py |-  
      views.py |-  
      __init__.py |-  
        email.py |  
        -models.py |-  
        migrations/ |-tests/  
        |__init__.py |-test*.py  
    |-venv/ |-  
      requirements.txt
```

```
| - config.py |
  manage.py
```

Essa estrutura tem quatro pastas de nível superior:

- O aplicativo Flask reside dentro de um pacote denominado *app genericamente*. • A pasta *migrations* contém os scripts de migração do banco de dados, como antes.
- Os testes unitários são escritos em um pacote de *testes*.
- A pasta *venv* contém o ambiente virtual Python, como antes.

Há também alguns novos arquivos:

- *requirements.txt* lista as dependências do pacote para que seja fácil gerar novamente um ambiente virtual idêntico em um computador diferente.
- *config.py* armazena as definições de configuração.
- *manage.py* inicia o aplicativo e outras tarefas do aplicativo.

Para ajudá-lo a entender completamente essa estrutura, as seções a seguir descrevem o processo para converter o aplicativo *hello.py* para ele.

## Opções de configuração

Os aplicativos geralmente precisam de vários conjuntos de configuração. O melhor exemplo disso é a necessidade de usar bancos de dados diferentes durante o desenvolvimento, teste e produção para que não interfiram entre si.

Em vez da simples configuração de estrutura semelhante a um dicionário usada pelo *hello.py*, uma hierarquia de classes de configuração pode ser usada. O [Exemplo 7-2](#) mostra o arquivo *config.py*.

*Exemplo 7-2. config.py: configuração do aplicativo*

```
import os
based = os.path.abspath(os.path.dirname(__file__))

class Config:
    SECRET_KEY = os.environ.get('SECRET_KEY') ou 'string difícil de adivinhar'
    SQLALCHEMY_COMMIT_ON_TEARDOWN =
        Verdadeiro FLASKY_MAIL_SUBJECT_PREFIX = '[Flasky]'
    FLASKY_MAIL_SENDER = 'Administrador Flasky <flasky@example.com>'
    FLASKY_ADMIN = os.environ.get('FLASKY_ADMIN')

    @staticmethod
    def init_app(app):
        pass

class DevelopmentConfig(Config):
    DEBUG = Verdadeiro
```

```

MAIL_SERVER = 'smtp.googlemail.com'
MAIL_PORT = 587
MAIL_USE_TLS =
Verdadeiro MAIL_USERNAME = os.environ.get('MAIL_USERNAME')
MAIL_PASSWORD = os.environ.get('MAIL_PASSWORD')
SQLALCHEMY_DATABASE_URI = os.environ.get('DEV_DATABASE_URL') ou \
    'sqlite:/// + os.path.join(basedir, 'data-dev.sqlite')

class TestingConfig(Config):
    TESTING = True
    SQLALCHEMY_DATABASE_URI = os.environ.get('TEST_DATABASE_URL') ou \
        'sqlite:/// + os.path.join(basedir, 'data-test.sqlite')

class ProductionConfig(Config):
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL') ou \
        'sqlite:/// + os.path.join(basedir, 'data.sqlite')

config =
{ 'development': DevelopmentConfig,
  'testing': TestingConfig, 'production':
ProductionConfig,
  'padrão': DevelopmentConfig
}

```

A classe base Config contém configurações comuns a todas as configurações; as diferentes subclasses definem configurações que são específicas para uma configuração. Configurações adicionais podem ser adicionadas conforme necessário.

Para tornar a configuração mais flexível e segura, algumas configurações podem ser importadas opcionalmente de variáveis de ambiente. Por exemplo, o valor de SECRET\_KEY, devido à sua natureza sensível, pode ser definido no ambiente, mas um valor padrão é fornecido caso o ambiente não o defina.

A variável SQLALCHEMY\_DATABASE\_URI recebe valores diferentes em cada uma das três configurações. Isso permite que o aplicativo seja executado em diferentes configurações, cada uma usando um banco de dados diferente.

As classes de configuração podem definir um método de classe init\_app() que recebe uma instância do aplicativo como argumento. Aqui a inicialização específica da configuração pode ser executada. Por enquanto, a classe Config básica implementa um método init\_app() vazio .

Na parte inferior do script de configuração, as diferentes configurações são registradas em um dicionário de configuração . Uma das configurações (a de desenvolvimento neste caso) também é registrada como padrão.

## Pacote de aplicativos

O pacote do aplicativo é onde residem todos os códigos, modelos e arquivos estáticos do aplicativo. Ele é chamado simplesmente *de aplicativo*, embora possa receber um nome específico do aplicativo, se desejado. Os *modelos* e as *pastas estáticas* fazem parte do pacote do aplicativo, portanto, essas duas pastas são movidas dentro *do app*. Os modelos de banco de dados e as funções de suporte de e-mail também são movidos dentro deste pacote, cada um em seu próprio módulo como *app/models.py* e *app/email.py*.

### Usando uma Fábrica de Aplicativos A

maneira como o aplicativo é criado na versão de arquivo único é muito conveniente, mas tem uma grande desvantagem. Como o aplicativo é criado no escopo global, não há como aplicar as alterações de configuração dinamicamente: no momento em que o script estiver em execução, a instância do aplicativo já foi criada, então já é tarde demais para fazer alterações na configuração. Isso é particularmente importante para testes de unidade porque às vezes é necessário executar o aplicativo em diferentes configurações para melhor cobertura de teste.

A solução para esse problema é atrasar a criação do aplicativo movendo-o para uma *função de fábrica* que pode ser invocada explicitamente a partir do script. Isso não apenas dá ao script tempo para definir a configuração, mas também a capacidade de criar várias instâncias do aplicativo, algo que também pode ser muito útil durante o teste. A função de fábrica do aplicativo, mostrada no [Exemplo 7-3](#), é definida no construtor do pacote *app* .

Esse construtor importa a maioria das extensões do Flask atualmente em uso, mas como não há nenhuma instância de aplicativo para inicializá-las, ele as cria não inicializadas, não passando argumentos para seus construtores. A função *create\_app()* é a fábrica de aplicativos, que recebe como argumento o nome de uma configuração a ser usada para o aplicativo. As configurações armazenadas em uma das classes definidas em *config.py* podem ser importadas diretamente para a aplicação usando o método *from\_object()* disponível no objeto de configuração *app.config* do Flask . O objeto de configuração é selecionado pelo nome no dicionário de configuração . Depois que um aplicativo é criado e configurado, as extensões podem ser inicializadas. Chamar *init\_app()* nas extensões que foram criadas anteriormente conclui sua inicialização.

*Exemplo 7-3. app/\_init\_.py: construtor do pacote de aplicativos*

```
do frasco import Flask, render_template do
frasco.ext.bootstrap import Bootstrap do
frasco.ext.mail import Correio do
frasco.ext.moment import Momento do
frasco.ext.sqlalchemy import SQLAlchemy do
config import config

bootstrap = Bootstrap() mail
= Mail() momento = Moment()
db = SQLAlchemy()
```

```

def create_app(config_name): app =
    Flask(__name__)
    app.config.from_object(config[config_name])
    config[config_name].init_app(app)

    bootstrap.init_app(app)
    mail.init_app(app)
    moment.init_app(app)
    db.init_app(app)

    # anexar rotas e páginas de erro personalizadas aqui

```

aplicativo de retorno

A função de fábrica retorna a instância do aplicativo criada, mas observe que os aplicativos criados com a função de fábrica em seu estado atual estão incompletos, pois estão faltando rotas e manipuladores de página de erro personalizados. Este é o tema da próxima seção.

### Implementando a funcionalidade do aplicativo em um blueprint A

conversão para uma fábrica de aplicativos apresenta uma complicação para rotas. Em aplicativos de script único, a instância do aplicativo existe no escopo global, portanto, as rotas podem ser facilmente definidas usando o decorador `app.route`. Mas agora que o aplicativo é criado em tempo de execução, o decorador `app.route` começa a existir somente depois que `create_app()` é invocado, o que é tarde demais. Assim como as rotas, os manipuladores de página de erro personalizados apresentam o mesmo problema, pois são definidos com o decorador `app.errorhandler`.

Felizmente, o Flask oferece uma solução melhor usando *blueprints*. Um blueprint é semelhante a um aplicativo, pois também pode definir rotas. A diferença é que as rotas associadas a um blueprint ficam em estado inativo até que o blueprint seja registrado em um aplicativo, momento em que as rotas se tornam parte dele. Usando um blueprint definido no escopo global, as rotas do aplicativo podem ser definidas quase da mesma maneira que no aplicativo de script único.

Assim como os aplicativos, os blueprints podem ser definidos em um único arquivo ou podem ser criados de forma mais estruturada com vários módulos dentro de um pacote. Para permitir maior flexibilidade, um subpacote dentro do pacote de aplicação será criado para hospedar o blueprint. O [Exemplo 7-4](#) mostra o construtor do pacote, que cria o blueprint.

*Exemplo 7-4. app/main/\_init\_.py: criação do blueprint*

**do projeto de importação de frascos**

```
main = Blueprint('principal', __name__)
```

```
de . importar visualizações, erros
```

Os blueprints são criados instanciando um objeto da classe Blueprint. O construtor para esta classe recebe dois argumentos obrigatórios: o nome do blueprint e o módulo ou pacote onde o blueprint está localizado. Assim como nos aplicativos, a variável `__name__` do Python é, na maioria dos casos, o valor correto para o segundo argumento.

As rotas do aplicativo são armazenadas em um módulo `app/main/views.py` dentro do pacote e os manipuladores de erro estão em `app/main/errors.py`. A importação desses módulos faz com que as rotas e os manipuladores de erros sejam associados ao blueprint. É importante observar que os módulos são importados na parte inferior do script `app/__init__.py` para evitar dependências circulares, pois `views.py` e `errors.py` precisam importar o blueprint principal .

O blueprint é registrado com o aplicativo dentro da função de fábrica `create_app()` , conforme mostrado no Exemplo 7-5.

*Exemplo 7-5. app/\_init\_.py: registro do blueprint*

```
def create_app(config_name):
    # ...

    from main import main como main_blueprint
    app.register_blueprint(main_blueprint)

    aplicativo de retorno
```

O Exemplo 7-6 mostra os manipuladores de erros.

*Exemplo 7-6. app/main/errors.py: Blueprint com manipuladores de erros*

```
do frasco importe render_template do .
importar principal

@main.app_errorhandler(404)
def page_not_found(e):
    return render_template('404.html'), 404

@main.app_errorhandler(500)
def internal_server_error(e): return
    render_template('500.html'), 500
```

Uma diferença ao escrever manipuladores de erro dentro de um blueprint é que, se o decorador `errorhandler` for usado, o manipulador será invocado apenas para erros originados no blueprint. Para instalar manipuladores de erros em todo o aplicativo, o `app_errorhandler` deve ser usado.

O Exemplo 7-7 mostra as rotas do aplicativo atualizadas para estarem no blueprint.

*Exemplo 7-7. app/main/views.py: Blueprint com rotas de aplicativos*

```
from datetime import datetime do
frasco import render_template, session, redirect, url_for
```

```

de . import principal
de .forms import NameForm
de .. import db de ..models
import User

@main.route('/', métodos=['GET', 'POST']) def
index(): form = NameForm() if
    form.validate_on_submit():

    ...
    # return redirect(url_for('.index'))
    return render_template('index.html',
        form=form, name=session.get('name'),
        known=session.get('known', False),
        current_time=datetime.utcnow())

```

Existem duas diferenças principais ao escrever uma função de visualização dentro de um blueprint. Primeiro, como foi feito anteriormente para manipuladores de erros, o decorador de rota vem do blueprint. A segunda diferença está no uso da função url\_for(). Como você deve se lembrar, o primeiro argumento para essa função é o nome do ponto de extremidade da rota, que para rotas baseadas em aplicativos tem como padrão o nome da função de visualização. Por exemplo, em um aplicativo de script único, a URL para uma função de visualização index() pode ser obtida com url\_for('index').

A diferença com os blueprints é que o Flask aplica um namespace a todos os endpoints provenientes de um blueprint para que vários blueprints possam definir funções de visualização com os mesmos nomes de endpoint sem colisões. O namespace é o nome do blueprint (o primeiro argumento para o construtor Blueprint), então a função de visualização index() é registrada com o nome do terminal main.index e sua URL pode ser obtida com url\_for('main.index').

A função url\_for() também suporta um formato mais curto para endpoints em blueprints nos quais o nome do blueprint é omitido, como url\_for('.index'). Com essa notação, o blueprint para a solicitação atual é usado. Isso significa efetivamente que os redirecionamentos dentro do mesmo blueprint podem usar o formato mais curto, enquanto os redirecionamentos entre blueprints devem usar o nome do endpoint com namespace.

Para concluir as alterações na página do aplicativo, os objetos de formulário também são armazenados dentro do blueprint em um módulo `app/main/forms.py`.

## Iniciar script

O arquivo `manage.py` na pasta de nível superior é usado para iniciar o aplicativo. Este script é mostrado no Exemplo 7-8.

*Exemplo 7-8. manage.py: inicia o script*

```
#!/usr/bin/env python
import os from app
import create_app, db de app.models
import User, Role de flask.ext.script
import Manager, Shell de flask.ext.migrate import
Migrate, MigrateCommand

app = create_app(os.getenv('FLASK_CONFIG') ou 'default') manager
= Manager(app) migrate = Migrate(app, db)

def make_shell_context():
    return dict(app=app, db=db, User=User, Role=Role)
manager.add_command("shell", Shell(make_context=make_shell_context))
manager.add_command('db', MigrateCommand)

if __name__ == '__main__':
    manager.run()
```

O script começa criando um aplicativo. A configuração utilizada é retirada da variável de ambiente FLASK\_CONFIG se estiver definida; se não, a configuração padrão é usada. Flask-Script, Flask-Migrate e o contexto personalizado para o shell do Python são inicializados.

Por conveniência, uma linha shebang é adicionada, para que em sistemas operacionais baseados em Unix o script possa ser executado como ./manage.py em vez do python mais detalhado manage.py.

## Arquivo de Requisitos

Os aplicativos devem incluir um arquivo *requirements.txt* que registre todas as dependências do pacote, com os números de versão exatos. Isso é importante caso o ambiente virtual precise ser regenerado em uma máquina diferente, como a máquina na qual o aplicativo será implantado para uso em produção. Este arquivo pode ser gerado automaticamente pelo pip com o seguinte comando:

```
(venv) $ pip freeze >requirements.txt
```

É uma boa ideia atualizar esse arquivo sempre que um pacote for instalado ou atualizado. Um exemplo de arquivo de requisitos é mostrado aqui:

```
Frasco==0.10.1
Flask-Bootstrap==3.0.3.1
Flask-Mail==0.9.0 Flask-
Migrate==1.1.0 Flask-
Moment==0.2.0 Flask-
SQLAlchemy==1.0 Flask-
Script==0.6.6 Flask-WTF=
=0,9,4
```

```
Jinja2==2.7.1
Mako==0.9.1
MarkupSafe==0.18
SQLAlchemy==0.8.4
WTForms==1.0.5
Tool==0.9.4
alambique==0.6.2
pisca-pisca==1.3
é perigoso==0.23
```

Quando você precisa construir uma réplica perfeita do ambiente virtual, você pode criar um novo ambiente virtual e executar o seguinte comando nele:

```
(venv) $ pip install -r requirements.txt
```

Os números de versão no arquivo `requirements.txt` de exemplo provavelmente estarão desatualizados quando você ler isso. Você pode tentar usar versões mais recentes dos pacotes, se quiser. Se você tiver algum problema, sempre poderá voltar para as versões especificadas no arquivo de requisitos, pois elas são compatíveis com o aplicativo.

### Testes de unidade

Esta aplicação é muito pequena, então não há muito o que testar ainda, mas como exemplo, dois testes simples podem ser definidos conforme mostrado no [Exemplo 7-9](#).

*Exemplo 7-9. `test/test_basics.py`: testes de unidade*

```
import unittest
do frasco import current_app do
app import create_app, db

class BasicsTestCase(unittest.TestCase):
    def setUp(self):
        self.app = create_app('testing')
        self.app_context = self.app.app_context()
        self.app_context.push() db.create_all()

    def tearDown(self):
        db.session.remove()
        db.drop_all()
        self.app_context.pop()

    def test_app_exists(self):
        self.assertFalse(current_app is None)

    def test_app_is_testing(self):
        self.assertTrue(current_app.config['TESTING'])
```

Os testes são escritos usando o pacote unittest padrão da biblioteca padrão do Python. Os métodos setUp() e tearDown() são executados antes e depois de cada teste, e quaisquer métodos que tenham um nome que comece com test\_ são executados como testes.



Se você quiser aprender mais sobre como escrever testes de unidade com o pacote *unittest* do Python , leia a [documentação oficial](#).

O método setUp() tenta criar um ambiente para o teste próximo ao de um aplicativo em execução. Ele primeiro cria um aplicativo configurado para teste e ativa seu contexto. Essa etapa garante que os testes tenham acesso a current\_app, como solicitações regulares.

Em seguida, ele cria um banco de dados totalmente novo que o teste pode usar quando necessário. O banco de dados e o contexto do aplicativo são removidos no método tearDown() .

O primeiro teste garante que a instância do aplicativo existe. O segundo teste garante que o aplicativo esteja sendo executado na configuração de teste. Para tornar a pasta de *testes* um pacote adequado, um arquivo *tests/\_\_init\_\_.py* precisa ser adicionado, mas este pode ser um arquivo vazio, pois o pacote unittest pode varrer todos os módulos e localizar os testes.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar git checkout 7a para verificar a versão convertida do aplicativo. Para garantir que você tenha todas as dependências instaladas, execute também pip install -r requirements.txt.

Para executar os testes de unidade, um comando personalizado pode ser adicionado ao script *manage.py* . O [Exemplo 7-10](#) mostra como adicionar um comando de teste .

#### *Exemplo 7-10. manage.py: comando do iniciador de teste de unidade*

```
@manager.command
def test(): """Execute
os testes de unidade."""
import unittest
tests = unittest.TestLoader().discover('tests')
unittest.TextTestRunner(verbosity=2).run(tests)
```

O decorador manager.command simplifica a implementação de comandos personalizados. O nome da função decorada é usado como nome do comando e a docstring da função é exibida nas mensagens de ajuda. A implementação da função test() invoca o executor de teste do pacote unittest .

Os testes unitários podem ser executados da seguinte forma:

```
(venv) $ python manage.py test
test_app_exists (test_basics.BasicsTestCase) ... test_app_is_testing OK
(test_basics.BasicsTestCase) ... OK
-----
Executou 2 testes em 0,001s
OK
```

### Configuração do banco de dados

O aplicativo reestruturado usa um banco de dados diferente da versão de script único.

A URL do banco de dados é obtida de uma variável de ambiente como primeira escolha, com um banco de dados SQLite padrão como alternativa. As variáveis de ambiente e os nomes de arquivo do banco de dados SQLite são diferentes para cada uma das três configurações. Por exemplo, na configuração de desenvolvimento a URL é obtida da variável de ambiente `DEV_DATABASE_URL`, e se ela não estiver definida então um banco de dados SQLite com o nome `data-dev.sqlite` é usado.

Independentemente da origem da URL do banco de dados, as tabelas do banco de dados devem ser criadas para o novo banco de dados. Ao trabalhar com o Flask-Migrate para acompanhar as migrações, as tabelas de banco de dados podem ser criadas ou atualizadas para a revisão mais recente com um único comando:

```
(venv) $ python manage.py db upgrade
```

Acredite ou não, você chegou ao final da Parte I. Agora você aprendeu os elementos básicos necessários para construir uma aplicação web com o Flask, mas provavelmente se sente inseguro sobre como todas essas peças se encaixam para formar uma aplicação real. O objetivo da Parte II é ajudar com isso, orientando você no desenvolvimento de um aplicativo completo.



**PARTE II**

---

**Exemplo: Um Social  
Aplicativo de blog**



---

## CAPÍTULO 8

# Autenticação de usuário

A maioria dos aplicativos precisa acompanhar quem são seus usuários. Quando os usuários se conectam ao aplicativo, eles se *autenticam* nele, um processo pelo qual tornam sua identidade conhecida. Uma vez que o aplicativo saiba quem é o usuário, ele pode oferecer uma experiência personalizada.

O método de autenticação mais usado exige que os usuários forneçam uma identificação (seja seu e-mail ou nome de usuário) e uma senha secreta. Neste capítulo, o sistema de autenticação completo para Flasky é criado.

## Extensões de autenticação para Flask

Existem muitos pacotes de autenticação Python excelentes, mas nenhum deles faz tudo. A solução de autenticação de usuário apresentada neste capítulo usa vários pacotes e fornece a cola que os faz funcionar bem juntos. Esta é a lista de pacotes que serão utilizados:

- Flask-Login: Gerenciamento de sessões de usuário para usuários logados •

Werkzeug: verificação e hash de senha • itsdangerous: geração e verificação de token criptograficamente segura

Além dos pacotes específicos de autenticação, as seguintes extensões de uso geral serão usadas:

- Flask-Mail: Envio de e-mails relacionados à autenticação • Flask-

Bootstrap: modelos HTML

- Flask-WTF: formulários da Web

## Segurança de senha

A segurança das informações do usuário armazenadas em bancos de dados geralmente é negligenciada durante o design de aplicativos da web. Se um invasor conseguir invadir seu servidor e acessar seu banco de dados de usuários, você arriscará a segurança de seus usuários e o risco será maior do que você imagina. É um fato conhecido que a maioria dos usuários usa a mesma senha em vários sites, portanto, mesmo que você não armazene nenhuma informação sensível, o acesso às senhas armazenadas em seu banco de dados pode dar ao invasor acesso a contas que seus usuários têm em outros sites locais.

A chave para armazenar senhas de usuário com segurança em um banco de dados não depende do armazenamento da senha em si, mas de um *hash* dela. Uma função de hashing de senha recebe uma senha como entrada e aplica uma ou mais transformações criptográficas a ela. O resultado é uma nova sequência de caracteres que não tem nenhuma semelhança com a senha original. Os hashes de senha podem ser verificados no lugar das senhas reais porque as funções de hash são repetíveis: dadas as mesmas entradas, o resultado é sempre o mesmo.



O hash de senha é uma tarefa complexa que é difícil de acertar. É recomendado que você não implemente sua própria solução, mas sim confie em bibliotecas respeitáveis que foram revisadas pela comunidade. Se você estiver interessado em aprender o que está envolvido na geração de hashes de senha segura, o artigo [Salted Password Hashing - Doing it Right](#) é uma leitura que vale a pena.

### Hashing de senhas com Werkzeug

O módulo de segurança do Werkzeug implementa convenientemente o hash de senha seguro. Essa funcionalidade é exposta com apenas duas funções, utilizadas nas fases de registro e verificação, respectivamente:

- `generate_password_hash(password, method=pbkdf2:sha1, salt_length=8)`: Esta função recebe uma senha de texto simples e retorna o hash da senha como uma string que pode ser armazenada no banco de dados do usuário. Os valores padrão para `method` e `salt_length` são suficientes para a maioria dos casos de uso.
- `check_password_hash(hash, password)`: Esta função pega um hash de senha recuperado do banco de dados e a senha digitada pelo usuário. Um valor de retorno `True` indica que a senha está correta.

O [Exemplo 8-1](#) mostra as alterações no modelo User criado no [Capítulo 5](#) para acomodar o hash de senha.

*Exemplo 8-1. app/models.py: hash de senha no modelo de usuário*

`de werkzeug.security importação generate_password_hash, check_password_hash`

```

class User(db.Model): #
    ...
    password_hash = db.Column(db.String(128))

    @property
    def password(self):
        raise AttributeError('senha não é um atributo legível')

    @password.setter
    def password(self, password):
        self.password_hash = generate_password_hash(senha)

    def check_password(self, password): return
        check_password_hash(self.password_hash, password)

```

A função de hash de senha é implementada por meio de uma propriedade somente de gravação chamada senha. Quando esta propriedade estiver configurada, o método setter chamará a função generate\_password\_hash() de Werkzeug e escreverá o resultado no campo password\_hash . A tentativa de ler a propriedade de senha retornará um erro, pois claramente a senha original não pode ser recuperada após o hash.

O método verify\_password pega uma senha e a passa para a função check\_password\_hash() de Werkzeug para verificação em relação à versão com hash armazenada no modelo User . Se este método retornar True, a senha está correta.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar git checkout 8a para verificar esta versão do aplicativo.

A funcionalidade de hash de senha agora está completa e pode ser testada no shell:

```

(venv) $ python manage.py shell >>> u
= User() >>> u.password = 'cat' >>>
u.password_hash

'pbkdf2:sha1:1000$duxMk0OF$4735b293e397d6eeaf650aaf490fd9091f928bed' >>>
u.verify_password ('gato')
Verdadeiro

>>> u.verify_password('cão')
Falso

>>> u2 = User()
>>> u2.password = 'cat' >>>
u2.password_hash

'pbkdf2:sha1:1000$UjvnGeTP$875e28eb0874f44101d6b332442218f66975ee89'

```

Observe como os usuários u e u2 têm hashes de senha completamente diferentes, mesmo que ambos usem a mesma senha. Para garantir que esta funcionalidade continue a funcionar no

No futuro, os testes acima podem ser escritos como testes unitários que podem ser repetidos facilmente. No [Exemplo 8-2](#) um novo módulo dentro do pacote de `testes` é mostrado com três novos testes que exercitam as mudanças recentes no modelo User .

*Exemplo 8-2. tests/test\_user\_model.py: testes de hash de senha*

```
import unittest
from app.models import User

class UserModelTestCase(unittest.TestCase):
    def test_password_setter(self):
        u = User(password = 'cat')
        self.assertTrue(u.password_hash != None)

    def test_no_password_getter(self):
        u = User(password = 'cat') com
        self.assertRaises(AttributeError): u.password

    def test_password_verification(self):
        u = User(password = 'cat')
        self.assertTrue(u.verify_password('cat'))
        self.assertFalse(u.verify_password('dog'))

    def test_password_salts_are_random(self):
        u = User(password='cat') u2 = User(password='cat')
        self.assertTrue(u.password_hash != u2.password_hash)
```

## Criando um Blueprint de Autenticação

Os blueprints foram introduzidos no [Capítulo 7](#) como uma forma de definir rotas no escopo global depois que a criação do aplicativo foi movida para uma função de fábrica. As rotas relacionadas ao sistema de autenticação do usuário podem ser adicionadas a um blueprint de autenticação . Usar diferentes esquemas para diferentes conjuntos de funcionalidades do aplicativo é uma ótima maneira de manter o código bem organizado.

O blueprint de autenticação será hospedado em um pacote Python com o mesmo nome. O construtor de pacote do blueprint cria o objeto blueprint e importa rotas de um módulo `views.py` . Isso é mostrado no [Exemplo 8-3](#).

*Exemplo 8-3. app/auth/\_\_init\_\_.py: criação do blueprint*

```
do projeto de importação de frascos

auth = Blueprint('auth', __name__)

de . importar visualizações
```

O módulo `app/auth/views.py`, mostrado no [Exemplo 8-4](#), importa o blueprint e define as rotas associadas à autenticação usando seu decorador de rota. Por enquanto, uma rota `/login` é adicionada, o que renderiza um modelo de espaço reservado com o mesmo nome.

*Exemplo 8-4. app/auth/views.py: rotas de blueprint e funções de visualização*

```
do frasco importe render_template do .
autenticação de importação
```

```
@auth.route('/login') def
login():
    return render_template('auth/login.html')
```

Observe que o arquivo de modelo fornecido a `render_template()` é armazenado dentro da pasta `auth`. Essa pasta deve ser criada dentro de `app/templates`, pois o Flask espera que os templates sejam relativos à pasta de templates do aplicativo. Ao armazenar os modelos de blueprint em sua própria pasta, não há risco de colisões de nomes com o blueprint principal ou qualquer outro blueprint que será adicionado no futuro.



Os blueprints também podem ser configurados para ter sua própria pasta independente para modelos. Quando várias pastas de templates são configuradas, a função `render_template()` procura primeiro a pasta de templates configurada para a aplicação e então procura as pastas de template definidas por blueprints.

O blueprint de autenticação precisa ser anexado ao aplicativo na função de fábrica `create_app()` conforme mostrado no [Exemplo 8-5](#).

*Exemplo 8-5. app/\_\_init\_\_.py: anexo do blueprint*

```
def create_app(config_name):
    ...
    # from .auth import auth as auth_blueprint
    app.register_blueprint(auth_blueprint, url_prefix='/auth')
```

aplicativo de retorno

O argumento `url_prefix` no registro do blueprint é opcional. Quando usado, todas as rotas definidas no blueprint serão registradas com o prefixo fornecido, neste caso `/auth`. Por exemplo, a rota `/login` será registrada como `/auth/login`, e a URL totalmente qualificada no servidor Web de desenvolvimento se tornará `http://localhost:5000/auth/login`.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar git checkout 8b para verificar esta versão do aplicativo.

## Autenticação do usuário com Flask-Login

Quando os usuários efetuam login no aplicativo, seu estado autenticado deve ser registrado para que seja lembrado à medida que navegam pelas diferentes páginas. Flask-Login é uma extensão pequena, mas extremamente útil, especializada em gerenciar esse aspecto específico de um sistema de autenticação de usuário, sem estar vinculado a um mecanismo de autenticação específico.

Para começar, a extensão precisa ser instalada no ambiente virtual:

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

### Preparando o modelo de usuário para logins Para

poder trabalhar com o modelo de usuário do aplicativo , a extensão Flask-Login requer que alguns métodos sejam implementados por ela. Os métodos necessários são mostrados na [Tabela 8-1](#).

*Tabela 8-1. Métodos de usuário Flask-Login*

Método	Descrição
is_authenticated()	Deve retornar True se o usuário tiver credenciais de login ou False caso contrário. está ativo()
	Deve retornar True se o usuário tiver permissão para efetuar login ou False caso contrário. Um valor de retorno False pode ser usado para contas desativadas.
is_anonymous()	Deve sempre retornar False para usuários regulares.
get_id()	Deve retornar um identificador exclusivo para o usuário, codificado como uma string Unicode.

Esses quatro métodos podem ser implementados diretamente como métodos na classe de modelo, mas como uma alternativa mais fácil, o Flask-Login fornece uma classe UserMixin que possui implementações padrão apropriadas para a maioria dos casos. O modelo de usuário atualizado é mostrado no [Exemplo 8-6](#).

*Exemplo 8-6. app/models.py: Atualizações no modelo de usuário para dar suporte a logins de usuários*

**de flask.ext.login importar UserMixin**

```
class User(UserMixin, db.Model):
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.String(64), unique=True, index=True)
    username = db.Column(db.String(64), unique=True, index=True)
    password_hash = db.Column(db.String(128))
    role_id = db.Column(db.Integer, db.ForeignKey('roles.id'))
```

Observe que um campo de email também foi adicionado. Neste aplicativo, os usuários farão login com seu e-mail, pois é menos provável que esqueçam seus endereços de e-mail do que seus nomes de usuário.

O Flask-Login é inicializado na função de fábrica do aplicativo, conforme mostrado no [Exemplo 8-7](#).

*Exemplo 8-7. app/\_\_init\_\_.py: Inicialização do Flask-Login*

```
de flask.ext.login importar LoginManager

login_manager = LoginManager()
login_manager.session_protection = 'forte'
login_manager.login_view = 'auth.login'

def create_app(config_name):
    # ...
    login_manager.init_app(app) #
    ...


```

O atributo session\_protection do objeto LoginManager pode ser definido como Nenhum, 'básico' ou 'forte' para fornecer diferentes níveis de segurança contra adulteração de sessão do usuário. Com a configuração 'forte', o Flask-Login acompanhará o endereço IP do cliente e o agente do navegador e desconectará o usuário se detectar uma alteração. O atributo login\_view configura o terminal para a página de login. Lembre-se de que, como a rota de login está dentro de um blueprint, ela precisa ser prefixada com o nome do blueprint.

Finalmente, Flask-Login requer que o aplicativo configure uma função de retorno de chamada que carrega um usuário, dado o identificador. Esta função é mostrada no [Exemplo 8-8](#).

*Exemplo 8-8. app/models.py: função de retorno de chamada do carregador do usuário*

```
de . importar login_manager

@login_manager.user_loader
def load_user(user_id):
    return User.query.get(int(user_id))


```

A função de retorno de chamada do carregador de usuário recebe um identificador de usuário como uma string Unicode. O valor de retorno da função deve ser o objeto do usuário, se disponível, ou Nenhum, caso contrário.

### Protegendo Rotas Para

proteger uma rota para que ela possa ser acessada apenas por usuários autenticados, o Flask-Login fornece um decorador login\_required. Segue um exemplo de seu uso:

```
de flask.ext.login importação login_required

@app.route('/secret')
@login_required def
secret(): return 'Somente
usuários autenticados são permitidos!'


```

Se essa rota for acessada por um usuário não autenticado, o Flask-Login interceptará a solicitação e enviará o usuário para a página de login.

### Adicionando um formulário

O formulário de login que será apresentado aos usuários tem um campo de texto para o endereço de e-mail, um campo de senha, uma caixa de seleção “lembre-me” e um botão de envio. A classe de formulário Flask-WTF é mostrada no [Exemplo 8-9](#).

*Exemplo 8-9. app/auth/forms.py: formulário de login*

```
from flask.ext.wtf import Form from
wtforms import StringField, PasswordField, BooleanField, SubmitField from
wtforms.validators import Obrigatório, Email

class LoginForm(Form):
    email = StringField('Email', validators=[Required(), Length(1, 64),
                                             Email()])
    password = PasswordField('Password', validators=[Required()])
    remember_me = BooleanField('Mantenha- me conectado') submit =
    SubmitField('Log In')
```

O campo email aproveita os validadores Length() e Email() fornecidos pelo WTForms. A classe PasswordField representa um elemento <input> com type="password". A classe BooleanField representa uma caixa de seleção.

O modelo associado à página de login é armazenado em *auth/login.html*. Este modelo só precisa renderizar o formulário usando a macro wtf.quick\_form() do Flask-Bootstrap .

A [Figura 8-1](#) mostra o formulário de login renderizado pelo navegador da web.

A barra de navegação no modelo *base.html* usa uma condicional Jinja2 para exibir os links “Sign In” ou “Sign Out” dependendo do estado conectado do usuário atual. A condicional é mostrada no [Exemplo 8-10](#).

*Exemplo 8-10. app/templates/base.html: links da barra de navegação Entrar e Sair*

```
<ul class="nav navbar-nav navbar-right">
    {% if current_user.is_authenticated() %} <li><a
    href="{{ url_for('auth.logout') }}> Sair</a></li> {% else %} <li> <a
    href="{{ url_for('auth.login') }}>Fazer login </a></li> {% endif %} </ul>
```

A variável `current_user` usada na condicional é definida pelo Flask-Login e está automaticamente disponível para visualizar funções e templates. Essa variável contém o usuário conectado no momento ou um objeto de usuário anônimo proxy se o usuário não estiver conectado.

Objetos de usuários anônimos respondem ao método `is_authenticated()` com `False`, portanto, essa é uma maneira conveniente de saber se o usuário atual está logado.

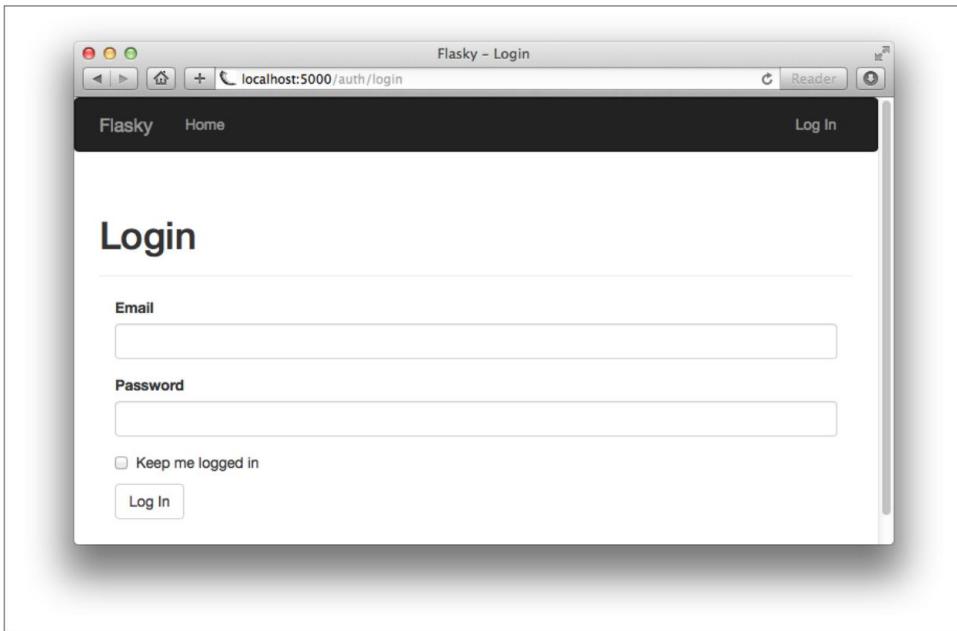


Figura 8-1. Forma de login

### Conectando usuários

A implementação da função de visualização login() é mostrada no [Exemplo 8-11](#).

*Exemplo 8-11. app/auth/views.py: rota de login*

```
de flask import render_template, redirect, request, url_for, flash
de flask.ext.login importar login_user
de . import auth
from ..models import User
from .forms import LoginForm
```

```
@auth.route('/login', métodos=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():

        user = User.query.filter_by(email=form.email.data).first()
        se user não for
        None e user.verify_password(form.password.data):
            login_user(user,
            form.remember_me.data)
            return redirect(request.args.get('next') ou
            url_for('main.index'))
            flash('Nome de usuário ou senha inválidos.')
            return
    render_template('auth/login.html', form=form)
```

A função view cria um objeto LoginForm e o utiliza como o formulário simples do [Capítulo 4](#). Quando a requisição é do tipo GET, a função view apenas renderiza o template, que

por sua vez, exibe o formulário. Quando o formulário é enviado em uma solicitação POST , a função validate\_on\_submit() do Flask-WTF valida as variáveis do formulário e tenta fazer o login do usuário.

Para fazer o login de um usuário, a função começa carregando o usuário do banco de dados usando o e-mail fornecido com o formulário. Se existir um usuário com o endereço de e-mail fornecido, seu método Verify\_password() será chamado com a senha que também veio com o formulário.

Se a senha for válida, a função login\_user() do Flask-Login é invocada para registrar o usuário como logado para a sessão do usuário. A função login\_user() leva o usuário para efetuar login e um booleano opcional “lembra-me”, que também foi enviado com o formulário.

Um valor False para este argumento faz com que a sessão do usuário expire quando a janela do navegador for fechada, de modo que o usuário terá que efetuar login novamente na próxima vez. Um valor True faz com que um cookie de longo prazo seja definido no navegador do usuário e com isso a sessão do usuário possa ser restaurada.

De acordo com o padrão Post/Redirect/Get discutido no [Capítulo 4](#), a solicitação POST que enviou as credenciais de login termina com um redirecionamento, mas há dois destinos de URL possíveis. Se o formulário de login foi apresentado ao usuário para impedir o acesso não autorizado a uma URL protegida, então o Flask-Login salvou a URL original no próximo argumento de string de consulta, que pode ser acessado a partir do dicionário request.args .

Se o próximo argumento de string de consulta não estiver disponível, um redirecionamento para a página inicial será emitido. Se o e-mail ou a senha fornecidos pelo usuário forem inválidos, uma mensagem flash será definida e o formulário será renderizado novamente para que o usuário tente novamente.



Em um servidor de produção, a rota de login deve ser disponibilizada em HTTP seguro para que os dados do formulário transmitidos ao servidor sejam encriptados. Sem HTTP seguro, as credenciais de login podem ser interceptadas durante o trânsito, anulando qualquer esforço de segurança de senhas no servidor.

O modelo de login precisa ser atualizado para renderizar o formulário. Essas mudanças são mostradas no [Exemplo 8-12.](#)

*Exemplo 8-12. app/templates/auth/login.html: Renderizar formulário de login*

```
{% extends "base.html" %} {%
import "bootstrap/wtf.html" como wtf %}

{% block title %}Flasky - Login{% endblock %}

{% block page_content %}
<div class="page-header">
<h1>Login</h1> </div>
<div class="col-md-4">
{{ wtf.quick_form(form) }}</div>
```

```
</div>
{%
  block final %}
```

## Desconectando usuários

A implementação da rota de logout é mostrada no [Exemplo 8-13](#).

*Exemplo 8-13. app/auth/views.py: rota de saída*

```
de flask.ext.login importar logout_user, login_required

@auth.route('/logout')
@login_required def
logout(): logout_user()
    flash('Você foi
desconectado.') return
    redirect(url_for('main.index'))
```

Para desconectar um usuário, a função `logout_user()` do Flask-Login é chamada para remover e redefinir a sessão do usuário. O logout é concluído com uma mensagem flash que confirma a ação e um redirecionamento para a página inicial.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar `git checkout 8c` para verificar esta versão do aplicativo.

Esta atualização contém uma migração de banco de dados, portanto, lembre-se de executar `python manage.py db upgrade` depois de verificar o código.

Para garantir que você tenha todas as dependências instaladas, execute também `pip install -r requirements.txt`.

## Testando logins

Para verificar se a funcionalidade de login está funcionando, a página inicial pode ser atualizada para saudar o usuário conectado pelo nome. A seção de modelo que gera a saudação é mostrada no [Exemplo 8-14](#).

*Exemplo 8-14. app/templates/index.html: Cumprimente o usuário conectado*

```
Olá,
{%
  if current_user.is_authenticated()
    {{ current_user.username }} {%
  else %}

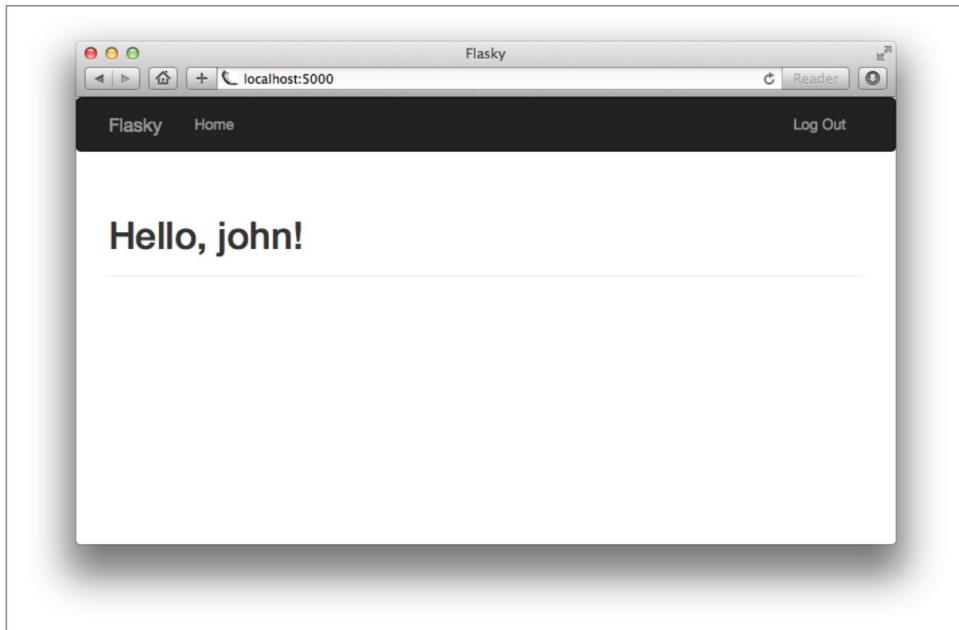
    Estranho
{%
  endif %}!
```

Neste modelo, mais uma vez, `current_user.is_authenticated()` é usado para determinar se o usuário está logado.

Como nenhuma funcionalidade de registro de usuário foi construída, um novo usuário pode ser registrado a partir do shell:

```
(venv) $ python manage.py shell >>> u
= User(email='john@example.com', username='john', password='cat') >>> db.session.add(u)
> >> db.session.commit()
```

O usuário criado anteriormente agora pode efetuar login. A [Figura 8-2](#) mostra a página inicial do aplicativo com o usuário conectado.



*Figura 8-2. Página inicial após login bem-sucedido*

## Registo de novo utilizador

Quando novos usuários desejam se tornar membros do aplicativo, eles devem se registrar nele para que sejam conhecidos e possam fazer login. Um link na página de login os enviará para uma página de registro, onde eles poderão inserir seu endereço de e-mail, nome de usuário, e senha.

### Adicionando um formulário de registro de

**usuário** O formulário que será usado na página de registro solicita que o usuário insira um endereço de e-mail, nome de usuário e senha. Este formulário é mostrado no [Exemplo 8-15](#).

*Exemplo 8-15. app/auth/forms.py: formulário de registro de usuário*

```
from flask.ext.wtf import Form from
wtforms import StringField, PasswordField, BooleanField, SubmitField
from wtforms.validators import Required, Length, Email, Regexp, EqualTo
```

```

de wtforms import ValidationError
de ..models import User

class RegistrationForm(Form):
    email = StringField('Email', validators=[Required(), Length(1, 64), Email()])
    username = StringField('Username',
                           validators=[ Required(), Length( 1, 64), Regexp('^[A-Za-z][A-Za-z0-9_.]*$', 0,
                           'Os nomes de usuário devem ter apenas
                           letras, 'números, pontos ou sublinhados')])
    password = PasswordField('Password', validators=[ Required(),
                           EqualTo('password2', message='Passwords must match.']])
    password2 = PasswordField('Confirm password', validators=[Required()])
    submit = SubmitField('Registrar')

    def validate_email(self, field):
        User.query.filter_by(email=field.data).first():
            raise ValidationError('Email já cadastrado.')

    def validate_username(self, field):
        User.query.filter_by(username=field.data).first():
            raise ValidationError('Username já em uso.')

```

Este formulário usa o validador Regexp do WTForms para garantir que o campo de nome de usuário contenha apenas letras, números, sublinhados e pontos. Os dois argumentos para o validador que seguem a expressão regular são os sinalizadores de expressão regular e a mensagem de erro a ser exibida em caso de falha.

A senha é digitada duas vezes como medida de segurança, mas esta etapa torna necessário validar que os dois campos de senha tenham o mesmo conteúdo, o que é feito com outro validador da WTForms chamado EqualTo. Este validador é anexado a um dos campos de senha com o nome do outro campo dado como argumento.

Este formulário também possui dois validadores personalizados implementados como métodos. Quando um formulário define um método com o prefixo validate\_ seguido pelo nome de um campo, o método é invocado além de quaisquer validadores definidos regularmente. Nesse caso, os validadores personalizados para e-mail e nome de usuário garantem que os valores fornecidos não sejam duplicados. Os validadores personalizados indicam um erro de validação lançando uma exceção ValidationError com o texto da mensagem de erro como argumento.

O template que apresenta este formato chama-se `/templates/auth/register.html`. Assim como o modelo de login, este também renderiza o formulário com `wtf.quick_form()`. A página de registro é mostrada na [Figura 8-3](#).

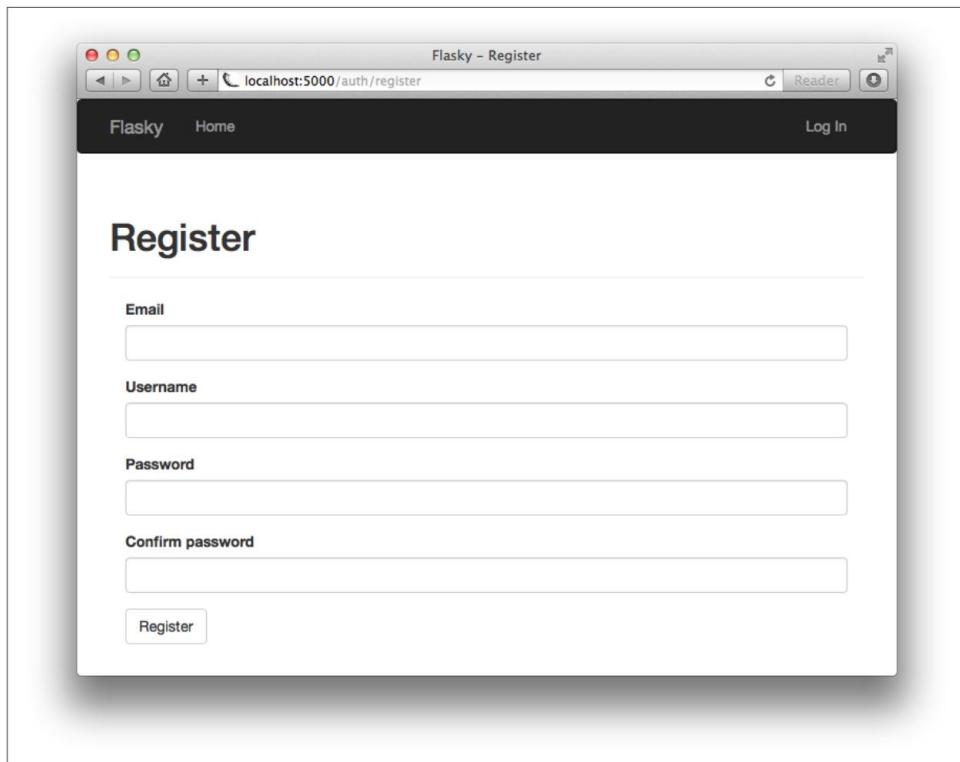


Figura 8-3. Formulário de cadastro de novo usuário

A página de registro precisa estar vinculada à página de login para que os usuários que não tenham uma conta possam encontrá-la facilmente. Essa mudança é mostrada no [Exemplo 8-16](#).

*Exemplo 8-16. app/templates/auth/login.html: Link para a página de registro*

```
<p>
    Novo usuário?
    <a href="{{ url_for('auth.register') }}> Clique aqui
        para se registrar </a> </p>
```

#### Cadastro de Novos Usuários O

tratamento de cadastros de usuários não traz grandes surpresas. Quando o formulário de registro é enviado e validado, um novo usuário é adicionado ao banco de dados usando as informações fornecidas pelo usuário. A função de visualização que executa esta tarefa é mostrada no [Exemplo 8-17](#).

*Exemplo 8-17. app/auth/views.py: rota de registro do usuário*

```
@auth.route('/register', métodos=['GET', 'POST']) def
register(): form = RegistrationForm() if form.validate_on_submit():
    user = User(email=form.email.data ,
    username=form.username.data,
    password=form.password.data) db.session.add(user)
    flash(' Agora você pode fazer login.')
    return redirect(url_for('auth.login'))
return render_template('auth/register.html',
form=form)
```



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar git checkout 8d para verificar esta versão do aplicativo.

## Confirmação da conta

Para determinados tipos de aplicativos, é importante garantir que as informações do usuário fornecidas durante o registro sejam válidas. Um requisito comum é garantir que o usuário possa ser contatado por meio do endereço de e-mail fornecido.

Para validar o endereço de e-mail, os aplicativos enviam um e-mail de confirmação aos usuários imediatamente após o registro. A nova conta é inicialmente marcada como não confirmada até que as instruções no e-mail sejam seguidas, o que prova que o usuário pode ser alcançado. O procedimento de confirmação da conta geralmente envolve clicar em um link de URL especialmente criado que inclui um token de confirmação.

### Gerando Tokens de Confirmação com seu perigo

O link de confirmação de conta mais simples seria um URL com o formato `http://www.example.com/auth/confirm/<id>` incluído no e-mail de confirmação, onde `id` é o id numérico atribuído ao usuário no banco de dados. Quando o usuário clica no link, a função de visualização que trata dessa rota recebe o ID do usuário para confirmar como argumento e pode atualizar facilmente o status confirmado do usuário.

Mas isso obviamente não é uma implementação segura, pois qualquer usuário que descobrir o formato dos links de confirmação poderá confirmar contas arbitrárias apenas enviando números aleatórios na URL. A ideia é substituir o `id` na URL por um token que contenha as mesmas informações criptografadas com segurança.

Se você se lembrar da discussão sobre sessões do usuário no [Capítulo 4](#), o Flask usa cookies assinados criptograficamente para proteger o conteúdo das sessões do usuário contra adulteração. Estes seguram

cookies são assinados por um pacote chamado *itsdangerous*. A mesma ideia pode ser aplicada aos tokens de confirmação.

A seguir está uma breve sessão de shell que mostra como seu perigo pode gerar um token que contém um ID de usuário dentro de:

```
(venv) $ python manage.py shell >>>
from manage import app >>> from
itsdangerous import TimedJSONWebSignatureSerializer as Serializer >>> s =
Serializer(app.config['SECRET_KEY'], expires_in = 3600) >>> token = s.dumps({'confirm': 23 })
>>> token 'eyJhbGciOiJIUzI1NilsImV4cCI6MTM4MTcxODU1OCwiaWF0IjoxMzgxNzE0OTU4fQ.eyJ ...
>>> data = s.loads(token) >>> data {'confirm': 23}
```

Itsdangerous fornece vários tipos de geradores de token. Entre eles, a classe TimedJSONWebSignatureSerializer gera JSON Web Signatures (JWS) com tempo de expiração. O construtor desta classe recebe como argumento uma chave de criptografia, que em uma aplicação Flask pode ser a SECRET\_KEY configurada.

O método dumps() gera uma assinatura criptográfica para os dados fornecidos como argumento e então serializa os dados mais a assinatura como uma string de token conveniente. O argumento expires\_in define um tempo de expiração para o token expresso em segundos.

Para decodificar o token, o objeto serializador fornece um método load() que recebe o token como seu único argumento. A função verifica a assinatura e o prazo de validade e, caso seja válido, retorna os dados originais. Quando o método load() recebe um token inválido ou um token válido que expirou, uma exceção é lançada.

A geração e verificação de token usando essa funcionalidade podem ser adicionadas ao modelo de usuário . As mudanças são mostradas no [Exemplo 8-18](#).

*Exemplo 8-18. app/models.py: confirmação da conta do usuário*

```
from itsdangerous import TimedJSONWebSignatureSerializer as Serializer from flask
import current_app from . importar banco de dados
```

```
class User(UserMixin, db.Model ):
    ...
    # confirmado = db.Column(db.Boolean, default=False)

    def generate_confirmation_token(self, expiração=3600):
        s = Serializer(current_app.config['SECRET_KEY'], expiração) return
        s.dumps({'confirm': self.id})

    def confirm(self, token):
        s = Serializer(current_app.config['SECRET_KEY']) tente:
```

```

dados = s.loads(token)
exceto:
    retornar False
if data.get('confirm') != self.id:
    return False
self.confirmed = True
db.session.add(self)
retornar Verdadeiro

```

O método generate\_confirmation\_token() gera um token com um tempo de validade padrão de uma hora. O método confirm() verifica o token e, se válido, define o novo atributo confirmado como True.

Além de verificar o token, a função confirm() verifica se o id do token corresponde ao usuário logado, que está armazenado em current\_user. Isso garante que, mesmo que um usuário mal-intencionado descubra como gerar tokens assinados, ele não poderá confirmar a conta de outra pessoa.



Como uma nova coluna foi adicionada ao modelo para rastrear o estado confirmado de cada conta, uma nova migração de banco de dados precisa ser gerada e aplicada.

Os dois novos métodos adicionados ao modelo User são facilmente testados em testes unitários. Você pode encontrar os testes de unidade no repositório do GitHub para o aplicativo.

#### **Enviando e-mails de confirmação**

A rota /register atual redireciona para /index após adicionar o novo usuário ao banco de dados.

Antes de redirecionar, esta rota agora precisa enviar o e-mail de confirmação. Essa mudança é mostrada no Exemplo 8-19.

*Exemplo 8-19. app/auth/views.py: rota de registro com e-mail de confirmação*

```
from ..email importar send_email
```

```
@auth.route('/register', métodos = ['GET', 'POST']) def register():
form = RegistrationForm() if form.validate_on_submit():
```

```

# ...
db.session.add(user)
db.session.commit()
token = user.generate_confirmation_token()
send_email(user.email, 'Confirme sua conta', 'auth/email/
    confirm', user=user, token=token) flash('Um e-mail de
confirmação foi enviado para você por e-mail.')

```

```
    return redirect(url_for('main.index')) return
    render_template('auth/register.html', form=form)
```

Observe que uma chamada db.session.commit() teve que ser adicionada, mesmo que o aplicativo tenha configurado confirmações automáticas de banco de dados no final da solicitação. O problema é que novos usuários recebem um id quando estão comprometidos com o banco de dados. Como o id é necessário para o token de confirmação, a confirmação não pode ser atrasada.

Os modelos de email usados pelo blueprint de autenticação serão adicionados na pasta `templates/auth/email` para mantê-los separados dos modelos HTML. Conforme discutido no [Capítulo 6](#), para cada e-mail são necessários dois modelos para as versões de texto simples e rich text do corpo. Como exemplo, o [Exemplo 8-20](#) mostra a versão em texto simples do modelo de e-mail de confirmação e você pode encontrar a versão HTML equivalente no repositório do GitHub.

*Exemplo 8-20. app/auth/templates/auth/email/confirm.txt: corpo do texto do e-mail de confirmação*

Caro {{ user.username }},

Bem-vindo ao Flasky!

Para confirmar sua conta, clique no link a seguir:

```
{{ url_for('auth.confirm', token=token, _external=True) }}
```

Sinceramente,

A equipe Flask

Observação: as respostas a este endereço de e-mail não são monitoradas.

Por padrão, `url_for()` gera URLs relativos, então, por exemplo, `url_for('auth.confirm', token='abc')` retorna a string '/auth/confirm/abc'. Isso, é claro, não é um URL válido que pode ser enviado por e-mail. Os URLs relativos funcionam bem quando são usados no contexto de uma página da Web porque o navegador os converte em absolutos adicionando o nome do host e o número da porta da página atual, mas ao enviar um URL por e-mail não existe esse contexto. O argumento `_external=True` é adicionado à chamada `url_for()` para solicitar uma URL totalmente qualificada que inclui o esquema (`http://` ou `https://`), nome do host e porta.

A função de visualização que confirma as contas é mostrada no [Exemplo 8-21](#).

*Exemplo 8-21. app/auth/views.py: confirme uma conta de usuário*

```
from flask import Blueprint, flash, redirect, render_template, url_for
from flask.ext.login import current_user
from .forms import LoginForm, RegistrationForm, ResetPasswordForm, \
    ChangePasswordForm, DeleteAccountForm
from .models import User
from .utils import flash_errors
```

```
def confirm(token): if
    current_user.confirmed: return
        redirect(url_for('main.index')) if
    current_user.confirm(token): flash('Você confirmou'
        sua conta. Obrigado!') else: flash('O link de confirmação é'
        inválido ou expirou.') return redirect(url_for('main.index'))
```

Esta rota é protegida com o decorador `login_required` do Flask-Login, para que quando os usuários clicarem no link do e-mail de confirmação eles sejam solicitados a fazer login antes de chegarem a esta função de visualização.

A função primeiro verifica se o usuário logado já está confirmado e, nesse caso, redireciona para a página inicial, pois obviamente não há nada a fazer. Isso pode evitar trabalho desnecessário se um usuário clicar no token de confirmação várias vezes por engano.

Como a confirmação real do token é feita inteiramente no modelo `User`, tudo o que a função view precisa fazer é chamar o método `confirm()` e, em seguida, exibir uma mensagem de acordo com o resultado. Quando a confirmação for bem-sucedida, o atributo `confirmed` do modelo `User` é alterado e adicionado à sessão, que será confirmada quando a solicitação terminar.

Cada aplicativo pode decidir o que os usuários não confirmados podem fazer antes de confirmar sua conta. Uma possibilidade é permitir que usuários não confirmados façam login, mas apenas mostre a eles uma página que solicita que eles confirmem suas contas antes que possam obter acesso.

Essa etapa pode ser feita usando o gancho `before_request` do Flask, que foi brevemente descrito no [Capítulo 2](#). De um blueprint, o gancho `before_request` se aplica apenas a solicitações que pertencem ao blueprint. Para instalar um gancho para todas as solicitações de aplicativo de um blueprint, o decorador `before_app_request` deve ser usado. [O Exemplo 8-22](#) mostra como esse manipulador é implementado.

*Exemplo 8-22. app/auth/views.py: filtre contas não confirmadas no manipulador before\_app\_request*

```
@auth.before_app_request
def before_request(): se
    current_user.is_authenticated() \ e não
        current_user.confirmed \ e
        request.endpoint[:5] != 'auth.': return
    redirect(url_for('auth.unconfirmed'))

@auth.route('/unconfirmed') def
unconfirmed(): se
    current_user.is_anonymous() ou current_user.confirmed:
        return redirect('main.index') return
    render_template('auth/unconfirmed.html')
```

O manipulador before\_app\_request interceptará uma solicitação quando três condições forem verdadeiro:

1. Um usuário está logado (`current_user.is_authenticated()` deve retornar True).
  2. A conta do usuário não está confirmada.
  3. O endpoint solicitado (acessível como `request.endpoint`) está fora do blueprint de autenticação.
- O acesso às rotas de autenticação precisa ser concedido, pois essas são as rotas que permitirão ao usuário confirmar a conta ou executar outras funções de gerenciamento de conta.

Se as três condições forem atendidas, um redirecionamento será emitido para uma nova rota `/auth/unconfirmed` que mostra uma página com informações sobre a confirmação da conta.



Quando um callback `before_request` ou `before_app_request` retorna uma resposta ou um redirecionamento, o Flask envia isso para o cliente sem invocar a função view associada à solicitação. Isso efetivamente permite que esses retornos de chamada interceptem uma solicitação quando necessário.

A página apresentada a usuários não confirmados (mostrada na [Figura 8-4](#)) apenas renderiza um modelo que fornece aos usuários instruções sobre como confirmar sua conta e oferece um link para solicitar um novo e-mail de confirmação, caso o e-mail original tenha sido perdido. A rota que reenvia o e-mail de confirmação é mostrada no [Exemplo 8-23](#).

*Exemplo 8-23. app/auth/views.py: reenviar e-mail de confirmação da conta*

```
@auth.route('/confirm')
@login_required def
resend_confirmation():
    token = current_user.generate_confirmation_token()
    send_email('auth/email/confirm', 'Confirme sua conta', user,
               token=token) flash('Um novo e-mail de confirmação
foi enviado a você por e-mail.')
    return redirect(url_for('principal.index'))
```

Essa rota repete o que foi feito na rota de registro usando `current_user`, o usuário que está logado, como usuário de destino. Essa rota também é protegida com `login_required` para garantir que, ao ser acessada, o usuário que está fazendo a solicitação seja conhecido.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar `git checkout 8e` para verificar esta versão do aplicativo. Esta atualização contém uma migração de banco de dados, portanto, lembre-se de executar `python manage.py db upgrade` depois de verificar o código.

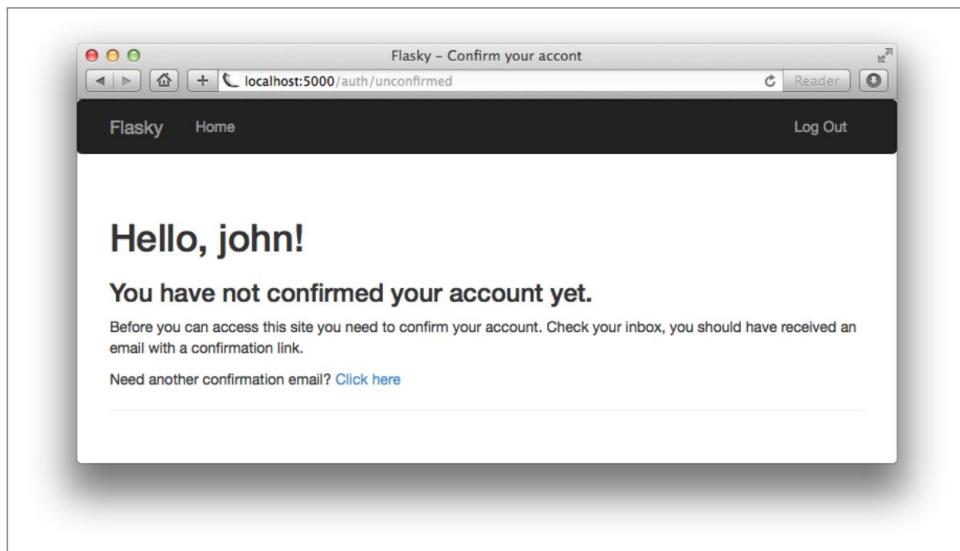


Figura 8-4. Página de conta não confirmada

## Gerenciamento de contas

Os usuários que possuem contas no aplicativo podem precisar fazer alterações em suas contas de tempos em tempos. As seguintes tarefas podem ser adicionadas ao blueprint de autenticação usando as técnicas apresentadas neste capítulo: *Atualizações de senha* Usuários preocupados com segurança podem querer alterar suas senhas periodicamente. Este é um recurso de fácil implementação, pois desde que o usuário esteja logado, é seguro apresentar um formulário que solicita a senha antiga e uma nova senha para substituí-la.

(Esse recurso é implementado como commit 8f no repositório GitHub.)

### Redefinições de

*senha* Para evitar bloquear usuários do aplicativo quando eles esquecem suas senhas, uma opção de redefinição de senha pode ser oferecida. Para implementar redefinições de senha de maneira segura, é necessário usar tokens semelhantes aos usados para confirmar contas. Quando um usuário solicita uma redefinição de senha, um e-mail com um token de redefinição é enviado para o endereço de e-mail registrado. O usuário então clica no link do e-mail e, após a verificação do token, é apresentado um formulário onde uma nova senha pode ser inserida. (Esse recurso é implementado como commit 8g no repositório GitHub.)

### Alterações de endereço

*de e-mail* Os usuários podem ter a opção de alterar o endereço de e-mail registrado, mas antes que o novo endereço seja aceito, ele deve ser verificado com um e-mail de confirmação. Para usar isso

recurso, o usuário insere o novo endereço de e-mail em um formulário. Para confirmar o endereço de e-mail, um token é enviado por e-mail para esse endereço. Quando o servidor recebe o token de volta, ele pode atualizar o objeto de usuário. Enquanto o servidor espera para receber o token, ele pode armazenar o novo endereço de e-mail em um novo campo de banco de dados reservado para endereços de e-mail pendentes ou pode armazenar o endereço no token junto com o id. (Este recurso é implementado como commit 8h no repositório GitHub.)

No próximo capítulo, o subsistema de usuário do Flasky será estendido através do uso de funções de usuário.

## CAPÍTULO 9

### Funções do usuário

Nem todos os usuários de aplicativos da web são criados iguais. Na maioria dos aplicativos, uma pequena porcentagem de usuários é confiável com poderes extras para ajudar a manter o aplicativo funcionando sem problemas. Os administradores são o melhor exemplo, mas em muitos casos também existem usuários avançados de nível médio, como moderadores de conteúdo.

Há várias maneiras de implementar funções em um aplicativo. O método apropriado depende em grande parte de quantos papéis precisam ser apoiados e quanto elaborados eles são.

Por exemplo, um aplicativo simples pode precisar de apenas duas funções, uma para usuários comuns e outra para administradores. Nesse caso, ter um campo booleano `is_administrator` no modelo User pode ser tudo o que é necessário. Um aplicativo mais complexo pode precisar de funções adicionais com níveis variados de poder entre usuários regulares e administradores.

Em algumas aplicações, pode até não fazer sentido falar sobre funções discretas; em vez disso, dar aos usuários uma combinação de *permissões* pode ser a abordagem correta.

A implementação de função de usuário apresentada neste capítulo é um híbrido entre funções e permissões discretas. Os usuários recebem uma função discreta, mas as funções são definidas em termos de permissões.

#### Representação do banco de dados de funções

Uma tabela de funções simples foi criada no [Capítulo 5](#) como um veículo para demonstrar relacionamentos um-para-muitos. O [Exemplo 9-1](#) mostra um modelo de papel aprimorado com algumas adições.

*Exemplo 9-1. app/models.py: permissões de função*

```
class Role(db.Model):
    __tablename__ = 'roles'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(64), unique=True)
    default = db.Column(db.Boolean, default=False, index=True)
```

```
    permissões = db.Column(db.Integer)
    users = db.relationship('User', backref='role', lazy='dynamic')
```

O campo padrão deve ser definido como True para apenas uma função e False para todas as outras.

A função marcada como padrão será aquela atribuída aos novos usuários no momento do registro.

A segunda adição ao modelo é o campo de permissões , que é um número inteiro que ser usados como sinalizadores de bits. Cada tarefa será atribuída a uma posição de bit e, para cada função, as tarefas que são permitidos para essa função terão seus bits definidos como 1.

A lista de tarefas para as quais as permissões são necessárias é obviamente específica do aplicativo. Por Flasky, a lista de tarefas é mostrada na [Tabela 9-1](#).

*Tabela 9-1. Permissões do aplicativo*

Nome da tarefa	Valor do bit	Descrição
Seguir usuários	0b00000001 (0x01)	Siga outros usuários
Comentar posts feitos por outros	0b00000010 (0x02)	Comentar artigos escritos por outros
Escrever artigos	0b00000100 (0x04)	Escrever artigos originais
Moderar comentários feitos por outros	0b00001000 (0x08)	Suprimir comentários ofensivos feitos por outros
Acesso de administração	0b10000000 (0x80)	Acesso administrativo ao site

Observe que um total de oito bits foi alocado para tarefas e até agora apenas cinco foram usados.

Os três restantes são deixados para expansão futura.

A representação do código da [Tabela 9-1](#) é mostrada no [Exemplo 9-2](#).

*Exemplo 9-2. app/models.py: constantes de permissão*

**permissão de classe :**

```
SEGUIR = 0x01
COMENTÁRIO = 0x02
WRITE_ARTICLES = 0x04
MODERATE_COMMENTS = 0x08
ADMINISTRADOR = 0x80
```

A [Tabela 9-2](#) mostra a lista de funções de usuário que serão suportadas, juntamente com a permissão bits que o definem.

*Tabela 9-2. Funções do usuário*

Papel do usuário	Permissões	Descrição
Anônimo	0b00000000 (0x00)	Usuário que não está conectado. Acesso somente leitura ao aplicativo.
Do visitante	0b00000111 (0x07)	Permissões básicas para escrever artigos e comentários e seguir outros usuários. Isto é o padrão para novos usuários.
Moderador	0b00001111 (0x0f)	Adiciona permissão para suprimir comentários considerados ofensivos ou inapropriados.
Administrador	0b11111111 (0xff)	Acesso total, que inclui permissão para alterar as funções de outros usuários.

Organizar as funções com permissões permite adicionar novas funções no futuro que usam diferentes combinações de permissões.

Adicionar as funções ao banco de dados manualmente é demorado e propenso a erros. Em vez disso, um método de classe será adicionado à classe Role para essa finalidade, conforme mostrado no [Exemplo 9-3](#).

*Exemplo 9-3. app/models.py: crie funções no banco de dados*

```
class Role(db.Model):
    @staticmethod def
        insert_roles():
            roles = {
                'User': (
                    (Permission.FOLLOW
                     | Permission.COMMENT |
                         Permission.WRITE_ARTICLES, True),
                    'Moderator': (Permission.FOLLOW | Permission.
                        COMMENT |
                        Permission.WRITE_ARTICLES |
                        Permission.MODERATE_COMMENTS, False),
                    'Administrador': (0xff, False)
                ) para r em papéis:
                    role = Role.query.filter_by(name=r).first() se role for
                    None: role = Role(name=r) role.permissions = roles[r]
                    [0] role.default = roles[r][1] db.session.add(role)
                    db.session.commit()
```

A função insert\_roles() não cria diretamente novos objetos de função. Em vez disso, ele tenta localizar as funções existentes por nome e atualizá-las. Um novo objeto de função é criado apenas para nomes de função que ainda não estão no banco de dados. Isso é feito para que a lista de funções possa ser atualizada no futuro quando for necessário fazer alterações. Para adicionar uma nova função ou alterar as atribuições de permissão para uma função, altere a matriz de funções e execute novamente a função. Observe que a função “Anônimo” não precisa ser representada no banco de dados, pois foi projetada para representar usuários que não estão no banco de dados.

Para aplicar essas funções ao banco de dados, uma sessão de shell pode ser usada:

```
(venv) $ python manage.py shell >>>
Role.insert_roles()
>>> Role.query.all()
[<Função u'Administrador', <Função u'Usuário', <Função u'Moderador']
```

## Atribuição de Função

Quando os usuários registram uma conta no aplicativo, a função correta deve ser atribuída a eles. Para a maioria dos usuários, a função atribuída no momento do registro será a função “Usuário”, conforme

essa é a função marcada como uma função padrão. A única exceção é feita para o administrador, que precisa ser atribuído à função “Administrador” desde o início. Este usuário é identificado por um endereço de e-mail armazenado na variável de configuração FLASKY\_ADMIN , portanto, assim que esse endereço de e-mail aparecer em uma solicitação de registro, ele poderá receber a função correta. O Exemplo 9-4 mostra como isso é feito no construtor do modelo User .

*Exemplo 9-4. app/models.py: defina uma função padrão para usuários*

```
class User(UserMixin, db.Model):
    # ...
    def __init__(self, **kwargs):
        super(User, self).__init__(**kwargs) se
        self.role for Nenhum:
            if self.email == current_app.config['FLASKY_ADMIN']:
                self.role = Role.query.filter_by(permissions=0xff).first()
            se self.role for Nenhum:
                self.role = Role.query.filter_by(default=True).first()
        # ...
```

O construtor User primeiro invoca os construtores das classes base, e se depois disso o objeto não tiver uma função definida, ele define as funções de administrador ou padrão dependendo do endereço de e-mail.

## Verificação de função

Para simplificar a implementação de funções e permissões, um método auxiliar pode ser adicionado ao modelo User que verifica se uma determinada permissão está presente, conforme mostrado no Exemplo 9-5.

*Exemplo 9-5. app/models.py: avalia se um usuário tem uma determinada permissão*

```
de flask.ext.login importar UserMixin, AnonymousUserMixin

class User(UserMixin, db.Model):
    # ...

    def can(self, permissions): return
        self.role não é None e \
            (self.role.permissions
            & permissions) == permissions

    def is_administrator(self): return
        self.can(Permission.ADMINISTER)

class AnonymousUser(AnonymousUserMixin):
    def can(self, permissions): return False

    def is_administrator(self): return
        False
```

```
login_manager.anonymous_user = AnonymousUser
```

O método can() adicionado ao modelo User executa uma *operação bit a bit* entre as permissões solicitadas e as permissões da função atribuída. O método retorna True se todos os bits solicitados estiverem presentes na função, o que significa que o usuário deve ter permissão para executar a tarefa. A verificação de permissões de administração é tão comum que também é implementada como um método is\_administrator() autônomo.

Para consistência, uma classe AnonymousUser personalizada que implementa os métodos can() e is\_administrator() é criada. Este objeto herda da classe AnonymousUserMixin do Flask-Login e é registrado como a classe do objeto que é atribuído a current\_user quando o usuário não está logado. Isso permitirá que o aplicativo chame livremente current\_user.can() e current\_user.is\_administrator() sem ter que verificar primeiro se o usuário está logado.

Para casos em que uma função de visualização inteira precisa ser disponibilizada apenas para usuários com determinadas permissões, um decorador personalizado pode ser usado. O [Exemplo 9-6](#) mostra a implementação de dois decoradores, um para verificações de permissões genéricas e outro que verifica especificamente para permissões de administrador.

*Exemplo 9-6. app/decorators.py: decoradores personalizados que verificam as permissões do usuário*

```
from functools import wraps
from flask import abort
from flask.ext.login import current_user

def permission_required(permission):
    decorator(f): @wraps(f) def
        decorated_function(*args, **kwargs):
            se não current_user.can(permission): abort(403)
            return f(*args, **kwargs)

    retornar decorated_function
    retornar decorator

def admin_required(f):
    return permission_required(Permission.ADMINISTER)(f)
```

Esses decoradores são construídos com a ajuda do pacote *functools* da biblioteca padrão do Python e retornam um código de erro 403, o erro HTTP “Proibido”, quando o usuário atual não possui as permissões solicitadas. No [Capítulo 3](#), páginas de erro personalizadas foram criadas para os erros 404 e 500, então agora uma página para o erro 403 também precisa ser adicionada.

A seguir estão dois exemplos que demonstram o uso desses decoradores:

```

de decoradores import admin_required, permission_required

@main.route('/admin')
@login_required
@admin_required def
for_admins_only(): return
    "Para administradores!"

@main.route('/moderator')
@login_required
@permission_required(Permission.MODERATE_COMMENTS)
def for_moderators_only(): return "Para moderadores de
    comentários!"

```

As permissões também podem precisar ser verificadas a partir de modelos, portanto, a classe Permission com todas as constantes de bits precisa estar acessível a elas. Para evitar ter que adicionar um argumento de modelo em cada chamada render\_template() , um processador de contexto pode ser usado. Os processadores de contexto tornam as variáveis globalmente disponíveis para todos os modelos. Essa mudança é mostrada no Exemplo 9-7.

*Exemplo 9-7. app/main/\_\_init\_\_.py: Adicionando a classe Permission ao template cony texto*

```

@app.context_processor
def inject_permissions(): return
    dict(Permission=Permission)

```

As novas funções e permissões podem ser exercidas em testes de unidade. O Exemplo 9-8 mostra dois testes simples que também servem como demonstração do uso.

*Exemplo 9-8. tests/test\_user\_model.py: testes de unidade para funções e permissões*

```

class UserModelTestCase(unittest.TestCase):
    # ...

    def test_roles_and_permissions(self):
        Role.insert_roles()
        u =
            User(email='john@example.com', password='cat')
        self.assertTrue(u.can(Permission.WRITE_ARTICLES))
        self.assertFalse(u.can(Permission.MODERATE_COMMENTS))

    def test_anonymous_user(self):
        u =
            AnonymousUser()
        self.assertFalse(u.can(Permission.FOLLOW))

```



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar git checkout 9a para verificar esta versão do aplicativo.  
Esta atualização contém uma migração de banco de dados, portanto, lembre-se de executar python manage.py db upgrade depois de verificar o código.

Antes de passar para o próximo capítulo, é uma boa ideia recriar ou atualizar o banco de dados de desenvolvimento para que todas as contas de usuário que foram criadas antes da existência das funções e permissões tenham uma função atribuída.

O sistema do usuário agora está bastante completo. O próximo capítulo fará uso dele para criar páginas de perfil de usuário.



## CAPÍTULO 10

# Perfis de usuário

Neste capítulo, os perfis de usuário para Flasky são implementados. Todos os sites socialmente conscientes dão aos seus usuários uma página de perfil, onde é apresentado um resumo da participação do usuário no site. Os usuários podem anunciar sua presença no site compartilhando o URL em sua página de perfil, por isso é importante que os URLs sejam curtos e fáceis de lembrar.

## Informação do Perfil

Para tornar as páginas de perfil do usuário mais interessantes, algumas informações adicionais sobre os usuários podem ser registradas. No [Exemplo 10-1](#), o modelo User é estendido com vários novos campos.

*Exemplo 10-1. app/models.py: campos de informações do usuário*

```
class User(UserMixin, db.Model):
    # ...
    nome = db.Column(db.String(64)) local
    = db.Column(db.String(64)) about_me =
    db.Column(db.Text()) member_since =
    db.Column(db.DateTime(), default=datetime.utcnow) last_seen =
    db.Column(db.DateTime(), default=datetime.utcnow)
```

Os novos campos armazenam o nome real do usuário, localização, descrição auto-escrita, data de registro e data da última visita. O campo about\_me é atribuído ao tipo db.Text().

A diferença entre db.String e db.Text é que db.Text não precisa de um comprimento máximo.

Os dois carimbos de data/hora recebem um valor padrão da hora atual. Observe que datetime.utcnow está faltando o () no final. Isso ocorre porque o argumento padrão para db.Column() pode receber uma função como um valor padrão, portanto, cada vez que um valor padrão precisa ser gerado, a função é invocada para produzi-lo. Esse valor padrão é tudo o que é necessário para gerenciar o campo member\_since .

O campo `last_seen` também é inicializado com a hora atual na criação, mas precisa ser atualizado toda vez que o usuário acessar o site. Um método na classe `User` pode ser adicionado para realizar esta atualização. Isso é mostrado no [Exemplo 10-2](#).

*Exemplo 10-2. app/models.py: atualiza o horário da última visita de um usuário*

```
class User(UserMixin, db.Model):
    # ...
```

```
def ping(self):
    self.last_seen = datetime.utcnow()
    db.session.add(self)
```

O método `ping()` deve ser chamado toda vez que uma solicitação do usuário for recebida. Como o manipulador `before_app_request` no blueprint de autenticação é executado antes de cada solicitação, ele pode fazer isso facilmente, conforme mostrado no [Exemplo 10-3](#).

*Exemplo 10-3. app/auth/views.py: ping do usuário logado*

```
@auth.before_app_request
def before_request():
    current_user.is_authenticated():
        current_user.ping() if not
        current_user.confirmed \ e
            request.endpoint[:5] != 'auth.': return
            redirect(url_for('auth. não confirmado'))
```

## Página de perfil do usuário

Criar uma página de perfil para cada usuário não apresenta novos desafios. O [Exemplo 10-4](#) mostra a definição da rota.

*Exemplo 10-4. app/main/views.py: rota da página de perfil*

```
@main.route('/user/<username>') def
user(username): user =
    User.query.filter_by(username=username).first() se user for None:
        abort(404) return render_template('user .html', usuário=usuário)
```

Esta rota é adicionada no blueprint principal. Para um usuário chamado john, a página de perfil estará em `http://localhost:5000/user/john`. O nome de usuário fornecido na URL é pesquisado no banco de dados e, se encontrado, o modelo `user.html` é renderizado com ele como argumento. Um nome de usuário inválido enviado para esta rota fará com que um erro 404 seja retornado. O `usuário.html`

template deve renderizar as informações armazenadas no objeto de usuário. Uma versão inicial deste modelo é mostrada no [Exemplo 10-5](#).

*Exemplo 10-5. app/templates/user.html: modelo de perfil de usuário*

```

{%
block page_content %}

<div class="page-header">
  <h1>{{ user.username }}</h1> {%
    if user.name ou user.location %}
    <p>
      {%
        if user.name %}{{ user.name }}{%
        endif %} {%
        if user.location %}
          De <a href="http://maps.google.com/?q={{ user.location }}">
            {{ user.location }} </a>
          {%
            endif %} </p> {%
            endif %}
        {%
          if current_user.is_administrator()
        %} <p><a href="mailto:
          {{ user.email }}">{{ user.email }}</a></
        p> {%
          endif %} {%
          if user.about_me %}
        <p>{{ user.about_me }}</p>{%
          endif %}

<p>
  Membro desde {{ moment(user.member_since).format('L') }}.
  Visto pela última vez {{ moment(user.last_seen).fromNow() }}.
</p> </div> {%
  bloco final %}

```

Este modelo tem alguns detalhes de implementação interessantes:

- Os campos name e location são renderizados dentro de um único elemento `<p>`. Somente quando pelo menos um dos campos é definido é que o elemento `<p>` é criado.
- O campo de localização do usuário é renderizado como um link para uma consulta do Google Maps. • Se o usuário conectado for um administrador, os endereços de e-mail serão exibidos, renderizados como um link `mailto`.

Como a maioria dos usuários deseja acesso fácil à sua própria página de perfil, um link para ela pode ser adicionado à barra de navegação. As alterações relevantes no modelo `base.html` são mostradas no [Exemplo 10-6](#).

*Exemplo 10-6. app/templates/base.html*

```

{%
  if current_user.is_authenticated() %} <li> <a
  href="{{ url_for('main.user',
    username=current_user.username) }}">
    Perfil <
  a> </li> {%
  endif %}

```

O uso de uma condicional para o link da página de perfil é necessário porque a barra de navegação também é renderizada para usuários não autenticados, caso em que o link do perfil é ignorado.

A [Figura 10-1](#) mostra a aparência da página de perfil no navegador. O novo link de perfil na barra de navegação também é exibido.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar git checkout 10a para verificar esta versão do aplicativo.

Esta atualização contém uma migração de banco de dados, portanto, lembre-se de executar python manage.py db upgrade depois de verificar o código.

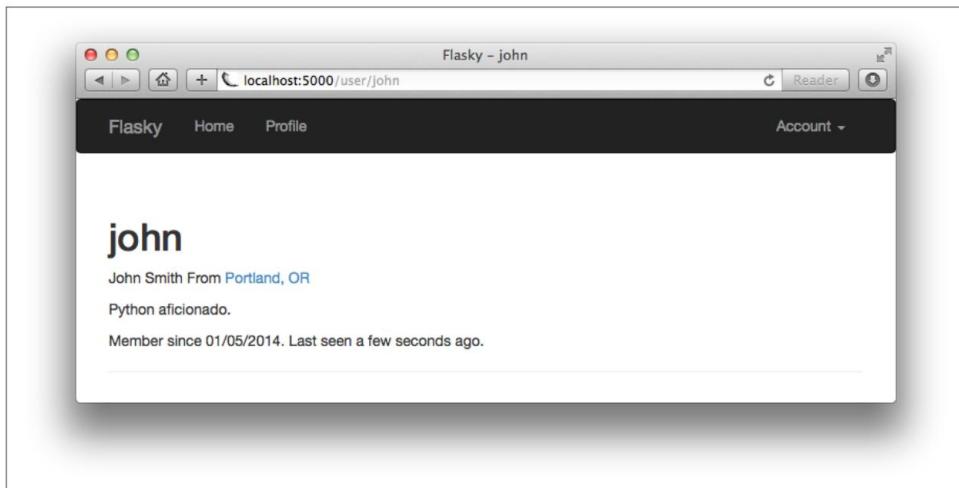


Figura 10-1. Página de perfil do usuário

## Editor de perfil

Existem dois casos de uso diferentes relacionados à edição de perfis de usuário. O mais óbvio é que os usuários precisam ter acesso a uma página onde possam inserir informações sobre si mesmos para apresentar em suas páginas de perfil. Um requisito menos óbvio, mas também importante, é permitir que os administradores editem o perfil de qualquer usuário – não apenas os itens de informações pessoais, mas também outros campos no modelo de usuário aos quais os usuários não têm acesso direto, como a função de usuário. Como os dois requisitos de edição de perfil são substancialmente diferentes, dois formulários diferentes serão criados.

### Editor de perfil de nível de usuário

O formulário de edição de perfil para usuários regulares é mostrado no [Exemplo 10-7](#).

*Exemplo 10-7. app/main/forms.py: formulário de edição de perfil*

**classe EditProfileForm (Formulário):**

```
name = StringField('Nome real', validators=[Length(0, 64)]) location =
StringField('Location', validators=[Length(0, 64)]) about_me = TextAreaField('Sobre
mim') submit = SubmitField('Enviar')
```

Observe que, como todos os campos deste formulário são opcionais, o validador de comprimento permite um comprimento igual a zero. A definição de rota que usa este formulário é mostrada no [Exemplo 10-8](#).

*Exemplo 10-8. app/main/views.py: rota de edição de perfil*

```
@main.route('/edit-profile', métodos=['GET', 'POST']) @login_required
def edit_profile(): form = EditProfileForm() if form.validate_on_submit():
current_user.name = form.name.data current_user.location =
form.location.data current_user.about_me = form.about_me.data
db.session.add(user) flash('Seu perfil foi atualizado.') return
redirect(url_for('.user', username =current_user.username))

form.name.data = current_user.name
form.location.data = current_user.location
form.about_me.data = current_user.about_me return
render_template('edit_profile.html', form=form)
```

Esta função de visualização define valores iniciais para todos os campos antes de apresentar o formulário. Para qualquer campo, isso é feito atribuindo o valor inicial a form.<field-name>.data. Quando form.validate\_on\_submit() é False, os três campos neste formulário são inicializados a partir dos campos correspondentes em current\_user. Então, quando o formulário é enviado, os atributos de dados dos campos do formulário contêm os valores atualizados, então eles são movidos de volta para os campos do objeto de usuário e o objeto é adicionado à sessão do banco de dados.

A [Figura 10-2](#) mostra a página Editar perfil.

Para facilitar o acesso dos usuários a esta página, um link direto pode ser adicionado na página de perfil, conforme mostrado no [Exemplo 10-9](#).

*Exemplo 10-9. app/templates/user.html: link de edição do perfil*

```
{% if user == current_user %} <a
class="btn btn-default" href="{{ url_for('.edit_profile') }}> Editar perfil </a> {% endif
%}
```

A condicional que inclui o link fará com que o link apareça apenas quando os usuários estiverem visualizando seus próprios perfis.

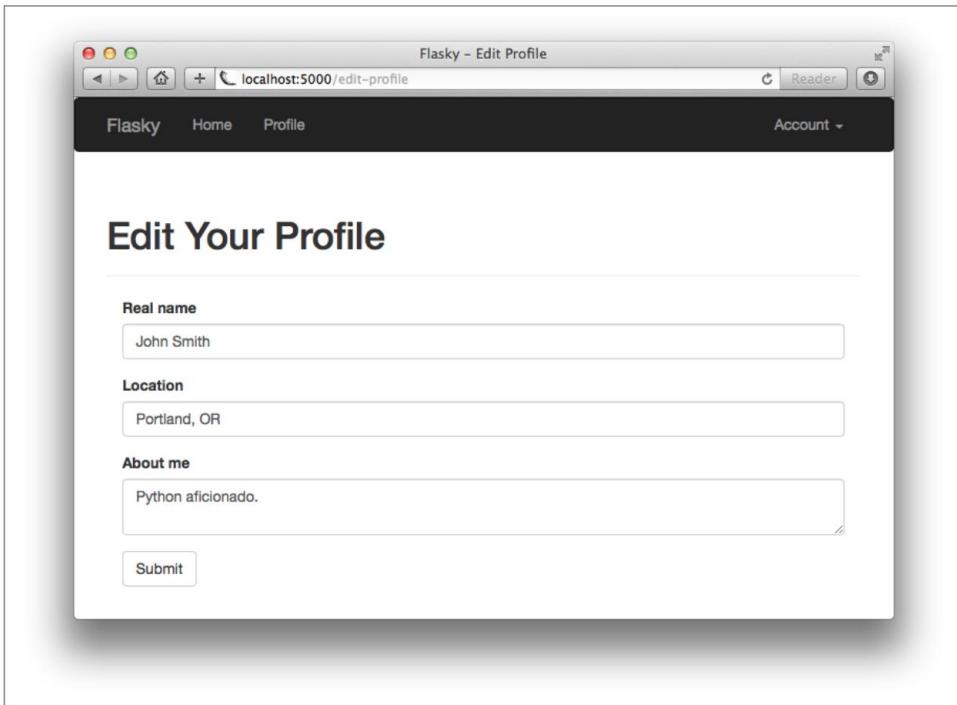


Figura 10-2. Editor de perfil

### Editor de perfil de nível de administrador

O formulário de edição de perfil para administradores é mais complexo do que o de usuários comuns. Além dos três campos de informações de perfil, este formulário permite que os administradores editem o e-mail, nome de usuário, status confirmado e função de um usuário. O formulário é mostrado no Exemplo 10-10.

*Exemplo 10-10. app/main/forms.py: formulário de edição de perfil para administradores*

```
class EditProfileAdminForm (Formulário):
    email = StringField('Email', validators=[Required(), Length(1, 64), Email()])
    nome_de_usuario = StringField('Username',
        validators=[ Required(), Length(1, 64), Regexp ('^A-Za-z][A-Za-z0-9_.]*$', 0,
            'Os nomes de usuário devem ter apenas
            letras, 'números, pontos ou sublinhados')])
    confirmado = BooleanField('Confirmed')
    role = SelectField('Role', coerce=int)
    name = StringField('Real name', validators=[Length(0, 64)])
    location = StringField('Location', validators= [Length(0, 64)])
    about_me =
    TextAreaField('Sobre mim')
    submit = SubmitField('Enviar')
```

```

def __init__(self, usuário, *args, **kwargs):
    super(EditProfileAdminForm, self).__init__(*args, **kwargs)
    self.role.choices = [(role.id, role.name) for role in Role.query.order_by(Role.name).all()]

    self.user = usuário

def validate_email(self, field):
    if field.data != self.user.email or \
        User.query.filter_by(email=field.data).first():
        raise ValidationError('Email já registrado.')

def validate_username(self, field):
    if field.data != self.user.username and \
        User.query.filter_by(username=field.data).first():
        raise ValidationError('Nome de usuário já em uso.')

```

O SelectField é o wrapper do WTForm para o controle de formulário HTML <select>, que implementa uma lista suspensa, usada neste formulário para selecionar uma função de usuário. Uma instância de SelectField deve ter os itens definidos em seu atributo de escolhas. Eles devem ser fornecidos como uma lista de tuplas, com cada tupla consistindo em dois valores: um identificador para o item e o texto a ser exibido no controle como uma string. A lista de opções é definida no construtor do formulário, com valores obtidos do Role model com uma consulta que classifica todos os papéis alfabeticamente por nome. O identificador de cada tupla é definido como o id de cada função e, como são inteiros, um argumento coerce =int é adicionado ao construtor SelectField para que os valores do campo sejam armazenados como inteiros em vez do padrão, que são strings.

Os campos de email e nome de usuário são construídos da mesma forma que nos formulários de autenticação, mas sua validação requer um tratamento cuidadoso. A condição de validação utilizada para ambos os campos deve primeiro verificar se foi feita uma alteração no campo, e somente quando houver alteração deve garantir que o novo valor não duplique o de outro usuário. Quando esses campos não são alterados, a validação deve passar. Para implementar essa lógica, o construtor do formulário recebe o objeto de usuário como argumento e o salva como uma variável de membro, que é posteriormente usada nos métodos de validação personalizados.

A definição de rota para o editor de perfil do administrador é mostrada no [Exemplo 10-11](#).

*Exemplo 10-11. app/main/views.py: rota de edição de perfil para administradores*

```

@main.route('/edit-profile/<int:id>', methods=['GET', 'POST']) @login_required
@admin_required def edit_profile_admin(id): user = User.query.get_or_404(id)
form = EditProfileAdminForm(user=user) if form.validate_on_submit():


```

```

user.email = form.email.data
user.username = form.username.data
user.confirmed = form.confirmed.data

```

```

user.role = Role.query.get(form.role.data) user.name
= form.name.data user.location = form.location.data

user.about_me = form.about_me.data
db.session.add(user) flash('O perfil foi
atualizado.') return redirect(url_for('.user',
username=user.username))
form.email.data = usuario.email
form.username.data = user.username
form.confirmed.data = user.confirmed
form.role.data = user.role_id form.name.data
= user.name form.location.data = user.location

form.about_me.data = user.about_me
return render_template ('edit_profile.html', form=form, user=user)

```

Esta rota tem basicamente a mesma estrutura que a mais simples para usuários regulares. Nesta função de visualização, o usuário é dado por seu id, então a função de conveniência get\_or\_404() do Flask-SQLAlchemy pode ser usada, sabendo que se o id for inválido a requisição retornará um erro de código 404.

O SelectField utilizado para o papel de usuário também merece ser estudado. Ao configurar o valor inicial para o campo, o role\_id é atribuído a field.role.data porque a lista de tuplas configurada no atributo options usa os identificadores numéricos para fazer referência a cada opção. Quando o formulário é enviado, o id é extraído do atributo data do campo e usado em uma consulta para carregar o objeto role pelo seu id. O argumento coerce =int usado na declaração SelectField no formulário garante que o atributo data deste campo seja

um inteiro.

Para vincular a esta página, outro botão é adicionado na página de perfil do usuário, conforme mostrado no [Exemplo 10-12](#).

*Exemplo 10-12. app/templates/user.html: link de edição de perfil para administrador*

```

{% if current_user.is_administrator() %} <a
class="btn btn-danger"
href="{{ url_for('.edit_profile_admin', id=user.id) }}> Editar perfil
[Admin] </a> {% endif %}

```

Este botão é renderizado com um estilo Bootstrap diferente para chamar a atenção para ele. A condicional neste caso faz com que o botão apareça nas páginas de perfil se o usuário logado for um administrador.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar git checkout 10b para verificar esta versão do aplicativo.

## Avatares de usuário

A aparência das páginas de perfil pode ser melhorada mostrando imagens de avatar dos usuários. Nesta seção, você aprenderá como adicionar avatares de usuários fornecidos pelo [Gravatar](#), o principal serviço de avatar. O Gravatar associa imagens de avatar a endereços de e-mail. Os usuários criam uma conta em <http://gravatar.com> e depois carregam suas imagens. Para gerar o URL do avatar para um determinado endereço de e-mail, seu hash MD5 é calculado:

```
(venv) $ python >>>
import hashlib >>>
hashlib.md5('john@example.com'.encode('utf-8')).hexdigest()
'd4c74594d841139328695756648b6bd6'
```

Os URLs de avatar são então gerados anexando o hash MD5 ao URL <http://www.gravatar.com/avatar/> ou <https://secure.gravatar.com/avatar/>. Por exemplo, você pode digitar <http://www.gravatar.com/avatar/d4c74594d841139328695756648b6bd6> na barra de endereço do seu navegador para obter a imagem do avatar para o endereço de e-mail john@example.com ou uma imagem gerada padrão se esse endereço de e-mail não ter um avatar registrado. A string de consulta da URL pode incluir vários argumentos que configuram as características da imagem do avatar, listadas na [Tabela 10-1](#).

*Tabela 10-1. Argumentos de string de consulta do Gravatar*

Nome do argumento	Descrição
s	Tamanho da imagem, em pixels.
r	Classificação da imagem. As opções são "g", "pg", "r" e "x".
d	O gerador de imagens padrão para usuários que não possuem avatares cadastrados no serviço Gravatar. As opções são "404" para retornar um erro 404, uma URL que aponta para uma imagem padrão ou um dos seguintes geradores de imagem: "mm", "identicon", "monsterid", "wavatar", "retro" ou "blank".
fd	Forçar o uso de avatares padrão.

O conhecimento de como construir uma URL Gravatar pode ser adicionado ao modelo User . A implementação é mostrada no [Exemplo 10-13](#).

*Exemplo 10-13. app/models.py: geração de URL do Gravatar*

```
importar hashlib
da solicitação de importação de frasco

class User(UserMixin, db.Model):
```

```

# ...
def gravatar(self, size=100, default='identicon', rating='g'): if request.is_secure:
    url = 'https://secure.gravatar.com/avatar' else: url = 'http:/ /
    www.gravatar.com/avatar'

hash = hashlib.md5(self.email.encode('utf-8')).hexdigest() return '{url}/
{hash}?s={size}&d={default}&r={rating}'. formato(
url = url, hash = hash, tamanho = tamanho, padrão = padrão, classificação = classificação)

```

Essa implementação seleciona a URL base padrão ou segura do Gravatar para corresponder à segurança da solicitação do cliente. A URL do avatar é gerada a partir da URL base, do hash MD5 do endereço de e-mail do usuário e dos argumentos, todos com valores padrão.

Com esta implementação, é fácil gerar URLs de avatar no shell do Python:

```
(venv) $ python manage.py shell >>> u
= User (email='john@example.com ') >>>
u.gravatar() ' http://www.gravatar.com/avatar/
d4c74594d84113932869575bd6?s = 100 & d = identicon & r = g ' >>> u.gravatar (tamanho = 256) '
http://www.gravatar.com/avatar/d4c74594d84113932869575bd6?s=256&d=identicon&r=g '
```

O método gravatar() também pode ser invocado a partir de templates Jinja2. O [Exemplo 10-14](#) mostra como um avatar de 256 pixels pode ser adicionado à página de perfil.

*Exemplo 10-14. app/tempaltes/user.html: Avatar na página de perfil*

```
...

    {% if current_user.can(Permission.WRITE_ARTICLES) %}
        {{ wtf.quick_form(form) }}
    {% endif %}
```

```

</div>
<ul class="posts"> {%
    para postagem em postagens
%} <li class="post"> <div
        class="profile-thumbnail">
            <a href="{{ url_for('.user', username=post.author.username) }}>
                {{ moment(post.timestamp).fromNow() }}</div> <div
        class="post-author">
            <a href="{{ url_for('.user', username=post.author.username) }}>
                {{ post.author.username }} </a> </div> <div class="post-body">{{ post.body }}</div> </li> {% endfor %}</ul>
...

```

Observe que o método User.can() é usado para ignorar o formulário de postagem do blog para usuários que não têm a permissão WRITE\_ARTICLES em sua função. A lista de posts do blog é implementada como uma lista HTML não ordenada, com classes CSS dando-lhe uma formatação mais agradável. Um pequeno avatar do autor é renderizado no lado esquerdo, e tanto o avatar quanto o nome de usuário do autor são renderizados como links para a página de perfil do usuário. Os estilos CSS usados são armazenados em um arquivo `styles.css` na pasta `estática` do aplicativo . Você pode revisar este arquivo no repositório do GitHub. A [Figura 11-1](#) mostra a página inicial com o formulário de envio e a lista de postagens do blog.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar `git checkout 11a` para verificar esta versão do aplicativo.  
Esta atualização contém uma migração de banco de dados, portanto, lembre-se de executar `python manage.py db upgrade` depois de verificar o código.

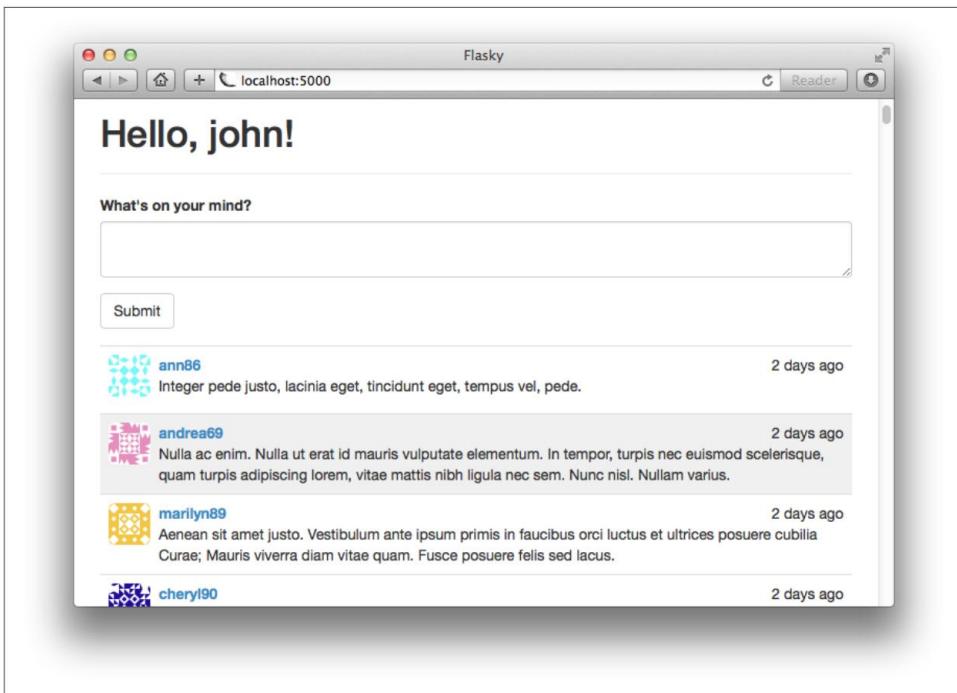


Figura 11-1. Página inicial com formulário de envio de blog e lista de postagens de blog

## Postagens de blog em páginas de perfil

A página de perfil do usuário pode ser melhorada mostrando uma lista de postagens de blog de autoria do usuário. O [Exemplo 11-5](#) mostra as alterações na função de visualização para obter a lista de postagens.

*Exemplo 11-5. app/main/views.py: rota da página de perfil com postagens do blog*

```
@main.route('/user/<username>')
def user(username):
    user = User.query.filter_by(username=username).first() se o usuário for
    Nenhum: abort(404) posts =
        user.posts.order_by(Post.timestamp.desc()).all() return
    render_template('user.html', user=user, posts=posts)
```

A lista de postagens de blog de um usuário é obtida do relacionamento `User.posts`, que é um objeto de consulta, portanto, filtros como `order_by()` também podem ser usados nele.

O modelo `user.html` requer a árvore HTML `<ul>` que renderiza uma lista de postagens de blog como a de `index.html`. Ter que manter duas cópias idênticas de um pedaço de HTML não é ideal, então para casos como este, a diretiva `include()` do Jinja2 é muito útil. O modelo `user.html` inclui a lista de um arquivo externo, conforme mostrado no [Exemplo 11-6](#).

*Exemplo 11-6. app/templates/user.html: modelo de página de perfil com postagens de blog*

```
...
<h3>Postagens de {{ user.username }}</h3>
{% include '_posts.html' %}
...
```

Para completar esta reorganização, a árvore `<ul>` de `index.html` é movida para o novo template `_posts.html`, e substituída por outra diretiva `include()`. Observe que o uso de um prefixo de sublinhado no nome do modelo `_posts.html` não é um requisito; isso é apenas uma convenção para distinguir modelos independentes e parciais.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar `git checkout 11b` para verificar esta versão do aplicativo.

## Paginando listas longas de postagens de blog

À medida que o site cresce e o número de postagens do blog aumenta, ficará lento e impraticável mostrar a lista completa de postagens nas páginas inicial e de perfil. Páginas grandes levam mais tempo para serem geradas, baixadas e renderizadas no navegador da Web, portanto, a qualidade da experiência do usuário diminui à medida que as páginas aumentam. A solução é *paginar* os dados e renderizá-los em pedaços.

### Criando dados falsos de postagens de

**blog** Para poder trabalhar com várias páginas de postagens de blog, é necessário ter um banco de dados de teste com um grande volume de dados. A adição manual de novas entradas de banco de dados é demorada e tediosa; uma solução automatizada é mais apropriada. Existem vários pacotes Python que podem ser usados para gerar informações falsas. Um bastante completo é o *ForgeryPy*, que é instalado com pip:

```
(venv) $ pip install forgerypy
```

O pacote ForgeryPy não é, estritamente falando, uma dependência da aplicação, pois é necessário apenas durante o desenvolvimento. Para separar as dependências de produção das dependências de desenvolvimento, o arquivo `requirements.txt` pode ser substituído por uma pasta de *requisitos* que armazena diferentes conjuntos de dependências. Dentro desta nova pasta um arquivo `dev.txt` pode listar as dependências que são necessárias para o desenvolvimento e um arquivo `prod.txt` pode listar as dependências que são necessárias na produção. Como há um grande número de dependências que estarão em ambas as listas, um arquivo `common.txt` é adicionado para elas e, em seguida, as listas `dev.txt` e `prod.txt` usam o prefixo `-r` para incluí-lo. O Exemplo 11-7 mostra o arquivo `dev.txt`.

*Exemplo 11-7. requirements/dev.txt: arquivo de requisitos de desenvolvimento*

```
-r common.txt
ForgeryPy==0.1
```

O **Exemplo 11-8** mostra métodos de classe adicionados aos modelos User e Post que podem gerar dados falsos.

*Exemplo 11-8. app/models.py: gera usuários e postagens de blog falsos*

```
class User(UserMixin, db.Model):
    # ...
    @staticmethod
    def generate_fake(count=100): from
        sqlalchemy.exc import IntegrityError from random
        import seed import forgery_py

        seed()
        for i in range(count): u =
            User(email=forgery_py.internet.email_address(),
                 username=forgery_py.internet.user_name(True),
                 password=forgery_py.lorem_ipsum.word(), confirmado=True,
                 name=forgery_py.name.full_name(),
                 location=forgery_py.address.city(),
                 about_me=forgery_py.lorem_ipsum_sentence(),
                 member_since=forgery_py.date.date(True))
            db.session.add(u) tente: db.session.commit() exceto
            IntegrityError:

        db.session.rollback()

class Post(db.Model):
    # ...
    @staticmethod
    def generate_fake(count=100): da
        semente de importação aleatória ,
        importação randint forgery_py

        seed()
        user_count = User.query.count() for i in
        range(count):
            u = User.query.offset(randint(0, user_count - 1)).first() p =
            Post(body=forgery_py.lorem_ipsum.sentences(randint(1, 3)),
                 timestamp=forgery_py.date.date(True ), autor=u) db.session.add(p)
            db.session.commit()
```

Os atributos desses objetos falsos são gerados com geradores de informações aleatórias ForgeryPy, que podem gerar nomes, e-mails, frases e muitos outros atributos de aparência real.

Os endereços de e-mail e nomes de usuários dos usuários devem ser únicos, mas como o ForgeryPy os gera de maneira completamente aleatória, existe o risco de haver duplicatas. Nesse evento improvável, a confirmação da sessão do banco de dados lançará uma exceção `IntegrityError`. Essa exceção é tratada revertendo a sessão antes de continuar. As iterações de loop que produzem uma duplicata não gravarão um usuário no banco de dados, portanto, o número total de usuários falsos adicionados pode ser menor que o número solicitado.

A geração de postagem aleatória deve atribuir um usuário aleatório para cada postagem. Para isso, o filtro de consulta `offset()` é usado. Este filtro descarta o número de resultados fornecidos como argumento. Ao definir um deslocamento aleatório e, em seguida, chamar `first()`, um usuário aleatório diferente é obtido a cada vez.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar `git checkout 11c` para verificar esta versão do aplicativo. Para garantir que você tenha todas as dependências instaladas, execute também `pip install -r requirements/dev.txt`.

Os novos métodos facilitam a criação de um grande número de usuários e postagens falsos do shell do Python:

```
(venv) $ python manage.py shell >>>
User.generate_fake(100)
>>> Post.generate_fake(100)
```

Se você executar o aplicativo agora, verá uma longa lista de postagens de blog aleatórias na página inicial.

### Renderizando dados em páginas

O Exemplo 11-9 mostra as alterações na rota da página inicial para dar suporte à paginação.

*Exemplo 11-9. app/main/views.py: paginar a lista de postagens do blog*

```
@main.route('/', métodos=['GET', 'POST']) def
index(): # page = request.args.get('page', 1, type=int)
    pagination =
        Post.query.order_by(Post.timestamp.desc()).paginate(page,
per_page=current_app.config['FLASKY_POSTS_PER_PAGE'], error_out=False)
        posts = pagination.items return render_template('index.html', form=form, posts
=postagens, paginação=paginação)
```

O número da página a ser renderizada é obtido da string de consulta da solicitação, que está disponível como `request.args`. Quando uma página explícita não é fornecida, uma página padrão de 1 (a primeira página) é usado. O argumento `type=int` garante que, se o argumento não puder ser convertido em um inteiro, o valor padrão é retornado.

Para carregar uma única página de registros, a chamada para `all()` é substituída por Flask-SQLAlchemy's `paginate()`. O método `paginate()` toma o número da página como o primeiro e único argumento necessário. Um argumento opcional por página pode ser fornecido para indicar o tamanho de cada página, em número de itens. Se este argumento não for especificado, o padrão é 20 itens por página. Outro argumento opcional chamado `error_out` pode ser definido como `True` (o padrão) para emitir um erro de código 404 quando uma página fora do intervalo válido for solicitada. Se `error_out` for `False`, as páginas fora do intervalo válido são retornadas com uma lista vazia de itens. Para tornar os tamanhos de página configuráveis, o valor do argumento `per_page` é lido de uma variável de configuração específica do aplicativo chamada `FLASKY_POSTS_PER_PAGE`.

Com essas alterações, a lista de postagens do blog na página inicial mostrará um número limitado de itens. Para ver a segunda página de postagens, adicione uma string de consulta `?page=2` ao URL na barra de endereço do navegador.

## Adicionando um widget de paginação

O valor de retorno de `paginate()` é um objeto da classe `Pagination`, uma classe definida pelo Flask SQLAlchemy. Este objeto contém várias propriedades que são úteis para gerar páginas links em um modelo, então ele é passado para o modelo como um argumento. Um resumo dos atributos do objeto de paginação são mostrados na [Tabela 11-1](#).

*Tabela 11-1. Atributos do objeto de paginação Flask-SQLAlchemy*

Atributo	Descrição
Itens	Os registros na página atual
inquerir	A consulta de origem que foi paginada
página	O número da página atual
núm_anterior	O número da página anterior
next_num	O número da próxima página
has_next	Verdadeiro se houver uma próxima página
has_prev	Verdadeiro se houver uma página anterior
Páginas	O número total de páginas para a consulta
por página	O número de itens por página
total	O número total de itens retornados pela consulta

O objeto de paginação também possui alguns métodos, listados na [Tabela 11-2](#).

*Tabela 11-2. Atributos do objeto de paginação Flask-SQLAlchemy*

Método	Descrição
iter_pages(left_edge=2, left_current=2, right_current=5, right_edge=2)	Um iterador que retorna esses números de sequência de páginas para exibição em um widget de paginação. A lista terá páginas left_edge à esquerda, páginas left_current à esquerda da página atual, páginas right_current à direita da página atual e páginas right_edge à direita. Por exemplo, para a página 50 de 100, este iterador configurado com valores padrão retornará as seguintes páginas: 1, 2, Nenhum, 48, 49, 50, 51, 52, 53, 54, 55, Nenhum, 99, 100. A Nenhum valor na sequência indica uma lacuna na sequência de páginas.
anterior()	Um objeto de paginação para a página anterior.
próximo()	Um objeto de paginação para a próxima página.

Armado com este poderoso objeto e as classes CSS de paginação do Bootstrap, é muito fácil construir um rodapé de paginação no template. A implementação mostrada no [Exemplo 11-10](#) é feita como uma macro Jinja2 reutilizável.

*Exemplo 11-10. app/templates/\_macros.html: macro de modelo de paginação*

```
{% macro pagination_widget(pagination, endpoint) %} <ul class="pagination">
    <li{% if not pagination.has_prev %}> <a href="{% if pagination.has_prev %}{{ url_for(endpoint,
        page = pagination.page - 1, **kwargs) }}{% else %}#{% endif %}"> &laquo; </a>
    </li> <li> <a href="{{ url_for(endpoint, page = p, **kwargs) }}>{{ p }}</a> </li> {% else %}
    <li> <a href="#">&hellip;</a></li> {% endif %} {% endfor %}
    <li{% if not pagination.has_next %}> <a href="{% if pagination.has_next %}{{ url_for(endpoint,
        page = pagination.page + 1, **kwargs) }}{% else %}#{% endif %}"> &raquo; </a>
    </li>
</ul>
```

```
</ul>
{%- endmacro %}
```

A macro cria um elemento de paginação Bootstrap, que é uma lista não ordenada estilizada. Ele define os seguintes links de página dentro dele:

- Um link de “página anterior”. Este link obtém a classe desabilitada se a página atual for a primeira página.
- Links para todas as páginas retornadas pelo `iter_pages()` do objeto de paginação .  
Essas páginas são renderizadas como links com um número de página explícito, dado como argumento para `url_for()`. A página exibida atualmente é destacada usando a classe CSS ativa . As lacunas na sequência de páginas são renderizadas com o caractere de reticências. • Um link “próxima página”.

Este link aparecerá desabilitado se a página atual for a última página.

As macros Jinja2 sempre recebem argumentos de palavras-chave sem precisar incluir `**kwargs` na lista de argumentos. A macro de paginação passa todos os argumentos de palavra-chave que recebe para a chamada `url_for()` que gera os links de paginação. Essa abordagem pode ser usada com rotas como a página de perfil que possui uma parte dinâmica.

A macro `pagination_widget` pode ser adicionada abaixo do modelo `_posts.html` incluído por `index.html` e `user.html`. O [Exemplo 11-11](#) mostra como ele é usado na página inicial do aplicativo.

*Exemplo 11-11. app/templates/index.html: rodapé de paginação para listas de postagens de blog*

```
{% extends "base.html" %} {%
import "bootstrap/wtf.html" como wtf %} {% import
"macros.html" como macros %}
...
{% include '_posts.html' %} <div
class="pagination">
{{ macros.pagination_widget(pagination, '.index') }} </div> {%
endif %}
```

A [Figura 11-2](#) mostra como os links de paginação aparecem na página.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar `git checkout 11d` para verificar esta versão do aplicativo.

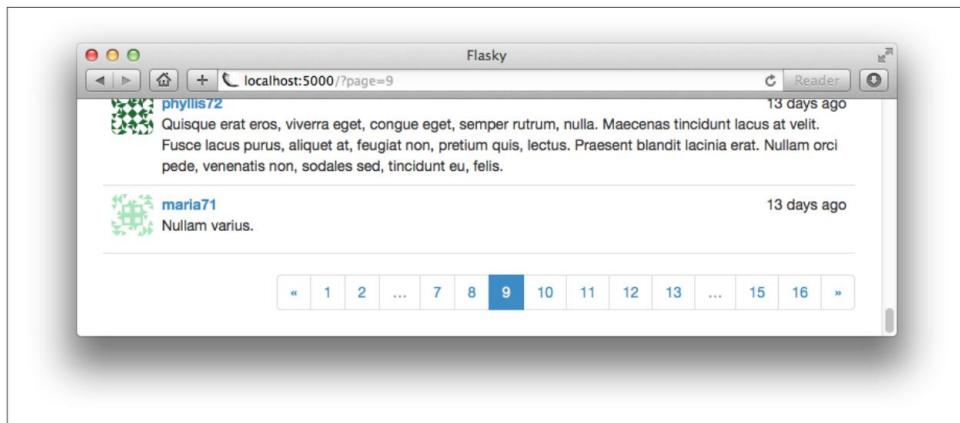


Figura 11-2. Paginação da postagem do blog

## Postagens Rich-Text com Markdown e Flask-PageDown

As postagens em texto simples são suficientes para mensagens curtas e atualizações de status, mas os usuários que desejam escrever artigos mais longos encontrarão a falta de formatação muito limitante. Nesta seção, o campo da área de texto onde as postagens são inseridas será atualizado para suportar o [Markdown](#) sintaxe e apresentar uma visualização em rich text da postagem.

A implementação deste recurso requer alguns novos pacotes:

- PageDown, um conversor Markdown-to-HTML do lado do cliente implementado em Java<sup>9</sup> Roteiro.
- Flask-PageDown, um wrapper PageDown para Flask que integra PageDown com formulários Flask-WTF.
- Markdown, um conversor de Markdown para HTML do lado do servidor implementado em Python.
- Bleach, um desinfetante HTML implementado em Python.

Os pacotes Python podem ser instalados com pip:

```
(venv) $ pip install flask-pagedown markdown bleach
```

### Usando Flask-PageDown A extensão

Flask-PageDown define uma classe PageDownField que tem a mesma interface que TextAreaField de WTForms. Antes que este campo possa ser usado, a extensão precisa ser inicializada conforme mostrado no [Exemplo 11-12](#).

*Exemplo 11-12. app/\_\_init\_\_.py: inicialização do Flask-PageDown*

```
from flask.ext.pagedown import PageDown #
...
```

```

pagedown = PageDown()
# ...
def create_app(config_name):
    ...
    # pagedown.init_app(app)
    # ...

```

Para converter o controle de área de texto na home page em um editor de rich text Markdown, o campo do corpo do PostForm deve ser alterado para um PageDownField conforme mostrado no [Exemplo 11-13](#).

*Exemplo 11-13. app/main/forms.py: formulário de postagem habilitado para Markdown*

```

de flask.ext.pagedown.fields importar PageDownField

class PostForm(Form):
    body = PageDownField("O que você está pensando?", validators=[Required()])
    submit = SubmitField('Submit')

```

A visualização do Markdown é gerada com a ajuda das bibliotecas PageDown, portanto, elas devem ser adicionadas ao modelo. O Flask-PageDown simplifica essa tarefa fornecendo uma macro de modelo que inclui os arquivos necessários de um CDN, conforme mostrado no [Exemplo 11-14](#).

*Exemplo 11-14. app/index.html: declaração de modelo Flask-PageDown*

```

{% block scripts %}
{{ super() }}
{{ pagedown.include_pagedown() }} {%
endblock %}

```



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar git checkout 11e para verificar esta versão do aplicativo. Para garantir que você tenha todas as dependências instaladas, execute pip install -r requirements/dev.txt.

Com essas alterações, o texto formatado em Markdown digitado no campo da área de texto será imediatamente renderizado como HTML na área de visualização abaixo. A [Figura 11-3](#) mostra o formulário de envio de blog com rich text.

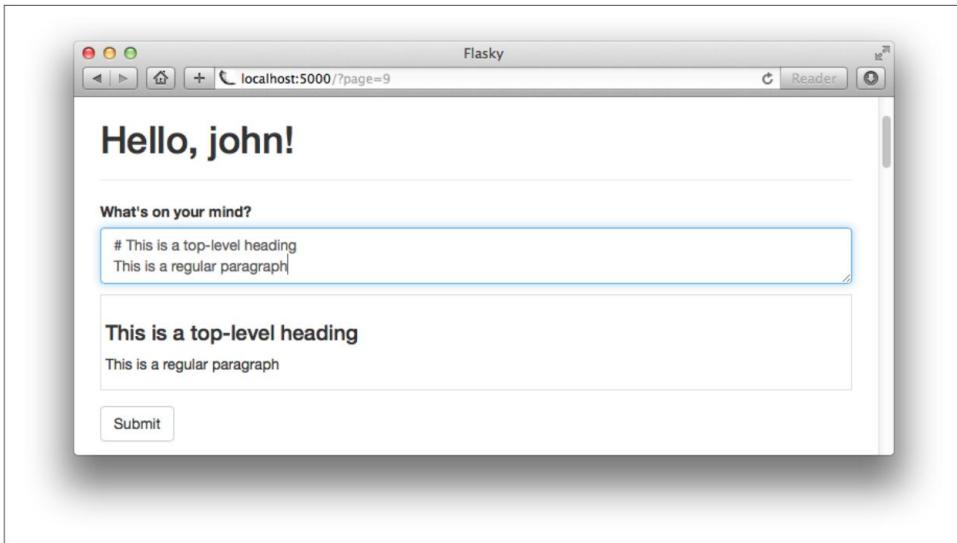


Figura 11-3. Formulário de postagem de blog em rich text

### Manipulando Rich Text no Servidor Quando

o formulário é enviado, apenas o texto bruto Markdown é enviado com a solicitação POST ; a visualização HTML que foi mostrada na página é descartada. O envio da visualização HTML gerada com o formulário pode ser considerado um risco de segurança, pois seria bastante fácil para um invasor construir sequências HTML que não correspondam à origem do Markdown e enviá-las. Para evitar quaisquer riscos, apenas o texto fonte do Markdown é enviado e, uma vez no servidor, é convertido novamente para HTML usando *Markdown*, um conversor de Markdown para HTML do Python. O HTML resultante será higienizado com *Bleach* para garantir que apenas uma pequena lista de tags HTML permitidas seja usada.

A conversão das postagens do blog Markdown para HTML pode ser emitida no modelo `_posts.html` , mas isso é ineficiente, pois as postagens terão que ser convertidas toda vez que forem renderizadas em uma página. Para evitar essa repetição, a conversão pode ser feita uma vez quando a postagem do blog é criada. O código HTML para a postagem de blog renderizada é armazenado em `cache` em um novo campo adicionado ao modelo Post que o modelo pode acessar diretamente. A fonte original do Markdown também é mantida no banco de dados caso a postagem precise ser editada. O [Exemplo 11-15](#) mostra as mudanças no modelo Post .

*Exemplo 11-15. app/models/post.py: manipulação de texto Markdown no modelo Post*

`de remarcação de importação remarcação  
de importação lixivia`

```
class Post(db.Model): #
    ...
    body_html = db.Column(db.Text)
```

```

...
# @staticmethod
def on_changed_body(target, value, oldvalue, initiator):
    allowed_tags = ['a', 'abbr', 'acronym', 'b', 'blockquote', 'code', 'em', 'i', 'li', 'ol', 'pre',
                    'strong', 'ul', 'h1', 'h2', 'h3', 'p']

    target.body_html = bleach.linkify(bleach.clean( markdown(value,
                                                          output_format='html'), tags=allowed_tags, strip=True))

db.event.listen(Post.body, 'set', Post.on_changed_body)

```

A função `on_changed_body` é registrada como ouvinte do evento “`set`” do SQLAlchemy para `body`, o que significa que ela será invocada automaticamente sempre que o campo `body` em qualquer instância da classe `Post` for definido com um novo valor. A função renderiza a versão HTML do corpo e a armazena em `body_html`, efetivamente tornando a conversão do texto Mark-down para HTML totalmente automática.

A conversão real é feita em três etapas. Primeiro, a função `markdown()` faz uma conversão inicial para HTML. O resultado é passado para `clean()`, junto com a lista de tags HTML aprovadas. A função `clean()` remove quaisquer tags que não estejam na lista branca. A conversão final é feita com `linkify()`, outra função fornecida pelo Bleach que converte qualquer URL escrita em texto simples em links `<a>` apropriados. Esta última etapa é necessária porque a geração automática de links não está oficialmente na especificação Markdown. PageDown o suporta como uma extensão, então `linkify()` é usado no servidor para corresponder.

A última alteração é substituir `post.body` por `post.body_html` no modelo quando disponível, conforme mostrado no [Exemplo 11-16](#).

*Exemplo 11-16. app/templates/\_posts.html: use a versão HTML dos corpos da postagem no modelo*

```

...
<div class="post-body">
    {% if post.body_html %}
        {{ post.body_html | safe }} {% else
    %} {{ post.body }} {% endif %} </div>

...

```

O `| safe` sufixo ao renderizar o corpo HTML está lá para dizer ao Jinja2 para não escapar dos elementos HTML. Jinja2 escapa de todas as variáveis de template por padrão como medida de segurança. O HTML gerado pelo Markdown foi gerado no servidor, portanto, é seguro renderizar.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar `git checkout 11f` para verificar esta versão do aplicativo.

Esta atualização também contém uma migração de banco de dados, portanto, lembre-se de executar `python manage.py db upgrade` depois de verificar o código. Para garantir que você tenha todas as dependências instaladas, execute `pip install -r requirements/dev.txt`.

## Links permanentes para postagens do blog

Os usuários podem querer compartilhar links para postagens específicas do blog com amigos nas redes sociais. Para isso, será atribuída a cada postagem uma página com um URL exclusivo que a referencia. As funções de rota e visualização que suportam links permanentes são mostradas no [Exemplo 11-17](#).

*Exemplo 11-17. app/main/views.py: links permanentes para postagens*

```
@main.route('/post/<int:id>')
def post(id):
    post = Post.query.get_or_404(id)
    return render_template('post.html', posts=[post])
```

As URLs que serão atribuídas às postagens do blog são construídas com o campo de id exclusivo atribuído quando a postagem é inserida no banco de dados.



Para alguns tipos de aplicativos, pode ser preferível criar links permanentes que usem URLs legíveis em vez de IDs numéricos. Uma alternativa ao ID numérico é atribuir a cada postagem do blog um *slug*, que é uma string exclusiva relacionada à postagem.

Observe que o modelo `post.html` recebe uma lista apenas com a postagem a ser renderizada. O envio de uma lista é necessário para que o template `_posts.html` referenciado por `index.html` e `user.html` também possa ser usado nesta página.

Os links permanentes são adicionados na parte inferior de cada postagem no modelo genérico `_posts.html`, conforme mostrado no [Exemplo 11-18](#).

*Exemplo 11-18. app/templates/\_posts.html: links permanentes para postagens*

```
<ul class="posts"> {%
    para postagem em postagens
%} <li class="post">
    ...
    <div class="post-content">
        ...
        <div class="post-footer">
            <a href="{{ url_for('.post', id=post.id) }}>
                <span class="label label-default">Link permanente </a>
```

```

</div>
</div> </li>
{% endfor %} <
ul>
```

O novo modelo *post.html* que renderiza a página de link permanente é mostrado no [Exemplo 11-19](#). Inclui o modelo de exemplo.

*Exemplo 11-19. app/templates/post.html: modelo de link permanente*

```

{% estende "base.html" %}

{% block title %}Flasky - Postar{% endblock %}

{% block page_content %}
{% include '_posts.html' %} {% endblock %}
```



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar git checkout 11g para verificar esta versão do aplicativo.

## Editor de postagem do blog

O último recurso relacionado às postagens do blog é um editor de postagens que permite aos usuários editar suas próprias postagens. O editor de postagem do blog ficará em uma página independente. No topo da página, a versão atual do post será mostrada para referência, seguida por um editor Markdown onde o Markdown fonte pode ser modificado. O editor será baseado no Flask PageDown, então uma prévia da versão editada da postagem do blog será mostrada na parte inferior da página. O modelo *edit\_post.html* é mostrado no [Exemplo 11-20](#).

*Exemplo 11-20. app/templates/edit\_post.html: Editar modelo de postagem do blog*

```

{% extends "base.html" %} {% import "bootstrap/wtf.html" como wtf %}

{% block title %}Flasky - Editar postagem{% endblock %}

{% block page_content %}
<div class="page-header">
<h1>Editar postagem</h1> <div>
{{ wtf.quick_form(form) }} <
div> {% endblock %}
```

```
{% block scripts %}
{{ super() }}
{{ pagedown.include_pagedown() }} {%
endblock %}
```

A rota que suporta o editor de postagem do blog é mostrada no [Exemplo 11-21](#).

*Exemplo 11-21. app/main/views.py: Editar rota de postagem do blog*

```
@main.route('/edit/<int:id>', métodos=['GET', 'POST'])
@login_required def edit(id): post = Post.query.get_or_404(id) if
current_user != post.author e \ não
    current_user.can(Permission.ADMINISTER): abort(403) form
    = PostForm () if form.validate_on_submit():

        post.body = form.body.data
        db.session.add(post) flash('O
        post foi atualizado.') return
        redirect(url_for('post', id=post.id)) form.body.data =
    post.body return render_template ('edit_post.html', form=form)
```

Essa função de visualização é codificada para permitir que apenas o autor de uma postagem do blog a edite, exceto os administradores, que têm permissão para editar postagens de todos os usuários. Se um usuário tentar editar uma postagem de outro usuário, a função de visualização responderá com um código 403. A classe de formulário PostForm web usada aqui é a mesma usada na página inicial.

Para completar o recurso, um link para o editor de postagem do blog pode ser adicionado abaixo de cada postagem do blog, próximo ao link permanente, conforme mostrado no [Exemplo 11-22](#).

*Exemplo 11-22. app/templates/\_posts.html: editar links de postagem do blog*

```
<ul class="posts"> {%
    para postagem em postagens
%} <li class="post">
    ...
    <div class="post-content">
        ...
        <div class="post-footer">
            ...
            {% if current_user == post.author %} <a
                href="{{ url_for('.edit', id=post.id) }}">
                    <span class="label label-primary">Editar </a> {% elif
                current_user.is_administrator() %} <a href="{{ url_for('.edit',
                id=post.id) }}">
                    ...
                    <span class="label label-danger">Editar [Admin] </a> {% endif
%}
```

```
</div>
</div> </li>
{% endfor %} </
ul>
```

Essa alteração adiciona um link “Editar” a qualquer postagem de blog criada pelo usuário atual.

Para administradores, o link é adicionado a todas as postagens. O link do administrador tem um estilo diferente como uma indicação visual de que este é um recurso de administração. A [Figura 11-4](#) mostra a aparência dos links Edit e Permalink no navegador da web.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar git checkout 11h para conferir esta versão do aplicativo.

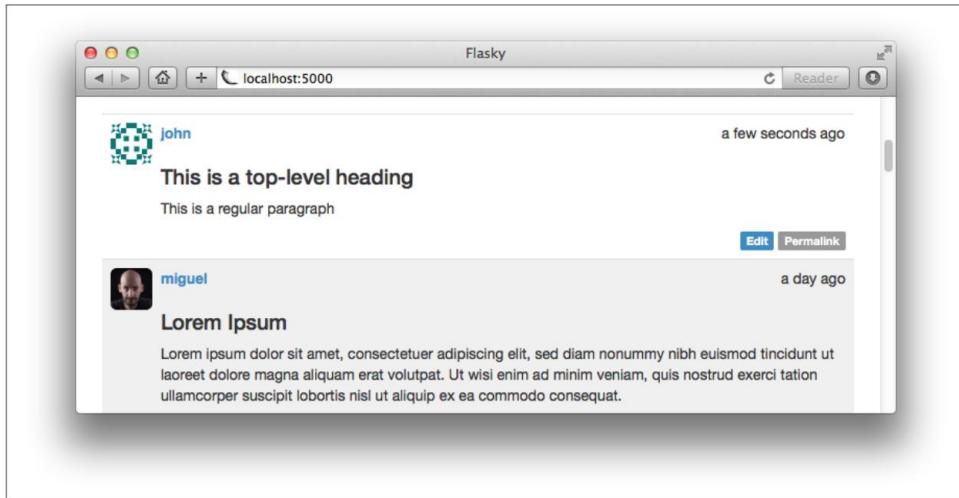


Figura 11-4. Editar e links permanentes nas postagens do blog.

---

## CAPÍTULO 12

# Seguidores

Os aplicativos da Web socialmente conscientes permitem que os usuários se conectem com outros usuários. Os aplicativos chamam esses relacionamentos *de seguidores*, *amigos*, *contatos*, *conexões* ou *amigos*, mas o recurso é o mesmo, independentemente do nome, e em todos os casos envolve acompanhar links direcionais entre pares de usuários e usar esses links no banco de dados. consultas.

Neste capítulo, você aprenderá como implementar um recurso de seguidor para o Flasky. Os usuários poderão “seguir” outros usuários e optar por filtrar a lista de postagens do blog na página inicial para incluir apenas aqueles dos usuários que seguem.

### Relacionamentos de banco de dados revisitados

Como discutimos no [Capítulo 5](#), bancos de dados estabelecem links entre registros usando *relacionamentos*. O relacionamento um-para-muitos é o tipo mais comum de relacionamento, em que um registro é vinculado a uma lista de registros relacionados. Para implementar esse tipo de relacionamento, os elementos do lado “muitos” possuem uma chave estrangeira que aponta para o elemento vinculado do lado “um”. O aplicativo de exemplo em seu estado atual inclui dois relacionamentos um-para-muitos: um que vincula as funções do usuário a listas de usuários e outro que vincula os usuários às postagens do blog que eles criaram.

A maioria dos outros tipos de relacionamento pode ser derivada do tipo um para muitos. O relacionamento *muitos para um* é um relacionamento um para muitos visto do ponto de vista do lado “muitos”. O tipo de relacionamento *um-para-um* é uma simplificação do tipo um-para-muitos, onde o lado “muitos” é restrito a ter no máximo um elemento. O único tipo de relacionamento que não pode ser implementado como uma simples variação do modelo um para muitos é o *muitos para muitos*, que possui listas de elementos em ambos os lados. Essa relação é descrita em detalhes na seção a seguir.

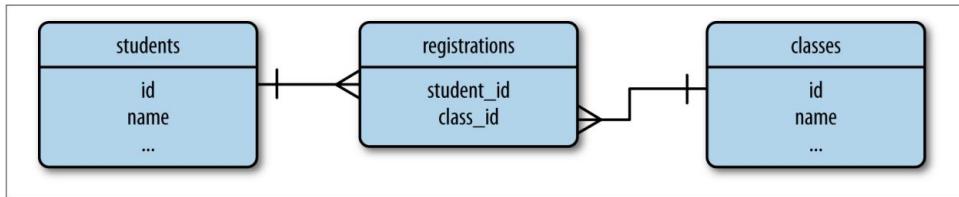
### Relacionamentos muitos para muitos Os

relacionamentos um para muitos, muitos para um e um para um têm pelo menos um lado com uma única entidade, de modo que os links entre registros relacionados são implementados com chaves estrangeiras apontando para aquele elemento. Mas como você implementa um relacionamento onde ambos os lados são “muitos” lados?

Considere o exemplo clássico de um relacionamento muitos-para-muitos: um banco de dados de alunos e as aulas que eles estão cursando. Claramente, você não pode adicionar uma chave estrangeira a uma classe na tabela de alunos, porque um aluno faz muitas aulas - uma chave estrangeira não é suficiente.

Da mesma forma, você não pode adicionar uma chave estrangeira ao aluno na tabela de turmas, pois as turmas têm mais de um aluno. Ambos os lados precisam de uma lista de chaves estrangeiras.

A solução é adicionar uma terceira tabela ao banco de dados, chamada de *tabela de associação*. Agora, o relacionamento muitos-para-muitos pode ser decomposto em dois relacionamentos um-para-muitos de cada uma das duas tabelas originais para a tabela de associação. A [Figura 12-1](#) mostra como o relacionamento muitos-para-muitos entre alunos e turmas é representado.



*Figura 12-1. Exemplo de relacionamento muitos para muitos*

A tabela de associação neste exemplo é chamada de registros. Cada linha nesta tabela representa um registro individual de um aluno em uma turma.

Consultar um relacionamento muitos-para-muitos é um processo de duas etapas. Para obter a lista de aulas que um aluno está cursando, você parte da relação um-para-muitos entre alunos e matrículas e obtém a lista de matrículas do aluno desejado. Em seguida, a relação um-para-muitos entre turmas e matrículas é percorrida na direção muitos-para-um para obter todas as classes associadas às matrículas recuperadas para o aluno. Da mesma forma, para encontrar todos os alunos em uma turma, você começa a partir da turma e obtém uma lista de inscrições e, em seguida, vincula os alunos a essas inscrições.

Atravessar dois relacionamentos para obter resultados de consulta parece difícil, mas para um relacionamento simples como o do exemplo anterior, SQLAlchemy faz a maior parte do trabalho.

A seguir está o código que representa o relacionamento muitos-para-muitos na [Figura 12-1](#):

```

inscrições = db.Table('registrations',
    db.Column('student_id', db.Integer, db.ForeignKey('students.id')),
    db.Column('class_id', db.Integer, db.ForeignKey('classes.id'))
)

```

```
class Aluno(db.Model):
    id = db.Column(db.Integer, primary_key=True) name
    = db.Column(db.String) classes = db.relationship('Class',
        secondary=registrations, backref=db.backref('students',
            lazy= 'dinâmico'),
            preguiçoso='dinâmico')

class Class(db.Model): id
    = db.Column(db.Integer, primary_key = True) name =
    db.Column(db.String)
```

O relacionamento é definido com a mesma construção db.relationship() que é usada para relacionamentos um-para-muitos, mas no caso de um relacionamento muitos-para-muitos o argumento secundário adicional deve ser definido para a tabela de associação . O relacionamento pode ser definido em qualquer uma das duas classes, com o argumento backref cuidando de expor o relacionamento do outro lado também. A tabela de associação é definida como uma tabela simples, não como um modelo, pois o SQLAlchemy gerencia essa tabela internamente.

O relacionamento de classes usa semântica de lista, o que torna extremamente fácil trabalhar com relacionamentos muitos para muitos configurados dessa maneira. Dado um aluno s e uma turma c, o código que registra o aluno para a turma é:

```
>>> s.classes.append(c)
>>> db.session.add(s)
```

As consultas que listam as turmas para as quais os alunos estão inscritos e a lista de alunos inscritos para a turma c também são muito simples:

```
>>> s.classes.all() >>>
c.students.all()
```

O relacionamento de alunos disponível no modelo Class é aquele definido no argumento db.backref() . Observe que neste relacionamento o argumento backref foi expandido para também ter um atributo lazy = 'dynamic' , então ambos os lados retornam uma consulta que pode aceitar filtros adicionais.

Se o aluno mais tarde decidir abandonar a aula c, você pode atualizar o banco de dados da seguinte forma:

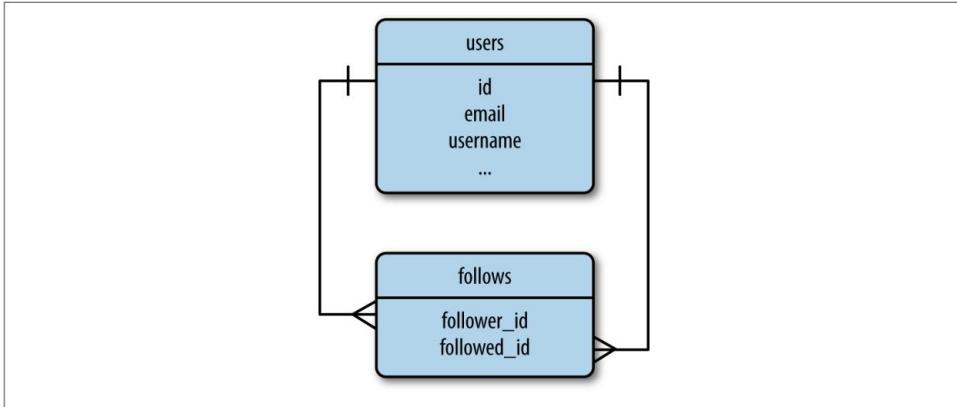
```
>>> s.classes.remove(c)
```

## Relacionamentos autorreferenciais

Um relacionamento muitos-para-muitos pode ser usado para modelar usuários seguindo outros usuários, mas há um problema. No exemplo de alunos e turmas, havia duas entidades claramente definidas ligadas entre si pela tabela de associação. No entanto, para representar usuários seguindo outros usuários, são apenas usuários – não há uma segunda entidade.

Uma relação em que ambos os lados pertencem à mesma tabela é dita *auto-referencial*. Neste caso as entidades do lado esquerdo do relacionamento são usuários, que

podem ser chamados de “seguidores”. As entidades do lado direito também são usuários, mas esses são os usuários “seguidos”. Conceitualmente, os relacionamentos autorreferenciais não são diferentes dos relacionamentos regulares, mas são mais difíceis de pensar. A Figura 12-2 mostra um diagrama de banco de dados para um relacionamento autorreferencial que representa usuários seguindo outros usuários.



*Figura 12-2. Seguidores, relacionamento muitos para muitos*

A tabela de associação neste caso é chamada de segue. Cada linha nesta tabela representa um usuário seguindo outro usuário. A relação um-para-muitos ilustrada no lado esquerdo associa os usuários à lista de linhas “seguidores” nas quais eles são os seguidores. O relacionamento um para muitos ilustrado no lado direito associa os usuários à lista de linhas “seguidores” nas quais eles são o usuário seguido.

### Relacionamentos muitos-para-muitos avançados Com

um relacionamento muitos-para-muitos auto-referencial configurado conforme indicado no exemplo anterior, o banco de dados pode representar seguidores, mas há uma limitação. Uma necessidade comum ao trabalhar com relacionamentos muitos para muitos é armazenar dados adicionais que se aplicam ao link entre duas entidades. Para o relacionamento de seguidores, pode ser útil armazenar a data em que um usuário começou a seguir outro usuário, pois isso permitirá que as listas de seguidores sejam apresentadas em ordem cronológica. O único local em que essas informações podem ser armazenadas é na tabela de associação, mas em uma implementação semelhante à dos alunos e turmas mostradas anteriormente, a tabela de associação é uma tabela interna totalmente gerenciada pelo SQLAlchemy.

Para poder trabalhar com dados personalizados no relacionamento, a tabela de associação deve ser promovida para um modelo adequado que o aplicativo possa acessar. O [Exemplo 12-1](#) mostra a nova tabela de associação, representada pelo modelo Follow .

*Exemplo 12-1. app/models/user.py: A tabela de associação a seguir como modelo*

```
class Follow(db.Model):
    __tablename__ = 'follows'
    follower_id = db.Column(db.Integer, db.ForeignKey('users.id'),
                           primary_key=True)
    seguido_id = db.Column(db.Integer, db.ForeignKey('users.id'),
                           primary_key=True)
    timestamp = db.Column(db.DateTime, default=datetime.utcnow)
```

SQLAlchemy não pode usar a tabela de associação de forma transparente porque isso não dará ao aplicativo acesso aos campos personalizados nela. Em vez disso, o relacionamento muitos-para-muitos deve ser decomposto em dois relacionamentos básicos um-para-muitos para os lados esquerdo e direito, e estes devem ser definidos como relacionamentos padrão. Isso é mostrado no [Exemplo 12-2](#).

*Exemplo 12-2. app/models/user.py: um relacionamento muitos para muitos implementado como dois relacionamentos um para muitos*

```
class User(UserMixin, db.Model):
    # ...
    seguido = db.relationship('Follow',
                             Foreign_keys=[Follow.follower_id],
                             backref=db.backref('follower', lazy='joined'), lazy='dynamic',
                             cascade='all, delete-orphan') seguidores =
    db.relationship('Follow', Foreign_keys=[Follow.followed_id],
                   backref=db.backref('followed', lazy='joined'), lazy='dynamic', cascade='all, delete-orphan')
```

Aqui os relacionamentos seguidos e seguidores são definidos como relacionamentos individuais de um para muitos. Observe que é necessário eliminar qualquer ambiguidade entre chaves estrangeiras especificando em cada relacionamento qual chave estrangeira usar por meio do argumento opcional `Foreign_keys`. Os argumentos `db.backref()` nesses relacionamentos não se aplicam uns aos outros; as referências anteriores são aplicadas ao modelo `Follow`.

O argumento `lazy` para as referências anteriores é especificado como associado. Esse modo lento faz com que o objeto relacionado seja carregado imediatamente da consulta de junção. Por exemplo, se um usuário estiver seguindo 100 outros usuários, chamar `user.followed.all()` retornará uma lista de 100 instâncias `Follow`, onde cada uma tem as propriedades de referência de seguidor e seguido de volta definidas para os respectivos usuários. O modo `lazy='joined'` permite que tudo isso aconteça a partir de uma única consulta de banco de dados. Se `lazy` estiver definido com o valor padrão de `select`, o seguidor e os usuários seguidos serão carregados lentamente quando forem acessados pela primeira vez e cada atributo exigirá uma consulta individual, o que significa que obter a lista completa de usuários seguidos exigiria 100 consultas de banco de dados adicionais.

O argumento preguiçoso do lado do usuário de ambos os relacionamentos tem necessidades diferentes. Estes estão no lado “um” e retornam no lado “muitos”; aqui é usado um modo de dinâmica , para que os atributos de relacionamento retornem objetos de consulta em vez de retornar os itens diretamente, de modo que filtros adicionais possam ser adicionados à consulta antes que ela seja executada.

O argumento cascade configura como as ações executadas em um objeto pai se propagam para objetos relacionados. Um exemplo de opção em cascata é a regra que diz que quando um objeto é adicionado à sessão do banco de dados, quaisquer objetos associados a ele por meio de relacionamentos também devem ser adicionados automaticamente à sessão. As opções de cascata padrão são apropriadas para a maioria das situações, mas há um caso em que as opções de cascata padrão não funcionam bem para esse relacionamento muitos-para-muitos. O comportamento padrão em cascata quando um objeto é excluído é definir a chave estrangeira em qualquer objeto relacionado que se vincule a ele para um valor nulo. Mas para uma tabela de associação, o comportamento correto é excluir as entradas que apontam para um registro que foi excluído, pois isso efetivamente destrói o link. Isso é o que a opção de cascata delete-orphan faz.



O valor atribuído a cascade é uma lista de opções de cascata separadas por vírgulas. Isso é um pouco confuso, mas a opção chamada all representa todas as opções em cascata, exceto delete-orphan. Usando o valor all, delete-orphan deixa as opções de cascata padrão habilitadas e adiciona o comportamento de exclusão para órfãos.

O aplicativo agora precisa trabalhar com os dois relacionamentos um-para-muitos para implementar a funcionalidade muitos-para-muitos. Como essas são operações que precisarão ser repetidas com frequência, é uma boa ideia criar métodos auxiliares no modelo User para todas as operações possíveis. Os quatro novos métodos que controlam essa relação são mostrados no [Exemplo 12-3](#).

#### *Exemplo 12-3. app/models/user.py: métodos auxiliares de seguidores*

```
class Usuário(db.Model):
    # ...
    def follow(self, user):
        não self.is_following(user):
            Follow(follower=self, seguido=user)
            db.session.add(f)

    def unfollow(self, user):
        f =
            self.followed.filter_by(followed_id=user.id).first() if f: db.session.delete(f)

    def is_following(self, user):
        return self.followed.filter_by(seguido_id
            =user.id).first() não é Nenhum

    def is_followed_by(self, user):
```

```
return
    self.followers.filter_by( follower_id=user.id).first() não é Nenhum
```

O método follow() insere manualmente uma instância Follow na tabela de associação que vincula um seguidor a um usuário seguido, dando ao aplicativo a oportunidade de definir o campo personalizado. Os dois usuários que estão se conectando são atribuídos manualmente à nova instância Follow em seu construtor e, em seguida, o objeto é adicionado à sessão do banco de dados como de costume.

Observe que não há necessidade de definir manualmente o campo de carimbo de data/hora porque ele foi definido com um valor padrão que define a data e hora atuais. O método unfollow() usa o relacionamento seguido para localizar a instância Follow que vincula o usuário ao usuário seguido que precisa ser desconectado. Para destruir o vínculo entre os dois usuários, o objeto Follow é simplesmente excluído. Os métodos is\_following() e is\_followed\_by() pesquisam os relacionamentos um-para-muitos do lado esquerdo e direito, respectivamente, para o usuário fornecido e retornam True se o usuário for encontrado.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar git checkout 12a para verificar esta versão do aplicativo.  
Esta atualização contém uma migração de banco de dados, portanto, lembre-se de executar python manage.py db upgrade depois de verificar o código.

A parte do banco de dados do recurso agora está concluída. Você pode encontrar um teste de unidade que exerce o relacionamento do banco de dados no repositório de código-fonte no GitHub.

## Seguidores na página de perfil

A página de perfil de um usuário precisa apresentar um botão “Seguir” se o usuário que a visualiza não for um seguidor, ou um botão “Parar de seguir” se o usuário for um seguidor. Também é uma boa adição para mostrar as contagens de seguidores e seguidos, exibir as listas de seguidores e usuários seguidos e mostrar um sinal de “segue você” quando apropriado. As alterações no modelo de perfil do usuário são mostradas no [Exemplo 12-4](#). A [Figura 12-3](#) mostra a aparência das adições na página de perfil.

*Exemplo 12-4. app/templates/user.html: aprimoramentos do seguidor no cabeçalho do perfil do usuário*

```
{% if current_user.can(Permission.FOLLOW) and user != current_user %}
    {% if not current_user.is_following(user) %} <a
        href="{{ url_for('.follow', username=user.username) }}" class="btn btn-
        primary">Seguir</a> { % else %} <a href="{{ url_for('.unfollow',
        username=user.username) }}" class="btn btn-default">Parar de
        seguir</a>
    {% fim se %}
    {% endif %}
    <a href="{{ url_for('.followers', username=user.username) }}>
```

```

Seguidores: <span class="badge">{{ user.followers.count() }}</a> <a href="{{ url_for('.followed_by', username=user.username) }}">

A seguir: <span class="badge">{{ user.followed.count() }}</a> {% if current_user.is_authenticated() and user != current_user and

user.is_following(current_user) %} | <span class="label label-default">Segue você {% endif %}

```

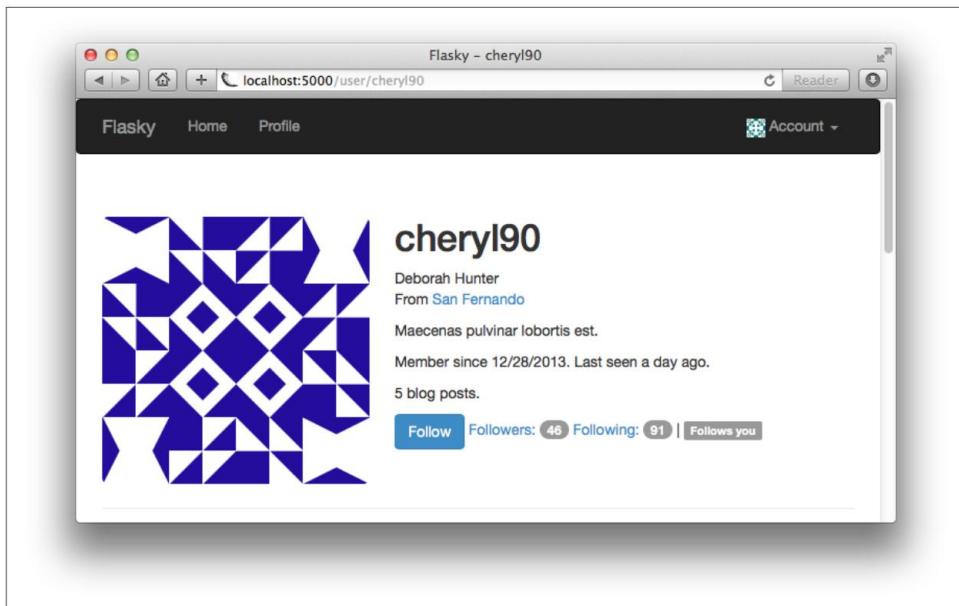


Figura 12-3. Seguidores na página de perfil

Há quatro novos endpoints definidos nessas alterações de modelo. A rota `/follow/<username>` é invocada quando um usuário clica no botão "Follow" na página de perfil de outro usuário. A implementação é mostrada no Exemplo 12-5.

*Exemplo 12-5. app/main/views.py: Siga a rota e veja a função*

```

@main.route('/follow/<username>')
@login_required
@permission_required(Permission.FOLLOW)
def follow(username):
    user = User.query.filter_by(username=username).first()
    if user is None:
        flash('Usuário inválido.')
        return redirect(url_for('.index'))

    if current_user.is_following(user):
        flash('Você já está seguindo este usuário.')

```

```

    return redirect(url_for('.user', username=username))
current_user.follow(user) flash(
Agora você está seguindo %s.' % username) return
redirect(url_for('.user', username=username))

```

Essa função de visualização carrega o usuário solicitado, verifica se ele é válido e se ainda não foi seguido pelo usuário conectado e, em seguida, chama a função auxiliar follow() no modelo User para estabelecer o link. A rota /unfollow/<username> é implementada de maneira semelhante.

A rota /followers/<username> é invocada quando um usuário clica na contagem de seguidores de outro usuário na página de perfil. A implementação é mostrada no [Exemplo 12-6](#).

*Exemplo 12-6. app/main/views.py: Rota dos seguidores e função de visualização*

```

@main.route('/followers/<username>') def
seguidores(username): user =
    User.query.filter_by(username=username).first() se user for None:
        flash('Invalid user.') return redirecionar(url_for('.index'))

    page = request.args.get('page', 1, type=int) pagination
    = user.followers.paginate(
        page, per_page=current_app.config['FLASKY_FOLLOWERS_PER_PAGE'],
        error_out=False) segue = [{'user': item.follower, 'timestamp': item.timestamp}
    para item em pagination.items] return render_template('followers.html', user=user,
    title="Seguidores de",
        endpoint='.followers', pagination=pagination, follow
        =follows)

```

Essa função carrega e valida o usuário solicitado e, em seguida, pagina seu relacionamento de seguidores usando as mesmas técnicas aprendidas no [Capítulo 11](#). Como a consulta de seguidores retorna instâncias de Follow , a lista é convertida em outra lista que possui campos de usuário e timestamp em cada entrada para que renderização é mais simples.

O modelo que renderiza a lista de seguidores pode ser escrito genericamente para que possa ser usado para listas de seguidores e usuários seguidos. O modelo recebe o usuário, um título para a página, o endpoint a ser usado nos links de paginação, o objeto de paginação e a lista de resultados.

O ponto de extremidade seguido\_por é quase idêntico. A única diferença é que a lista de usuários é obtida do relacionamento user.followed . Os argumentos do modelo também são ajustados de acordo.

O modelo `seguidores.html` é implementado com uma tabela de duas colunas que mostra nomes de usuários e seus avatares à esquerda e carimbos de data e hora do Flask-Moment à direita. Você pode consultar o repositório de código-fonte no GitHub para estudar a implementação em detalhes.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar git checkout 12b para verificar esta versão do aplicativo.

## Consultar postagens seguidas usando uma associação de banco de dados

A página inicial do aplicativo atualmente mostra todas as postagens no banco de dados em ordem cronológica decrescente. Com o recurso de seguidores agora completo, seria uma boa adição dar aos usuários a opção de visualizar as postagens do blog apenas dos usuários que eles seguem.

A maneira óbvia de carregar todas as postagens criadas por usuários seguidos é primeiro obter a lista desses usuários e, em seguida, obter as postagens de cada um e classificá-las em uma única lista. É claro que essa abordagem não se adapta bem; o esforço para obter essa lista combinada aumentará à medida que o banco de dados crescer, e operações como paginação não podem ser feitas com eficiência. A chave para obter as postagens do blog com bom desempenho é fazê-lo com uma única consulta.

A operação de banco de dados que pode fazer isso é chamada de *junção*. Uma operação de junção usa duas ou mais tabelas e localiza todas as combinações de linhas que satisfazem uma determinada condição. As linhas combinadas resultantes são inseridas em uma tabela temporária que é o resultado da junção.

A melhor maneira de explicar como as junções funcionam é por meio de um exemplo.

A Tabela 12-1 mostra um exemplo de tabela de usuários com três usuários.

Tabela 12-1. tabela de usuários

nome de usuário
1 João
2 susan
3 Davi

A Tabela 12-2 mostra a tabela de postagens correspondente , com algumas postagens do blog.

Tabela 12-2. Tabela de postagens

id	author_id	corpo
1	2	Postagem no blog de susan
2	1	Postagem no blog de João
3	3	Postagem no blog de Davi
4	1	Segunda postagem do blog por john

Finalmente, a Tabela 12-3 mostra quem está seguindo quem. Nesta tabela você pode ver que *John* está seguindo *David*, *Susan* está seguindo *John* e *David* não está seguindo ninguém.

Tabela 12-3. Segue tabela

id_seguidor	id_seguidor
1	3
2	1
2	3

Para obter a lista de posts seguidos pela usuária *susan*, as tabelas de posts e seguidores devem ser combinado. Primeiro, a tabela a seguir é filtrada para manter apenas as linhas que têm *susan* como o seguidor, que neste exemplo são as duas últimas linhas. Em seguida, uma tabela de junção temporária é criado a partir de todas as combinações possíveis de linhas dos posts e filtrado **segue tabelas em que o author\_id do post é o mesmo que o follow\_id de a seguir**, selecionando efetivamente quaisquer postagens que apareçam na lista de usuários que *Susan* está seguindo. A Tabela 12-4 mostra o resultado da operação de junção. As colunas que foram usadas para executar a junção são marcadas com \* nesta tabela.

Tabela 12-4. Tabela unida

id autor_id* corpo		id_seguidor	id_seguidor*
2 1	Postagem no blog de João	2	1
3 3	Postagem no blog de Davi	2	3
4 1	Segunda postagem do blog por john 2		1

Esta tabela contém exatamente a lista de postagens de blog criadas por usuários que *susan* está seguindo. A consulta Flask-SQLAlchemy que literalmente executa a operação de junção conforme descrito é bastante complexo:

```
return db.session.query(Post).select_from(Follow).  
    filter_by(follower_id=self.id).  
    join(Post, Follow.followed_id == Post.author_id)
```

Todas as consultas que você viu até agora começam no atributo de consulta do modelo que é consultado. Esse formato não funciona bem para esta consulta, porque a consulta precisa retornar linhas de postagens , mas a primeira operação que precisa ser feita é aplicar um filtro ao segue tabela. Portanto, uma forma mais básica da consulta é usada. Para entender completamente esta consulta, cada parte deve ser analisada individualmente:

- db.session.query(Post) especifica que esta será uma consulta que retornará Post objetos.
- select\_from(Follow) diz que a consulta começa com o modelo Follow .
- filter\_by(follower\_id=self.id) realiza a filtragem da tabela a seguir por o usuário seguidor.

- join(Post, Follow.followed\_id == Post.author\_id) une os resultados de filter\_by() com os objetos Post .

A consulta pode ser simplificada trocando a ordem do filtro e da junção:

```
return Post.query.join(Follow, Follow.followed_id == Post.author_id)\n    .filter(Followfollower_id == self.id)
```

Emitindo a operação de junção primeiro, a consulta pode ser iniciada a partir de Post.query, então agora os únicos dois filtros que precisam ser aplicados são join() e filter(). Mas isso é o mesmo? Pode parecer que fazer a junção primeiro e depois a filtragem daria mais trabalho, mas na realidade essas duas consultas são equivalentes. O SQLAlchemy primeiro coleta todos os filtros e depois gera a consulta da maneira mais eficiente. As instruções SQL nativas para essas duas consultas são idênticas. A versão final dessa consulta é adicionada ao Postmodel, conforme mostrado no [Exemplo 12-7](#).

*Exemplo 12-7. app/models/user.py: obter postagens seguidas*

```
class User(db.Model):\n    ...\n\n    @property\n    def seguido_posts (self):\n        return Post.query.join(Follow, Follow.followed_id == Post.author_id)\n            .filter(Followfollower_id == self.id)
```

Observe que o método follow\_posts() é definido como uma propriedade para que não precise do (). Dessa forma, todos os relacionamentos têm uma sintaxe consistente.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar git checkout 12c para verificar esta versão do aplicativo.

As junções são extremamente difíceis de entender; você pode precisar experimentar o código de exemplo em um shell antes que tudo seja absorvido.

## Mostrar postagens seguidas na página inicial

A página inicial agora pode dar aos usuários a opção de visualizar todas as postagens do blog ou apenas as de usuários seguidos. O [Exemplo 12-8](#) mostra como esta escolha é implementada.

*Exemplo 12-8. app/main/views.py: Mostrar todas as postagens seguidas*

```
@app.route('/', métodos = ['GET', 'POST']) def index():\n\n    # ...\n    show_followed = Falso
```

```

if current_user.is_authenticated():
    show_followed = bool(request.cookies.get('show_followed', ""))
    if show_followed:
        query = current_user.followed_posts else:

            query = Post.query
            paginação = query.order_by(Post.timestamp.desc()).paginate(
                page, per_page=current_app.config['FLASKY_POSTS_PER_PAGE'],
                error_out=False)
            posts = pagination.items return render_template('index.html',
                form=form, posts=posts,
                show_followed=show_followed, paginação=paginação)

```

A opção de mostrar todas as postagens seguidas é armazenada em um cookie chamado `show_followed` que, quando definido como uma string não vazia, indica que apenas as postagens seguidas devem ser mostradas. Os cookies são armazenados no objeto de solicitação como um dicionário `request.cookies`. O valor da string do cookie é convertido em um booleano e, com base em seu valor, uma variável local de consulta é definida para a consulta que obtém as listas completas ou filtradas de postagens do blog. Para mostrar todas as postagens, a consulta de nível superior `Post.query` é usada e a propriedade `User.followed_posts` adicionada recentemente é usada quando a lista deve ser restrita a seguidores.

A consulta armazenada na variável local da consulta é então paginada e os resultados enviados para o modelo como antes.

O cookie `show_followed` é definido em duas novas rotas, mostradas no [Exemplo 12-9](#).

*Exemplo 12-9. app/main/views.py: Seleção de todas as postagens seguidas*

```

@main.route('/all')
@login_required def
show_all(): resp =
    make_response(redirect(url_for('.index')))
    resp.set_cookie('show_followed', "", max_age=30*24*60 *60) retorno resp

@main.route('/followed')
@login_required def
show_followed(): resp =
    make_response(redirect(url_for('.index')))
    resp.set_cookie('show_followed', '1', max_age=30*24* 60*60) retorno resp

```

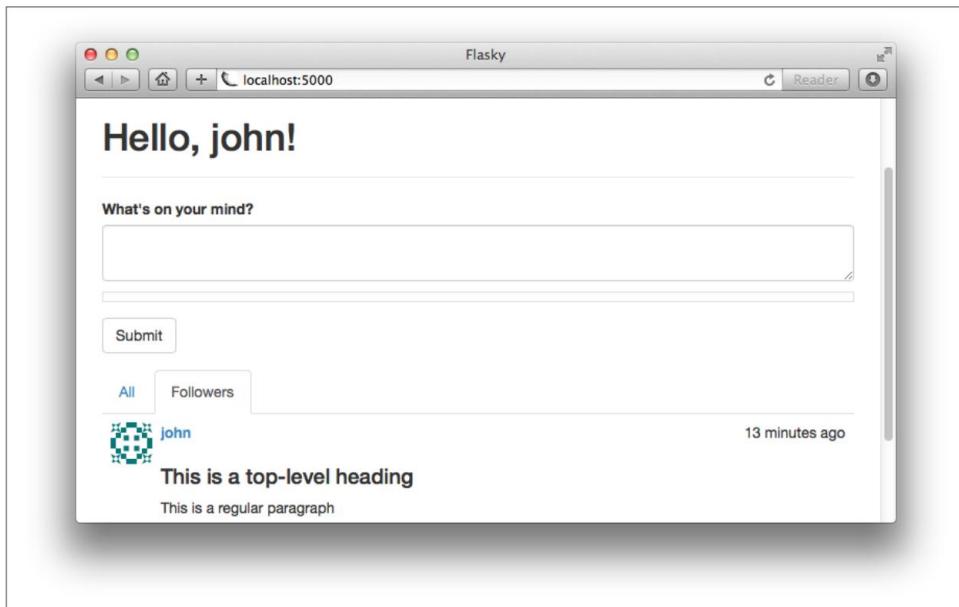
Links para essas rotas são adicionados ao modelo de página inicial. Quando eles são invocados, o cookie `show_followed` é definido com o valor apropriado e um redirecionamento de volta para a página inicial é emitido.

Os cookies podem ser definidos apenas em um objeto de resposta, portanto, essas rotas precisam criar um objeto de resposta por meio de `make_response()` em vez de permitir que o Flask faça isso.

A função `set_cookie()` recebe o nome do cookie e o valor como os dois primeiros argumentos. O argumento opcional `max_age` define o número de segundos até que o cookie

expira. Não incluir este argumento faz com que o cookie expire quando a janela do navegador for fechada. Nesse caso, uma idade de 30 dias é definida para que a configuração seja lembrada mesmo que o usuário não retorne ao aplicativo por vários dias.

As alterações no modelo adicionam duas guias de navegação na parte superior da página que invocam as rotas `/all` ou `/followed` para definir as configurações corretas na sessão. Você pode inspecionar as alterações do modelo em detalhes no repositório de código-fonte no GitHub. A [Figura 12-4](#) mostra a aparência da página inicial com essas alterações.



*Figura 12-4. Posts seguidos na página inicial*



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar `git checkout 12d` para verificar esta versão do aplicativo.

Se você experimentar o aplicativo neste momento e alternar para a lista de postagens seguidas, notará que suas próprias postagens não aparecem na lista. Claro que isso está correto, porque os usuários não são seguidores de si mesmos.

Mesmo que as consultas estejam funcionando conforme o planejado, a maioria dos usuários espera ver suas próprias postagens quando estiver vendo as de seus amigos. A maneira mais fácil de resolver esse problema é registrar todos os usuários como seus próprios seguidores no momento em que são criados. Esse truque é mostrado no [Exemplo 12-10](#).

*Exemplo 12-10. app/models/user.py: faça dos usuários seus próprios seguidores na construção*

```
class User(UserMixin, db.Model):
    ...
    # def __init__(self, **kwargs):
    # ...
    self.follow(self)
```

Infelizmente, você provavelmente tem vários usuários no banco de dados que já foram criados e não estão seguindo a si mesmos. Se o banco de dados for pequeno e fácil de regenerar, ele poderá ser excluído e recriado, mas se isso não for uma opção, adicionar uma função de atualização que corrija os usuários existentes é a solução adequada. Isso é mostrado no [Exemplo 12-11](#).

*Exemplo 12-11. app/models/user.py: Faça dos usuários seus próprios seguidores*

```
class User(UserMixin, db.Model):
    ...
    # @staticmethod
    def add_self_follows(): para
        usuário em User.query.all():
            se não for user.is_following(user):
                user.follow(user)
                db.session.add(user)
                db.session.commit()
    # ...
```

Agora, o banco de dados pode ser atualizado executando a função de exemplo anterior do shell:

```
(venv) $ python manage.py shell >>>
User.add_self_follows()
```

A criação de funções que introduzem atualizações no banco de dados é uma técnica comum usada para atualizar aplicativos implantados, pois executar uma atualização com script é menos propenso a erros do que atualizar bancos de dados manualmente. No [Capítulo 17](#), você verá como essa função e outras semelhantes podem ser incorporadas a um script de implantação.

Tornar todos os usuários auto-seguidores torna o aplicativo mais utilizável, mas essa mudança apresenta algumas complicações. As contagens de seguidores e usuários seguidos mostradas na página de perfil do usuário agora são aumentadas em um devido aos links de auto-seguidor. Os números precisam ser reduzidos em um para serem precisos, o que é fácil de fazer diretamente no modelo renderizando {{ user.followers.count() - 1 }} e {{ user.followed.count() - 1 }}. As listas de seguidores e usuários seguidos também devem ser ajustadas para não mostrar o mesmo usuário, outra tarefa simples de se fazer no template com uma condicional. Finalmente, quaisquer testes de unidade que verificam contagens de seguidores também são afetados pelos links de auto-seguidores e devem ser ajustados para levar em conta os auto-seguidores.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar checkout 12e para verificar esta versão do aplicativo.

No próximo capítulo, o subsistema de comentários do usuário será implementado - outro recurso muito importante de aplicativos socialmente conscientes.

## CAPÍTULO 13

### Comentários do usuário

Permitir que os usuários interajam é a chave para o sucesso de uma plataforma de blog social. Neste capítulo, você aprenderá como implementar comentários de usuários. As técnicas apresentadas são genéricas o suficiente para serem diretamente aplicáveis a um grande número de aplicações socialmente habilitadas.

### Representação do banco de dados de comentários

Os comentários não são muito diferentes das postagens do blog. Ambos têm um corpo, um autor e um carimbo de data/hora e, nesta implementação específica, ambos são escritos com a sintaxe Markdown. A [Figura 13-1](#) mostra um diagrama da tabela de comentários e seus relacionamentos com outras tabelas do banco de dados.

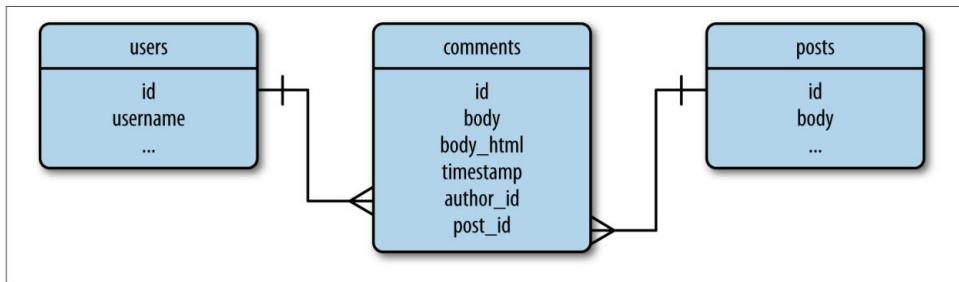


Figura 13-1. Representação de banco de dados de comentários de postagem de blog

Os comentários aplicam postagens de blog específicas, portanto, é definido um relacionamento um-para-muitos da tabela de postagens . Esse relacionamento pode ser usado para obter a lista de comentários associados a uma postagem de blog específica.

A tabela de comentários também está em um relacionamento um-para-muitos com a tabela de usuários . Essa relação dá acesso a todos os comentários feitos por um usuário e, indiretamente, quantos comentários um usuário escreveu, uma informação que pode ser interessante mostrar nas páginas de perfil do usuário. A definição do modelo Comentário é mostrada no [Exemplo 13-1](#).

*Exemplo 13-1. app/models.py: modelo de comentários*

```
class Comment(db.Model):
    __tablename__ = 'comentários'
    id = db.Column(db.Integer, primary_key=True)
    body = db.Column(db.Text)
    body_html = db.Column(db.Text)
    timestamp = db.Column(db.DateTime, index=True,
        default=datetime.utcnow)
    disabled = db.Column(db.Boolean)
    author_id = db.Column(db.Integer, db.ForeignKey('users.id'))
    post_id = db.Column(db.Integer, db.ForeignKey('posts.id'))

    @staticmethod
    def on_changed_body(target, value, oldvalue, initiator):
        allowed_tags = ['a', 'abbr', 'acronym', 'b', 'code', 'em', 'i', 'strong']
        target.body_html = bleach.linkify(bleach.clean(
            markdown(value, output_format='html'),
            tags=allowed_tags, strip=True))

db.event.listen(Comment.body, 'set', Comment.on_changed_body)
```

Os atributos do modelo Comment são quase os mesmos do Post. Uma adição é o campo desabilitado , um booleano que será usado pelos moderadores para suprimir comentários inapropriados ou ofensivos. Como foi feito para as postagens do blog, os comentários definem um evento que aciona sempre que o campo do corpo muda, automatizando a renderização do texto Markdown para HTML. O processo é idêntico ao que foi feito para postagens de blog no [Capítulo 11](#), mas como os comentários tendem a ser curtos, a lista de tags HTML permitidas na conversão do Markdown é mais restritiva, as tags relacionadas a parágrafos foram removidas e restam apenas as tags de formatação de caracteres.

Para concluir as alterações do banco de dados, os modelos User e Post devem definir os relacionamentos um para muitos com a tabela de comentários , conforme mostrado no [Exemplo 13-2](#).

*Exemplo 13-2. app/models/user.py: relacionamentos um-para-muitos de usuários e postagens para comentários*

```
class Usuário(db.Model):
    # ...
    comentários = db.relationship('Comentário', backref='autor',
        preguiçoso='dinâmico')

class Post(db.Model):
    ...
    comentários = db.relationship('Comentário', backref='post',
        lazy='dynamic')
```

## Envio e exibição de comentários

Neste aplicativo, os comentários são exibidos nas páginas de postagem de blog individuais que foram adicionadas como links permanentes no [Capítulo 11](#). Um formulário de envio também está incluído nesta página. O [Exemplo 13-3](#) mostra o formulário da web que será usado para inserir comentários – um formulário extremamente simples que possui apenas um campo de texto e um botão de envio.

*Exemplo 13-3. app/main/forms.py: formulário de entrada de comentários*

**class CommentForm(Formulário):**

```
body = StringField("", validators=[Required()])
submit = SubmitField('Submit')
```

O [Exemplo 13-4](#) mostra a rota `/post/<int:id>` atualizada com suporte para comentários.

*Exemplo 13-4. app/main/views.py: suporte a comentários de postagem do blog*

```
@main.route('/post/<int:id>', métodos=['GET', 'POST'])
def post(id):
    post = Post.query.get_or_404(id)
    form = CommentForm()
    if form.validate_on_submit():
        comment = Comment(body=form.body.data,
                           post=post,
                           author=current_user._get_current_object())
        db.session.add(comment)
        flash('Seu comentário foi publicado.')
        redirect(url_for('.post', id=post.id, page=-1))
    página = request.args.get('page', 1, type=int)
    if page == -1:
        page = (post.comments.count() - 1) // current_app.config['FLASKY_COMMENTS_PER_PAGE'] + 1
    comentários = post.comments.order_by(Comment.timestamp.asc()).paginate(
        page, per_page=current_app.config['FLASKY_COMMENTS_PER_PAGE'], error_out=False)
    return render_template('post.html', comentários=comentários, paginação=paginação,
                           posts=[post], form=form,
```

comentários=comentários, paginação=paginação)

Essa função de visualização instancia o formulário de comentário e o envia para o modelo `post.html` para renderização. A lógica que insere um novo comentário quando o formulário é enviado é semelhante ao tratamento das postagens do blog. Como no caso `Post`, o autor do comentário não pode ser definido diretamente como `current_user` porque este é um objeto proxy de variável de contexto.

A expressão `current_user._get_current_object()` retorna o objeto `User` real.

Os comentários são classificados por carimbo de data/hora em ordem cronológica, para que novos comentários sejam sempre adicionados na parte inferior da lista. Quando um novo comentário é inserido, o redirecionamento que finaliza a solicitação volta para a mesma URL, mas a função `url_for()` define a página como `-1`, um número de página especial que é usado para solicitar a última página de comentários para que o comentário recém-inserido é visto na página. Quando o número da página é obtido

da string de consulta e encontrado como -1, um cálculo com o número de comentários e o tamanho da página é feito para obter o número de página real a ser usado.

A lista de comentários associados à postagem é obtida por meio da relação `post.comments` um-para-muitos, classificada por carimbo de data/hora do comentário e paginada com as mesmas técnicas usadas para postagens de blog. Os comentários e o objeto de paginação são enviados ao modelo para renderização. A variável de configuração `FLASKY_COMMENTS_PER_PAGE` é adicionada ao `config.py` para controlar o tamanho de cada página de comentários.

A renderização de comentários é definida em um novo modelo `_comments.html` que é semelhante a `_posts.html`, mas usa um conjunto diferente de classes CSS. Este modelo é incluído por `_post.html` abaixo do corpo da postagem, seguido por uma chamada para a macro de paginação. Você pode revisar as alterações nos modelos no repositório GitHub do aplicativo.

Para completar esse recurso, as postagens do blog exibidas nas páginas inicial e de perfil precisam de um link para a página com os comentários. Isso é mostrado no [Exemplo 13-5](#).

*Exemplo 13-5. `_app/templates/_posts.html`: Link para comentários da postagem do blog*

```
<a href="{{ url_for('post', id=post.id) }}#comments"> <span
    class="label label-primary">
        {{ post.comments.count() }} Comentários </
    a>
```

Observe como o texto do link inclui o número de comentários, que é facilmente obtido a partir do relacionamento um-para-muitos entre as postagens e as tabelas de comentários usando o filtro `count()` do SQLAlchemy.

Também é interessante a estrutura do link para a página de comentários, que é construída como o link permanente para a postagem com um sufixo `#comments` adicionado. Esta última parte é chamada de *fragmento de URL* e é usada para indicar uma posição de rolagem inicial para a página. O navegador da web procura um elemento com o id fornecido e rola a página para que o elemento apareça no topo da página. Essa posição inicial é definida para o cabeçalho de comentários no modelo `post.html`, que é escrito como `<h4 id="comments">Comentários</h4>`.

A [Figura 13-2](#) mostra como os comentários aparecem na página.

Uma alteração adicional foi feita na macro de paginação. Os links de paginação para comentários também precisam do fragmento `#comments` adicionado, portanto, um argumento de fragmento foi adicionado à macro e passado na invocação de macro do modelo `post.html`.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar `git checkout 13a` para verificar esta versão do aplicativo. Esta atualização contém uma migração de banco de dados, portanto, lembre-se de executar `python manage.py db upgrade` depois de verificar o código.

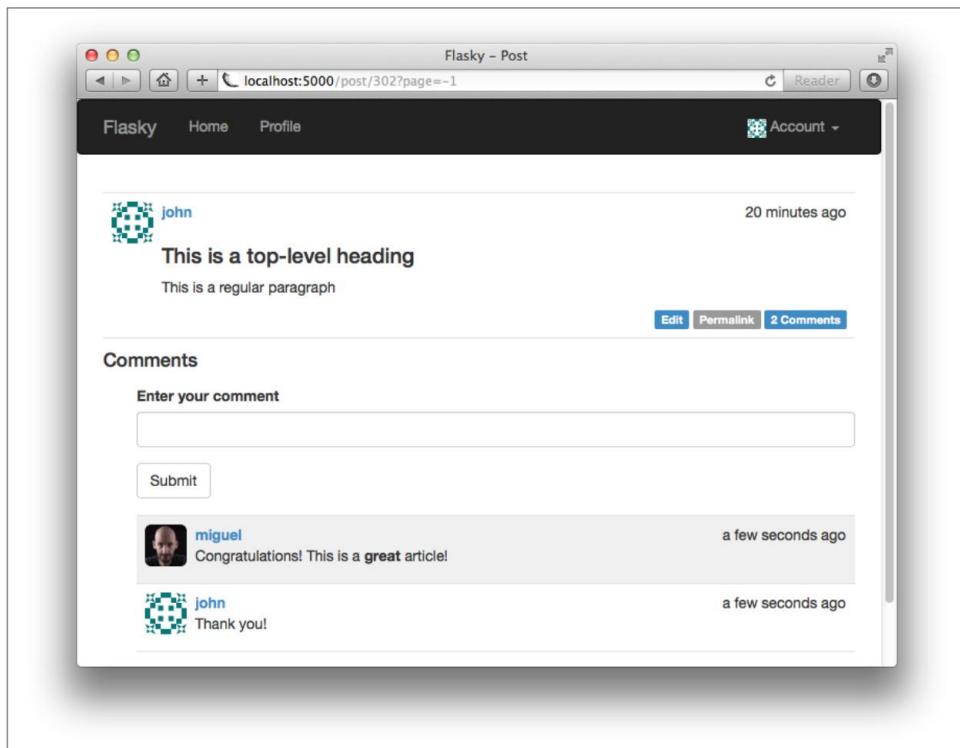


Figura 13-2. Comentários da postagem do blog

## Moderação dos comentários

No [Capítulo 9](#) foi definida uma lista de funções de usuário, cada uma com uma lista de permissões. Uma das permissões foi `Permission.MODERATE_COMMENTS`, que dá aos usuários que têm em suas funções o poder de moderar comentários feitos por outras pessoas.

Esse recurso será exposto como um link na barra de navegação que aparece apenas para usuários autorizados a usá-lo. Isso é feito no modelo `base.html` usando uma condicional, conforme mostrado no [Exemplo 13-6](#).

*Exemplo 13-6. app/templates/base.html: link de comentários moderados na barra de navegação*

```
...
{%
if current_user.can(Permission.MODERATE_COMMENTS) %}
<li><a href="{{ url_for('main.moderate') }}">Moderar comentários</a></li>
{%
endif %}
...

```

A página de moderação mostra os comentários de todas as postagens na mesma lista, com os comentários mais recentes exibidos primeiro. Abaixo de cada comentário há um botão que pode alternar o atributo desativado . A rota /moderate é mostrada no [Exemplo 13-7](#).

*Exemplo 13-7. app/main/views.py: rota de moderação de comentários*

```
@main.route('/moderate')
@login_required
@permission_required(Permission.MODERATE_COMMENTS)
def moderate(): page = request.args.get('page', 1, type=int)
    pagination =
        Comment.query.order_by(Comment.timestamp.desc()).paginate(page,
            per_page=current_app.config['FLASKY_COMMENTS_PER_PAGE'], error_out=False)
    comments = pagination.items return render_template('moderate.html',
        comments=comments,
        paginação=paginação, página=página)
```

Esta é uma função muito simples que lê uma página de comentários do banco de dados e os passa para um template para renderização. Junto com os comentários, o template recebe o objeto de paginação e o número da página atual.

O modelo *mode.html* , mostrado no [Exemplo 13-8](#), também é simples porque se baseia no *submodelo\_comments.html* criado anteriormente para a renderização dos comentários.

*Exemplo 13-8. app/templates/moderate.html: modelo de moderação de comentários*

```
{% extends "base.html" %}
import "_macros.html" como macros %

{% block title %}Flasky - Moderação de comentários{% endblock %}

{% block page_content %}
<div class="page-header">
    <h1>Moderação de comentários</h1>
    <div> {% set mode = True %} {% include '_comments.html' %} {% if
paginação %} <div class="pagination">
    {{ macros.pagination_widget(pagination,
'moderate') }} </div> {% endif %} {% endblock %}
```

Este template adia a renderização dos comentários para o template *\_comments.html* , mas antes de passar o controle para o template subordinado ele usa a diretiva set do Jinja2 para definir uma variável de template moderada configurada como True. Esta variável é usada pelo template *\_comments.html* para determinar se os recursos de moderação precisam ser renderizados.

A parte do modelo `_comments.html` que renderiza o corpo de cada comentário precisa ser modificada de duas maneiras. Para usuários regulares (quando a variável moderada não está definida), quaisquer comentários marcados como desabilitados devem ser suprimidos. Para moderadores (quando moderado é definido como True), o corpo do comentário deve ser renderizado independentemente do estado desabilitado e abaixo do corpo deve ser incluído um botão para alternar o estado.

O Exemplo 13-9 mostra essas mudanças.

*Exemplo 13-9. app/templates/\_comments.html: renderização dos corpos dos comentários*

```
...
<div class="comment-body">
    {% if comment.disabled %}
        <p></p><i>Este comentário foi desativado por um moderador.</i></p> {% endif %}
    {% if moderado or not comment.disabled %} {% if comment.body_html %}
        {{ comment.body_html | safe }} {% else %} {{ comment.body }} {% endif %}
    {% endif %}
</div> {% if
moderado %} <br>
    {% if
comment.disabled %} <a
class="btn btn-default btn-xs" href="{{ url_for('.moderate_enable',
id=comment.id, page=page) }}">Ativar</a> {% else %} <a class="btn btn-danger btn-xs"
href="{{ url_for('.moderate_disable', id=comment.id, page=page) }}">Desativar</a>
    {% fim se %}
    {% fim se %}
...

```

Com essas alterações, os usuários verão um breve aviso para comentários desabilitados. Os moderadores verão o aviso e o corpo do comentário. Os moderadores também verão um botão para alternar o estado desabilitado abaixo de cada comentário. O botão invoca uma das duas novas rotas, dependendo de qual dos dois estados possíveis o comentário está mudando.

O Exemplo 13-10 mostra como essas rotas são definidas.

*Exemplo 13-10. app/main/views.py: rotas de moderação de comentários*

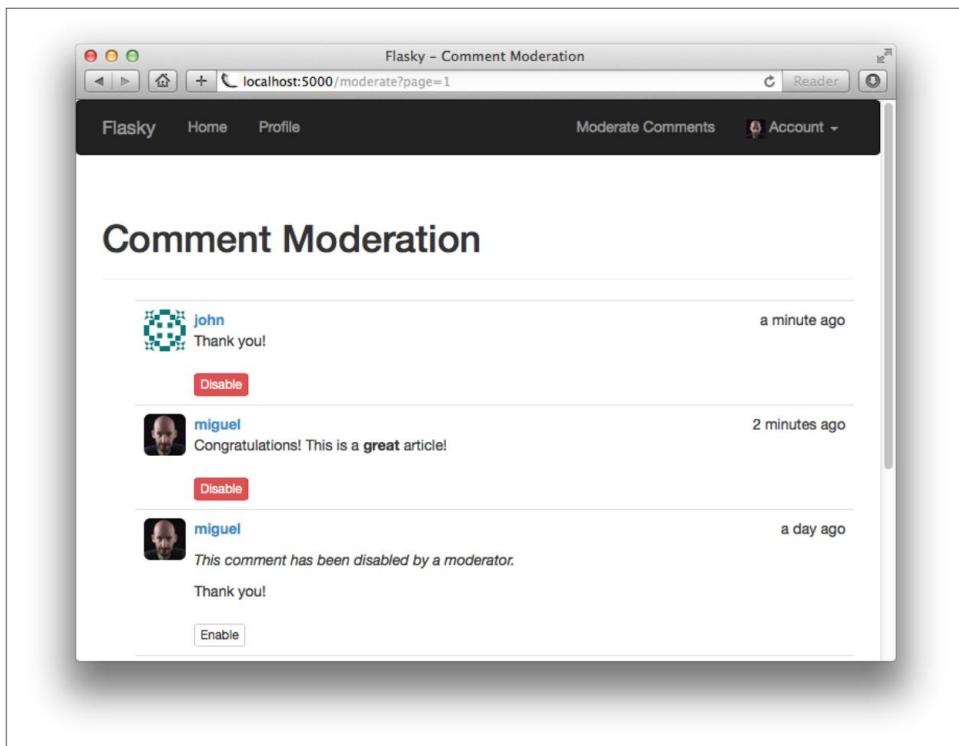
```
@main.route('/moderate/enable/<int:id>')
@login_required
@permission_required(Permission.MODERATE_COMMENTS)
def moderate_enable(id): comment =
    Comment.query.get_or_404(id) comment.disabled = False
    db.session.add(comentário) return
    redirect(url_for('.moderate', page=request.args.get('page',
1, type=int)))
```

```

@main.route('/moderate/disable/<int:id>')
@login_required
@permission_required(Permission.MODERATE_COMMENTS)
def mode_disable (id):
    comment = Comment.query.get_or_404(id)
    comment.disabled = Verdadeiro
    db.session.add(comentário) return
    redirect(url_for('.moderate',
                    page=request.args.get('page', 1, type=int)))

```

As rotas de habilitação e desabilitação de comentários carregam o objeto de comentário, definem o campo desabilitado com o valor apropriado e o gravam de volta no banco de dados. No final, eles redirecionam de volta para a página de moderação de comentários (mostrada na [Figura 13-3](#)) e, se um argumento de página foi fornecido na string de consulta, eles o incluem no redirecionamento. Os botões no *template\_comments.html* foram renderizados com o argumento page para que o redirecionamento traga o usuário de volta para a mesma página.



*Figura 13-3. Página de moderação de comentários*



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar checkout 13b para verificar esta versão do aplicativo.

O tópico das características sociais é completado com este capítulo. No próximo capítulo, você aprenderá como expor a funcionalidade do aplicativo como uma API que clientes que não sejam navegadores da Web podem usar.



---

CAPÍTULO 14

# Interfaces de programação de aplicativos

Nos últimos anos, tem havido uma tendência nas aplicações web de mover cada vez mais a lógica de negócios para o lado do cliente, produzindo uma arquitetura que é conhecida como Rich Internet Application (RIA). Em RIAs, a função principal (e às vezes única) do servidor é fornecer ao aplicativo cliente serviços de recuperação e armazenamento de dados. Nesse modelo, o servidor se torna um *web service* ou *Application Programming Interface (API)*.

Existem vários protocolos pelos quais as RIAs podem se comunicar com um serviço da web. Protocolos RPC (Remote Procedure Call) como XML-RPC ou seu derivado Simplified Object Access Protocol (SOAP) eram escolhas populares há alguns anos. Mais recentemente, a arquitetura Representational State Transfer (REST) surgiu como a favorita para aplicações web por ser construída no modelo familiar da World Wide Web.

Flask é um framework ideal para construir web services *RESTful* devido à sua natureza leve. Neste capítulo, você aprenderá como implementar uma API RESTful baseada em Flask.

## Introdução ao REST

Ph.D. de Roy Fielding . [dissertação](#) introduz o estilo arquitetural REST para web services listando suas seis características definidoras:

### Servidor cliente

Deve haver uma separação clara entre os clientes e o servidor.

### Stateless

Uma solicitação de cliente deve conter todas as informações necessárias para realizá-la.

O servidor não deve armazenar nenhum estado sobre o cliente que persista de uma solicitação para a próxima.

As

respostas de *cache* do servidor podem ser rotuladas como cacheable ou noncacheable para que os clientes (ou intermediários entre clientes e servidores) possam usar um cache para fins de otimização.

#### *Interface uniforme* O

protocolo pelo qual os clientes acessam os recursos do servidor deve ser consistente, bem definido e padronizado. A interface uniforme comumente usada de web services REST é o protocolo HTTP.

Servidores, caches

ou gateways do *System Proxy* em camadas podem ser inseridos entre clientes e servidores conforme necessário para melhorar o desempenho, a confiabilidade e a escalabilidade.

Clientes *Code-on-*

*Demand* podem opcionalmente fazer download do código do servidor para executar em seu contexto.

#### **Recursos são tudo** O conceito de

*recursos* é fundamental para o estilo de arquitetura REST. Nesse contexto, um recurso é um item de interesse no domínio da aplicação. Por exemplo, no aplicativo de blogging, usuários, postagens de blog e comentários são todos recursos.

Cada recurso deve ter um URL exclusivo que o represente. Continuando com o exemplo de blog, uma postagem de blog pode ser representada pela URL `/api/posts/12345`, em que 12345 é um identificador exclusivo para a postagem, como a chave primária do banco de dados da postagem. O formato ou conteúdo do URL não importa muito; tudo o que importa é que cada URL de recurso identifique exclusivamente um recurso.

Uma coleção de todos os recursos em uma classe também tem uma URL atribuída. A URL para a coleção de postagens do blog pode ser `/api/posts/` e a URL para a coleção de todos os comentários pode ser `/api/comments/`.

Uma API também pode definir URLs de coleção que representam subconjuntos lógicos de todos os recursos em uma classe. Por exemplo, a coleção de todos os comentários na postagem de blog 12345 pode ser representada pela URL `/api/posts/12345/comments/`. É uma prática comum definir URLs que representam coleções de recursos com uma barra final, pois isso lhes dá uma representação de “pasta”.



Esteja ciente de que o Flask aplica um tratamento especial às rotas que terminam com uma barra. Se um cliente solicitar uma URL sem uma barra final e a única rota correspondente tiver uma barra no final, o Flask responderá automaticamente com um redirecionamento para a URL com barra final. Nenhum redirecionamento é emitido para o caso inverso.

## Métodos de solicitação

O aplicativo cliente envia solicitações ao servidor nas URLs de recursos estabelecidas e usa o *método* de solicitação para indicar a operação desejada. Para obter a lista de postagens de blog disponíveis na API de blogs, o cliente enviará uma solicitação GET para <http://www.example.com/api/posts/> e, para inserir uma nova postagem, enviará uma solicitação POST para a mesma URL, com o conteúdo da postagem do blog no corpo da solicitação. Para recuperar a postagem de blog 12345, o cliente enviará uma solicitação GET para <http://www.example.com/api/posts/12345>. A [Tabela 14-1](#) lista os métodos de solicitação que são comumente usados em APIs RESTful com seus significados.

*Tabela 14-1. Métodos de solicitação HTTP em APIs RESTful*

Método de solicitação	Alvo	Descrição	HTTP código de estado
OBTER	Recurso individual URL	Obtenha o recurso.	200
OBTER	Coleta de recursos URL	Obtenha a coleção de recursos (ou uma página dela se o servidor implementar a paginização).	200
PUBLICAR	Coleta de recursos URL	Crie um novo recurso e adicione à coleção. O servidor escolhe o URL do novo recurso e o retorna em um cabeçalho de localização na resposta.	201
POR	Recurso individual URL	Modifique um recurso existente. Alternativamente, este método também pode ser usado para criar um novo recurso quando o cliente pode escolher esse recurso URL.	200
EXCLUIR	Recurso individual URL	Excluir um recurso.	200
EXCLUIR	Coleta de recursos URL	Exclua todos os recursos da coleção.	200



A arquitetura REST não requer que todos os métodos sejam implementados para um recurso. Se o cliente invocar um método que não tem suporte para um determinado recurso, uma resposta com o código de status 405 para “Método não permitido” deve ser retornada. O Flask trata esse erro automaticamente.

## Corpos de solicitação e resposta Os

recursos são enviados entre cliente e servidor nos corpos de solicitações e respostas, mas o REST não especifica o formato a ser usado para codificar recursos. O cabeçalho Content-Type em solicitações e respostas é usado para indicar o formato no qual um recurso é codificado no corpo. Os mecanismos de negociação de conteúdo padrão no protocolo HTTP podem ser usados entre cliente e servidor para concordar com um formato que ambos suportam.

Os dois formatos comumente usados com web services RESTful são JavaScript Object Notation (JSON) e Extensible Markup Language (XML). Para RIAs baseadas na web, o JSON é atraente por causa de seus laços estreitos com o JavaScript, a linguagem de script do lado do cliente usada pelos navegadores da web. Voltando à API de exemplo de blog, um recurso de postagem de blog pode ser representado em JSON da seguinte forma:

```
{
  "url": "http://www.example.com/api/posts/12345", "title": "Escrevendo APIs RESTful em Python", "author": "http://www.example.com/api/users/2", "body": "... texto do artigo aqui ...", "comments": "http://www.example.com/api/posts/12345/comments"
}
```

Observe como os campos de URL, autor e comentários na postagem do blog acima são URLs de recursos totalmente qualificados. Isso é importante porque esses URLs permitem que o cliente descubra novos Recursos.

Em uma API RESTful bem projetada, o cliente apenas conhece uma pequena lista de URLs de recursos de nível superior e descobre o restante dos links incluídos nas respostas, semelhante a como você pode descobrir novas páginas da Web enquanto navega na Web clicando em links que aparecem em páginas que você conhece.

## Versionamento

Em um aplicativo da Web tradicional centrado no servidor, o servidor tem controle total do aplicativo. Quando um aplicativo é atualizado, basta instalar a nova versão no servidor para atualizar todos os usuários, pois até mesmo as partes do aplicativo que são executadas no navegador da Web do usuário são baixadas do servidor.

A situação com RIAs e serviços da Web é mais complicada, porque geralmente os clientes são desenvolvidos independentemente do servidor - talvez até por pessoas diferentes. Considere o caso de um aplicativo em que o serviço da Web RESTful é usado por uma variedade de clientes, incluindo navegadores da Web e clientes de smartphone nativos. O cliente do navegador da web pode ser atualizado no servidor a qualquer momento, mas os aplicativos do smartphone não podem ser atualizados à força; o proprietário do smartphone precisa permitir que a atualização aconteça. Mesmo que o proprietário do smartphone esteja disposto a atualizar, não é possível cronometrar a implantação dos aplicativos de smartphone atualizados em todas as lojas de aplicativos para coincidir exatamente com a implantação do novo servidor.

Por esses motivos, os serviços da Web precisam ser mais tolerantes do que os aplicativos da Web comuns e poder trabalhar com versões antigas de seus clientes. Uma maneira comum de resolver esse problema é a versão das URLs tratadas pelo serviço da web. Por exemplo, a primeira versão do serviço da web de blogs pode expor a coleção de postagens de blog em `/api/v1.0/posts/`.

Incluir a versão do serviço da Web na URL ajuda a manter os recursos antigos e novos organizados para que o servidor possa fornecer novos recursos a novos clientes enquanto continua a oferecer suporte

clientes antigos. Uma atualização no serviço de blog poderia alterar o formato JSON das postagens do blog e agora expor as postagens do blog como `/api/v1.1/posts/`, mantendo o formato JSON mais antigo para clientes que se conectam a `/api/v1.0/Postagens/`. Por um período de tempo, o servidor trata todos os URLs em suas variações `v1.1` e `v1.0`.

Embora o suporte a várias versões do servidor possa se tornar um fardo de manutenção, há situações em que essa é a única maneira de permitir que o aplicativo cresça sem causar problemas às implantações existentes.

## Web Services RESTful com Flask

O Flask facilita muito a criação de serviços web RESTful. O decorador `route()` familiar , juntamente com seu argumento opcional de métodos , pode ser usado para declarar as rotas que tratam das URLs de recursos expostas pelo serviço. Trabalhar com dados JSON também é simples, pois os dados JSON incluídos em uma solicitação são expostos automaticamente como um dicionário Python `request.json` e uma resposta que precisa conter JSON pode ser facilmente gerada a partir de um dicionário Python usando a função auxiliar `jsonify()` do Flask .

As seções a seguir mostram como o Flasky pode ser estendido com um serviço Web RESTful que oferece aos clientes acesso a postagens de blog e recursos relacionados.

### Criando um Blueprint de API As

rotas associadas a uma API RESTful formam um subconjunto independente do aplicativo, portanto, colocá-los em seu próprio blueprint é a melhor maneira de mantê-los bem organizados. A estrutura geral do blueprint da API dentro do aplicativo é mostrada no [Exemplo 14-1](#).

#### *Exemplo 14-1. Estrutura do blueprint da API*

```
|-flasky |-  
  app/ |-  
    api_1_0 |-  
      __init__.py |-  
      user.py |-  
      post.py |-  
      comment.py |-  
      authentication.py |-  
      errors.py |-decorators.py
```

Observe como o pacote usado para a API inclui um número de versão em seu nome. Quando uma versão incompatível com versões anteriores da API precisa ser introduzida, ela pode ser adicionada como outro pacote com um número de versão diferente e ambas as APIs podem ser atendidas ao mesmo tempo.

Este blueprint de API implementa cada recurso em um módulo separado. Módulos a tomar cuidados com a autenticação, tratamento de erros e fornecer decoradores personalizados também são importantes. incluído. O construtor do blueprint é mostrado no [Exemplo 14-2](#).

*Exemplo 14-2. app/api\_1\_0/\_\_init\_\_.py: construtor do blueprint da API*

**do projeto de importação de frascos**

```
api = Blueprint('api', __name__)
```

**de . autenticação de importação , postagens, usuários, comentários, erros**

O registro do blueprint da API é mostrado no [Exemplo 14-3](#).

*Exemplo 14-3. app/\_init\_.py: registro do blueprint da API*

```
def create_app(config_name):
```

```
...
```

```
# de .api_1_0 importar api como api_1_0_blueprint
app.register_blueprint(api_1_0_blueprint, url_prefix='/api/v1.0')
# ...
```

### Tratamento de erros

Um web service RESTful informa o cliente sobre o status de uma solicitação enviando o aplicativo código de status HTTP apropriado na resposta mais qualquer informação adicional no corpo de resposta. Os códigos de status típicos que um cliente pode esperar ver em um serviço da web estão listados na [Tabela 14-2](#).

*Tabela 14-2. Códigos de status de resposta HTTP normalmente retornados por APIs*

Status HTTP código	Nome	Descrição
200	OK	A solicitação foi concluída com sucesso.
201	Criado	A solicitação foi concluída com sucesso e um novo recurso foi criado como resultado.
400	Pedido ruim	A solicitação é inválida ou inconsistente.
401	Não autorizado	A solicitação não inclui informações de autenticação.
403	Proibido	As credenciais de autenticação enviadas com a solicitação são insuficientes para a solicitação.
404	Não encontrado	O recurso referenciado na URL não foi encontrado.
405	Método não permitido	O método de solicitação solicitado não é compatível com o recurso fornecido.
500	Erro interno do servidor	Ocorreu um erro inesperado ao processar a solicitação.

O manuseio dos códigos de status 404 e 500 apresenta uma pequena complicação, pois estes erros são gerados pelo Flask por conta própria e geralmente retornarão uma resposta HTML, o que provavelmente confundirá um cliente de API.

Uma maneira de gerar respostas apropriadas para todos os clientes é fazer com que os manipuladores de erros adaptem suas respostas com base no formato solicitado pelo cliente, uma técnica chamada *negociação de conteúdo*. O Exemplo 14-4 mostra um manipulador de erro 404 aprimorado que responde com JSON para clientes de serviço da Web e com HTML para outros. O manipulador de erro 500 é escrito de maneira semelhante.

*Exemplo 14-4. app/main/errors.py: manipuladores de erros com negociação de conteúdo HTTP*

```
@main.app_errorhandler(404)
def page_not_found(e):
    se request.accept_mimetypes.accept_json e \
        não request.accept_mimetypes.accept_html:
        response = jsonify({'error': 'not found'})
        response.status_code = 404 return response return
        render_template('404.html'), 404
```

Essa nova versão do manipulador de erros verifica o cabeçalho da solicitação Accept , que Werkzeug decodifica em request.accept\_mimetypes, para determinar em qual formato o cliente deseja a resposta gerada apenas para clientes que aceitam JSON e não aceitam HTML.

Os códigos de status restantes são gerados explicitamente pelo serviço da Web, para que possam ser implementados como funções auxiliares dentro do blueprint no módulo errors.py .

O Exemplo 14-5 mostra a implementação do erro 403; os outros são semelhantes.

*Exemplo 14-5. app/api/errors.py: manipulador de erros da API para código de status 403*

```
def proibido(mensagem):
    response = jsonify({'error': 'forbidden', 'message': message})
    response.status_code = 403 return response
```

Agora as funções de visualização no serviço da web podem invocar essas funções auxiliares para gerar respostas de erro.

## Autenticação do usuário com Flask-HTTPAuth

Os serviços da Web, como aplicativos da Web comuns, precisam proteger as informações e garantir que elas não sejam fornecidas a terceiros não autorizados. Por esse motivo, os RIAs devem solicitar credenciais de login a seus usuários e passá-las ao servidor para verificação.

Foi mencionado anteriormente que uma das características dos web services RESTful é que eles são stateless, o que significa que o servidor não tem permissão para “lembra” nada sobre o cliente entre as requisições. Os clientes precisam fornecer todas as informações necessárias para realizar uma solicitação na própria solicitação, portanto, todas as solicitações devem incluir as credenciais do usuário.

A funcionalidade de login atual implementada com a ajuda do Flask-Login armazena dados na sessão do usuário, que o Flask armazena por padrão em um cookie do lado do cliente, para que o servidor não armazene nenhuma informação relacionada ao usuário; ele pede ao cliente para armazená-lo em vez disso. Parece que esta implementação está em conformidade com o requisito sem estado do REST, mas o uso de cookies em serviços da Web RESTful fica em uma área cinzenta, pois pode ser complicado para clientes que não são navegadores da Web implementá-los. Por esse motivo, geralmente é visto como uma má escolha de design usá-los.



O requisito sem estado do REST pode parecer excessivamente rigoroso, mas não é arbitrário. Servidores sem estado podem ser *dimensionados* com muita facilidade. Quando os servidores armazenam informações sobre os clientes, é necessário ter um cache compartilhado acessível a todos os servidores para garantir que o mesmo servidor sempre receba requisições de um determinado cliente. Ambos são problemas complexos de resolver.

Como a arquitetura RESTful é baseada no protocolo *HTTP*, a autenticação *HTTP* é o método preferencial usado para enviar credenciais, seja em seus sabores Basic ou Digest.

Com a autenticação HTTP, as credenciais do usuário são incluídas em um cabeçalho de autorização com todas as solicitações.

O protocolo de autenticação HTTP é simples o suficiente para ser implementado diretamente, mas a extensão Flask-HTTPAuth fornece um wrapper conveniente que oculta os detalhes do protocolo em um decorador semelhante ao `login_required` do Flask-Login.

Flask-HTTPAuth é instalado com pip:

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

Para inicializar a extensão para autenticação HTTP Basic, deve-se criar um objeto da classe `HTTPBasicAuth`. Assim como o Flask-Login, o Flask-HTTPAuth não faz suposições sobre o procedimento necessário para verificar as credenciais do usuário, portanto, essa informação é fornecida em uma função de retorno de chamada. O [Exemplo 14-6](#) mostra como o ramal é inicializado e fornecido com um retorno de verificação.

*Exemplo 14-6. app/api\_1\_0/authentication.py: inicialização Flask-HTTPAuth*

```
from flask.ext.httpauth import HTTPBasicAuth auth =
HTTPBasicAuth()

@auth.verify_password
def Verify_password(email, password): if email
== "": g.current_user = AnonymousUser()
return True

user = User.query.filter_by(email = email).first() se não for user:
    retornar Falso
g.current_user = user
return user.verify_password(password)
```

Como esse tipo de autenticação de usuário será usado apenas no blueprint da API, a extensão Flask HTTPAuth é inicializada no pacote do blueprint e não no pacote do aplicativo, como outras extensões.

O email e a senha são verificados usando o suporte existente no modelo User . O retorno de chamada de verificação retorna True quando o login é válido ou False caso contrário. Logins anônimos são suportados, para os quais o cliente deve enviar um campo de e-mail em branco.

O retorno de chamada de autenticação salva o usuário autenticado no objeto global g do Flask para que a função de visualização possa acessá-lo posteriormente. Observe que quando um login anônimo é recebido, a função retorna True e salva uma instância da classe AnonymousUser usada com Flask-Login em g.current\_user.



Como as credenciais do usuário estão sendo trocadas a cada solicitação, é extremamente importante que as rotas da API sejam expostas em HTTP seguro para que todas as solicitações e respostas sejam criptografadas.

Quando as credenciais de autenticação são inválidas, o servidor retorna um erro 401 ao cliente. Flask-HTTPAuth gera uma resposta com esse código de status por padrão, mas para garantir que a resposta seja consistente com outros erros retornados pela API, a resposta de erro pode ser personalizada conforme mostrado no [Exemplo 14-7](#).

*Exemplo 14-7. \_app/api\_1\_0/authentication.py: manipulador de erros Flask-HTTPAuth*

```
@auth.error_handler
def auth_error(): return
    não autorizado(' Credenciais inválidas ')
```

Para proteger uma rota, o decorador auth.login\_required é usado:

```
@api.route('/posts/')
@auth.login_required def
get_posts():
    passar
```

Mas como todas as rotas no blueprint precisam ser protegidas da mesma maneira, o decorador login\_required pode ser incluído uma vez em um manipulador before\_request para o blueprint, conforme mostrado no [Exemplo 14-8](#).

*Exemplo 14-8. app/api\_1\_0/authentication.py: manipulador before\_request com autenticação*

```
de .errors import_proibido_erro_

@api.before_request
@auth.login_required
def before_request():
    pass
```

```
se não for g.current_user.is_anonymous e \ não
    g.current_user.confirmed: return
        proibido('Conta não confirmada')
```

Agora as verificações de autenticação serão feitas automaticamente para todas as rotas no blueprint. Como uma verificação adicional, o manipulador before\_request também rejeita usuários autenticados que não confirmaram suas contas.

## Autenticação baseada em token

Os clientes devem enviar credenciais de autenticação com cada solicitação. Para evitar a transferência constante de informações confidenciais, uma solução de autenticação baseada em token pode ser oferecida.

Na autenticação baseada em token, o cliente envia as credenciais de login para uma URL especial que gera tokens de autenticação. Depois que o cliente tiver um token, ele poderá usá-lo no lugar das credenciais de login para autenticar solicitações. Por motivos de segurança, os tokens são emitidos com uma expiração associada. Quando um token expira, o cliente deve se autenticar novamente para obter um novo. O risco de um token cair nas mãos erradas é limitado devido à sua curta vida útil. O [Exemplo 14-9](#) mostra os dois novos métodos adicionados ao modelo User que suportam geração e verificação de tokens de autenticação usando seu perigoso.

*Exemplo 14-9. app/models.py: suporte à autenticação baseada em token*

```
class User(db.Model):
    ...
    def generate_auth_token(self, expiração):
        s = Serializer(current_app.config['SECRET_KEY'],
                      expires_in=expiração)
        return s.dumps({'id': self.id})

    @staticmethod
    def Verify_auth_token(token):
        s = Serializer(current_app.config['SECRET_KEY'])
        try:
            data = s.loads(token)
        except:
            return Nenhum

        return User.query.get(data['id'])
```

O método generate\_auth\_token() retorna um token assinado que codifica o campo id do usuário. Um tempo de expiração dado em segundos também é usado. O método verify\_auth\_token() recebe um token e, se for válido, retorna o usuário armazenado nele. Este é um método estático, pois o usuário só será conhecido depois que o token for decodificado.

Para autenticar solicitações que vêm com um token, o retorno de chamada Verify\_password para Flask-HTTPAuth deve ser modificado para aceitar tokens e credenciais regulares. O retorno de chamada atualizado é mostrado no [Exemplo 14-10](#).

*Exemplo 14-10. app/api\_1\_0/authentication.py: verificação de autenticação aprimorada com suporte a token*

```
@auth.verify_password
def Verify_password(email_or_token, senha):
    if email_or_token == "":
        g.current_user = AnonymousUser()
        return True
    if password == "":
        g.current_user = User.verify_auth_token(email_or_token) g.token_used =
        = True return g.current_user não é Nenhum user =
        User.query.filter_by(email=email_or_token).first() se não for usuário:

    retornar Falso
    g.current_user = usuário
    g.token_used = Falso
    return user.verify_password(password)
```

Nesta nova versão, o primeiro argumento de autenticação pode ser o endereço de e-mail ou um token de autenticação. Se este campo estiver em branco, um usuário anônimo será assumido, como antes. Se a senha estiver em branco, o campo email\_or\_token será considerado um token e validado como tal. Se ambos os campos não estiverem vazios, a autenticação regular de e-mail e senha é assumida. Com essa implementação, a autenticação baseada em token é opcional; cabe a cada cliente usá-lo ou não. Para dar às funções de visualização a capacidade de distinguir entre os dois métodos de autenticação, uma variável g.token\_used é adicionada.

A rota que retorna os tokens de autenticação para o cliente também é adicionada ao blueprint da API. A implementação é mostrada no [Exemplo 14-11](#).

*Exemplo 14-11. app/api\_1\_0/authentication.py: geração de token de autenticação*

```
@api.route('/token') def
get_token(): if
    g.current_user.is_anonymous() ou g.token_used: return não
        autorizado(' Credenciais inválidas') return jsonify({'token':
    g.current_user.generate_auth_token( expiração=3600), 'expiração': 3600})
```

Como essa rota está no blueprint, os mecanismos de autenticação adicionados ao manipulador before\_request também se aplicam a ela. Para evitar que os clientes usem um token antigo para solicitar um novo, a variável g.token\_used é verificada e, dessa forma, as solicitações autenticadas com um token podem ser rejeitadas. A função retorna um token na resposta JSON com um período de validade de uma hora. O período também está incluído no JSON resposta.

## Serializando recursos de e para JSON Uma

necessidade frequente ao escrever um serviço da Web é converter a representação interna de recursos de e para JSON, que é o formato de transporte usado em solicitações e respostas HTTP. O Exemplo 14-12 mostra um novo método `to_json()` adicionado à classe `Post`.

*Exemplo 14-12. app/models.py: converte uma postagem em um dicionário serializável JSON*

```
class Post(db.Model):
    ...
    def to_json(self):
        json_post = {
            'url': url_for('api.get_post', id=self.id, _external=True), 'body': self.body,
            'body_html': self.body_html, 'timestamp': self.timestamp, 'autor':
            url_for('api.get_user', id=self.author_id, _external=True),
            ...
            'comments': url_for('api.get_post_comments', id=self.id, _external=True)
            'comment_count': self.comments.count()
        }
        return json_post
```

Os campos `url`, `author` e `comments` precisam retornar as URLs para os respectivos recursos, para que sejam gerados com chamadas `url_for()` para rotas que serão definidas no blueprint da API. Observe que `_external=True` é adicionado a todas as chamadas `url_for()` para que as URLs totalmente qualificadas sejam retornadas em vez das URLs relativas que normalmente são usadas no contexto de um aplicativo da Web tradicional.

Este exemplo também mostra como é possível retornar atributos “inventados” na representação de um recurso. O campo `comment_count` retorna o número de comentários que existem para a postagem do blog. Embora este não seja um atributo real do modelo, ele é incluído na representação do recurso para conveniência do cliente.

O método `to_json()` para modelos de usuário pode ser construído de maneira semelhante ao `Post`. Este método é mostrado no Exemplo 14-13.

*Exemplo 14-13. app/models.py: converte um usuário em um dicionário serializável JSON*

```
class User(UserMixin, db.Model):
    ...
    def to_json(self):
        json_user = {
            'url': url_for('api.get_post', id=self.id, _external=True), 'username':
            self.username, 'member_since': self.member_since, 'last_seen':
            self.last_seen, 'posts': url_for('api.get_user_posts', id=self.id,
            _external=True), 'followed_posts': url_for('api.get_user_followed_posts',
            ...
            id=self.id, _external=True),
        }
        return json_user
```

```

    'post_count': self.posts.count()

} return json_user

```

Observe como neste método alguns dos atributos do usuário, como email e função, são omitidos da resposta por motivos de privacidade. Este exemplo demonstra novamente que a representação de um recurso oferecido aos clientes não precisa ser idêntica à representação interna do modelo de banco de dados correspondente.

Converter uma estrutura JSON em um modelo apresenta o desafio de que alguns dos dados provenientes do cliente podem ser inválidos, errados ou desnecessários. O [Exemplo 14-14](#) mostra o método que cria um modelo Post de JSON.

*Exemplo 14-14. app/models.py: crie uma postagem no blog a partir de JSON*

`de app.exceptions import ValidationError`

```

class Post(db.Model): #
    @staticmethod def
        from_json(json_post):
            body = json_post.get('body') if
                body is None ou body == "": raise
                ValidationError('post not have a
                    body') return Post(corpo=corpo)

```

Como você pode ver, essa implementação opta por usar apenas o atributo body do dicionário JSON. O atributo body\_html é ignorado, pois a renderização Markdown do lado do servidor é automaticamente acionada por um evento SQLAlchemy sempre que o atributo body é modificado. O atributo timestamp não precisa ser fornecido, a menos que o cliente tenha permissão para retroceder postagens, o que não é um recurso deste aplicativo. O campo autor não é usado porque o cliente não tem autoridade para selecionar o autor de uma postagem de blog; o único valor possível para o campo de autor é o do usuário autenticado. Os atributos comments e comment\_count são gerados automaticamente a partir de um relacionamento de banco de dados, portanto, não há informações úteis neles que sejam necessárias para criar um modelo. Finalmente, o campo url é ignorado porque nesta implementação as URLs dos recursos são definidas pelo servidor, não pelo cliente.

Observe como a verificação de erros é feita. Se o campo do corpo estiver ausente ou vazio, uma exceção ValidationError será gerada. Lançar uma exceção é neste caso a maneira apropriada de lidar com o erro porque este método não tem conhecimento suficiente para lidar adequadamente com a condição. A exceção passa efetivamente o erro para o chamador, permitindo que o código de nível superior faça o tratamento do erro. A classe ValidationError é implementada como uma subclasse simples de ValueError do Python. Esta implementação é mostrada no [Exemplo 14-15](#).

*Exemplo 14-15. app/exceptions.py: exceção ValidationError*

```
class ValidationError(ValueError):
    pass
```

O aplicativo agora precisa lidar com essa exceção fornecendo a resposta apropriada ao cliente. Para evitar ter que adicionar código de captura de exceção nas funções de exibição, um manipulador de exceção global pode ser instalado. Um manipulador para a exceção ValidationError é mostrado no [Exemplo 14-16](#).

*Exemplo 14-16. app/api\_1\_0/errors.py: manipulador de erros da API para ValidationError, exceto exceções*

```
@api.errorhandler(ValidationError) def
validation_error(e): return
    bad_request(e.args[0])
```

O decorador errorhandler é o mesmo que é usado para registrar manipuladores para códigos de status HTTP, mas neste uso ele recebe uma classe Exception como argumento. A função decorada será invocada sempre que uma exceção de determinada classe for levantada. Observe que o decorador é obtido do blueprint da API, portanto, esse manipulador será invocado somente quando a exceção for levantada enquanto uma rota do blueprint estiver sendo processada.

Usando esta técnica, o código nas funções de visualização pode ser escrito de forma muito limpa e concisa, sem a necessidade de incluir verificação de erros. Por exemplo:

```
@api.route('/posts/', métodos=['POST']) def
new_post(): post = Post.from_json(request.json)
    post.author = g.current_user
    db.session.add(post) db.session.commit()
    return jsonify(post.to_json())
```

### Implementação de endpoints de recursos

O que resta é implementar as rotas que tratam dos diferentes recursos. As solicitações GET geralmente são as mais fáceis porque apenas retornam informações e não precisam fazer alterações. O [Exemplo 14-17](#) mostra os dois manipuladores GET para postagens de blog.

*Exemplo 14-17. app/api\_1\_0/posts.py: GET manipuladores de recursos para postagens*

```
@api.route('/posts/')
@auth.login_required def
get_posts(): posts =
    Post.query.all() return
    jsonify({ 'posts': [post.to_json() for post in posts] })

@api.route('/posts/<int:id>')
@auth.login_required
```

```
def get_post(id):
    post = Post.query.get_or_404(id) return
        jsonify(post.to_json())
```

A primeira rota trata da solicitação da coleção de postagens. Esta função usa uma compreensão de lista para gerar a versão JSON de todas as postagens. A segunda rota retorna uma única postagem no blog e responde com um erro de código 404 quando o id fornecido não é encontrado no banco de dados.



O manipulador para erros 404 está no nível do aplicativo, mas fornecerá uma resposta JSON se o cliente solicitar esse formato. Se uma resposta customizada para o serviço da web for desejada, o manipulador de erro 404 poderá ser substituído no blueprint.

O manipulador POST para recursos de postagem de blog insere uma nova postagem de blog no banco de dados. Esta rota é mostrada no [Exemplo 14-18](#).

*Exemplo 14-18. app/api\_1\_0/posts.py: manipulador de recursos POST para postagens*

```
@api.route('/posts/', métodos=['POST'])
@permission_required(Permission.WRITE_ARTICLES)
def new_post(): post = Post.from_json(request.json)
    post.author = g.current_user db. session.add(post)
    db.session.commit() return jsonify(post.to_json()),
        201, \
            {'Local': url_for('api.get_post', id=post.id, _external=True)}
```

Essa função de visualização é encapsulada em um decorador permission\_required (mostrado em um próximo exemplo) que garante que o usuário autenticado tenha permissão para escrever postagens no blog. A criação real da postagem do blog é direta devido ao suporte de tratamento de erros que foi implementado anteriormente. Uma postagem de blog é criada a partir dos dados JSON e seu autor é explicitamente atribuído como o usuário autenticado. Depois que o modelo é gravado no banco de dados, um código de status 201 é retornado e um cabeçalho Location é adicionado com a URL do recurso recém-criado.

Observe que, para conveniência do cliente, o corpo da resposta inclui o novo recurso. Isso evitaria que o cliente precise emitir um recurso GET para ele imediatamente após a criação do recurso.

O decorador permission\_required usado para impedir que usuários não autorizados criem novas postagens no blog é semelhante ao usado no aplicativo, mas é personalizado para o blueprint da API. A implementação é mostrada no [Exemplo 14-19](#).

*Exemplo 14-19. app/api\_1\_0/decorators.py: permission\_required decorador*

```
def permission_required(permission):
    def decorator(f): @wraps(f) def
        decorated_function(*args, **kwargs):

            se não g.current_user.can(permission): return
                proibido('Permissões insuficientes')
            return f(*args, **kwargs)
        retornar decorated_função
    retornar decorator
```

O manipulador PUT para postagens de blog, usado para editar recursos existentes, é mostrado no [Exemplo 14-20](#).

*Exemplo 14-20. app/api\_1\_0/posts.py: PUT gerenciador de recursos para postagens*

```
@api.route('/posts/<int:id>', métodos=['PUT'])
@permission_required(Permission.WRITE_ARTICLES)
def edit_post(id): post = Post.query.get_or_404(id) if g.
    current_user != post.author e \ não
        g.current_user.can(Permission.ADMINISTER):

            return proibido('Permissões insuficientes')
        post.body = request.json.get('body', post.body)
        db.session.add(post) return jsonify(post.to_json())
```

As verificações de permissão são mais complexas neste caso. A verificação padrão de permissão para escrever postagens de blog é feita com o decorador, mas para permitir que um usuário edite uma postagem de blog, a função também deve garantir que o usuário seja o autor da postagem ou um administrador. Essa verificação é adicionada explicitamente à função de visualização. Se essa verificação tivesse que ser adicionada em muitas funções de visualização, construir um decorador para ela seria uma boa maneira de evitar a repetição de código.

Como o aplicativo não permite a exclusão de postagens, o manipulador para o método de solicitação DELETE não precisa ser implementado.

Os manipuladores de recursos para usuários e comentários são implementados de maneira semelhante. A [Tabela 14-3](#) lista o conjunto de recursos implementados para este aplicativo. A implementação completa está disponível para você estudar no [repositório do GitHub](#).

*Tabela 14-3. Recursos da API Flasky*

URL do recurso	Métodos	Descrição
/users/<int:id>	OBTER	Um usuário
/users/<int:id>/posts/ GET		As postagens do blog escritas por um usuário
/users/<int:id>/timeline/ GET		As postagens do blog seguidas por um usuário
/Postagens/	GET, POST	Todas as postagens do blog

URL do recurso	Métodos	Descrição
/posts/<int:id>	GET, PUT	Uma postagem no blog
/posts/<int:id>/comments/	GET, POST	Os comentários em uma postagem de blog
/comentários/	OBTER	Todos os comentários
/comments/<int:id>	OBTER	Um comentário

Observe que os recursos que foram implementados permitem que um cliente ofereça um subconjunto da funcionalidade que está disponível por meio do aplicativo da web. A lista de recursos suportados pode ser expandida se necessário, como expor seguidores, habilitar a moderação de comentários e implementar quaisquer outros recursos que um cliente possa precisar.

## Paginação de grandes coleções de recursos

As solicitações GET que retornam uma coleção de recursos podem ser extremamente caras e difíceis de gerenciar para coleções muito grandes. Assim como os aplicativos da Web, os serviços da Web podem optar por paginar as coleções.

O Exemplo 14-21 mostra uma possível implementação de paginação para a lista de postagens do blog.

*Exemplo 14-21. app/api\_1\_0/posts.py: paginação da postagem*

```
@api.route('/posts/')
def get_posts():
    page = request.args.get('page', 1, type=int)
    pagination = Post.query.paginate(page,
                                      per_page=current_app.config['FLASKY_POSTS_PER_PAGE'],
                                      error_out=False)
    posts = pagination.items
    prev = None
    if pagination.has_prev:
        prev = url_for('api.get_posts', page=page-1, _external=True)

    next = None
    if pagination.has_next:
        next = url_for('api.get_posts', page=page+1, _external=True)
    return jsonify({
        'posts': [post.to_json() for post in posts],
        'prev': prev,
        'next': next,
        'count': pagination.total
    })
```

O campo de postagens na resposta JSON contém os itens de dados como antes, mas agora é apenas uma parte do conjunto completo. Os itens anterior e seguinte contêm os URLs de recursos para as páginas anterior e seguinte, quando disponíveis. O valor de contagem é o número total de itens na coleção.

Essa técnica pode ser aplicada a todas as rotas que retornam coleções.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar git checkout 14a para verificar esta versão do aplicativo. Para garantir que você tenha todas as dependências instaladas, execute também pip install -r requirements/dev.txt.

### Testando Serviços da Web com HTTPie

Para testar um serviço da Web, um cliente HTTP deve ser usado. Os dois clientes mais usados para testar serviços da Web a partir da linha de comando são *curl* e *HTTPie*. Este último tem uma linha de comando muito mais concisa e legível. *HTTPie* é instalado com pip:

```
(venv) $ pip install httpie
```

Uma solicitação GET pode ser emitida da seguinte forma:

```
(venv) $ http --json --auth <email>:<password> GET \> http://127.0.0.1:5000/api/v1.0/posts
HTTP/1.0 200 OK Content-Length: 7018
Content-Type: application/json Data: Dom, 22 de dezembro de 2013 08:11:24 GMT
Servidor: Werkzeug/0.9.4 Python/2.7.3
```

```
{
    "postagens": [
        ...
    ],
    "prev": null
    "next": "http://127.0.0.1:5000/api/v1.0/posts/?page=2", "count": 150
}
```

Observe os links de paginação incluídos na resposta. Como esta é a primeira página, uma página anterior não é definida, mas uma URL para obter a próxima página e uma contagem total foram retornadas.

A mesma solicitação pode ser emitida por um usuário anônimo enviando um e-mail e senha vazios:

```
(venv) $ http --json --auth : GET http://127.0.0.1:5000/api/v1.0/posts/
```

O comando a seguir envia uma solicitação POST para adicionar uma nova postagem no blog:

```
(venv) $ http --auth <email>:<password> --json POST \> http://127.0.0.1:5000/api/v1.0/posts/ \> "body=Estou adicionando uma postagem da *linha de comando*."
HTTP/1.0 201 CREATED
Content-Length: 360
Content-Type: application/json Data:
Dom, 22 de dezembro de 2013 08:30:27
GMT Local: http://127.0.0.1:5000/api/v1.0/posts/ Servidor 111:
Werkzeug/0.9.4 Python/2.7.3
```

```
{
    "author": "http://127.0.0.1:5000/api/v1.0/users/1", "body": "Estou
    adicionando uma postagem da *linha de comando*. ", "body_html":
    "<p>Estou adicionando uma postagem da <em>linha de comando</em>.</p>", "comments":
    "http://127.0.0.1:5000/api/v1.0/posts /111/comments", "comment_count": 0, "timestamp":
    "Dom, 22 de dezembro de 2013 08:30:27 GMT", "url": "http://127.0.0.1:5000/api/v1.0 /posts/
    111"
}
```

Para usar tokens de autenticação, uma solicitação para `/api/v1.0/token` é enviada:

```
(venv) $ http --auth <email>:<password> --json GET \> http://
127.0.0.1:5000/api/v1.0/token HTTP/1.0 200 OK Content-Length:
162 Content-Type: application/json Data: sáb, 04 de janeiro de
2014 08:38:47 GMT Servidor: Werkzeug/0.9.4 Python/3.3.3
```

```
{
    "expiration": 3600,
    "token": "eyJpYXQiOjEzODg4MjQ3MjcsImV4cCI6MTM4ODgyODMyNywiYWxnIjoiSFMy...
}
```

E agora o token retornado pode ser usado para fazer chamadas para a API pela próxima hora, passando-o junto com um campo de senha vazio:

```
(venv) $ http --json --auth eyJpYXQ...: GET http://127.0.0.1:5000/api/v1.0/posts/
```

Quando o token expirar, as solicitações serão retornadas com um erro de código 401, indicando que um novo token precisa ser obtido.

Parabéns! Este capítulo conclui a Parte II e, com isso, a fase de desenvolvimento de recursos do Flasky está concluída. O próximo passo é obviamente implantá-lo, e isso traz um novo conjunto de desafios que são objeto da Parte III.



**PARTE III**

---

**A última milha**



---

## CAPÍTULO 15

# Teste

Há duas razões muito boas para escrever testes de unidade. Ao implementar uma nova funcionalidade, testes de unidade são usados para confirmar que o novo código está funcionando da maneira esperada. O mesmo resultado pode ser obtido testando manualmente, mas é claro que os testes automatizados economizam tempo e esforço.

Uma segunda razão mais importante é que cada vez que o aplicativo é modificado, todos os testes de unidade construídos em torno dele podem ser executados para garantir que não haja *regressões* no código existente; em outras palavras, que as novas mudanças não afetaram a forma como o código antigo funciona.

Os testes unitários fazem parte do Flasky desde o início, com testes projetados para exercitar recursos específicos do aplicativo implementados nas classes do modelo de banco de dados.

Essas classes são fáceis de testar fora do contexto de um aplicativo em execução, portanto, considerando que exige pouco esforço, implementar testes de unidade para todos os recursos implementados nos modelos de banco de dados é a melhor maneira de garantir que pelo menos parte do aplicativo inicie robusto e fica assim.

Este capítulo discute maneiras de melhorar e estender o teste de unidade.

### Obtenção de relatórios de cobertura de código

Ter uma suíte de testes é importante, mas é igualmente importante saber se ela é boa ou ruim. As ferramentas de cobertura de código medem quanto do aplicativo é exercido por testes de unidade e podem fornecer um relatório detalhado que indica quais partes do código do aplicativo não estão sendo testadas. Essas informações são inestimáveis, pois podem ser usadas para direcionar o esforço de escrever novos testes para as áreas que mais precisam.

Python tem uma excelente ferramenta de cobertura de código apropriadamente chamada de *cobertura*. Você pode instalá-lo com pip:

```
(venv) cobertura de instalação de $ pip
```

Essa ferramenta vem como um script de linha de comando que pode iniciar qualquer aplicativo Python com cobertura de código habilitada, mas também fornece acesso de script mais conveniente para iniciar o mecanismo de cobertura programaticamente. Para ter métricas de cobertura bem integradas ao script de inicialização *manage.py* , o comando de teste personalizado adicionado no **Capítulo 7** pode ser expandido com um argumento opcional `--coverage` . A implementação desta opção é mostrada no [Exemplo 15-1](#).

*Exemplo 15-1. manage.py: métricas de cobertura*

```
#!/usr/bin/env python
import os COV = Nenhum
if
os.environ.get('FLASK_COVERAGE'):
    importar cobertura
    COV = cobertura.cobertura(branch=True, include='app/*')
    COV.início()

# ...

@manager.command
def test(coverage=False):
    """Execute os testes de
    unidade."""" se cobertura e não os.environ.get('FLASK_COVERAGE'):
        import sys os.environ['FLASK_COVERAGE'] = '1'
        os.execvp(sys.executable, [sys.executable] + sys.argv)
        import unittest tests = unittest.TestLoader().discover('tests')
        unittest.TextTestRunner(verbosity=2).run(tests) if COV:

            COV.stop()
            COV.save()
            print('Resumo da Cobertura:')
            COV.report ()
            basedir = os.path.abspath (os.path.dirname (__file__)) covdir =
            os.path.join (basedir, 'tmp / cobertura')
            COV.html_report(directory=covdir)
            print('HTML versão: file:///%s/index.html' % covdir)
            COV.apagar ()

# ...
```

O Flask-Script facilita muito a definição de comandos personalizados. Para adicionar uma opção booleana ao comando `test` , adicione um argumento booleano à função `test()` . O Flask-Script deriva o nome da opção do nome do argumento e passa True ou False para a função de acordo.

Mas integrar a cobertura de código com o script *manage.py* apresenta um pequeno problema. Quando a opção `--coverage` é recebida na função `test()` , já é tarde demais para ativar as métricas de cobertura; a essa altura, todo o código no escopo global já

executado. Para obter métricas precisas, o script se reinicia após definir o Variável de ambiente FLASK\_COVERAGE . Na segunda execução, a parte superior do script encontra que a variável de ambiente está definida e ativa a cobertura desde o início.

A função coverage.coverage() inicia o mecanismo de cobertura. O branch=True opção permite a análise de cobertura de filiais, que, além de rastrear quais linhas de código execute, verifica se para cada condicional ambos os casos True e False têm excedido. A opção include é usada para limitar a análise de cobertura aos arquivos que estão dentro o pacote de aplicativos, que é o único código que precisa ser medido. Sem o **opção include**, todas as extensões instaladas no ambiente virtual e o código para os testes em si seriam incluídos nos relatórios de cobertura - e isso acrescentaria muito ruído ao relatório.

Após a execução de todos os testes, a função text() grava um relatório no console e também grava um relatório HTML melhor no disco. A versão HTML é muito boa para exibir cobertura visualmente porque mostra as linhas de código-fonte codificadas por cores de acordo com sua usar.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar git checkout 15a para verificar esta versão do aplicativo.

Para garantir que você tenha todas as dependências instaladas, execute também pip instale -r requirements/dev.txt.

Segue um exemplo do relatório baseado em texto:

```
(venv) $ python manage.py test --coverage
...
-----
Realizou 19 testes em 50.609s

OK

Resumo da cobertura:
Nome           Stmts Miss Branch BrMiss Cover ausente
...
-----
app/__init__.py          33      0      0      0 100%
api_1_0/__init__.py      3       0      0      0 100%
authentication/app/api_1_0/comments 30      19     11      11    27%
app/api_1_0/decorators app/api_1_0/ 40      30     12      12    19%
errors/app/api_1_0/posts app/api_1_0/ 11      3      2      2    62%
users/app/auth/__init__.py app/auth/forms 17      10     0      0    41%
app/auth/views app/decorators      35      23     9      9    27%
                                         30      24     12     12    14%
                                         3       0      0      0 100%
                                         45      8      8      8    70%
                                         109     84     41     41    17%
                                         14      3      2      2    69%
```

app/email	15	9	0	0	40%
app/exceptions app/	2	0	0	0	100%
main/__init__ app/main/	6	1	0	0	83%
errors app/main/forms	20	15	9	9	17%
app/main/views app/	39	7	8	8	68%
models	169	131	36	36	19%
	243	62	44	17	72%
<hr/>					
TOTAL	864	194	versão HTML: file:///home/flask/flasky/tmp/109	average/	167
index.html					44%

O relatório mostra uma cobertura geral de 44%, o que não é terrível, mas não é muito bom ou. As classes de modelo, que receberam toda a atenção dos testes de unidade até agora, contêm um total de 243 afirmações, das quais 72% são contempladas em provas. Obviamente o `views.py` arquivos nos blueprints principal e de autenticação e as rotas no blueprint `api_1_0` têm cobertura muito baixa, uma vez que estes não são exercidos em nenhum dos testes unitários existentes.

Armado com este relatório, é fácil determinar quais testes precisam ser adicionados ao teste suite para melhorar a cobertura, mas infelizmente nem todas as partes do aplicativo podem ser testadas tão facilmente quanto os modelos de banco de dados. As próximas duas seções discutem testes mais avançados estratégias que podem ser aplicadas para visualizar funções, formulários e modelos.

Observe que o conteúdo da coluna *Missing* foi omitido no relatório de exemplo para melhorar a formatação. Esta coluna indica as linhas de código-fonte que foram perdidas pelos testes como uma longa lista de intervalos de números de linha.

## O cliente de teste do frasco

Algumas partes do código do aplicativo dependem muito do ambiente criado por um aplicativo em execução. Por exemplo, você não pode simplesmente invocar o código em uma função de exibição para testá-lo, pois a função pode precisar acessar globais de contexto do Flask, como solicitação ou sessão, ele pode estar esperando dados de formulário em uma solicitação POST e algumas funções de visualização também pode exigir um usuário logado. Em resumo, as funções de visualização podem ser executadas apenas dentro do contexto de uma solicitação e um aplicativo em execução.

O Flask vem equipado com um *cliente de teste* para tentar resolver esse problema, pelo menos parcialmente. O cliente de teste replica o ambiente que existe quando um aplicativo está em execução dentro de um servidor web, permitindo que testes atuem como clientes e enviem requisições.

As funções de visualização não apresentam grandes diferenças quando executadas no cliente de teste; solicitações são recebidas e roteadas para as funções de visualização apropriadas, das quais são geradas e devolvidas. Depois que uma função de visualização é executada, sua resposta é passada para o teste, que pode verificar se está correto.

### Testando aplicativos da Web O Exemplo

[15-2](#) mostra uma estrutura de teste de unidade que usa o cliente de teste.

*Exemplo 15-2. tests/test\_client.py: Framework para testes usando o cliente de teste Flask*

```
import unittest
from app import create_app, db
from app.models import User, Role

class FlaskClientTestCase(unittest.TestCase):
    def setUp(self):
        self.app = create_app('testing')
        self.app_context = self.app.app_context()
        self.app_context.push()
        db.create_all()

        Role.insert_roles()
        self.client = self.app.test_client(use_cookies=True)

    def tearDown(self):
        db.session.remove()
        db.drop_all()
        self.app_context.pop()

    def test_home_page(self):
        resposta = self.client.get(url_for('main.index'))
        self.assertTrue('Stranger' in response.get_data(as_text=True))
```

A variável de instância self.client adicionada ao caso de teste é o objeto cliente de teste do Flask. Este objeto expõe métodos que emitem solicitações para o aplicativo. Quando o cliente de teste é criado com a opção use\_cookies habilitada, ele aceitará e enviará cookies da mesma forma que os navegadores, portanto, a funcionalidade que depende de cookies para recuperar o contexto entre as solicitações pode ser usada. Em particular, essa abordagem permite o uso de sessões de usuário, portanto, é necessário efetuar login e logout dos usuários.

O teste test\_home\_page() é um exemplo simples do que o cliente de teste pode fazer. Neste exemplo, uma solicitação para a página inicial é emitida. O valor de retorno do método get() do cliente de teste é um objeto Flask Response , contendo a resposta retornada pela função de visualização invocada. Para verificar se o teste foi bem-sucedido, no corpo da resposta, obtida de response.get\_data(), é pesquisada a palavra "Stranger", que faz parte do campo "Hello, Stranger!" saudação mostrada a usuários anônimos. Observe que get\_data() retorna o corpo da resposta como uma matriz de bytes por padrão; passar as\_text=True retorna uma string Unicode que é muito mais fácil de trabalhar.

O cliente de teste também pode enviar solicitações POST que incluem dados de formulário usando o método post() , mas o envio de formulários apresenta uma pequena complicaçāo. Os formulários gerados pelo Flask-WTF possuem um campo oculto com um token CSRF que precisa ser enviado junto com o formulário. Para replicar essa funcionalidade, um teste precisaria solicitar a página que inclui o formulário, analisar o texto HTML retornado na resposta e extrair o token para que ele possa enviar o token com os dados do formulário. Para evitar o incômodo de lidar

com tokens CSRF em testes, é melhor desabilitar a proteção CSRF na configuração de teste. Isto é como mostrado no [Exemplo 15-3](#).

*Exemplo 15-3. config.py: Desabilite a proteção CSRF na configuração de teste*

```
class TestingConfig(Config):
```

```
    #...
```

```
    WTF_CSRF_ENABLED = Falso
```

O [Exemplo 15-4](#) mostra um teste de unidade mais avançado que simula um novo usuário registrando uma conta, efetuando login, confirmando a conta com um token de confirmação e, finalmente, efetuando logout.

*Exemplo 15-4. tests/test\_client.py: Simulação de um novo fluxo de trabalho do usuário com o cliente de teste Flask*

```
class FlaskClientTestCase(unittest.TestCase):
```

```
    ...
```

```
    # def test_register_and_login(self): #
```

```
        registra uma nova conta response =
```

```
        self.client.post(url_for('auth.register'), data={
```

- 'email': 'john@example.com',
- 'username': 'john', 'password': 'cat',
- 'password2': 'cat'

```
    })
```

```
    self.assertTrue(response.status_code == 302)
```

```
    # login com a nova resposta da
```

```
    conta = self.client.post(url_for('auth.login'), data={
```

- 'email': 'john@example.com',
- 'password': 'cat' },

```
    follow_redirects=True) data =
```

```
    response.get_data(as_text=True)
```

```
    self.assertTrue(re.search('Hello,(s+john !', data))
```

```
    self.assertTrue('Você ainda não confirmou sua conta' em data)
```

```
    # envia um token de confirmação
```

```
    user = User.query.filter_by(email='john@example.com').first() token =
```

```
    user.generate_confirmation_token() response = self.client.get(url_for('auth.confirm',
```

```
    token=token), follow_redirects=True) data = response.get_data(as_text=True)
```

```
    self.assertTrue('Você confirmou sua conta' em
```

```
    data)
```

```
    # sair
```

```
    resposta = self.client.get(url_for('auth.logout'),
```

```
        follow_redirects=True) data
```

```
    = response.get_data(as_text=True) self.assertTrue('Você foi
```

```
    desconectado' nos dados)
```

A prova começa com o envio de um formulário para a rota de inscrição. O argumento `data` para `post()` é um dicionário com os campos do formulário, que devem corresponder exatamente aos nomes dos campos definidos no formulário. Como a proteção CSRF agora está desabilitada na configuração de teste, não há necessidade de enviar o token CSRF com o formulário.

A rota `/auth/register` pode responder de duas maneiras. Se os dados de registro forem válidos, um redirecionamento envia o usuário para a página de login. No caso de um registro inválido, a resposta torna a página com o formulário de registro novamente, incluindo as mensagens de erro apropriadas. Para validar que o registro foi aceito, o teste verifica se o código de status da resposta é 302, que é o código de um redirecionamento.

A segunda seção do teste emite um login no aplicativo usando o e-mail e a senha recém cadastrados. Isso é feito com uma solicitação POST para a rota `/auth/login`. Desta vez, um argumento `follow_redirects=True` é incluído na chamada `post()` para fazer o cliente de teste funcionar como um navegador e emitir automaticamente uma solicitação GET para a URL redirecionada. Com esta opção, o código de status 302 não será retornado; em vez disso, a resposta do URL redirecionado é retornada.

Uma resposta bem-sucedida ao envio de login agora teria uma página que cumprimenta o usuário pelo nome de usuário e indica que a conta precisa ser confirmada para obter acesso. Duas declarações `assert` verificam que esta é a página, e aqui é interessante notar que uma busca pela string 'Hello, john!' não funcionaria porque esta string é montada a partir de porções estáticas e dinâmicas, com espaço em branco extra entre as partes.

Para evitar um erro neste teste devido ao espaço em branco, uma expressão regular mais flexível é usada.

O próximo passo é confirmar a conta, que apresenta outro pequeno obstáculo. O URL de confirmação é enviado ao usuário por e-mail durante o registro, portanto, não há uma maneira fácil de acessá-lo a partir do teste. A solução apresentada no exemplo ignora o token que foi gerado como parte do registro e gera outro diretamente da instância `User`. Outra possibilidade seria extrair o token analisando o corpo do email, que o `Flask-Mail` salva ao executar em uma configuração de teste.

Com o token em mãos, a terceira parte do teste é simular o usuário clicando na URL do token de confirmação. Isso é feito enviando uma solicitação GET para a URL de confirmação com o token anexado. A resposta a esta solicitação é um redirecionamento para a página inicial, mas mais uma vez `follow_redirects=True` foi especificado, então o cliente de teste solicita a página redirecionada automaticamente. A resposta é verificada para a saudação e uma mensagem piscando que informa ao usuário que a confirmação foi bem-sucedida.

A etapa final neste teste é enviar uma solicitação GET para a rota de logout; para confirmar que isso funcionou, o teste procura uma mensagem piscando na resposta.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar git checkout 15b para verificar esta versão do aplicativo.

## Testando Web Services O

cliente de teste Flask também pode ser usado para testar Web Services RESTful. O Exemplo 15-5 mostra uma classe de teste de unidade de exemplo com dois testes.

*Exemplo 15-5. tests/test\_api.py: teste de API RESTful com o cliente de teste Flask*

```
class APITestCase(unittest.TestCase):
    # ...
    def get_api_headers(self, username, password):
        { 'Authorization': 'Basic ' + b64encode( (username +
            ':' + password).encode('utf-8')).decode('utf-8') ,
            'Aceitar': 'application/json', 'Content-Type':
            'application/json' }

    }
    def test_no_auth(self):
        response = self.client.get(url_for('api.get_posts'),
                                    content_type='application/json')
        self.assertTrue(response.status_code == 401)

    def test_posts(self): # adiciona um usuário
        r = Role.query.filter_by(name='User').first()
        self.assertIsNotNone(r) u = User(email='john@example.com',
                                        password='cat', confirm =True, role= r)
        db.session.add(u) db.session.commit()

        # escreva uma
        resposta de postagem =
            self.client.post( url_for('api.new_post'),
                headers=self.get_auth_header('john@example.com', 'cat'), data=json.dumps({ 'body' : 'corpo da postagem do blog' })
                self.assertTrue(response.status_code == 201) url =
                    response.headers.get('Location') self.assertIsNotNone(url)

        # obtém a nova
        resposta do post =
            self.client.get( url,
                headers=self.get_auth_header('john@example.com', 'cat'))
```

```

self.assertTrue(response.status_code == 200)
json_response = json.loads(response.data.decode('utf-8'))
self.assertTrue(json_response['url'] == url)
self.assertTrue(json_response['body'] == 'corpo da postagem do *blog*')
self.assertTrue(json_response['body_html'] ==
    '<p>corpo da postagem do <em>blog</em></p>')

```

Os métodos `setUp()` e `tearDown()` para testar a API são os mesmos do aplicativo normal, mas o suporte a cookies não precisa ser configurado porque a API não o utiliza. O método `get_api_headers()` é um método auxiliar que retorna os cabeçalhos comuns que precisam ser enviados com todas as solicitações. Isso inclui as credenciais de autenticação e os cabeçalhos relacionados ao tipo MIME. A maioria dos testes precisa enviar esses cabeçalhos.

O teste `test_no_auth()` é um teste simples que garante que uma solicitação que não inclua credenciais de autenticação seja rejeitada com o código de erro 401. O teste `test_posts()` adiciona um usuário ao banco de dados e, em seguida, usa a API RESTful para inserir uma postagem no blog e então leia de volta. Quaisquer solicitações que enviam dados no corpo devem codificá-los com `json.dumps()`, porque o cliente de teste Flask não codifica automaticamente para JSON. Da mesma forma, os corpos de resposta também são retornados no formato JSON e devem ser decodificados com `json.loads()` antes de serem inspecionados.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar `git checkout 15c` para verificar esta versão do aplicativo.

## Testes de ponta a ponta com selênio

O cliente de teste do Flask não pode emular totalmente o ambiente de um aplicativo em execução. Por exemplo, qualquer aplicativo que dependa do código JavaScript executado no navegador do cliente não funcionará, pois o código JavaScript incluído nas respostas retornadas ao teste não será executado como seria em um cliente de navegador da web real.

Quando os testes exigem o ambiente completo, não há outra opção a não ser usar um navegador web real conectado ao aplicativo rodando em um servidor web real. Felizmente, a maioria dos navegadores da web pode ser automatizado. **Selênio** é uma ferramenta de automação de navegador da web que suporta os navegadores da web mais populares nos três principais sistemas operacionais.

A interface Python para Selenium é instalada com pip:

```
(venv) $ pip install selenium
```

O teste com o Selenium requer que o aplicativo esteja sendo executado dentro de um servidor Web que esteja ouvindo solicitações HTTP reais. O método que será mostrado nesta seção inicia a aplicação com o servidor de desenvolvimento em um thread em segundo plano enquanto os testes são executados

no fio principal. Sob o controle dos testes, o Selenium inicia um navegador web e faz com que ele se conecte ao aplicativo para realizar as operações necessárias. maneira, para que as tarefas em segundo plano, como o mecanismo de cobertura de código, possam concluir seu trabalho de maneira limpa. O servidor web Werkzeug tem uma opção de desligamento, mas como o servidor está sendo executado isolado em seu próprio thread, a única maneira de pedir ao servidor para desligar é enviando uma solicitação HTTP regular. O [Exemplo 15-6](#) mostra a implementação de uma rota de desligamento do servidor.

*Exemplo 15-6. \_app/main/views.py: rota de desligamento do servidor*

```
@main.route('/shutdown') def
server_shutdown(): se não
    current_app.testing: abort(404)
        shutdown =
    request.environ.get('werkzeug.server.shutdown') se não for shutdown:
        abort(500)
    shutdown() return
'Desligando...'
```

A rota de desligamento funcionará apenas quando o aplicativo estiver sendo executado no modo de teste; invocá-lo em outras configurações falhará. O procedimento de desligamento real envolve chamar uma função de desligamento que Werkzeug expõe no ambiente. Depois de chamar essa função e retornar da solicitação, o servidor web de desenvolvimento saberá que precisa sair normalmente.

O [Exemplo 15-7](#) mostra o layout de um caso de teste configurado para executar testes com Selenium.

*Exemplo 15-7. tests/test\_selenium.py: Framework para testes usando Selenium*

**do webdriver de importação de selênio**

```
class SeleniumTestCase(unittest.TestCase): cliente
    = Nenhum

    @classmethod
    def setUpClass(cls): #
        inicia o Firefox try:
        cls.client =
            webdriver.Firefox() exceto: pass

        # ignore esses testes se o navegador não puder ser iniciado se
        cls.client: # crie o aplicativo cls.app = create_app('testing')
        cls.app_context = cls.app.app_context() cls.app_context.push()
```

```

# suprimir o registro para manter a saída do unittest limpa
import
logging logger = logging.getLogger('werkzeug')
logger.setLevel("ERROR")

# cria o banco de dados e preenche com alguns dados falsos
db.create_all()
Role.insert_roles()
User.generate_fake(10)
Post.generate_fake(10)

# adiciona um usuário administrador
admin_role = Role.query.filter_by(permissions=0xff).first() admin =
User(email='john@example.com', username='john', password='cat',
role=admin_role, confirmado=True) db.session.add(admin)
db.session.commit()

# inicia o servidor Flask em um thread
threading.Thread(target=cls.app.run).start()

@classmethod
def tearDownClass(cls): if
cls.client: # pare o
    servidor flask e o navegador cls.client.get('http://
localhost:5000/shutdown') cls.client.close()

# destrói o banco de
dados db.drop_all()
db.session.remove()

# remove o contexto do aplicativo
cls.app_context.pop()

def setUp(self): se
não self.client:
    self.skipTest(' Navegador da Web não disponível')

def tearDown(self):
    passar

```

Os métodos das classes `setUpClass()` e `tearDownClass()` são invocados antes e depois da execução dos testes nesta classe. A configuração envolve iniciar uma instância do Firefox através da API webdriver do Selenium e criar um aplicativo e um banco de dados com alguns dados iniciais para testes a serem usados. O aplicativo é iniciado em um thread usando o método `app.run()` padrão . No final, o aplicativo recebe uma solicitação para `/shutdown`, que faz com que o thread em segundo plano termine. O navegador é então fechado e o banco de dados de teste removido.



O Selenium suporta muitos outros navegadores além do Firefox. Consulte a [documentação do Selenium](#) se você deseja usar um navegador da web diferente.

O método `setUp()` que é executado antes de cada teste ignora os testes se o Selenium não puder iniciar o navegador da web no método `setUpClass()`. No [Exemplo 15-8](#) você pode ver um exemplo de teste construído com Selenium.

*Exemplo 15-8. tests/test\_selenium.py: Exemplo de teste de unidade de selênia*

```
class SeleniumTestCase(unittest.TestCase):
    # ...

    def test_admin_home_page(self): #
        # navega para a página inicial
        self.client.get('http://localhost:5000/')
        self.assertTrue(re.search('Hello,\s+Stranger!', self.client.page_source))

        # navegue até a página de
        login self.client.find_element_by_link_text('Log In').click()
        self.assertTrue('<h1>Login</h1>' em self.client.page_source)

        # login
        self.client.find_element_by_name('email').\
            send_keys('john@example.com')
        self.client.find_element_by_name('password').send_keys('cat')
        self.client.find_element_by_name('submit').click() self.assertTrue(re.search('Hello,
        \s+john!', self.client.page_source))

        # navegue até a página de perfil do usuário
        self.client.find_element_by_link_text('Perfil').click()
        self.assertTrue('<h1>john</h1>' em self.client.page_source)
```

Este teste efetua login no aplicativo usando a conta de administrador que foi criada em `setUpClass()` e abre a página de perfil. Observe como a metodologia de teste é diferente do cliente de teste Flask. Ao testar com o Selenium, os testes enviam comandos para o navegador da Web e nunca interagem diretamente com o aplicativo. Os comandos correspondem às ações que um usuário real executaria com o mouse ou o teclado.

O teste começa com uma chamada para `get()` com a página inicial do aplicativo. No navegador, isso faz com que a URL seja inserida na barra de endereços. Para verificar esta etapa, a fonte da página é verificada para "Hello, Stranger!" saudações.

Para ir para a página de login, o teste procura o link "Log In" usando `find_element_by_link_text()` e então chama `click()` nele para acionar um clique real em

o navegador. O Selenium fornece vários métodos de conveniência `find_element_by...`() que podem pesquisar elementos de diferentes maneiras.

Para efetuar login no aplicativo, o teste localiza os campos de formulário de e-mail e senha por seus nomes usando `find_element_by_name()` e, em seguida, grava texto neles com `send_keys()`. O formulário é enviado chamando `click()` no botão enviar. A saudação personalizada é verificada para garantir que o login foi bem-sucedido e que o navegador está agora na página inicial.

A parte final do teste localiza o link “Perfil” na barra de navegação e clica nele. Para verificar se a página de perfil foi carregada, o cabeçalho com o nome de usuário é pesquisado na origem da página.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar `git checkout 15d` para verificar esta versão do aplicativo. Esta atualização contém uma migração de banco de dados, portanto, lembre-se de executar `python manage.py db upgrade` depois de verificar o código. Para garantir que você tenha todas as dependências instaladas, execute também `pip install -r requirements/dev.txt`.

## Vale a pena?

Até agora você pode estar se perguntando se o teste usando o cliente de teste Flask ou o Selenium realmente vale a pena. É uma pergunta válida e não tem uma resposta simples.

Quer você goste ou não, seu aplicativo será testado. Se você não testar por conta própria, seus usuários se tornarão os testadores relutantes que encontrarão os bugs, e você terá que corrigir os bugs sob pressão. Testes simples e focados como os que exercitam modelos de banco de dados e outras partes da aplicação que podem ser executadas fora do contexto de uma aplicação devem ser sempre utilizados, pois têm um custo muito baixo e garantem o bom funcionamento das peças centrais do lógica de aplicação.

Testes de ponta a ponta do tipo que o cliente de teste Flask e o Selenium podem realizar às vezes são necessários, mas devido à maior complexidade para escrevê-los, eles devem ser usados apenas para funcionalidades que não podem ser testadas isoladamente. O código do aplicativo deve ser organizado de forma que seja possível inserir o máximo possível da lógica de negócios em modelos de banco de dados ou outras classes auxiliares que sejam independentes do contexto do aplicativo e, portanto, possam ser testadas facilmente. O código que existe nas funções de visualização deve ser simples e apenas atuar como uma camada fina que aceita requisições e invoca as ações correspondentes em outras classes ou funções que encapsulam a lógica da aplicação.

Então, sim, o teste vale absolutamente a pena. Mas é importante projetar uma estratégia de teste eficiente e escrever um código que possa aproveitá-la.



---

CAPÍTULO 16

# atuação

Ninguém gosta de aplicativos lentos. As longas esperas pelo carregamento das páginas frustram os usuários, por isso é importante detectar e corrigir problemas de desempenho assim que eles aparecerem. Neste capítulo, dois aspectos importantes de desempenho de aplicativos da Web são considerados.

## Registro de desempenho lento do banco de dados

Quando o desempenho do aplicativo se deteriora lentamente com o tempo, provavelmente é devido a consultas lentas ao banco de dados, que pioram à medida que o tamanho do banco de dados aumenta. A otimização de consultas de banco de dados pode ser tão simples quanto adicionar mais índices ou tão complexa quanto adicionar um cache entre o aplicativo e o banco de dados. A instrução de explicação , disponível na maioria das linguagens de consulta de banco de dados, mostra as etapas que o banco de dados executa para executar uma determinada consulta, muitas vezes expondo ineficiências no design do banco de dados ou do índice.

Mas antes de começar a otimizar as consultas, é necessário determinar quais são as que valem a pena otimizar. Durante uma solicitação típica, várias consultas de banco de dados podem ser emitidas, portanto, muitas vezes é difícil identificar quais de todas as consultas são as lentas.

Flask-SQLAlchemy tem uma opção para registrar estatísticas sobre consultas de banco de dados emitidas durante uma solicitação. No [Exemplo 16-1](#) , você pode ver como esse recurso pode ser usado para *registrar* consultas mais lentas que um limite configurado.

*Exemplo 16-1. app/main/views.py: relatar consultas de banco de dados lentas*

```
de flask.ext.sqlalchemy importação get_debug_queries

@main.after_app_request
def after_request(response): para
    consulta em get_debug_queries():
        if query.duration >= current_app.config['FLASKY_SLOW_DB_QUERY_TIME']:
            current_app.logger.warning( 'Consulta lenta: %s\nParâmetros: %s\nDuration:
                %fs\nContexto: %s\n' % (query.statement , query.parâmetros, query.duration,
```

```
    consulta.contexto))
resposta de retorno
```

Essa funcionalidade é anexada a um manipulador `after_app_request`, que funciona de maneira semelhante ao manipulador `before_app_request`, mas é invocado após o retorno da função `view` que trata da solicitação. O Flask passa o objeto de resposta para o manipulador `after_app_request` caso precise ser modificado.

Nesse caso, o manipulador `after_app_request` não modifica a resposta; ele apenas obtém os tempos de consulta registrados pelo Flask-SQLAlchemy e registra qualquer um dos lento.

A função `get_debug_queries()` retorna as consultas emitidas durante a solicitação como uma lista. As informações fornecidas para cada consulta são mostradas na [Tabela 16-1](#).

*Tabela 16-1. Estatísticas de consulta registradas pelo Flask-SQLAlchemy*

Nome	Descrição
instrução	A instrução SQL
parâmetros	Os parâmetros usados com a instrução SQL
start_time	A hora em que a consulta foi emitida end_time
	A hora em que a consulta retornou
duração	A duração da consulta em segundos
contexto	Uma string que indica o local do código-fonte onde a consulta foi emitida

O manipulador `after_app_request` percorre a lista e registra todas as consultas que duraram mais do que um limite fornecido na configuração. O log é emitido no nível de aviso.

Alterar o nível para "erro" faria com que todas as ocorrências de consultas lentas também fossem enviadas por e-mail.

A função `get_debug_queries()` é habilitada apenas no modo de depuração por padrão. Infelizmente, problemas de desempenho de banco de dados raramente aparecem durante o desenvolvimento porque bancos de dados muito menores são usados. Por esse motivo, é útil usar essa opção em produção. O [Exemplo 16-2](#) mostra as alterações de configuração necessárias para habilitar o desempenho da consulta do banco de dados no modo de produção.

*Exemplo 16-2. config.py: configuração para relatórios de consultas lentas*

```
configuração de classe :
# ...
SQLALCHEMY_RECORD_QUERIES =
    Verdadeiro FLASKY_DB_QUERY_TIMEOUT =
        0,5 #.
```

`SQLALCHEMY_RECORD_QUERIES` informa ao Flask-SQLAlchemy para habilitar a gravação de estatísticas de consulta. O limite de consulta lenta é definido como meio segundo. Ambas as variáveis de configuração foram incluídas na classe `Config base`, portanto, serão habilitadas para todas as configurações.

Sempre que uma consulta lenta for detectada, uma entrada será gravada no registrador de aplicativos do Flask. Para poder armazenar essas entradas de log, o registrador deve ser configurado. A configuração de registro depende em grande parte da plataforma que hospeda o aplicativo. Alguns exemplos são mostrados no [Capítulo 17](#).



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar git checkout 16a para verificar esta versão do aplicativo.

## Perfil do código-fonte

Outra possível fonte de problemas de desempenho é o alto consumo de CPU, causado por funções que executam computação pesada. Os criadores de perfil de código-fonte são úteis para localizar as partes mais lentas de um aplicativo. Um criador de perfil observa um aplicativo em execução e registra as funções que são chamadas e quanto tempo cada uma leva para ser executada. Em seguida, ele produz um relatório detalhado mostrando as funções mais lentas.



A criação de perfil geralmente é feita em um ambiente de desenvolvimento. Um criador de perfil de código-fonte torna o aplicativo mais lento porque ele precisa observar e anotar tudo o que está acontecendo. A criação de perfil em um sistema de produção não é recomendada, a menos que um criador de perfil leve especificamente projetado para ser executado em um ambiente de produção seja usado.

O servidor web de desenvolvimento do Flask, que vem de Werkzeug, pode habilitar opcionalmente o criador de perfil Python para cada solicitação. O [Exemplo 16-3](#) adiciona uma nova opção de linha de comando ao aplicativo que inicia o criador de perfil.

*Exemplo 16-3. manage.py: execute o aplicativo no criador de perfil de solicitação*

```
@manager.command
def profile(length=25, profile_dir=None):
    """Iniciar o
    aplicativo no criador de perfil de código."""
    de
    werkzeug.contrib.profiler import ProfilerMiddleware
    app.wsgi_app =
    ProfilerMiddleware(app.wsgi_app, restrições= [comprimento], profile_dir=profile_dir)

    app.run()
```



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar git checkout 16b para verificar esta versão do aplicativo.

Quando o aplicativo é iniciado com o perfil `python manage.py`, o console mostrará as estatísticas do criador de perfil para cada solicitação, que incluirá as 25 funções mais lentas. A opção `--length` pode ser usada para alterar o número de funções mostradas no relatório.

Se a opção `--profile-dir` for fornecida, os dados do perfil de cada solicitação serão salvos em um arquivo no diretório fornecido. Os arquivos de dados do criador de perfil podem ser usados para gerar relatórios mais detalhados que incluem um *gráfico de chamadas*. Para obter mais informações sobre o criador de perfil do Python, consulte a [documentação oficial](#).

Os preparativos para a implantação estão concluídos. O próximo capítulo lhe dará uma visão geral do que esperar ao implantar seu aplicativo.

## CAPÍTULO 17

Desdobramento, desenvolvimento

O servidor de desenvolvimento web que acompanha o Flask não é robusto, seguro ou eficiente o suficiente para funcionar em um ambiente de produção. Neste capítulo, as opções de implantação para aplicativos Flask são examinadas.

### Fluxo de trabalho de implantação

Independentemente do método de hospedagem utilizado, há uma série de tarefas que devem ser executadas quando o aplicativo é instalado em um servidor de produção. O melhor exemplo é a criação ou atualização das tabelas do banco de dados.

Ter que executar essas tarefas manualmente toda vez que o aplicativo é instalado ou atualizado é propenso a erros e demorado, portanto, um comando que executa todas as tarefas necessárias pode ser adicionado ao *manage.py*.

O Exemplo 17-1 mostra uma implementação de comando deploy apropriada para Flasky.

#### *Exemplo 17-1. manage.py: comando deploy*

```
@manager.command
def deploy():
    """Executar
       tarefas de implantação."""
    from flask.ext.migrate import upgrade
    from app.models import Role, User

    # migrar banco de dados para atualização de revisão
    # mais recente()
    # cria funções de usuário
    Role.insert_roles()
    # cria auto-seguidores para todos os usuários
    User.add_self_follows()
```

As funções invocadas por este comando foram todas criadas antes; eles são apenas invocados todos juntos.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar git checkout 17a para verificar esta versão do aplicativo.

Essas funções são todas projetadas de uma maneira que não causa problemas se forem executadas várias vezes. Projetar as funções de atualização dessa maneira possibilita a execução apenas deste comando deploy sempre que uma instalação ou atualização é feita.

## Registro de erros durante a produção

Quando a aplicação está sendo executada no modo de depuração, o depurador interativo do Werkzeug aparece sempre que ocorre um erro. O rastreamento de pilha do erro é exibido na página da web e é possível ver o código-fonte e até avaliar expressões no contexto de cada quadro de pilha usando o depurador interativo baseado na web do Flask.

O depurador é uma excelente ferramenta para depurar problemas de aplicativos durante o desenvolvimento, mas obviamente não pode ser usado em uma implantação de produção. Erros que ocorrem na produção são silenciados e, em vez disso, o usuário recebe uma página de erro de código 500. Mas, felizmente, os rastreamentos de pilha desses erros não são completamente perdidos, pois o Flask os grava em um *arquivo de log*.

Durante a inicialização, o Flask cria uma instância da classe logging.Logger do Python e a anexa à instância do aplicativo como app.logger. No modo de depuração, esse registrador grava no console, mas no modo de produção não há manipuladores configurados para ele por padrão, portanto, a menos que um manipulador seja adicionado, os logs não são armazenados. As alterações no [Exemplo 17-2](#) configuraram um manipulador de log que envia os erros que ocorrem durante a execução no modo de produção para os e-mails do administrador da lista configurados na configuração FLASKY\_ADMIN .

*Exemplo 17-2. config.py: Enviar e-mail para erros de aplicativo*

```
class ProductionConfig(Config):
    # ...
    @classmethod
    def init_app(cls, app):
        Config.init_app(app)

        # erros de e-mail para os administradores
        importar log de
        logging.handlers importar credenciais SMTPHandler =
        Nenhum
        seguro = Nenhum
        se getattr(cls, 'MAIL_USERNAME', Nenhum) não for Nenhum:
            credenciais = (cls.MAIL_USERNAME, cls.MAIL_PASSWORD)
```

```

if getattribute(cls, 'MAIL_USE_TLS', Nenhum):
    secure = () mail_handler = SMTPHandler(
        mailhost=(cls.MAIL_SERVER, cls.MAIL_PORT),
        fromaddr=cls.FLASKY_MAIL_SENDER,
        toaddrs=[cls.FLASKY_ADMIN],
        subject=cls.FLASKY_MAIL_SUBJECT_PREFIX + ' Erro de aplicação',
        credenciais=credentials, secure=secure)
    mail_handler.setLevel(logging.ERROR) app.
    logger.addHandler(mail_handler)

```

Lembre-se de que todas as instâncias de configuração têm um método estático `init_app()` que é invocado por `create_app()`. Na implementação desse método para a classe `ProductionConfig`, o registrador de aplicativos é configurado para registrar erros em um registrador de e-mail.

O nível de log do registrador de e-mail é definido como `logging.ERROR`, portanto, apenas problemas graves são enviados por e-mail. As mensagens registradas em níveis menores podem ser registradas em um arquivo, `syslog` ou qualquer outro método com suporte adicionando os manipuladores de log apropriados. O método de registro a ser usado para essas mensagens depende muito da plataforma de hospedagem.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar `git checkout 17b` para verificar esta versão do aplicativo.

## Implantação de nuvem

A última tendência em hospedagem de aplicativos é hospedar na “nuvem”. Essa tecnologia, formalmente conhecida como Platform as a Service (PaaS), libera o desenvolvedor do aplicativo das tarefas mundanas de instalação e manutenção das plataformas de hardware e software nas quais o aplicativo é executado. No modelo PaaS, um provedor de serviços oferece uma plataforma totalmente gerenciada na qual os aplicativos podem ser executados. O desenvolvedor do aplicativo usa ferramentas e bibliotecas do provedor para integrar o aplicativo à plataforma. O aplicativo é então carregado para os servidores mantidos pelo provedor e geralmente é implantado em segundos. A maioria dos provedores de PaaS oferece maneiras de “escalar” dinamicamente o aplicativo adicionando ou removendo servidores conforme necessário para acompanhar o número de solicitações recebidas.

As implantações na nuvem oferecem grande flexibilidade e são relativamente simples de configurar, mas é claro que tudo isso tem um preço. Heroku, um dos provedores de PaaS mais populares que oferece suporte muito bom para Python, é estudado em detalhes na seção a seguir.

## A plataforma Heroku

A Heroku foi um dos primeiros provedores de PaaS, estando no mercado desde 2007. A plataforma Heroku é muito flexível e suporta uma longa lista de linguagens de programação. Para implantar um aplicativo no Heroku, o desenvolvedor usa o Git para enviar o aplicativo para o próprio servidor Git do Heroku. No servidor, o comando git push aciona automaticamente a instalação, configuração e implantação.

O Heroku usa unidades de computação chamadas *dynos* para medir o uso e cobrar pelo serviço. O tipo mais comum de dinamômetro é o *dinamômetro da web*, que representa uma instância do servidor web. Um aplicativo pode aumentar sua capacidade de manipulação de solicitações usando mais dynos da web. O outro tipo de dinamômetro é o *dinamômetro do trabalhador*, que é usado para executar trabalhos em segundo plano ou outras tarefas de suporte.

A plataforma oferece um grande número de plug-ins e complementos para bancos de dados, suporte por e-mail e muitos outros serviços. As seções a seguir expandem alguns dos detalhes envolvidos na implantação do Flasky no Heroku.

### Preparando o aplicativo Para trabalhar

com o Heroku, o aplicativo deve ser hospedado em um repositório Git. Se você estiver trabalhando com um aplicativo hospedado em um servidor Git remoto, como GitHub ou BitBucket, clonar o aplicativo criará um repositório Git local perfeito para usar com o Heroku. Se o aplicativo ainda não estiver hospedado em um repositório Git, um deve ser criado para ele em sua máquina de desenvolvimento.



Se você planeja hospedar seu aplicativo no Heroku, é uma boa ideia começar a usar o Git desde o início. O GitHub tem guias de instalação e configuração para os três principais sistemas operacionais em seu [guia de ajuda](#).

### Criando uma conta no

**Heroku** Você deve criar uma conta no Heroku antes de poder usar o serviço. Você pode se inscrever e hospedar aplicativos no nível de serviço mais baixo sem nenhum custo, portanto, essa é uma ótima plataforma para experimentar.

### Instalando o Heroku Toolbelt A

maneira mais conveniente de gerenciar seus aplicativos Heroku é através do Heroku **Toolbelt** utilitários de linha de comando. O Toolbelt é composto por dois aplicativos Heroku:

- heroku: O cliente Heroku, usado para criar e gerenciar aplicativos • capataz: Uma ferramenta que pode simular o ambiente Heroku em seu próprio computador para testar

Observe que, se você ainda não tiver um cliente Git instalado, o instalador do Toolbelt também instalará o Git para você.

O utilitário do cliente Heroku precisa ter as credenciais da sua conta Heroku antes de se conectar ao serviço. O comando heroku login cuida disso:

```
$ heroku login
Insira suas credenciais Heroku.
Email: <your-email-address>
Senha (a digitação será ocultada): <your-password> Carregando
chave pública ssh .../id_rsa.pub
```



É importante que sua chave pública SSH seja carregada no Heroku, pois é isso que habilita o comando git push . Normalmente, o comando login cria e carrega uma chave pública SSH automaticamente, mas o comando heroku keys:add pode ser usado para carregar sua chave pública separadamente do comando login ou se você precisar carregar chaves adicionais.

### Criando um aplicativo

A próxima etapa é criar um aplicativo usando o cliente Heroku. Para fazer isso, primeiro certifique-se de que seu aplicativo esteja sob controle de origem do Git e execute o seguinte comando no diretório de nível superior:

```
$ heroku create <appname>
Criando <appname>... feito, pilha é cedro http://
<appname>.herokuapp.com/ | git@heroku.com:<appname>.git Git remote heroku
adicionado
```

Os nomes dos aplicativos Heroku devem ser exclusivos, portanto, encontre um nome que não seja usado por nenhum outro aplicativo. Conforme indicado pela saída do comando create , uma vez implantado o aplicativo estará disponível em <http://<appname>.herokuapp.com>. Os nomes de domínio personalizados também podem ser anexados ao aplicativo.

Como parte da criação do aplicativo, o Heroku aloca um servidor Git: `git@heroku.com:<appname>.git`. O comando create adiciona este servidor ao seu repositório Git local como um git remote com o nome heroku.

### Provisionando um banco de

**dados** O Heroku suporta bancos de dados Postgres como um complemento. Um pequeno banco de dados de até 10.000 linhas pode ser adicionado a um aplicativo sem nenhum custo:

```
$ heroku addons:add heroku-postgresql:dev Adicionando
heroku-postgresql:dev em <appname>... done, v3 (free)
Anexado como HEROKU_POSTGRESQL_BROWN_URL
O banco de dados foi criado e está disponível
! Este banco de dados está vazio. Se atualizar, você pode transferir! dados
de outro banco de dados com pgbackups:restore.
Use `heroku addons:docs heroku-postgresql:dev` para ver a documentação.
```

A referência HEROKU\_POSTGRESQL\_BROWN\_URL é para o nome da variável de ambiente que possui a URL do banco de dados. Observe que, ao tentar isso, você pode obter uma cor diferente do marrom. O Heroku suporta vários bancos de dados por aplicativo, cada um recebendo uma cor diferente na URL. Um banco de dados pode ser promovido e isso expõe sua URL em uma variável de ambiente DATABASE\_URL . O comando a seguir promove o banco de dados marrom criado anteriormente para primário:

```
$ heroku pg:promover HEROKU_POSTGRESQL_BROWN_URL
Promovendo HEROKU_POSTGRESQL_BROWN_URL para DATABASE_URL... feito
```

O formato da variável de ambiente DATABASE\_URL é exatamente o que SQLAlchemy espera. Lembre-se de que o script *config.py* usa o valor de DATABASE\_URL se estiver definido, então a conexão com o banco de dados Postgres agora funcionará automaticamente.

## Configurando o

**registro** O registro de erros fatais por email foi adicionado anteriormente, mas além disso é muito importante configurar o registro de categorias de mensagens menores. Um bom exemplo desse tipo de mensagem são os avisos para consultas lentas ao banco de dados adicionados no [Capítulo 16](#).

Com o Heroku, os logs devem ser gravados em stdout ou stderr. A saída de registro é capturada e acessível por meio do cliente Heroku com o comando `heroku logs` .

A configuração de registro pode ser adicionada à classe `ProductionConfig` em seu método estático `init_app()` , mas como esse tipo de registro é específico do Heroku, uma nova configuração pode ser criada especificamente para essa plataforma, deixando `ProductionConfig` como uma configuração de linha de base para diferentes tipos de plataformas de produção. A classe `HerokuConfig` é mostrada no [Exemplo 17-3](#).

### *Exemplo 17-3. config.py: configuração do Heroku*

```
class HerokuConfig(ProductionConfig):
    @classmethod def init_app(cls, app):
        ProductionConfig.init_app(app)

        # log to stderr import
        logging from
        logging import StreamHandler file_handler =
        StreamHandler()
        file_handler.setLevel(logging.WARNING)
        app.logger.addHandler(file_handler)
```

Quando o aplicativo é executado pelo Heroku, ele precisa saber que esta é a configuração que precisa ser usada. A instância do aplicativo criada em *manage.py* usa a variável de ambiente FLASK\_CONFIG para saber qual configuração usar, portanto, essa variável precisa ser definida no ambiente Heroku. As variáveis de ambiente são definidas usando o comando config:set do cliente Heroku:

```
$ heroku config:set FLASK_CONFIG=heroku
Configurando vars de configuração e reiniciando <appname>... feito, v4
FLASK_CONFIG: heroku
```

### **Configurando e-**

**mail** O Heroku não fornece um servidor SMTP, portanto, um servidor externo deve ser configurado. Existem vários complementos de terceiros que integram o suporte de envio de e-mail pronto para produção com o Heroku, mas para fins de teste e avaliação é suficiente usar a configuração padrão do Gmail herdada da classe Config base .

Como pode ser um risco de segurança incorporar credenciais de login diretamente no script, o nome de usuário e a senha para acessar o servidor SMTP do Gmail são fornecidos como variáveis de ambiente:

```
$ heroku config:set MAIL_USERNAME=<your-gmail-username> $
heroku config:set MAIL_PASSWORD=<your-gmail-password>
```

### **Executando um servidor web de**

**produção** O Heroku não fornece um servidor web para os aplicativos que hospeda. Em vez disso, espera que os aplicativos iniciem seus próprios servidores e escutem o número da porta definido na variável de ambiente PORT.

O servidor web de desenvolvimento que vem com o Flask terá um desempenho muito ruim porque não foi projetado para ser executado em um ambiente de produção. Dois servidores web prontos para produção que funcionam bem com aplicativos Flask são **Gunicorn** e **uWSGI**.

Para testar a configuração do Heroku localmente, é uma boa ideia instalar o servidor web no ambiente virtual. Por exemplo, Gunicorn é instalado da seguinte forma:

```
(venv) $ pip install gunicorn
```

Para executar o aplicativo no Gunicorn, use o seguinte comando:

```
(venv) $ gunicorn manage:app
2013-12-03 09:52:10 [14363] [INFO] Iniciando gunicorn 18.0 2013-12-03
09:52:10 [14363] [INFO] Ouvindo em: http:// 127.0.0.1:8000 (14363)
03/12/2013 09:52:10 [14363] [INFO] Usando o trabalhador: sync
03/12/2013 09:52:10 [14368] [INFO] Inicializando o trabalhador com pid: 14368
```

O argumento *manage:app* indica o pacote ou módulo que define o aplicativo à esquerda dos dois pontos e o nome da instância do aplicativo dentro desse pacote à direita. Observe que o Gunicorn usa a porta 8000 por padrão, não 5000 como o Flask.

## Adicionando um arquivo

**de requisitos** O Heroku carrega as dependências do pacote de um arquivo *requirements.txt* armazenado na pasta de nível superior. Todas as dependências neste arquivo serão importadas para um ambiente virtual criado pelo Heroku como parte da implantação.

O arquivo de requisitos do Heroku deve incluir todos os requisitos comuns para a versão de produção do aplicativo, o pacote *psycopg2* para habilitar o suporte ao banco de dados Postgres e o servidor web Gunicorn.

O Exemplo 17-4 mostra um arquivo de requisitos de exemplo.

*Exemplo 17-4. requirements.txt: arquivo de requisitos do Heroku*

```
-r requirements/prod.txt
gunicorn==18.0 psycopg2==2.5.1
```

## Adicionando um

**Procfile** Heroku precisa saber qual comando usar para iniciar o aplicativo. Este comando é fornecido em um arquivo especial chamado *Procfile*. Este arquivo deve ser incluído na pasta de nível superior do aplicativo.

O Exemplo 17-5 mostra o conteúdo deste arquivo.

*Exemplo 17-5. Procfile: Heroku Procfile*

```
web: gunicorn gerenciar: app
```

O formato do Procfile é muito simples: em cada linha é dado um nome de tarefa, seguido de dois pontos e depois o comando que executa a tarefa. O nome da tarefa web é especial; é reconhecido pelo Heroku como a tarefa que inicia o servidor web. O Heroku dará a essa tarefa uma variável de ambiente PORT definida para a porta na qual o aplicativo precisa ouvir solicitações. Gunicorn por padrão honra a variável PORT se estiver configurada, então não há necessidade de incluí-la no comando de inicialização.



Os aplicativos podem declarar tarefas adicionais com nomes diferentes da web no Procfile. Estes podem ser outros serviços necessários à aplicação. O Heroku inicia todas as tarefas listadas no Procfile quando o aplicativo é implantado.

## Testando com o Foreman

Heroku Toolbelt inclui um segundo utilitário chamado *Foreman*, usado para executar o aplicativo localmente através do Procfile para fins de teste. As variáveis de ambiente como FLASK\_CONFIG que são definidas por meio do cliente Heroku estão disponíveis apenas no Heroku

servidores, então eles também devem ser definidos localmente para que o ambiente de teste no Foreman seja semelhante. Foreman procura por essas variáveis de ambiente em um arquivo chamado .env no diretório de nível superior do aplicativo. Por exemplo, o arquivo .env pode conter as seguintes variáveis:

```
FLASK_CONFIG=heroku
MAIL_USERNAME=<seu-nome de usuário>
MAIL_PASSWORD=<sua-senha>
```



Como o arquivo .env contém senhas e outras informações confidenciais da conta, ele nunca deve ser adicionado ao repositório Git.

O capataz tem várias opções, mas as duas principais são a execução do capataz e a partida do capataz. O comando run pode ser usado para executar comandos arbitrários no ambiente do aplicativo e é perfeito para executar o comando deploy que o aplicativo usa para criar o banco de dados:

```
(venv) $ foreman executar python manage.py deploy
```

O comando start lê o Procfile e executa todas as tarefas nele:

```
(venv) $ capataz início
22:55:08 web.1 | começou com pid 4246 22:55:08
web.1 | 2013-12-03 22:55:08 [4249] [INFO] Iniciando gunicorn 18.0 22:55:08 web.1 | 2013-12-03
22:55:08 [4249] [INFO] Ouvindo em: http://... 22:55:08 web.1 | 2013-12-03 22:55:08 [4249] [INFO]
Usando o trabalhador: sincronização 22:55:08 web.1 | 2013-12-03 22:55:08 [4254] [INFO] Inicializando
o trabalhador com pid: 4254
```

O Foreman consolida a saída de log de todas as tarefas iniciadas e a despeja no console, com cada linha prefixada com um carimbo de data/hora e o nome da tarefa.

É possível simular vários dynos usando a opção -c . Por exemplo, o comando a seguir inicia três web workers, cada um escutando em uma porta diferente:

```
(venv) $ capataz start -c web=3
```

### Habilitando HTTP seguro com Flask-SSLify Quando o

usuário efetua login no aplicativo enviando um nome de usuário e uma senha em um formulário da web, esses valores podem ser interceptados durante a viagem por terceiros, conforme discutido várias vezes antes. Para evitar que as credenciais do usuário sejam roubadas dessa forma, é necessário usar HTTP seguro, que criptografa todas as comunicações entre clientes e o servidor usando criptografia de chave pública.

A Heroku disponibiliza todos os aplicativos acessados no domínio [herokuapp.com](http://herokuapp.com) em <http://> e <https://> sem qualquer configuração usando o próprio certificado SSL da Heroku

cate. A única ação necessária é que o aplicativo intercepte todas as solicitações enviadas para a interface `http://` e redirecione-as para `https://`, e é isso que a extensão Flask SSLify faz.

A extensão precisa ser adicionada ao arquivo `requirements.txt`. O código do [Exemplo 17-6](#) é usado para ativar o ramal.

*Exemplo 17-6. app/\_\_init\_\_.py: redireciona todas as solicitações para HTTP seguro*

```
def create_app(config_name):
    ...
    # se não app.debug e não app.testing e não app.config['SSL_DISABLE']:
    from flask.ext.sslify import SSLify
    = SSLify(app)
    # ...
```

O suporte para SSL precisa ser ativado apenas no modo de produção e somente quando a plataforma for compatível. Para facilitar a ativação e desativação do SSL, uma nova variável de configuração chamada `SSL_DISABLE` foi adicionada. A classe Config base a define como True, para que o SSL não seja usado por padrão e a classe HerokuConfig a substitua. A implementação desta variável de configuração é mostrada no [Exemplo 17-7](#).

*Exemplo 17-7. config.py: Configure o uso de SSL*

```
configuração de classe :
# ...
SSL_DISABLE = Verdadeiro

class HerokuConfig(ProductionConfig):
    ...
    # SSL_DISABLE = bool(os.environ.get('SSL_DISABLE'))
```

O valor de `SSL_DISABLE` em HerokuConfig é obtido de uma variável de ambiente com o mesmo nome. Se a variável de ambiente for definida como algo diferente de uma string vazia, a conversão para Boolean retornará True, desabilitando o SSL. Se a variável de ambiente não existir ou estiver definida como uma string vazia, a conversão para Boolean fornecerá um valor False . Para evitar que o SSL seja ativado ao usar o Foreman, é necessário adicionar `SSL_DISABLE=1` ao arquivo `.env` .

Com essas mudanças, os usuários serão obrigados a utilizar o servidor SSL, mas há mais um detalhe que precisa ser tratado para que o suporte seja completo. Ao usar o Heroku, os clientes não se conectam diretamente aos aplicativos hospedados, mas a um servidor proxy reverso que redireciona as solicitações para os aplicativos. Nesse tipo de configuração, apenas o servidor proxy é executado no modo SSL; os aplicativos recebem todas as solicitações do servidor proxy sem SSL porque não há necessidade de usar segurança forte para solicitações internas à rede Heroku. Isso é um problema quando o aplicativo precisa gerar URLs absolutos que correspondam à segurança da solicitação, pois `request.is_secure` sempre será False quando um servidor proxy reverso for usado.

Um exemplo de quando isso se torna um problema é a geração de URLs de avatar. Se você se lembrar do [Capítulo 10](#), o método `gravatar()` do modelo `User` que gera as URLs do Gravatar verifica `request.is_secure` para gerar a versão segura ou não segura da URL. A geração de um avatar não seguro quando a página foi solicitada por SSL faria com que alguns navegadores exibissem um aviso de segurança para o usuário, portanto, todos os componentes de uma página devem ter segurança correspondente.

Os servidores proxy passam informações que descrevem a solicitação original do cliente para os servidores da Web redirecionados por meio de cabeçalhos HTTP personalizados, portanto, é possível determinar se o usuário está se comunicando com o aplicativo por SSL observando-os. Werkzeug fornece um *middleware* WSGI que verifica os cabeçalhos personalizados do servidor proxy e atualiza o objeto de solicitação de acordo para que, por exemplo, `request.is_secure` refleja a segurança da solicitação que o cliente enviou ao servidor proxy reverso e não a solicitação que o servidor proxy enviado para o aplicativo.

[O Exemplo 17-8](#) mostra como adicionar o middleware `ProxyFix` ao aplicativo.

*Exemplo 17-8. config.py: Suporte para servidores proxy*

```
class HerokuConfig(ProductionConfig):
    ...
    # @classmethod
    def init_app(cls, app):
        ...
        # manipula os cabeçalhos do
        # servidor proxy de werkzeug.contrib.fixers import
        ProxyFix app.wsgi_app = ProxyFix(app.wsgi_app)
```

O middleware é adicionado no método de inicialização para a configuração do Heroku. Middlewares WSGI, como `ProxyFix`, são adicionados ao encapsular o aplicativo WSGI. Quando uma solicitação chega, os middlewares têm a chance de inspecionar o ambiente e fazer alterações antes que a solicitação seja processada. O middleware `ProxyFix` é necessário não apenas para Heroku, mas em qualquer implantação que use um servidor proxy reverso.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar `git checkout 17c` para verificar esta versão do aplicativo. Para garantir que você tenha todas as dependências instaladas, execute também `pip install -r requirements.txt`.

## Implantando com git push A

etapa final do processo é enviar o aplicativo para os servidores Heroku. Certifique-se de que todas as alterações sejam confirmadas no repositório Git local e, em seguida, use `git push heroku master` para fazer upload do aplicativo para o `heroku remote`:

```
$ git push heroku master
Contando objetos: 645, feito.
Compressão delta usando até 8 threads.
Comprimindo objetos: 100% (315/315), feito.
Objetos de escrita: 100% (645/645), 95,52 KiB, feito.
Total 645 (delta 369), reutilizado 457 (delta 288)

.---> Aplicativo Python
detectado .----> Nenhum runtime.txt fornecido; assumindo python-2.7.4.
.----> Preparando o tempo de execução do Python (python-2.7.4)
...
-----> Tamanho do slug compilado: 32,8
MB -----> Iniciando... concluido, v8 http://
                <appname>.herokuapp.com implantado no Heroku

Para git@heroku.com:<appname>.git *
[novo branch]                      mestre -> mestre
```

O aplicativo agora está implementado e em execução, mas não funcionará corretamente porque o comando deploy não foi executado. O cliente Heroku pode executar este comando da seguinte forma:

```
$ heroku run python manage.py deploy
Executando `python manage.py predeploy` anexado ao terminal... up, run.8449 INFO
[alembic.migration] Context impl PostgresqlImpl.
INFO [alembic.migration] Assumirá DDL transacional.
...
```

Depois que as tabelas do banco de dados são criadas e configuradas, o aplicativo pode ser reiniciado para que seja iniciado de forma limpa:

```
$ heroku restart
Reiniciando dynos... feito
```

O aplicativo deve agora estar totalmente implantado e online em <https://<appname>.herokuapp.com>.

### **Revisando Logs** A saída

de log gerada pelo aplicativo é capturada pelo Heroku. Para visualizar o conteúdo do log, use o comando logs :

```
$ heroku logs
```

Durante o teste, também pode ser conveniente seguir o arquivo de log, o que pode ser feito da seguinte maneira:

```
$ heroku logs -t
```

#### **Implantando uma atualização** Quando um

aplicativo Heroku precisa ser atualizado, o mesmo processo precisa ser repetido.

Depois que todas as alterações foram confirmadas no repositório Git, os seguintes comandos executam uma atualização:

```
$ heroku Maintenance:on  
git push heroku master  
heroku run python manage.py deploy  
restart $ heroku Maintenance:off
```

A opção de manutenção disponível no cliente Heroku deixará o aplicativo offline durante a atualização e mostrará uma página estática que informa aos usuários que o site voltará em breve.

## **Hospedagem Tradicional**

A opção de hospedagem tradicional envolve a compra ou aluguel de um servidor, físico ou virtual, e a configuração de todos os componentes necessários. Isso geralmente é mais barato do que hospedar na nuvem, mas obviamente muito mais trabalhoso. As seções a seguir lhe darão uma idéia do trabalho envolvido.

#### **Configuração do servidor**

Existem várias tarefas de administração que devem ser executadas no servidor antes que ele possa hospedar aplicativos:

- Instale um servidor de banco de dados como *MySQL* ou *Postgres*. Usar um banco de dados *SQLite* também é possível, mas não é recomendado para um servidor de produção devido às suas muitas limitações.
- Instale um Mail Transport Agent (MTA), como o *Sendmail*, para enviar e-mails aos usuários.
- Instale um servidor web pronto para produção, como *Gunicorn* ou *uWSGI*.
- Compre, instale e configure um certificado SSL para habilitar HTTP seguro.
- (Opcional, mas altamente recomendado) Instale um servidor web de proxy reverso de front-end, como *nginx* ou *Apache*. Esse processo servirá arquivos estáticos diretamente e encaminhará quaisquer outras solicitações para o servidor web do aplicativo que está escutando em uma porta privada no *localhost*.

- Proteção do servidor. Isso agrupa várias tarefas que têm o objetivo de reduzir vulnerabilidades no servidor, como instalar firewalls, remover software e serviços não utilizados e assim por diante.

### Importando variáveis de ambiente

Da mesma forma que o Heroku, um aplicativo executado em um servidor autônomo depende de determinadas configurações, como URL do banco de dados, credenciais do servidor de e-mail e nome de configuração. Eles são armazenados em variáveis de ambiente que devem ser importadas antes do início do aplicativo.

Como não há Heroku ou Foreman para importar essas variáveis, essa tarefa precisa ser feita pelo próprio aplicativo durante a inicialização. O bloco de código abreviado no [Exemplo 17-9](#) carrega e analisa um arquivo .env semelhante ao usado com o Foreman. Esse código pode ser adicionado ao script de inicialização `manage.py` antes que a instância do aplicativo seja criada.

*Exemplo 17-9. manage.py: Importar ambiente do arquivo .env*

```
if os.path.exists('.env'):
    print('Importing environment from .env...')
    for line in open('.env'):
        var = line.strip().split('=')
        if len(var) == 2:
            os.environ[var[0]] = var[1]
```

O arquivo .env deve conter pelo menos a variável FLASK\_CONFIG que seleciona a configuração. ração para usar.

### Configurando o Log

Para servidores baseados em Unix, o log pode ser enviado ao daemon `syslog`. Uma nova configuração específica para Unix pode ser criada como uma subclasse de `ProductionConfig`, conforme mostrado no [Exemplo 17-10](#).

*Exemplo 17-10. config.py: configuração de exemplo Unix*

```
class UnixConfig(ProductionConfig):
    @classmethod
    def init_app(cls, app):
        ProductionConfig.init_app(app)

        # log para syslog
        importação log
        de logging.handlers import SysLogHandler
        syslog_handler = SysLogHandler()
        syslog_handler.setLevel(logging.WARNING)
        app.logger.addHandler(syslog_handler)
```

Com essa configuração, os logs do aplicativo serão gravados em `/var/log/messages`. O serviço syslog pode ser configurado para gravar um arquivo de log separado ou para enviar os logs para uma máquina diferente, se necessário.



Se você clonou o repositório Git do aplicativo no GitHub, você pode executar `git checkout 17d` para verificar esta versão do aplicativo.



---

CAPÍTULO 18

# Recursos adicionais

Você está praticamente pronto com este livro. Parabéns! Espero que os tópicos que abordei tenham lhe dado uma base sólida para começar a construir seus próprios aplicativos com o Flask. Os exemplos de código são de código aberto e têm uma licença permissiva, portanto, você pode usar o quanto quiser do meu código para semear seus projetos, mesmo que sejam de natureza comercial. Neste pequeno capítulo final, quero dar a você uma lista de dicas e recursos adicionais que podem ser úteis à medida que você continua trabalhando com o Flask.

## Usando um ambiente de desenvolvimento integrado (IDE)

O desenvolvimento de aplicativos Flask em um ambiente de desenvolvimento integrado (IDE) pode ser muito conveniente, pois recursos como conclusão de código e um depurador interativo podem acelerar consideravelmente o processo de codificação. Alguns dos IDEs que funcionam bem com o Flask estão listados aqui:

- **PyCharm:** IDE comercial da JetBrains com edições Community (gratuita) e Professional (paga), ambas compatíveis com aplicativos Flask. Disponível em Linux, Mac OS X e Windows.
- **PyDev:** IDE de código aberto baseado no Eclipse. Disponível em Linux, Mac OS X e Janelas.
- **Ferramentas Python para Visual Studio:** IDE gratuito criado como uma extensão do ambiente Visual Studio da Microsoft. Apenas para Microsoft Windows.



Ao configurar um aplicativo Flask para iniciar em um depurador, adicione as opções `--passthrough-errors --no-reload` ao comando `runserver`. A primeira opção desabilita a captura de erros pelo Flask para que as exceções lançadas enquanto uma solicitação é tratada sejam enviadas até o depurador. O segundo desabilita o módulo recarregador, o que confunde alguns depuradores.

## Encontrando extensões do Flask

Os exemplos neste livro contam com várias extensões e pacotes, mas há muitos outros que também são úteis e não foram discutidos. Segue uma pequena lista de alguns pacotes adicionais que valem a pena explorar:

- [Frasco-Babel](#): Suporte à internacionalização e localização •
- [Flask-RESTful](#): Ferramentas para construir APIs RESTful • [Aipo](#): Fila de tarefas para processar trabalhos em segundo plano •
- [Frozen-Flask](#): Conversão de um aplicativo Flask para um site estático •
- [Flask-DebugToolbar](#): Ferramentas de depuração no navegador • [Flask-Assets](#): Mesclando, minificando e compilando ativos CSS e JavaScript • [Flask-OAuth](#): Autenticação contra provedores OAuth • [Flask-OpenID](#): Autenticação contra provedores OpenID • [Flask-WhooshAlchemy](#): Pesquisa de texto completo para modelos Flask-SQLAlchemy com base em [Whoosh](#)
- [Sessão Flask-KV](#): Implementação alternativa de sessões de usuário que usam server-side armazenar

Se a funcionalidade de que você precisa para o seu projeto não for coberta por nenhuma das extensões e pacotes mencionados neste livro, seu primeiro destino para procurar extensões adicionais deve ser o [Flask Extension Registry oficial](#). Outros bons lugares para pesquisar são o [Python Package Index](#), [GitHub](#), e [BitBucket](#).

## Envolvendo-se com o Flask

O Flask não seria tão incrível sem o trabalho feito por sua comunidade de desenvolvedores. Como você está se tornando parte desta comunidade e se beneficiando do trabalho de tantos voluntários, você deve considerar encontrar uma maneira de retribuir. Aqui estão algumas ideias para ajudá-lo a começar:

- Revise a documentação do Flask ou seu projeto relacionado favorito e envie correções ou melhorias.

- Traduza a documentação para um novo idioma. •

Responda a perguntas em sites de perguntas e respostas, como o

**Stack Overflow.** • Fale sobre seu trabalho com seus colegas em reuniões ou conferências de grupos de usuários. • Contribuir com correções de bugs ou melhorias nos pacotes que você usa. • Escreva novas extensões do Flask e libere-as como código aberto. • Libere seus aplicativos como código aberto.

Espero que você decida se voluntariar de uma dessas maneiras ou de qualquer outra que seja significativa para você. Se sim, obrigado!



---

# Índice

## Símbolos

arquivo .env, 222, 228

Recursos de interfaces de programação **de** aplicativos  
(APIs), 188 versionamento, 178 autenticação, 181,  
184

Nuvem **C**,  
cobertura de código  
217, configuração 197, 211, 216, 228

## D

tabela de  
associação de banco de  
dados, 150 filter\_by query filter,  
159 join query filter, 159 joins,  
158 migrations, 64

NoSQL, 50  
desempenho, 211  
modelo relacional, 49  
relacionamentos, 56, 61, 149, 166  
SQL, 49  
depuração, 216

decoradores, 115

## E

e-mail, 221  
tratamento de erros, 180

## F

Flask, 3  
funções abort, 16, 180 função  
add\_url\_route, 14 gancho  
after\_app\_request, 211 função de  
fábrica de aplicativos, 78 decorador  
app\_errorhandler, 80, 180 gancho  
before\_app\_request, 107 gancho before\_request,  
15, 183 blueprints, 79, 92, 179 objeto de  
configuração, 78 contexto processadores, 63,  
116 contextos, 12, 84 cookies, 161 variável de  
contexto current\_app, 13, 84 argumento de  
depuração, 9 rotas dinâmicas, 8 decorador de  
tratamento de erros, 29, 79, 80, 188 registro  
de extensão, 232 função flash, 46 classe Flask,  
7 espaço de nomes frasco.ext, 17, 26

*Gostaríamos de ouvir suas sugestões para melhorar nossos índices. Envie um e-mail para index@oreilly.com.*

- g variável de contexto, [13](#), [15](#)
- função de modelo `get_flashed_messages`, [47](#) função
- `jsonify`, [179](#) função `make_response`, [16](#), [161](#) argumentos de métodos, [42](#) função de redirecionamento, [16](#), [45](#) função `render_template`, [22](#), [46](#) variável de contexto de solicitação, [12](#), [13](#)
  
- Classe de resposta,  
decorador de [16](#) rotas, [8](#), [14](#), [79](#)  
métodos de execução, [9](#)
- Configuração `SECRET_KEY`, desligamento de [76](#) servidores, variável de contexto de [206](#) sessões, [13](#), [45](#) métodos `set_cookie`, [16](#) arquivos estáticos, [32](#) pastas estáticas, [33](#) pastas de templates, [22](#) clientes de teste, [200](#)
  
- Mapa de URL, [14](#)  
função `url_for`, [32](#), [45](#), [81](#), [106](#) argumento `url_prefix`, [93](#)
- Flask-Bootstrap, [26](#)  
blocos, [28](#) macro  
`quick_form`, [40](#)
- Flask-HTTPAuth, [181](#)
- Flask-Login, [94](#)  
Classe `AnonymousUserMixin`, [115](#) variável de contexto `current_user`, [96](#)  
Classe `LoginManager`, [95](#)  
decorador `login_required`, [95](#), [107](#) função `login_user`, [98](#) função `logout_user`, [99](#)
  
- Classe `UserMixin`, [94](#)  
decorador `user_loader`, [95](#)
- Flask-Mail, [69](#)  
envios assíncronos, [72](#)
- Configuração do Gmail, [69](#)
- Flask-Migrar, [64](#)
- Flask-Moment, método  
de [33](#) formatos, método  
[34](#) `fromNow`, método [34](#)  
`lang`, [35](#)
- Frasco-Script, [17](#)
- Flask-SQLAlchemy, [52](#) método  
add session, [58](#), [60](#) opções de coluna, [55](#) tipos de coluna, [54](#)  
método `create_all`, [58](#)
  
- delete session method, [60](#) `drop_all` method, [58](#) `filter_by` query filter, [63](#) `get_debug_queries` function, [211](#) models, [54](#)
  
- Configuração do MySQL, método de consulta de paginação [52](#), [191](#)
- Configuração do Postgres, [52](#)  
executores de consulta, [61](#) filtros de consulta, [61](#) objeto de consulta, [60](#)
- `SQLALCHEMY_COMMIT_ON_TEARDOWN`  
Configuração PARA BAIXO, [53](#)
- Configuração `SQLALCHEMY_DATABASE_URI`, [53](#), [76](#)
  
- configuração SQLite, [52](#)
- Flask-SSLify, [223](#)
- Frasco-WTF, [37](#)  
Classe `BooleanField`, [96](#)  
Falsificação de solicitação entre sites (CSRF), [37](#)  
validadores personalizados, [101](#)  
Validador de e-mail, [96](#)
- EqualTo validador, [101](#)  
Classe de formulário, [38](#)  
campos de formulário, [39](#)
- Classe `PasswordField`, [96](#)  
Validador `Regexp`,  
renderização [101](#), [40](#)  
Validador obrigatório, [38](#)  
classe `StringField`, [38](#)  
Classe `SubmitField`, [38](#)  
função `validate_on_submit`, [98](#) método `validate_on_submit`, [42](#) validadores, [38](#), [39](#)
  
- Capataz, [218](#), [222](#)
  
- G**
- Git, [xiii](#), [218](#), [225](#)
- Gunicorn, [221](#), [222](#)
  
- H**
- Heroku, [218](#)  
Cliente Heroku, [218](#)  
Cinto de ferramentas Heroku, [218](#)  
Códigos de status HTTP, [180](#)  
HTTPie, [192](#)

**E**u integrei ambientes de desenvolvimento (IDEs),

231

é perigoso, 104, 184

**J**JavaScript Object Notation (JSON), 177 serialização,  
186

Jinja2, 3, 22

diretiva de bloco, 25, 27

diretiva de extensão, 25, 27

filtros, 23 para diretiva, 24 se

diretiva, 24, 41 diretiva de

importação, 24, 41 diretiva de

inclusão, 25 diretiva de macro,

24 filtro seguro, 24 diretiva de

conjunto, 170 super macro, 25

herança de modelo, 25 variáveis,

23

**P**erfilagem L , 211, 211, 216, 220, 226, 228**M**

manage.py, 76, 81, 228

comando de cobertura, 197

comando db, 64 comando

deploy, 215, 222 comando profile,

213 comando runserver, 18 comando

shell, 18, 63 comando test, 84

pip, 6

plataforma como serviço (PaaS), 217

padrão post/redirect/get, 44

Procfile, 222

código-fonte de criação de perfil,

213 servidores proxy, 225

**R**

Representational State Transfer (REST), arquivo de 175

requisitos, 76, 82, 222

Rich Internet Applications (RIAs), 175

**S**

HTTP seguro, 223

Selenium, 205

criador de perfil de código-fonte,

213 syslog, 228

**T**este T , 192, 197

testes de unidade, 83, 92,

116 aplicativos da web, 200

serviços da web, 204

Bootstrap do Twitter, 26

DENTRO

teste unitário, 83

fragmento de URL, 168

funções de usuário, 111

uWSGI, 221

DENTRO

virtualenv, 4

comandos de ativação, 5

comandos de desativação, 6

**P**

paginação, 191

senhas de segurança, hashing, 90

performances, 213 permissões, 112

Dentro

Interface de Gateway de Servidor Web (WSGI), 7

Ferramenta, 3, 90, 213, 216

Middleware ProxyFix WSGI, 225

## Sobre o autor

---

Miguel Grinberg tem mais de 25 anos de experiência como engenheiro de software. No trabalho, ele lidera uma equipe de engenheiros que desenvolve software de vídeo para a indústria de transmissão. Ele tem um blog (<http://blog.miguelgrinberg.com>) onde escreve sobre uma variedade de tópicos, incluindo desenvolvimento web, robótica, fotografia e, ocasionalmente, resenhas de filmes. Ele mora em Portland, Oregon com sua esposa, quatro filhos, dois cachorros e um gato.

### Colophon O

---

animal na capa da *Flask Web Development* é um Mastim dos Pirinéus (uma raça de *Canis lupus familiaris*). Esses cães espanhóis gigantes são descendentes de um antigo cão guardião de gado chamado Molossus, que foi criado pelos gregos e romanos e agora está extinto. No entanto, este ancestral é conhecido por ter desempenhado um papel na criação de muitas raças que são comuns hoje, como o Rottweiler, Dogue Alemão, Terra Nova e Cane Corso. Os Mastins dos Pireneus só foram reconhecidos como uma raça pura desde 1977, e o Pyrenean Mastiff Club of America está trabalhando para promover esses cães como animais de estimação nos Estados Unidos.

Após a Guerra Civil Espanhola, a população de Mastins dos Pirineus em sua terra natal despencou, e a raça só sobreviveu devido ao trabalho dedicado de alguns criadores dispersos por todo o país. O pool genético moderno para os Pireneus deriva dessa população do pós-guerra, tornando-os propensos a doenças genéticas como a displasia da anca.

Hoje, os donos responsáveis certificam-se de que seus cães sejam testados para doenças e radiografados para procurar anormalidades no quadril antes de serem criados.

Mastiffs dos Pireneus machos adultos podem atingir mais de 200 libras quando totalmente crescidos, portanto, possuir este cão requer um compromisso com um bom treinamento e muito tempo ao ar livre.

Apesar do seu tamanho e história como caçadores de ursos e lobos, o Pirinéu tem um temperamento muito calmo e é um excelente cão de família. Eles podem ser confiáveis para cuidar das crianças e proteger a casa, ao mesmo tempo em que são dóceis com outros cães.

Com socialização adequada e liderança forte, os Mastins dos Pireneus prosperam em um ambiente doméstico e serão um excelente guardião e companheiro.

A imagem da capa é do Wood's *Animate Creation*. As fontes da capa são URW Typewriter e Guardian Sans. A fonte do texto é Adobe Minion Pro; a fonte do título é Adobe Myriad Condensed; e a fonte do código é Ubuntu Mono da Dalton Maag.