



Construindo REST APIs com Flask

Crie serviços Web Python
com MySQL

Kunal Relan



www.allitebooks.com

Construindo APIs REST com Flask

Crie serviços Web Python com MySQL

Kunal Relan

Apress®

Construindo APIs REST com Flask: Crie serviços Web Python com MySQL

Kunal Relan

Nova Deli, Deli, Índia

ISBN-13 (pbk): 978-1-4842-5021-1

ISBN-13 (eletrônico): 978-1-4842-5022-8

<https://doi.org/10.1007/978-1-4842-5022-8>

Copyright © 2019 por Kunal Relan

Este trabalho está sujeito a direitos autorais. Todos os direitos são reservados à Editora, quer se trate da totalidade ou de parte do material, especificamente os direitos de tradução, reimpressão, reutilização de ilustrações, recitação, transmissão, reprodução em microfilmes ou de qualquer outra forma física, e transmissão ou armazenamento de informações e recuperação, adaptação eletrônica, software de computador ou por metodologia semelhante ou diferente agora conhecida ou desenvolvida posteriormente.

Nomes, logotipos e imagens de marcas registradas podem aparecer neste livro. Em vez de usar um símbolo de marca registrada em cada ocorrência de um nome, logotipo ou imagem de marca registrada, usamos os nomes, logotipos e imagens apenas de forma editorial e para o benefício do proprietário da marca registrada, sem intenção de violar a marca registrada.

O uso nesta publicação de nomes comerciais, marcas registradas, marcas de serviço e termos semelhantes, mesmo que não sejam identificados como tal, não deve ser tomado como uma expressão de opinião sobre se estão ou não sujeitos a direitos de propriedade.

Embora os conselhos e as informações contidas neste livro sejam considerados verdadeiros e precisos na data de publicação, nem os autores, nem os editores, nem o editor podem aceitar qualquer responsabilidade legal por quaisquer erros ou omissões que possam ser cometidos. O editor não oferece nenhuma garantia, expressa ou implícita, com relação ao material aqui contido.

Diretor administrativo, Apress Media LLC: Welmoed Spahr

Editor de aquisições: Nikhil Karkal

Editora de Desenvolvimento: Laura Berendson

Editora Coordenadora: Divya Modi

Capa desenhada por eStudioCalamar

Imagem da capa desenhada por Freepik (www.freepik.com)

Distribuído para o comércio de livros em todo o mundo pela Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Telefone 1-800-SPRINGER, fax (201) 348-4505, pedidos por e-mail-ny@springer-sbm.com ou visite www.springeronline.com. Apress Media, LLC é uma LLC da Califórnia e o único membro (proprietário) é Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc é uma corporação de Delaware .

Para obter informações sobre traduções, envie um e-mail para rights@apress.com ou visite <http://www.apress.com/rights-permissions>.

Os títulos da Apress podem ser adquiridos em grandes quantidades para uso acadêmico, corporativo ou promocional. Versões e licenças de e-books também estão disponíveis para a maioria dos títulos. Para obter mais informações, consulte nossa página de vendas em massa de impressão e e-books em <http://www.apress.com/bulk-sales>.

Qualquer código-fonte ou outro material suplementar referenciado pelo autor neste livro está disponível aos leitores no GitHub através da página do produto do livro, localizada em www.apress.com/9781484250211.

Para obter informações mais detalhadas, visite <http://www.apress.com/source-code>.

Impresso em papel sem ácido

*Dedicado à cafeína e ao açúcar, meus companheiros
durante muitas longas noites escrevendo, e
créditos extras para minha mãe.*

Índice

Sobre o autor	1
Sobre o Revisor Técnico	1
Agradecimentos	xiii
Introdução	xv
Capítulo 1: Começando com o Flask	1
Introdução ao Flask	1
Frasco inicial	2
Componentes do frasco abordados neste livro	3
Introdução aos serviços RESTful	4
Interface Uniforme	7
Representações	8
Mensagens	9
Links entre recursos	12
Cache	13
Apátrida	13
API REST de planejamento	14
Design de API	15
Configurando o Ambiente de Desenvolvimento	16
Trabalhando com PIP	16
Escolhendo o IDE	16
Compreendendo ambientes virtuais Python	19

Índice

Configurando o Flask	24
Instalando o Flask	25
Conclusão	26
Capítulo 2: Modelagem de banco de dados no Flask	27
Introdução	27
Bancos de Dados SQL	28
Bancos de dados NoSQL	28
Principais diferenças: MySQL vs. MongoDB	29
Criando uma aplicação Flask com SQLAlchemy	30
Criando um banco de dados de autores	33
Frasco de amostra do aplicativo MongoEngine	46
Conclusão	58
Capítulo 3: Aplicação CRUD com Flask (Parte 1)	59
Autenticação do usuário	88
Conclusão	96
Capítulo 4: Aplicação CRUD com Flask (Parte 2)	97
Introdução	97
Verificação de e-mail	98
Upload de arquivo	109
Documentação da API	114
Blocos de construção da documentação da API	115
Especificação OpenAPI	116
Conclusão	134

Índice

Capítulo 5: Teste no Flask 135

Introdução 135

Configurando testes de unidade 136

Endpoints de usuário de teste de unidade 139

Cobertura de teste 155

Conclusão 157

Capítulo 6: Implantando aplicativos Flask 160

Implantando Flask com uWSGI e Nginx no Alibaba Cloud ECS 160

Implantando Flask no Gunicorn com Apache no Alibaba Cloud ECS 167

Implantando Flask no AWS Elastic Beanstalk 172

Implantando o aplicativo Flask no Heroku 176

Adicionando um perfil 177

Implantando o aplicativo Flask no Google App Engine 180

Conclusão 182

Capítulo 7: Monitorando Aplicações Flask 183

Monitoramento de aplicativos 183

Sentinela 185

Painel de monitoramento de frasco 187

Nova Relíquia 189

Serviços de bônus 192

Conclusão 194

Índice 195

Sobre o autor



Kunal Relan é pesquisador de segurança iOS e desenvolvedor full stack com mais de quatro anos de experiência em vários campos da tecnologia, incluindo segurança de rede, DevOps, infraestrutura em nuvem e desenvolvimento de aplicativos, trabalhando como consultor para start-ups em todo o mundo. Ele é um MVP do Alibaba Cloud e autor de *Penetração no iOS Testing* (Apress) e uma variedade de white papers. Kunal é um entusiasta da tecnologia e um palestrante ativo. Ele contribui regularmente para comunidades de código aberto e escreve artigos para Digital Ocean e Alibaba Techshare.

Sobre o Revisor Técnico



Saurabh Badhwar é um engenheiro de software apaixonado por construir sistemas distribuídos escaláveis. Ele trabalha principalmente para resolver desafios relacionados ao desempenho de software em grande escala e esteve envolvido na construção de soluções que ajudam outros desenvolvedores a analisar e comparar rapidamente o desempenho de seus sistemas quando executados em escala. Ele também é apaixonado por trabalhar com comunidades de código aberto e tem sido participando ativamente como contribuidor em vários domínios, que envolvem desenvolvimento, testes e envolvimento da comunidade. Saurabh também tem sido um palestrante ativo em várias conferências onde tem falado sobre o desempenho de sistemas de grande escala.

Agradecimentos

Gostaria de agradecer à Apress por me fornecer esta plataforma, sem a qual isto teria sido muito mais difícil. Gostaria também de agradecer ao Sr. Nikhil Karkal pela sua ajuda e à Sra. Divya Modi pela sua perseverança, sem as quais este teria sido um projecto clarividente.

Gostaria de mencionar a forte comunidade Python que me ajudou a entender os conceitos básicos em meus primeiros anos de programação, o que me inspirou a contribuir de volta para a comunidade com este livro.

Por último, mas certamente não menos importante, gostaria de agradecer a todas as pessoas que constantemente me lembraram dos prazos e me ajudaram a escrever este livro, especialmente à minha família e a Aparna Abhijit por me ajudarem na edição.

Introdução

Flask é uma microestrutura leve para aplicativos da web construída sobre Python, que fornece uma estrutura eficiente para a construção de aplicativos baseados na web usando a flexibilidade do Python e forte suporte da comunidade com capacidade de escala para atender milhões de usuários.

Flask tem excelente suporte comunitário, documentação e bibliotecas de apoio; ele foi desenvolvido para fornecer uma estrutura básica para desenvolvedores, dando-lhes a liberdade de construir seus aplicativos usando seu conjunto preferido de bibliotecas e ferramentas.

Este livro conduz você por diferentes estágios de um processo de desenvolvimento de aplicativo baseado em API REST usando flask, que explica os fundamentos da estrutura Flask, presumindo que os leitores entendam Python. Abordaremos integração de banco de dados, compreensão de serviços REST, APIs REST executando operações CRUD, autenticação de usuário, integrações de bibliotecas de terceiros, testes, implantação e monitoramento de aplicativos.

Ao final deste livro, você terá uma boa compreensão da estrutura Flask, REST, teste, implantação e gerenciamento de aplicativos Flask, o que abrirá portas para a compreensão do desenvolvimento da API REST.

CAPÍTULO 1

Começando com o frasco

Flask é um microframework Python licenciado por BSD baseado em Werkzeug e Jinja2. Ser um microframework não o torna menos funcional; Flask é uma estrutura muito simples, mas altamente extensível. Isso dá aos desenvolvedores o poder de escolher a configuração que desejam, facilitando assim a criação de aplicativos ou plug-ins. O Flask foi originalmente criado por Pocoo, uma equipe de desenvolvedores de código aberto em 2010, e agora é desenvolvido e mantido pelo The Pallets Project, que alimenta todos os componentes por trás do Flask. Flask é apoiado por uma comunidade de desenvolvedores ativa e útil, incluindo um canal de IRC ativo e uma lista de discussão.

Introdução ao frasco

O Flask tem dois componentes principais, Werkzeug e Jinja2. Embora Werkzeug seja responsável por fornecer roteamento, depuração e Web Server Gateway Interface (WSGI), o Flask utiliza o Jinja2 como mecanismo de modelo. Nativamente o Flask não suporta acesso a banco de dados autenticação de usuário ou qualquer outro utilitário de alto nível mas fornece suporte para integração de extensões para adicionar todas essas funcionalidades tornando o Flask uma estrutura micro mas pronta para produção para o desenvolvimento de aplicações web e Serviços. Um aplicativo Flask simples pode caber em um único arquivo Python ou pode ser modularizado para criar um aplicativo pronto para produção. A ideia por trás do Flask é construir uma boa base para todos os aplicativos, deixando todo o rest

Capítulo 1 Começando com Flask

A comunidade Flask é bastante grande e ativa, com centenas de fontes abertas extensões. A equipe principal do Flask revisa continuamente as extensões e garante que as extensões aprovadas sejam compatíveis com as versões futuras. O Flask, sendo um microframework, oferece flexibilidade aos desenvolvedores para escolher as decisões de design apropriadas ao seu projeto. Ele mantém um registro de extensões que é atualizado regularmente e mantido continuamente.

Frasco Inicial

O Flask, assim como todas as outras bibliotecas Python, pode ser instalado a partir do Python Package Index (PPI) e é realmente fácil de configurar e começar a desenvolver, e leva apenas alguns minutos para começar a usar o Flask. Para poder acompanhar este livro, você deve estar familiarizado com Python, linha de comando (ou pelo menos PIP) e MySQL.

Como prometido, o Flask é realmente fácil de começar, e apenas cinco linhas de código permitem que você comece com um aplicativo Flask mínimo.

Listagem 1-1. Aplicação Básica de Frasco

```
do frasco importar frasco
app = Frasco(__nome__)
@app.route('/')
def olá_mundo():
    retornar 'Olá, do Flask!'
se __nome__ == '__main__':
    app.run()
```

O código anterior importa a biblioteca Flask, inicia o aplicativo criando uma instância da classe Flask, declara a rota e, em seguida, define a função a ser executada quando a rota é chamada. Este código é o suficiente para iniciar seu primeiro aplicativo Flask.

O código a seguir inicia um servidor integrado muito simples, que é bom o suficiente para testes, mas provavelmente não quando você deseja entrar em produção, mas abordaremos isso nos capítulos posteriores.

Quando esta aplicação for iniciada, a rota do índice, mediante solicitação, retornará "Hello From Flask!" conforme mostrado na Figura 1-1.

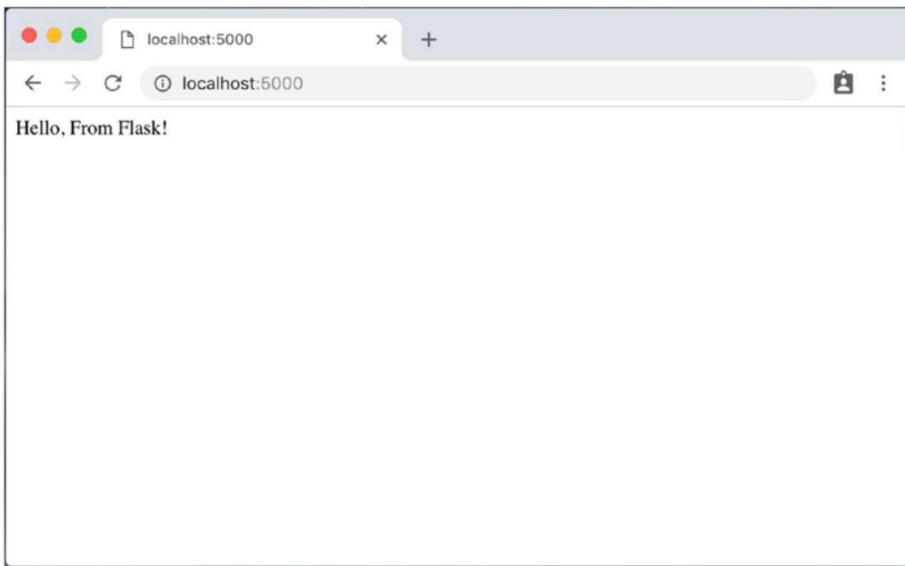


Figura 1-1. Aplicação mínima do frasco

Componentes do frasco abordados neste livro

Agora que você conheceu o Flask, discutiremos os componentes que abordaremos no desenvolvimento da API REST do Flask neste livro.

Este livro servirá como um guia prático para o desenvolvimento de API REST usando Flask, e usaremos MySQL como banco de dados backend. Como já discutido, o Flask não vem com suporte nativo para acesso ao banco de dados e, para preencher essa lacuna, usaremos uma extensão do Flask chamada Flask-SQLAlchemy que adiciona suporte para SQLAlchemy no Flask. SQLAlchemy é essencialmente

Capítulo 1 Começando com Flask

um kit de ferramentas Python SQL e mapeador relacional de objetos que fornece aos desenvolvedores todo o poder e flexibilidade do SQL.

SQLAlchemy fornece suporte completo para padrões de design de nível empresarial e foi projetado para acesso a bancos de dados de alto desempenho, mantendo a eficiência e a facilidade de uso. Construiremos um módulo de autenticação de usuário, APIs REST CRUD (Criar, Ler, Atualizar e Excluir) para criação, recuperação, manipulação e exclusão de objetos. Também integraremos um utilitário de documentação chamado Swagger para criar documentação de API, escrever testes unitários e de integração, aprender a depuração de aplicativos e, finalmente, verificar diferentes métodos de implantação e monitoramento de nossas APIs REST em plataformas de nuvem para uso em produção.

Para testes de unidade, usaremos pytest, que é um teste Python completo - pytest é fácil de escrever testes e ainda é escalonável para suportar casos de uso complexos. Também usaremos o Postman, que é uma plataforma REST API completa. O Postman fornece ferramentas de integração para cada estágio do ciclo de vida da API, tornando o desenvolvimento da API mais fácil e confiável.

A implantação e o monitoramento da API são partes críticas do desenvolvimento da API REST; O paradigma de desenvolvimento muda drasticamente quando se trata de dimensionar as APIs para casos de uso de produção e, para fins deste livro, implantaremos nossas APIs REST usando uWSGI e Nginx em um servidor Ubuntu na nuvem. Também implantaremos nossas APIs REST no Heroku, que é uma plataforma em nuvem que facilita a implantação e expansão de aplicativos Flask imediatamente.

Por último, mas não menos importante, discutiremos a depuração de erros e avisos comuns do Flask e a depuração de solicitações Nginx e verificaremos o monitoramento de aplicativos Flask, garantindo o mínimo de tempo de inatividade para uso em produção.

Introdução aos serviços RESTful

Representational State Transfer (REST) é um estilo de arquitetura de software para serviços web que fornece um padrão para comunicação de dados entre diferentes tipos de sistemas. Os serviços da Web são de padrão aberto

aplicativos da web que interagem com outros aplicativos com o objetivo de trocar dados, tornando-os uma parte essencial da arquitetura cliente-servidor em aplicativos móveis e da web modernos. Em termos simples, REST é um padrão para troca de dados pela Web para fins de interoperabilidade entre sistemas de computador. Os serviços da Web que estão em conformidade com o estilo arquitetônico REST são chamados de serviços da Web RESTful, que permitem solicitar sistemas para acessar e manipular os dados usando um conjunto uniforme e predefinido de operações sem estado.

Desde a sua criação em 2000 por Roy Feilding, a arquitetura RESTful cresceu muito e foi implementada em milhões de sistemas desde então.

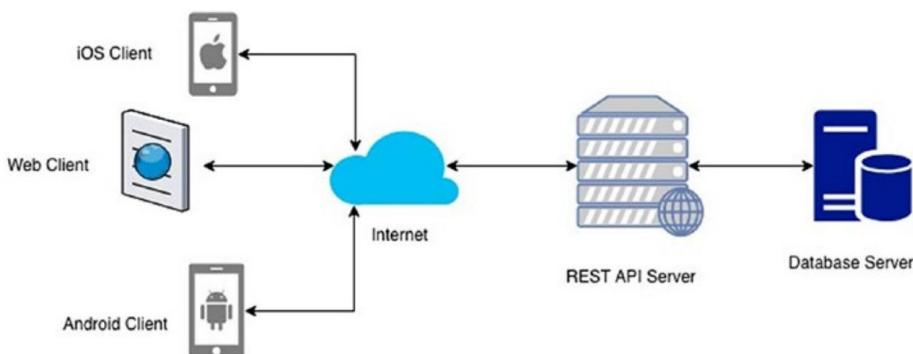
REST tornou-se agora uma das tecnologias mais importantes para aplicações baseadas na web e provavelmente crescerá ainda mais com a sua integração também em aplicações móveis e baseadas em IoT. Todas as principais linguagens de desenvolvimento possuem estruturas para a construção de serviços da web REST. Os princípios REST são o que o torna popular e muito utilizado. REST não tem estado, facilitando o uso de qualquer tipo de sistema e também possibilitando que cada solicitação seja atendida por um sistema diferente.

REST nos permite distinguir entre o cliente e o servidor, permitindo-nos implementar o cliente e o servidor de forma independente. A característica mais importante do REST é a ausência de estado, o que significa simplesmente que nem o cliente nem o servidor precisam saber o estado um do outro para ser capaz de se comunicar. Desta forma, tanto o cliente quanto o servidor podem compreender qualquer mensagem recebida sem ver a mensagem anterior.

Já que estamos falando sobre serviços web RESTful, vamos nos aprofundar nos serviços web e comparar outros padrões de serviços web.

Os serviços Web, numa definição simples, são serviços oferecidos por um dispositivo eletrônico a outro, possibilitando a comunicação através da World Wide Web. Na prática, os serviços da Web fornecem uma interface baseada na Web e orientada a recursos para um servidor de banco de dados e assim por diante, utilizado por outro cliente da Web. Os serviços da Web fornecem uma plataforma para diferentes tipos de sistemas se comunicarem entre si, usando uma solução para que os programas possam se comunicar entre si em uma linguagem que eles entendam (Figura 1-2).

Capítulo 1 Começando com Flask

**Figura 1-2.** Diagrama de arquitetura REST

SOAP (Simple Object Access Protocol) é outro protocolo de comunicação de serviço web que foi ultrapassado pelo REST nos últimos anos. Os serviços REST agora dominam a indústria, representando mais de 70% das APIs públicas, de acordo com Stormpath. Eles operam expondo uma interface consistente para acessar recursos nomeados. O SOAP, entretanto, expõe componentes da lógica da aplicação como serviços, e não como dados.

SOAP agora é um protocolo legado originalmente criado pela Microsoft e possui muitas outras restrições quando comparado ao REST. O SOAP troca dados apenas por XML e o REST fornece a capacidade de trocar dados por meio de uma variedade de formatos de dados. Os serviços RESTful são comparativamente mais rápidos e consomem menos recursos. No entanto, o SOAP ainda tem seus próprios casos de uso nos quais é um protocolo preferido em relação ao REST.

SOAP é preferido quando uma segurança robusta é essencial, pois fornece suporte para Web Services Security (WS-Security), que é uma especificação que define como as medidas de segurança são implementadas em serviços web para protegê-los contra ataques externos. Outra vantagem do SOAP sobre REST é sua lógica de repetição integrada para compensar solicitações com falha, ao contrário do REST, em que o cliente precisa lidar com solicitações com falha por meio de novas tentativas. SOAP é altamente extensível com outras tecnologias e protocolos como WS-Security, WS-endereçamento, WS-coordenação e assim por diante, o que lhe proporciona uma vantagem sobre outros protocolos de serviços da web.

Agora, depois de discutirmos brevemente os serviços da Web – REST e SOAP – vamos discutir os recursos do protocolo REST. Em geral, os serviços REST são definidos e implementados utilizando os seguintes recursos:

1. Interface uniforme
2. Representações
3. Mensagens
4. Ligações entre recursos
5. Cache
6. Apátrida

Interface Uniforme

Os serviços RESTful devem ter uma interface uniforme para acessar recursos e, como o nome sugere, a interface das APIs para o sistema deve ser uniforme em todo o sistema. Um sistema URI lógico com formas uniformes de buscar e manipular dados é o que torna o REST fácil de trabalhar. HTTP/1.1 fornece um conjunto de métodos para trabalhar em recursos baseados em substantivos; os métodos são geralmente chamados de verbos para esse propósito.

Na arquitetura REST, existe um conceito de métodos seguros e idempotentes. Métodos seguros são aqueles que não modificam recursos como um método GET ou HEAD. Um método idempotente é um método que produz o mesmo resultado, não importa quantas vezes seja executado. A Tabela 1-1 fornece uma lista de verbos HTTP comumente usados em serviços RESTful.

Capítulo 1 Começando com Flask

Tabela 1-1. Verbos HTTP comumente usados, úteis em serviços RESTful

Verbo CRUD	Operação	Seguro	Idempotente
PEGAR	Ler	Buscar um único ou vários recursos	sim Sim
POSTAGEM	criada	Insira um novo recurso	Não não
COLLOCAR	Atualizar/	Insira um novo recurso ou atualize o	Não Sim
	Criar	existente	
EXCLUIR	Excluir	Excluir um único ou vários recursos	Não Sim
OPÇÕES	LEIA	Listar operações permitidas em um recurso	Sim Sim
LEITURA DA CABEÇA		Retorna apenas cabeçalhos de resposta e nenhum corpo	sim Sim
Atualização do patch/		Atualize apenas as alterações fornecidas no	Não não
Modificar		recurso	

Representações

Os serviços RESTful concentram-se nos recursos e no fornecimento de acesso aos recursos.

Um recurso pode ser facilmente pensado como um objeto em OOP. A primeira coisa a fazer ao projetar serviços RESTful é identificar diferentes recursos e determinar a relação entre eles. Uma representação é uma explicação legível por máquina que define o estado atual de um recurso.

Uma vez identificados os recursos, as representações são o próximo curso de ação. REST nos fornece a capacidade de usar qualquer formato para representar os recursos do sistema. Ao contrário do SOAP, que nos restringe o uso de XML para representar os dados, podemos usar JSON ou XML. Normalmente, JSON é o método preferido para representar os recursos a serem chamados por clientes móveis ou web, mas XML pode ser usado para representar recursos mais complexos.

Aqui está um pequeno exemplo de representação de recursos em ambos os formatos.

Listagem 1-2. Representação XML de um recurso de livro

```
<?xml versão="1.0" codificação="UTF-8"?>
<Livro>
  <ID> 1 </ID>
  <Name> Construindo APIs REST com Flask </Name>
  <Autor> Kunal Relan </Autor>
  <Editora> Apress </Editora>
</Livro>
```

Listagem 1-3. Representação JSON de um recurso de livro

```
{
  "ID": "1",
  "Name": "Construindo APIs REST com Flask",
  "Autor": "Kunal Relan",
  "Editora": "Apress"
}
```

Em sistemas REST, você pode usar qualquer um dos métodos ou ambos os métodos dependendo do cliente solicitante para representar os dados.

Mensagens

Na arquitetura REST, que essencialmente estabeleceu uma forma de comunicação de dados no estilo cliente-servidor, as mensagens são uma chave importante. O cliente e o servidor conversam entre si por meio de mensagens nas quais o cliente envia uma mensagem ao servidor, que geralmente é chamada de solicitação, e o servidor envia uma resposta. Além dos dados reais trocados entre o cliente e o servidor na forma de solicitação e corpo de resposta, existem alguns metadados trocados pelo cliente e pelo servidor na forma de cabeçalhos de solicitação e resposta. O HTTP 1.1 define formatos de cabeçalhos de solicitação e resposta da seguinte maneira, a fim de obter uma forma uniforme de comunicação de dados entre diferentes tipos

Capítulo 1 Começando com Flask



Figura 1-3. Solicitud de amostra HTTP

Na Figura 1-4, GET é o método de solicitação, "/comments" é o caminho no servidor, "postId=1" é um parâmetro de solicitação, "HTTP/1.1" é a versão do protocolo que o cliente está solicitando, "jsonplaceholder .typicode.com" é o host do servidor e o tipo de conteúdo faz parte dos cabeçalhos da solicitação. Tudo isso combinado é o que faz uma solicitação HTTP que o servidor entende.

Em troca, o servidor HTTP envia a resposta para o pedido recursos.

```
[  
 {  
   "postId": 1,  
   "id": 1,  
   "nome": "id labore ex et quam laborum",  
   "e-mail": "Eliseo@gardner.biz",  
   "body": "laudantium enim quasi est quidem magnam voluptate ipsam  
           eos\\ntempora quo necessitatibus\\ndolor quam autem  
           quasi\\nreiciendis et nam sapiente accusantium"  
 },  
 {  
   "postId": 1,  
   "id": 2,  
   "nome": "quo vero reiciendis velit similique earum",  
   "e-mail": "Jayne_Kuhic@sydney.com",
```

```
"corpo": "est natus enim nihil est dolore omnis voluptatem  
numquam\net omnis occaecati quod ullam at\nvoluptatem  
error expedita pariatur\nnihil sint nostrum voluptatem reiciendis  
et"  
,  
{  
    "postId": 1,  
    "id": 3,  
    "nome": "odio adipisci rerum aut animi",  
    "e-mail": "Nikita@garfield.biz",  
    "body": "quia molestiae reprehenderit quasi aspernatur\naut expedita  
occaecati aliquam eveniet laudantium\nomnis quibusdam  
delectus saepe quia acusamus maiores nam est\nncum et  
ducimus et vero voluptates excepturi deleniti ratione"  
,  
{  
    "postId": 1,  
    "id": 4,  
    "nome": "alias odio sit",  
    "e-mail": "Lew@alysha.tv",  
    "body": "non et atque\noccaecati deserunt quas accusantium  
unde odit nobis qui voluptatem\nquia voluptas  
consequuntur itaque dolor\net qui rerum deleniti ut occaecati"  
,  
{  
    "postId": 1,  
    "id": 5,  
    "nome": "vero eaque aliquid doloribus et culpa",  
    "e-mail": "Hayden@althea.biz",
```

Capítulo 1 Começando com Flask

```
"body": "harum non quasi et ratione\n tempore iure ex voluptates in
ratione\n harum arquiteto fugit inventore cupiditate\n voluptates
magni quo et"
}]
```

```

HTTP/2 200
date: Mon, 14 Jan 2019 09:41:49 GMT
content-type: application/json; charset=utf-8
set-cookie: _uid=7d4uid7e00b0e0e91a59feeb7a272f83182b1547458999; expires=Tue, 14-Jan-20 09:41:49 GMT; path=/; domain=.typicode.com; HttpOnly
x-powered-by: Express
vary: Origin, Accept-Encoding
access-control-allow-credentials: true
access-control-max-age: 14400
pragma: no-cache
expires: Mon, 14 Jan 2019 13:41:49 GMT
x-content-type-options: nosniff
cf-ipcountry: US
cf-ray: 498f252a7b16a9c0-SIN
cf-cache-status: HIT
expect-ct: max-age=604800, report-uri="https://report-uri.cloudflare.com/cdn-cgi/beacon/expect-ct"
server: cloudflare
x-edge: 1.1 vugur
cf-cache-status: HIT
expect-ct: max-age=604800, report-uri="https://report-uri.cloudflare.com/cdn-cgi/beacon/expect-ct"
server: cloudflare
cf-ray: 498f252a7b16a9c0-SIN
{
  "postId": 1,
  "id": 1,
  "name": "et labore ex et quam laborum",
  "email": "ElieenGardner.Biz",
  "body": "laudantium enim quasi est quidem magnam voluptate ipsam eos\n tempora quo necessitatibus dolor quam autem quasi\n reiciendis et nam sapiente accusantium"
},
{
  "postId": 1,
  "id": 2,
  "name": "quo vero reiciendis velit similique earum",
  "email": "Jayne_Kuhic@sydney.com",
  "body": "tempore nostrum nihil est dolore omnis voluptatem numquam\n net omnis occaecati quo ullam at\n voluptatem error expedita pariatur\n nihil sint nostra\n voluptatem reiciendis et"
},
{
  "postId": 1,
  "id": 3,
  "name": "odio adipisci rerum aut animi",
  "email": "NikitaGarfield.biz",
  "body": "quia molestiae reprehenderit quasi aspernatur\n naut expedita occaecati aliquam eveniet laudantium\n omnis quibusdam delectus ssepe quia accusamus\n priores nam estinam et ducimus et vero voluptates excepturi deleniti ratione"
},
{
  "postId": 1,
  "id": 4,
  "name": "alias odio sit",
  "email": "LewellynShaw.tv",
  "body": "non et atque\\noccaecati deserunt quas accusantium unde odit nobis qui\n voluptatem\\nquia voluptas consequuntur itaque dolorin qui rerum deleniti"
}
```

Figura 1-4. Exemplo de resposta HTTP

Na figura anterior, “HTTP/2” é a versão HTTP da resposta e “200” é o código de resposta. A parte abaixo até “cf-ray” são os cabeçalhos de resposta, e a matriz de comentários de postagem abaixo de “cf-ray” é o corpo da resposta da solicitação.

Links entre recursos

Um recurso é o conceito fundamental no mundo da arquitetura REST.

Um recurso é um objeto com um tipo, dados associados e relacionamentos com outros recursos junto com um conjunto de métodos que podem ser executados nele. O recurso em uma API REST pode conter links para outros recursos que deve conduzir o fluxo do processo. Como no caso de uma página HTML em

quais os links na página inicial orientam o fluxo do usuário, os recursos na API REST devem ser capazes de conduzir o fluxo sem que o usuário conheça o mapa do processo.

Listagem 1-4. Um livro com link para comprar

```
{  
    "ID": "1",  
    "Name": "Construindo APIs REST com Flask",  
    "Autor": "Kunal Relan",  
    "Editora": "Apress",  
    "URI": "https://apress.com/us/book/123456789"  
}
```

Cache

Cache é uma técnica que armazena uma cópia de um determinado recurso e a devolve quando solicitado, economizando chamadas extras ao banco de dados e tempo de processamento. Isso pode ser feito em diferentes níveis, como cliente, servidor ou servidor proxy de middleware. O cache é uma ferramenta importante para aumentar o desempenho da API e dimensionar a aplicação; no entanto, se não for gerenciado adequadamente, o cliente receberá resultados antigos. O armazenamento em cache em APIs REST é controlado usando cabeçalhos HTTP. Os cabeçalhos de cache têm sido uma parte essencial das especificações dos cabeçalhos HTTP e uma parte importante do dimensionamento de serviços da Web com eficiência. Na especificação REST, quando um método seguro é usado em uma URL de recurso, geralmente o proxy reverso armazena em cache os resultados para usar os dados armazenados em cache quando o mesmo recurso for solicitado na próxima vez.

Apátrida

Cada solicitação do cliente para o servidor deve conter todas as informações necessárias para entender a solicitação e não pode tirar proveito de nenhum contexto armazenado no servidor.

O estado da sessão é, portanto, mantido inteiramente no cliente

—Roy Fielding

Capítulo 1 Começando com Flask

A apatridia aqui significa que cada resposta HTTP é uma entidade completa em si mesma e suficiente para servir ao propósito de fornecer informações a serem executadas sem qualquer necessidade de outra solicitação HTTP. O objetivo da apatridia é derrotar o propósito de acordo com o servidor, permitindo a flexibilidade pretendida na infraestrutura. Para facilitar o mesmo, os servidores REST fornecem informações suficientes na resposta HTTP que o cliente pode necessitar. A apatridia é uma parte essencial da capacidade de dimensionar a infraestrutura, permitindo-nos implantar vários servidores para atender milhões de usuários simultâneos, dado o fato de que não há dependência de estado de sessão do servidor. Ele também habilita o recurso de cache da infraestrutura REST, pois permite que o servidor de cache decida se deseja armazenar a solicitação em cache ou não, apenas observando a solicitação específica, independentemente de quaisquer solicitações anteriores.

API REST de planejamento

Aqui está uma lista de coisas que precisamos verificar ao planejar a criação de APIs REST:

1. Compreender o caso de uso. É muito importante saber por que você está construindo a API e quais serviços ela fornecerá.
2. Listando os recursos da API para entender tudo ações que suas APIs farão. Isso também inclui listar ações e agrupá-las para lidar com endpoints redundantes.
3. Identifique diferentes plataformas que usarão a API e fornecer suporte adequadamente.
4. Planear a longo prazo o apoio ao crescimento e a expansão da infra-estrutura.
5. Planeje a estratégia de versionamento de API garantindo continuidade o suporte é mantido em diferentes versões das APIs.

6. Planeje a estratégia de acesso à API, ou seja, autenticação, ACL e limitação.

7. Planeje a documentação e os testes da API.

8. Entenda como usar hipermídia com suas APIs.

Portanto, essas são as oito coisas importantes a serem garantidas ao planejar sua API e são realmente cruciais para o desenvolvimento de um sistema API estável e focado na produção.

Projeto de API

Agora vamos examinar o design da API. Aqui abordaremos os padrões de design de APIs REST tendo em mente a lista de coisas que acabamos de falar.

Implementação de longo prazo

A implementação de longo prazo ajuda a analisar as falhas de design antes da implementação real. Isso ajuda os desenvolvedores a escolher o tipo certo de plataformas e ferramentas para construir, garantindo que o mesmo sistema possa ser dimensionado para mais usuários posteriormente.

Desenvolvimento baseado em especificações

O desenvolvimento orientado a especificações impõe o design da API usando definição e não apenas o código, o que garante que as alterações sejam feitas na base de código enquanto o design da API está intacto. É uma boa prática usar uma ferramenta como o API Designer para entender o design da API antes do desenvolvimento, o que também permite prever as falhas. Ferramentas como swagger ou RAML permitem manter o design da API padronizado e portar a API para diferentes plataformas, se necessário.

Capítulo 1 Começando com Flask

Prototipagem

Depois que as especificações da API são implementadas, a prototipagem ajuda a visualizar a API antes do desenvolvimento real, permitindo que os desenvolvedores criem a API MOCK para ajudá-los a entender todos os aspectos potenciais da API.

Autenticação e autorização

A autenticação envolve o processo de verificação para saber quem é a pessoa, mas só não envolve dar acesso a todos os recursos ainda, e é aí que entra a autorização, que envolve autorizar uma pessoa autenticada a manter uma verificação dos recursos permitidos para acessar usando uma lista de controle de acesso (ACL).

Temos diferentes formas de autenticar e autorizar usuários, como autenticação básica, HMAC e OAuth. No entanto, OAuth 2.0 é um método preferido para o mesmo e é um protocolo padrão usado por empresas, bem como por pequenas empresas, para autenticação e autorização em suas APIs REST.

Então, esses são os principais recursos da infraestrutura REST, e vamos discutir mais sobre como o REST funciona e permite uma melhor comunicação em capítulos posteriores.

Agora, começaremos configurando nosso ambiente de desenvolvimento e entender alguns fatores-chave do desenvolvimento de aplicativos com Python.

Configurando o ambiente de desenvolvimento

Nesta parte, discutiremos a configuração do ambiente de desenvolvimento Python para um aplicativo Flask. Usaremos ambientes virtuais para um ambiente isolado separado para nossas dependências. Usaremos PIP para instalar e gerenciar nossas dependências e alguns outros utilitários úteis no processo de configuração de nosso ambiente de desenvolvimento. Neste livro, faremos tudo no macOS Mojave e Python 2.7, mas você

sinta-se à vontade para usar qualquer sistema operacional conforme sua conveniência. Portanto, se você não possui a versão correta do Python instalada em seu sistema operacional, você pode prosseguir com a instalação do Python no sistema operacional de sua escolha usando este link: www.python.org/downloads/ (Figura 1-5).

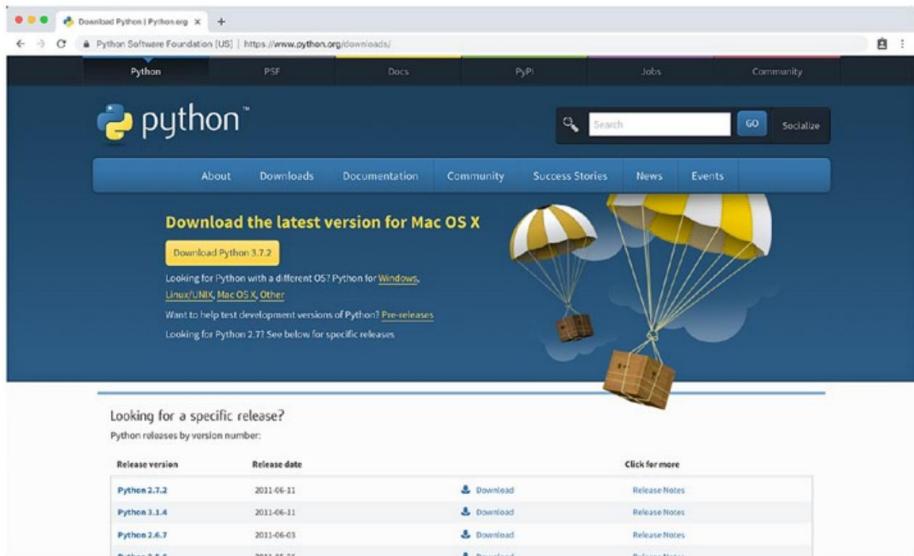


Figura 1-5. Baixar Python

Trabalhando com PIP

PIP é uma ferramenta recomendada pelo PyPi para gerenciamento de dependências de projetos. O PIP vem pré-instalado com o Python se você estiver usando o Python baixado de www.python.org.

No entanto, se você não tiver o PIP instalado em seu sistema, siga as instruções guia aqui para instalar o PIP.

Para instalar o PIP, baixe get-pip.py usando o seguinte comando em seu terminal (ou linha de comando no Windows).

```
$ curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
```

Capítulo 1 Começando com Flask

Assim que tiver o arquivo get-pip.py, instale e execute o próximo comando:

```
$ python get-pip.py
```

O comando anterior instalará PIP, setuptools (necessário para instalando distribuições fonte) e wheel.

Se você já possui o PIP, pode atualizar para a versão mais recente do pip usando o seguinte comando:

```
$ pip instalar -U pip
```

Para testar sua instalação, você deve executar o seguinte comando (Figura 1-6) em seu terminal (ou linha de comando no Windows):

```
$ python-V  
$pip-V
```



```
[Kunals-MacBook-Pro:~ kunalrelan$ pip -V  
pip 18.1 from /Library/Python/2.7/site-packages/pip (python 2.7)  
Kunals-MacBook-Pro:~ kunalrelan$ ]
```

Figura 1-6. Verificando a instalação do Python e PIP

Escolhendo o IDE

Antes de começarmos a escrever o código, precisaremos de algo para escrever. Ao longo deste livro, usaremos o Visual Studio Code, que é um IDE de código aberto e gratuito, disponível em todos os principais sistemas operacionais. O Visual Studio Code está disponível para download em www.code.visualstudio.com, e fornece um bom suporte para o desenvolvimento de aplicativos Python com muitos plug-ins úteis para facilitar o desenvolvimento. Você pode optar por usar seu próprio editor de texto ou IDE preferido para seguir este livro (Figura 1-7).

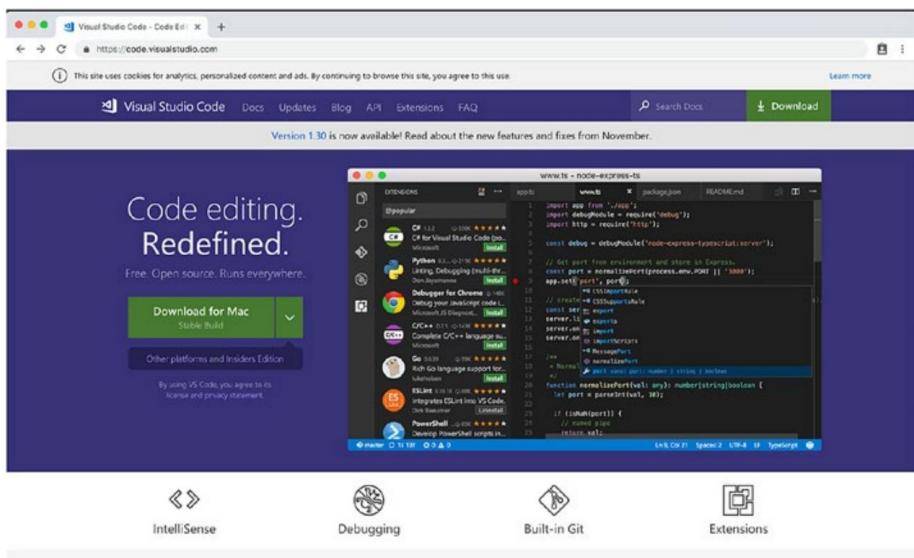


Figura 1-7. Código do Visual Studio

Depois de configurar o IDE, podemos prosseguir com a instalação e configuração o ambiente virtual.

Compreendendo os ambientes virtuais Python

Python, assim como outras linguagens de programação modernas, fornece uma grande quantidade de bibliotecas e SDKs de terceiros. Diferentes aplicativos podem precisar de várias versões específicas de módulos de terceiros, e não será possível que uma instalação do Python atenda a esses requisitos de cada aplicativo. Portanto, no mundo do Python, a solução para esse problema é o ambiente virtual, que cria uma árvore de diretórios independente separada contendo uma instalação do Python da versão necessária junto com os pacotes necessários.

Basicamente, o objetivo principal de um ambiente virtual é criar um ambiente isolado para conter uma instalação do Python e os pacotes necessários para o aplicativo. Não há limite para o número de ambientes virtuais que você pode criar e é muito fácil criá-los.

Usando ambientes virtuais

No Python 2.7, precisamos de um módulo chamado **virtualenv** que é instalado usando PIP para começar a usar ambientes virtuais Python.

Nota No Python 3, o módulo venv vem pré-enviado como parte da biblioteca padrão.

Para instalar o virtualenv, digite o seguinte comando em seu terminal (ou linha de comando no caso do Windows).

```
$ pip instalar virtualenv
```

Assim que tivermos o módulo virtualenv instalado em nosso sistema, a seguir iremos crie um novo diretório e crie um ambiente virtual nele.

Agora digite o seguinte comando para criar um novo diretório e abri-lo em seu terminal.

```
$ mkdir pyenv && cd pyenv
```

O comando anterior criará um diretório e o abrirá em seu terminal, e então usaremos o módulo virtualenv para criar um novo ambiente virtual dentro do diretório.

```
$ virtualenv venv
```

O comando anterior usará o módulo virtualenv e criará um ambiente virtual chamado venv. Você pode dar qualquer nome ao seu ambiente virtual, mas neste livro usaremos venv apenas por uma questão de uniformidade.

Assim que a execução deste comando parar, você verá um diretório chamado venv. Este diretório agora conterá seu ambiente virtual.

A estrutura de diretórios da pasta venv deve ser semelhante àquela na Figura 1-8.



```
[Kunals-MacBook-Pro:venv kunalrelan$ tree -L 2
.
└── bin
    ├── activate
    ├── activate.csh
    ├── activate.fish
    ├── activate_this.py
    ├── easy_install
    ├── easy_install-2.7
    ├── pip
    ├── pip2
    ├── pip2.7
    ├── python
    ├── python-config
    ├── python2 -> python
    ├── python2.7 -> python
    └── wheel
└── include
    └── python2.7 -> /System/Library/Frameworks/Python.framework/Versions/2.7/include/python2.7
└── lib
    └── python2.7
        └── pip-selfcheck.json
5 directories, 15 files
Kunals-MacBook-Pro:venv kunalrelan$ ]]
```

Figura 1-8. Estrutura de diretório do ambiente virtual

Aqui está o que cada pasta da estrutura contém:

1. bin: Arquivos para interagir com o ambiente virtual.
2. inclua: cabeçalhos C para compilar os pacotes Python.
3. lib: Esta pasta contém uma cópia da versão Python e todos os outros módulos de terceiros.

A seguir, há cópias ou links simbólicos para diferentes ferramentas Python para certifique-se de que todo o código e comandos Python sejam executados no ambiente atual. A parte importante aqui são os scripts de ativação na pasta bin, que configura o shell para usar o Python do ambiente virtual e os pacotes do site. Para fazer isso, você precisa ativar o ambiente virtual digitando o seguinte comando em seu terminal.

```
$ fonte venv/bin/ativar
```

Depois que este comando for executado, seu prompt do shell será prefixado pelo nome do ambiente virtual, assim como na Figura 1-9.

Capítulo 1 Começando com Flask



```
[Kunals-MacBook-Pro:pyenv kunalrelian$ source venv/bin/activate  
(venv) Kunals-MacBook-Pro:pyenv kunalrelian$ ]
```

Figura 1-9. Ativando ambiente virtual

Agora, vamos instalar o Flask em nosso ambiente virtual usando o seguinte comando:

```
$ pip instalar frasco
```

O comando anterior deve instalar o Flask em nosso ambiente virtual. Usaremos o mesmo código que usamos em nosso exemplo de aplicativo Flask.

```
$ nano app.py
```

E digite o seguinte código no editor de texto nano:

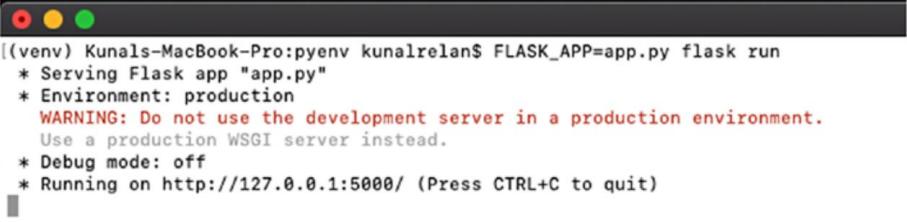
```
do frasco importar frasco
app = Frasco(__nome__)

@app.route('/')
def olá_mundo():
    retornar 'Olá, do Flask!'
```

Agora, tente executar seu app.py usando o comando python app.py.

```
$ FLASK_APP = execução do frasco app.py
```

Com o comando anterior, você poderá executar o simples Flask, e você deverá ver uma saída semelhante em seu terminal (Figura 1-10).



```
(venv) Kunals-MacBook-Pro:pyenv kunalrelan$ FLASK_APP=app.py flask run
 * Serving Flask app "app.py"
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Figura 1-10. Executando o aplicativo Flask em ambiente virtual

Agora, para desativar o ambiente virtual, é necessário executar o seguinte comando:

```
$ desativar
```

Após a execução desse comando, o prefixo (venv) do shell desaparecerá e, se você tentar executar o aplicativo novamente, ocorrerá um erro (Figura 1.11).



```
[Kunals-MacBook-Pro:pyenv kunalrelan$ FLASK_APP=app.py flask run
-bash: flask: command not found
Kunals-MacBook-Pro:pyenv kunalrelan$ ]
```

Figura 1-11. Executando o aplicativo Flask sem ambiente virtual

Agora que você entende o conceito de ambientes virtuais, podemos vá um pouco mais fundo e entenda o que está acontecendo dentro do ambiente virtual ambiente.

Compreender como funcionam os ambientes virtuais pode realmente ajudá-lo a depurar o aplicativo e compreender o ambiente de execução. Para começar, vamos dar uma olhada no executável Python com ambiente virtual ativado e desativado, para entender a diferença básica.

Vamos executar o seguinte comando com ambiente virtual ativado (Figura 1-12):

```
$ qual python
```

Capítulo 1 Começando com Flask



```
(venv) Kunals-MacBook-Pro:pyenv kunalrelan$ which python
/Users/kunalrelan/Documents/Apress/Flask-book/scripts/pyenv/venv/bin/python
(venv) Kunals-MacBook-Pro:pyenv kunalrelan$ █
```

Figura 1-12. Verificando o executável Python com ambiente virtual

Como você pode ver na figura a seguir, o shell está usando o executável Python do ambiente virtual e, se você desativar o ambiente e executar novamente o comando Python, notará que o shell agora está usando o Python do sistema (Figura 1.13).



```
[Kunals-MacBook-Pro:pyenv kunalrelan$ which python
/usr/bin/python
Kunals-MacBook-Pro:pyenv kunalrelan$ █
```

Figura 1-13. Verificando o executável Python sem ambiente virtual

Então, depois de ativar o ambiente virtual, o ambiente \$path variável é modificada para apontar para nosso ambiente virtual e, portanto, o Python em nosso ambiente virtual é usado em vez do sistema. No entanto, uma coisa importante a notar aqui é que é basicamente uma cópia ou um link simbólico para o executável Python do sistema.

Configurando o frasco

Já instalamos o Flask no módulo anterior, mas vamos recomeçar e configurar o microframework Flask.

Instalando o frasco

Com o ambiente virtual ativado, execute o seguinte comando para instalar a versão mais recente do Flask.

balão de instalação \$ pip

O comando anterior instalará o Flask em seu ambiente virtual.

No entanto, se você deseja trabalhar com o Flask mais recente antes do lançamento, instale/atualize o módulo Flask usando o branch master de seu repositório executando o seguinte comando:

```
$pip instalar -U https://github.com/pallets/flask/archive/  
mestre.tar.gz
```

Quando você instala o Flask, as seguintes distribuições são instaladas com o quadro principal:

1. Trabalho (<http://werkzeug.pocoo.org/>):

Werkzeug implementa WSGI, a interface Python padrão entre a aplicação e o servidor.

2. Jinja (<http://jinja.pocoo.org/>): Jinja é o

mecanismo de modelagem no Flask que renderiza as páginas para o aplicativo.

3. MarkupSafe (<https://pypi.org/project/MarkupSafe/>):

Markupsafe vem pré-fornecido com Jinja, que ajuda a escapar de uma entrada de usuário não confiável para escalar ataques de injeção.

4. É perigoso (<https://pythonhosted.org/itsdangerous/>)

é perigoso/: ItsDangerous é responsável por assinar dados com segurança para garantir a integridade dos dados e é usado para proteger os cookies de sessão do Flask.

Capítulo 1 Começando com Flask

5. Clique em (<http://click.pocoo.org/>): Click é um estrutura para escrever aplicativos CLI. Ele fornece o comando CLI “Flask”.

Conclusão

Depois de instalar o Flask em seu ambiente virtual, você estará pronto para passar para a próxima etapa da fase de desenvolvimento. Antes de fazermos isso, discutiremos sobre MySQL e Flask-SQLAlchemy, que é o ORM que usaremos em nosso aplicativo Flask. O banco de dados é uma parte essencial de um aplicativo REST e, no próximo capítulo, discutiremos o banco de dados MySQL e o Flask-SQLAlchemy ORM e também aprenderemos como conectar nosso aplicativo Flask ao Flask-SQLAlchemy.

CAPÍTULO 2

Modelagem de banco de dados em Flask

Este capítulo aborda um dos aspectos mais importantes do desenvolvimento de aplicações REST, ou seja, conectar e interagir com sistemas de banco de dados.

Neste capítulo, discutiremos sobre bancos de dados NoSQL e SQL, conectando-os e interagindo com eles.

Neste capítulo abordaremos os seguintes tópicos:

1. Bancos de dados NoSQL vs. SQL
2. Conectando-se com Flask-SQLAlchemy
3. Interagindo com banco de dados MySQL usando Flask-SQLAlchemy
4. Conectando com Flask-MongoEngine
5. Interagindo com MongoDB usando Flask-MongoEngine

Introdução

O Flask, sendo um microframework, fornece flexibilidade de fonte de dados para aplicativos e também fornece suporte de biblioteca para interagir com diferentes tipos de fontes de dados. Existem bibliotecas para conectar-se a SQL e NoSQL

Capítulo 2 Modelagem de banco de dados no Flask

bancos de dados baseados em Flask. Ele também fornece flexibilidade para interagir com bancos de dados usando bibliotecas de banco de dados brutas ou usando ORM (Object Relational Mapper) /ODM (Mapeador de Documentos de Objeto). Neste capítulo, discutiremos brevemente os bancos de dados baseados em NoSQL e SQL e aprenderemos como usar a camada ORM para nosso aplicativo Flask usando Flask-SQLAlchemy, após o qual usaremos a camada ODM usando Flask-MongoEngine.

A maioria dos aplicativos precisa de bancos de dados em algum momento, e MySQL e MongoDB são apenas duas das muitas ferramentas para fazer isso. A escolha do caminho certo para sua aplicação dependerá inteiramente dos dados que você irá armazenar. Se seus conjuntos de dados em tabelas estiverem relacionados entre si, os bancos de dados SQL são a melhor opção ou os bancos de dados NoSQL também podem servir a esse propósito.

Agora, vamos dar uma breve olhada nos bancos de dados SQL vs. NoSQL.

Bancos de dados SQL

Os bancos de dados SQL usam Structured Query Language (SQL) para manipulação e definição de dados. SQL é uma opção versátil, amplamente utilizada e aceita, o que o torna uma ótima opção para armazenamento de dados. Os sistemas SQL funcionam muito bem quando os dados em uso precisam ser relacionais e o esquema é predefinido. No entanto, um esquema predefinido também serve como uma desvantagem, pois exige que todo o conjunto de dados siga a mesma estrutura, o que pode ser difícil em algumas situações. Os bancos de dados SQL armazenam dados em formas de tabelas compostas de linhas e colunas e são escalonáveis verticalmente.

Bancos de dados NoSQL

Os bancos de dados NoSQL têm um esquema dinâmico para dados não estruturados e armazenam dados de diferentes maneiras, desde baseados em colunas (Apache Cassandra), baseados em documentos (MongoDB) e baseados em gráficos (Neo4J) ou como armazenamento de valor-chave (Redis). Isso proporciona flexibilidade para armazenar dados sem uma estrutura predefinida e versatilidade para adicionar campos à estrutura de dados em trânsito. Ser sem esquema é a principal distinção dos bancos de dados NoSQL,

e também os torna mais adequados para sistemas distribuídos. Ao contrário dos bancos de dados SQL, os bancos de dados NoSQL são escalonáveis horizontalmente.

Agora que explicamos brevemente os bancos de dados SQL e NoSQL, passaremos para as diferenças funcionais entre MySQL e MongoDB, já que esses são os dois mecanismos de banco de dados que veremos neste capítulo.

Principais diferenças: MySQL vs. MongoDB

Conforme discutido anteriormente, MySQL é um banco de dados baseado em SQL que armazena dados em tabelas com colunas e linhas e só funciona em dados estruturados. O MongoDB, por outro lado, pode lidar com dados não estruturados e armazenar documentos semelhantes a JSON em vez de tabelas e usar a linguagem de consulta MongoDB para se comunicar com o banco de dados. MySQL é um banco de dados extremamente estabelecido com uma enorme comunidade e grande estabilidade, e MongoDB é uma tecnologia relativamente nova com uma comunidade crescente e é desenvolvida pela MongoDB Inc. MySQL é verticalmente escalável em que a carga no servidor único pode ser aumentada atualizando a RAM, SSD ou CPU, enquanto no caso do MongoDB, que é escalonável horizontalmente, ele precisa compartilhar e adicionar mais servidores para aumentar a carga do servidor. O MongoDB é a escolha preferida para altas cargas de gravação e grandes conjuntos de dados, e o MySQL é perfeito para aplicativos que dependem muito de transações de várias linhas, como sistemas de contabilidade. MongoDB é uma ótima opção para aplicações com estrutura dinâmica e alta carga de dados, como uma aplicação analítica em tempo real ou um sistema de gerenciamento de conteúdo.

Flask fornece suporte para interação com MySQL e MongoDB. Existem vários drivers nativos, bem como ORM/ODM para comunicação com o banco de dados. Flask-MySQL é uma extensão Flask que permite conexão nativa ao MySQL; Flask-PyMongo é uma extensão nativa para trabalhar com MongoDB no Flask e também é recomendada pelo MongoDB. Flask-MongoEngine é uma extensão Flask, ODM para Flask funcionar com MongoDB. Flask-SQLAlchemy é uma camada ORM para aplicativos Flask se conectarem ao MySQL.

Capítulo 2 Modelagem de banco de dados no Flask

A seguir, discutiremos sobre Flask-SQLAlchemy e Flask-MongoEngine e criar aplicativos Flask CRUD usando-os.

Criando um aplicativo Flask com SQLAlchemy

Flask-SQLAlchemy é uma extensão para flask que adiciona suporte para SQLAlchemy ao aplicativo. SQLAlchemy é um kit de ferramentas Python e mapeador relacional de objetos que fornece acesso ao banco de dados SQL usando Python. SQLAlchemy vem com padrões de persistência de nível empresarial e acesso de banco de dados eficiente e de alto desempenho. Flask-SQLAlchemy fornece suporte para os seguintes mecanismos de banco de dados baseados em SQL, desde que o driver DBAPI apropriado esteja instalado:

- PostgreSQL
- MySQL
- Oráculo
- SQLite
- Microsoft SQL Server
- SyBase Firebird

Usaremos MySQL como mecanismo de banco de dados em nosso aplicativo, então vamos começar a instalar o SQLAlchemy e a configurar nosso aplicativo.

Vamos criar um novo diretório chamado flask-MySQL, criar um ambiente virtual e então instalar o flask-sqlalchemy.

```
$ mkdir frasco-mysql && cd frasco-mysql
```

Agora, crie um ambiente virtual dentro do diretório usando o seguinte comando:

```
$ virtualenv venv
```

Conforme discutido anteriormente, podemos ativar o ambiente virtual usando o seguinte comando:

```
$ fonte venv/bin/ativar
```

Assim que o ambiente virtual estiver ativado, vamos instalar o flask-sqlalchemy.

Flask e Flask-SQLAlchemy podem ser instalados usando PIP com o seguinte comando.

```
(venv)$ pip instalar balão flask-sqlalchemy
```

Além do SQLite, todos os outros mecanismos de banco de dados precisam de bibliotecas separadas para ser instalado junto com o Flask-SQLAlchemy para que funcione. SQLAlchemy usa MySQL-Python como DBAPI padrão para conexão com MySQL.

Agora, vamos instalar o PyMySQL para habilitar a conexão MySQL com Flask-SQLAlchemy.

```
(venv) $ pip instalar pymysql
```

Agora, devemos ter tudo o que precisamos para criar nosso frasco de amostra. Aplicativo MySQL com.

Vamos começar criando app.py que conterá o código da nossa aplicação. Após criar o arquivo, iniciaremos a aplicação Flask.

```
do frasco importar frasco
```

```
de flask_sqlalchemy importar SQLAlchemy
```

```
app = Frasco(__nome__)
app.config['SQLALCHEMY_DATABASE_URI'] =
'mysql+pymysql://<mysql_username>:<mysql_password>@<mysql_
host>:<porta_mysql>/<db_mysql>'
banco de dados = SQLAlchemy(aplicativo)

se __nome__ == "__main__":
    app.run(deputação=Verdadeiro)
```

Capítulo 2 Modelagem de banco de dados no Flask

Aqui, importamos a estrutura Flask e o Flask-SQLAlchemy e, em seguida, iniciamos uma instância do Flask. Depois disso, configuramos o URI do banco de dados SQLAlchemy para usar nosso URI de banco de dados MySQL e, em seguida, criamos um objeto SQLAlchemy denominado db, que tratará de nossas atividades relacionadas ao ORM.

Agora, se você estiver usando MySQL, certifique-se de fornecer strings de conexão de um servidor MySQL em execução e de que o nome do banco de dados fornecido exista.

Nota Use variáveis de ambiente para fornecer cadeias de conexão de banco de dados em seus aplicativos.

Certifique-se de ter um servidor MySQL em execução para seguir esta aplicação. No entanto, você também pode usar o SQLite em seu lugar, fornecendo os detalhes de configuração do SQLite no URI do banco de dados SQLAlchemy, que deve ser semelhante a este:

```
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///tmp/ <db_name>.db'
```

Para executar o aplicativo, você precisa executar o seguinte código em seu terminal:

```
(venv) $ python app.py
```

E se não houver erros, você deverá ver uma saída semelhante em seu terminal:

```
(venv) $ python app.py
```

- * Servindo o aplicativo Flask "app" (carregamento lento)

- * Meio Ambiente: produção

AVISO: Não utilize o servidor de desenvolvimento em um ambiente de produção.

Use um servidor WSGI de produção.

* Modo de depuração: ativado

* Executando em http://127.0.0.1:5000/ (pressione CTRL+C para sair)

- * Reiniciando com estatística
- * O depurador está ativo!
- * PIN do depurador: 779-301-240

Criando um banco de dados de autores

Agora criaremos um aplicativo de banco de dados de autor que fornecerá APIs RESTful CRUD. Todos os autores serão armazenados em uma tabela intitulada “autores”.

Após o objeto db declarado, adicione as seguintes linhas de código para declarar uma classe como Autores que conterá o esquema da tabela autores:

Autor da classe (db.Model):

```
id = db.Column(db.Integer, chave_primária=True)
nome = db.Column(db.String(20))
especialização = db.Column(db.String(50))

def __init__(self, nome, especialização):
    self.name = nome
    self.specialisation = especialização
def __repr__(auto):
    retornar '<Produto %d>' % self.id

db.create_all()
```

Com este código, criamos um modelo intitulado “Autores” que possui três campos – ID, nome e especialização. Nome e especialização são strings, mas ID é um número inteiro autogerado e incrementado automaticamente que servirá como chave primária. Observe a última linha “db.create_all()” que instrui o aplicativo a criar todas as tabelas e bancos de dados especificados no aplicativo.

Para servir a resposta JSON de nossa API usando os dados retornados pelo SQLAlchemy, precisamos de outra biblioteca chamada marshmallow, que é um complemento do SQLAlchemy para serializar objetos de dados retornados pelo SQLAlchemy para JSON.

```
(venv)$ pip instalar frasco-marshmallow
```

Capítulo 2 Modelagem de banco de dados no Flask

O comando a seguir instalará a versão Flask do marshmallow em nosso aplicativo e definiremos nosso esquema de saída do modelo Authors usando marshmallow.

Adicione as seguintes linhas na parte superior, abaixo das outras importações no arquivo do seu aplicativo para importar o marshmallow.

```
de marshmallow_sqlalchemy importar ModelSchema  
de campos de importação de marshmallow
```

Após db.create_all(), defina seu esquema de saída usando o seguinte código:

```
classe AutorSchema (ModelSchema):
```

```
    classe Meta(ModelSchema.Meta):  
        modelo = Autores  
  
        sqla_session=db.sessão  
  
        id = campos.Number(dump_only=True)  
        nome = campos.String(required=True)  
        especialização = campos.String(required=True)
```

O código anterior mapeia o atributo variável para objetos de campo e, em Meta, definimos o modelo para se relacionar ao nosso esquema. Portanto, isso deve nos ajudar a retornar JSON do SQLAlchemy.

Depois de configurar nosso modelo e esquema de retorno, podemos começar a criar nossos pontos finais. Vamos criar nosso primeiro endpoint GET /authors para retornar todos os autores registrados. Este endpoint consultará todos os objetos no modelo Autores e os retornará em JSON ao usuário. Mas antes de escrevermos o endpoint, edite a primeira linha de importação para a seguinte para importar jsonify, make_response e request do Flask.

```
do frasco importar Flask, request, jsonify, make_response
```

E após o AuthorSchema, escreva seu primeiro endpoint /authors com o seguinte código:

```
@app.route('/autores', métodos = ['GET'])

índice de definição():

    get_authors = Autores.query.all()
    autor_schema = AuthorSchema(muitos=True)
    autores, erro = autor_schema.dump(get_authors)
    return make_response(jsonify({"autores": autores}))
```

Neste método, estamos buscando todos os autores no banco de dados, despejando-os no AuthorSchema e retornando o resultado em JSON.

Se você iniciar o aplicativo e atingir o endpoint agora, ele retornará um array vazio, já que ainda não adicionamos nada no banco de dados, mas vamos tentar o endpoint.

Execute o aplicativo usando Python app.py e consulte o endpoint usando seu cliente REST preferido. Usarei o Postman para solicitar o endpoint.

Então basta abrir seu Postman e GET <http://localhost:5000/authors> para consultar o endpoint (Figura 2-1).

Capítulo 2 Modelagem de banco de dados no Flask

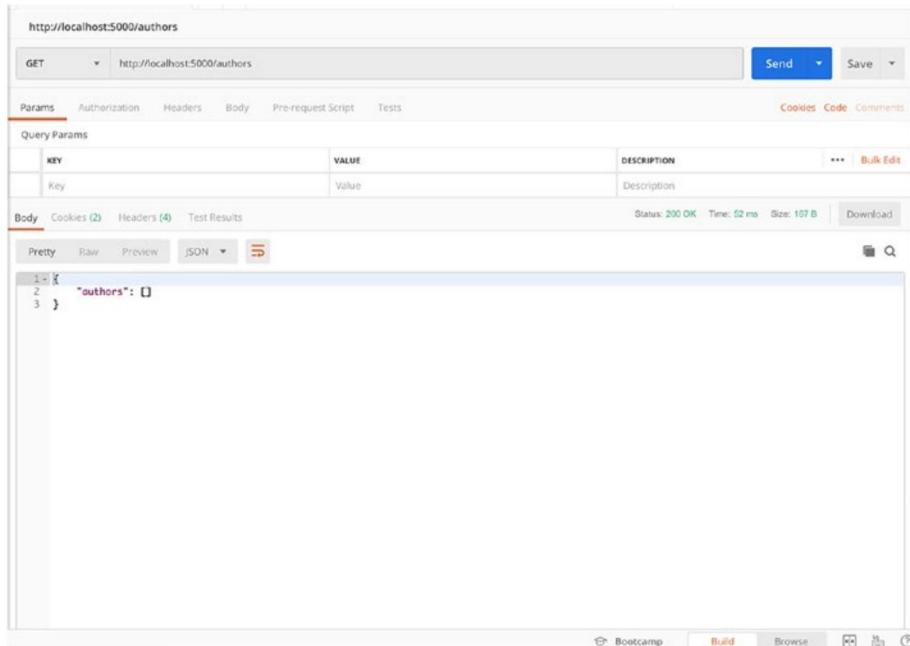


Figura 2-1. GET / resposta dos autores

Você deverá ver um resultado semelhante em seu cliente Postman. Agora vamos criar o endpoint POST para adicionar autores ao nosso banco de dados.

Podemos adicionar um objeto à tabela criando diretamente uma classe Authors em nosso método ou criando um classMethod para criar um novo objeto na classe Authors e então chamando o método em nosso endpoint. Vamos adicionar a classe Method na classe Authors para criar um novo objeto.

Adicione o seguinte código na classe Authors após a definição dos campos:

```
def criar(auto):
    db.session.add(self)
    db.sessão.commit()
    retornar a si mesmo
```

O método anterior cria um novo objeto com os dados e depois retorna o objeto criado. Agora sua classe Autores deve ficar assim:

Autores de classe (db.Model):

```
id = db.Column(db.Integer, primary_key=True) nome =
db.Column(db.String(20)) especialização
= db.Column(db.String(50))

def criar(self):
    db.session.add(self)
    db.session.commit()
    return self

def __init__(self, nome, especialização): self.name =
    nome
    self.specialisation = especialização def
__repr__(self): return
'<Autor %d>' % self.id
```

Agora criaremos nosso endpoint de autores POST e escreveremos o seguinte código após o endpoint GET:

```
@app.route('/autores', métodos = ['POST']) def
create_author(): dados =
    request.get_json() autor_schema
    = AuthorsSchema() autor, erro =
    autor_schema.load(dados) resultado =
    autor_schema.dump (autor.create()).data return
    make_response(jsonify({"autor": autores}),201)
```

O método anterior pegará os dados da solicitação JSON, carregará os dados no esquema marshmallow e, em seguida, chamará o método create que criamos na classe Authors, que retornará o objeto criado com o código de status 201.

Então, vamos solicitar o endpoint POST com dados de amostra e verificar a resposta. Vamos abrir Postman e POST/authors com corpo de solicitação JSON.

Capítulo 2 Modelagem de banco de dados no Flask

Precisamos adicionar campos de nome e especialização em nosso corpo para criar o objeto. Nossa exemplo de corpo de solicitação deve ser parecido com o seguinte:

```
{
    "nome": "Kunal Relan",
    "especialização": "Python"
}
```

Assim que solicitarmos o endpoint, obteremos o objeto Autor em resposta com nosso autor recém-criado. Observe que, neste caso, o código de status de retorno é 201, que é o código de status de um novo objeto (Figura 2-2).

The screenshot shows a Postman interface with the following details:

- URL:** http://localhost:5000/authors
- Method:** POST
- Body:** JSON (application/json)


```
1 - {
2   "name": "Kunal Relan",
3   "specialisation": "Python"
4 }
```
- Response Status:** 201 OK
- Response Body (Pretty JSON):**

```
1 - {
2   "author": {
3     "id": 1,
4     "name": "Kunal Relan",
5     "specialisation": "Python"
6   }
7 }
```

Figura 2-2. Ponto de extremidade POST /autores

Então agora, se solicitarmos nosso endpoint GET /authors, obteremos nosso autor recém-criado na resposta.

Visite novamente a guia GET /authors no Postman e clique na solicitação novamente; esse Neste momento você deverá obter uma série de autores com nosso autor recém-criado (Figura 2-3).

The screenshot shows the Postman interface with the following details:

- Request URL:** http://localhost:5000/authors
- Method:** GET
- Headers:** (1)
- Body:** (0)
- Query Params:**

KEY	VALUE	DESCRIPTION
Key	Value	Description
- Tests:**
- Responses:**
 - Pretty
 - Raw
 - Preview
 - JSON
- Status:** 200 OK | Time: 153 ms | Size: 264 B | Download

Figura 2-3. GET todos os autores com novo objeto

Até agora, criamos endpoints para registrar novos autores e buscar uma lista de autores. Em seguida, criaremos um endpoint para retornar o autor usando o ID do autor e, em seguida, atualizaremos o endpoint para atualizar os detalhes do autor usando o ID do autor e o último endpoint para excluir um autor usando o ID do autor.

Para GET autor por ID, teremos uma rota como /authors/<id> que pegará o ID do autor do parâmetro de solicitação e encontrará o autor correspondente.

Adicione o seguinte código para o endpoint GET autor por ID abaixo do seu OBTERNA a rota de todos os autores.

```
@app.route('/autores/<id>', métodos = ['GET'])
def get_author_by_id(id):
```

Capítulo 2 Modelagem de banco de dados no Flask

```
get_author = Autores.query.get(id)
autor_schema = AuthorsSchema()
autor, erro = autor_schema.dump(get_autor)
return make_response(jsonify({"autor": autor}))
```

Em seguida, precisamos testar esse endpoint e solicitaremos o autor com ID

1, como vemos na resposta anterior da API GET todos os autores, então vamos abrir o Postman novamente e solicitar /authors/1 em nosso servidor de aplicativos para verificar a resposta.

The screenshot shows the Postman interface with a GET request to `http://localhost:5000/authors/1`. The response body is a JSON object:

```
1. {
2.     "author": {
3.         "id": 1,
4.         "name": "Kunal Relan",
5.         "specialisation": "Python"
6.     }
7. }
```

Figura 2-4. GET autor por endpoint de ID

Como você pode ver na captura de tela anterior, estamos retornando um objeto com uma chave autor contendo o objeto autor com ID 1. Agora você pode adicionar mais autores usando o endpoint POST e buscá-los usando o ID retornado.

A seguir, precisamos criar um endpoint para atualizar o nome do autor ou especialização e, para atualizar qualquer objeto, usaremos o verbo HTTP PUT conforme discutimos na seção “Introdução aos serviços RESTful”. Este ponto final será semelhante ao ponto final GET autores por ID, mas usará o verbo PUT em vez do verbo GET.

Aqui está o código para o endpoint PUT para atualizar um objeto de autor

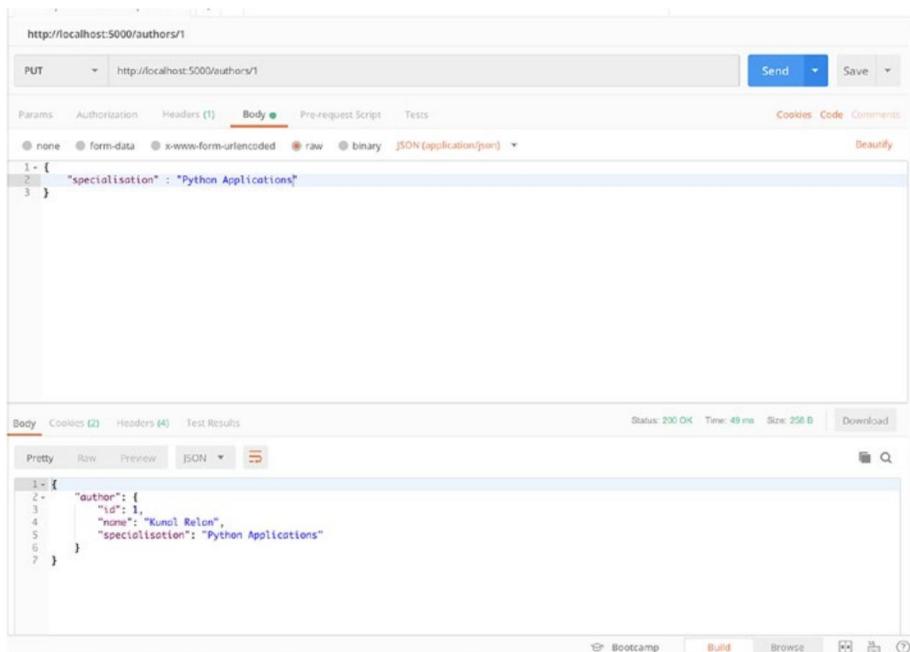
```
@app.route('/autores/<id>', métodos = ['PUT'])
def update_author_by_id(id):
    dados=solicitação.get_json()
    get_author = Autores.query.get(id)
    if data.get('especialização'):
        get_author.specialisation = dados['especialização']
    if data.get('nome'):
        get_author.name = dados['nome']
    db.session.add(get_author)
    db.sessão.commit()
    autor_schema = AuthorsSchema(only=['id', 'nome',
        'especialização'])
    autor, erro = autor_schema.dump(get_autor)
    return make_response(jsonify({"autor": autor}))
```

Então, vamos testar nosso endpoint PUT e alterar a especialização de ID do autor 1.

COLOCAREmos o seguinte corpo JSON para atualizar a especialização do autor.

```
{
    "especialização": "Aplicativos Python"
}
```

Capítulo 2 Modelagem de banco de dados no Flask

**Figura 2-5. ATUALIZAR autor por endpoint de ID**

Como você pode ver na Figura 2-5, atualizamos o autor com ID 1 e agora a especialização foi atualizada para “Aplicativos Python”.

Agora, o último endpoint para remover um autor do banco de dados. Adicione o código a seguir para adicionar um endpoint de exclusão que será semelhante a get author by Endpoint de ID, mas usará o verbo DELETE e retornará o código de status 204 sem conteúdo.

```

@app.route('/autores/<id>', métodos = ['DELETE'])
def delete_author_by_id(id):
    get_author = Autores.query.get(id)
    db.session.delete(get_author)
    db.sessão.commit()
    retornar make_response("",204)
  
```

E agora solicitaremos o endpoint delete para remover nosso autor com ID 1 (Figura 2-6).

The screenshot shows the Postman application interface. At the top, the URL is set to `http://localhost:5000/authors/1`. The method dropdown is set to `DELETE`. Below the URL, there are tabs for `Params`, `Authorization`, `Headers (1)`, `Body`, `Pre-request Script`, and `Tests`. The `Body` tab is selected, showing a table with one row and three columns: `KEY`, `VALUE`, and `DESCRIPTION`. The `KEY` column contains the value `Key`, the `VALUE` column contains the value `Value`, and the `DESCRIPTION` column contains the value `Description`. Below the table, there are buttons for `Pretty`, `Raw`, `Preview`, and `HTML`. The status bar at the bottom right indicates `Status: 204 NO CONTENT`, `Time: 19 ms`, and `Size: 162 B`.

Figura 2-6. EXCLUIR autor por ID

E agora, se você solicitar o endpoint GET todos os autores, ele retornará um matriz vazia.

Agora seu app.py deve ter o seguinte código:

```
de flask import Flask, request, jsonify, make_response de flask_sqlalchemy  
import SQLAlchemy de marshmallow_sqlalchemy  
import ModelSchema de campos de importação de marshmallow
```

```
app = Flask(__name__)  
app.config["SQLALCHEMY_DATABASE_URI"] = 'mysql+pymysql://<nome de usuário  
mysql>:<mysql_password>@<mysql_host>:<mysql_port>/<mysql_db>'
```

Capítulo 2 Modelagem de banco de dados no Flask

```
banco de dados = SQLAlchemy(aplicativo)
```

Autores de classe (db.Model):

```
id = db.Column(db.Integer, primária_key=True) nome  
= db.Column(db.String(20))  
especialização = db.Column(db.String(50))
```

```
def criar(self):  
    db.session.add(self)  
    db.session.commit()  
    retornar self
```

```
def __init__(self, nome, especialização): self.name  
    = nome  
    self.specialisation = especialização
```

```
def __repr__(self):  
    return '<Autor %d>' % self.id
```

```
db.create_all()
```

classe AuthorsSchema (ModelSchema):

```
classe Meta (ModelSchema.Meta):  
    modelo = Autores  
    sqla_session=db.sessão  
  
id = campos.Number(dump_only=True)  
nome = campos.String(required=True)  
especialização = campos.String(required=True)
```

```
@app.route('/authors', methods = ['GET']) def  
index():  
    get_authors = Authors.query.all()  
    autor_schema = AuthorsSchema(many=True)  
    autores, erro = autor_schema.dump(get_authors) return  
    make_response(jsonify({"autores": autores}))
```

```
@app.route('/authors/<id>', methods = ['GET']) def
get_author_by_id(id): get_author
    = Authors.query.get(id) author_schema =
    AuthorsSchema() autor, erro =
    autor_schema.dump (get_author) return
    make_response(jsonify({"autor": autor}))

@app.route('/authors/<id>', methods = ['PUT']) def
update_author_by_id(id): data =
    request.get_json() get_author
    = Authors.query.get(id) if
    data.get( 'especialização'):
        get_author.specialisation = dados['especialização'] if
    data.get('nome'):
        get_author.name = dados['nome']
    db.session.add(get_author)
    db.session.commit()
    autor_schema = AuthorsSchema(only=['id', 'nome',
    'especialização'])
    autor, erro = autor_schema.dump(get_author) return
    make_response(jsonify({"autor": autor}))

@app.route('/authors/<id>', methods = ['DELETE']) def
delete_author_by_id(id): get_author
    = Authors.query.get(id)
    db.session.delete(get_author)
    db.session. commit()
    retorna make_response("",204)

@app.route('/autores', métodos = ['POST']) def
create_author(): dados
    = request.get_json()
    autor_schema = AuthorsSchema()
```

Capítulo 2 Modelagem de banco de dados no Flask

```

autor, erro = autor_schema.load(dados)
resultado = autor_schema.dump(autor.create()).data
return make_response(jsonify({"autor": resultado}),200)

se __nome__ == "__main__":
    app.run(depuracao=Verdadeiro)

```

Então, agora criamos e testamos nosso exemplo Flask-MySQL CRUD aplicativo. Abordaremos relacionamentos de objetos complexos usando Flask-SQLAlchemy nos capítulos posteriores e, a seguir, criaremos um aplicativo Flask CRUD semelhante usando MongoEngine.

Exemplo de aplicativo Flask MongoEngine

MongoDB, como discutimos, é um poderoso banco de dados NoSQL baseado em documentos. Ele usa uma estrutura de esquema de documento semelhante a JSON e é altamente escalonável. Neste exemplo, criaremos novamente um aplicativo CRUD de banco de dados de Autores, mas desta vez usaremos MongoEngine em vez de SQLAlchemy. MongoEngine adiciona suporte MongoDB para Flask e é bastante semelhante ao SQLAlchemy, mas carece de alguns recursos devido ao fato de que MongoDB ainda não é amplamente usado com Flask.

Vamos começar a configurar nosso projeto para o aplicativo flask-mongodb. Assim como da última vez, crie um novo diretório flask-mongodb e inicie um novo ambiente virtual nele.

```
$ mkdir frasco-mongodb && cd frasco-mongodb
```

Após criar o diretório, vamos gerar nosso ambiente virtual e ative-o.

```

$ virtualenv venv
$ fonte venv/bin/ativar

```

Agora vamos instalar as dependências do nosso projeto usando PIP.

```
(venv) $ pip install balão
```

Precisaremos do Flask-MongoEngine e do Flask-marshmallow, então vamos instalar eles também.

```
(venv) $ pip install flask-mongoengine (venv) $ pip  
install flask-marshmallow
```

Depois de instalarmos as dependências, podemos criar nosso aplicativo.py e comece a escrever o código.

Portanto, o código a seguir é o esqueleto do aplicativo onde estão importados o flask, criam uma instância do aplicativo e, em seguida, importam o MongoEngine para criar uma instância de banco de dados.

```
do flask importar Flask, request, jsonify, make_response do flask_mongoengine  
importar MongoEngine do marshmallow importar Esquema,  
campos, post_load do bson importar ObjectId
```

```
app = Flask(__name__)  
app.config['MONGODB_DB'] = 'autores' db =  
MongoEngine(app)  
  
Esquema.TYPE_MAPPING[ObjectId] = campos.String  
  
se __name__ == "__main__":  
    app.run(debug=True)
```

Aqui, TYPE_MAPPING ajuda o marshmallow a entender o tipo ObjectId enquanto serializa e desserializa os dados.

Nota Não precisamos de db.create_all() aqui, pois o MongoDB irá criá-lo dinamicamente, durante a primeira vez que você salvar o valor em sua coleção.

Capítulo 2 Modelagem de banco de dados no Flask

Se você executar o aplicativo agora, seu servidor deverá iniciar, mas não terá nada para processar, apenas crie a instância do banco de dados e faça a conexão. A seguir, vamos criar um modelo de autor usando MongoEngine.

O código para criar o modelo do autor é bastante simples neste caso e se parece com isto:

Autores de classe (db.Document):

```
nome = db.StringField()  
especialização = db.StringField()
```

Vamos agora criar o esquema marshmallow que precisaremos descartar nossos objetos db em JSON serializado.

classe AutoresSchema(Esquema):

```
nome = campos.String(required=True)  
especialização = campos.String(required=True)
```

O código anterior nos permite criar o esquema que usaremos para mapear nosso objeto db para marshmallow. Observe que aqui não estamos usando marshmallow-sqlalchemy que possui uma camada extra de suporte para SQLAlchemy e o código parece ligeiramente alterado devido a isso aqui.

Agora podemos escrever nosso endpoint GET para buscar todos os autores de nosso banco de dados.

```
@app.route('/autores', métodos = ['GET'])
```

índice de definição():

```
get_authors = Autores.objects.all()  
author_schema = AuthorsSchema(muitos=True,only=['id','nome',  
'especialização'])  
autores, erro = autor_schema.dump(get_authors)  
return make_response(jsonify({"autores": autores}))
```

Nota MongoEngine retorna o ObjectId exclusivo no campo “id” que é gerado automaticamente e, portanto, não especificado no esquema.

Agora, vamos iniciar o aplicativo novamente usando o seguinte comando.

```
(venv) $ python app.py
```

Se não houver erros, você deverá ver a seguinte saída e seu aplicativo deverá estar instalado e funcionando.

```
(venv) $ python app.py *
```

```
Serving Flask app "app" (carregamento lento)
```

```
* Meio Ambiente: produção
```

```
AVISO: Não utilize o servidor de desenvolvimento em um ambiente de produção.
```

Use um servidor WSGI de produção.

```
* Modo de depuração:
```

```
ativado * Executando em http://127.0.0.1:5000/ (pressione CTRL+C para sair)
```

```
* Reiniciando com stat * O
```

```
depurador está ativo!
```

```
* PIN do depurador: 779-301-240
```

Capítulo 2 Modelagem de banco de dados no Flask

The screenshot shows the Postman application interface. At the top, the URL is set to `http://localhost:5000/authors`. Below the URL, there's a dropdown menu showing "GET". On the right side of the header, there are "Send" and "Save" buttons. Underneath the URL, there are tabs for "Params", "Authorization", "Headers (1)", "Body (1)", "Pre-request Script", and "Tests". The "Body" tab is selected, showing a table with one row. The table has columns for "KEY", "VALUE", and "DESCRIPTION". A single entry is present: "Key" is "authors", "Value" is an empty array, and "Description" is "Description". Below the table, there are buttons for "Pretty", "Raw", "Preview", and "JSON". The "JSON" button is highlighted. The main content area displays the JSON response:

```
1 - [  
2   "authors": []  
3 ]
```

. Above this content area, status information is shown: "Status: 200 OK", "Time: 8 ms", and "Size: 167 B". On the far right, there are "Download" and "Bulk Edit" buttons. At the bottom of the window, there are several icons: "Boatcamp", "Build", "Browse", and others.

Figura 2-7. Solicitando GET /autores

Agora que nosso endpoint GET está funcionando (Figura 2-7), vamos criar um endpoint POST /autores para registrar autores no banco de dados.

```
@app.route('/autores', métodos = ['POST']) def
create_author(): dados =
    request.get_json() autor =
        Autores(nome=dados['nome'],especialização=dados ['especialização'])
        autor.save() autor_schema
    =
        AuthorsSchema(only=['nome', 'especialização']) autores,
        erro =
        autor_schema.dump(autor) return
make_response(jsonify({"autor": autores}),201)
```

O código anterior coloca os dados JSON da solicitação na variável data, cria um objeto da classe Authors e invoca o método save() nele. Em seguida, ele cria um esquema usando AuthorsSchema e despeja o novo objeto para devolvê-lo ao usuário, confirmando que o usuário foi criado com um código de status 201.

Agora execute novamente o aplicativo e solicite o endpoint POST com amostra de detalhes do autor para registro.

Usaremos os mesmos dados JSON para postar neste aplicativo, como fizemos no outro aplicativo.

```
{
    "nome": "Kunal Relan",
    "especialização": "Python"
}
```

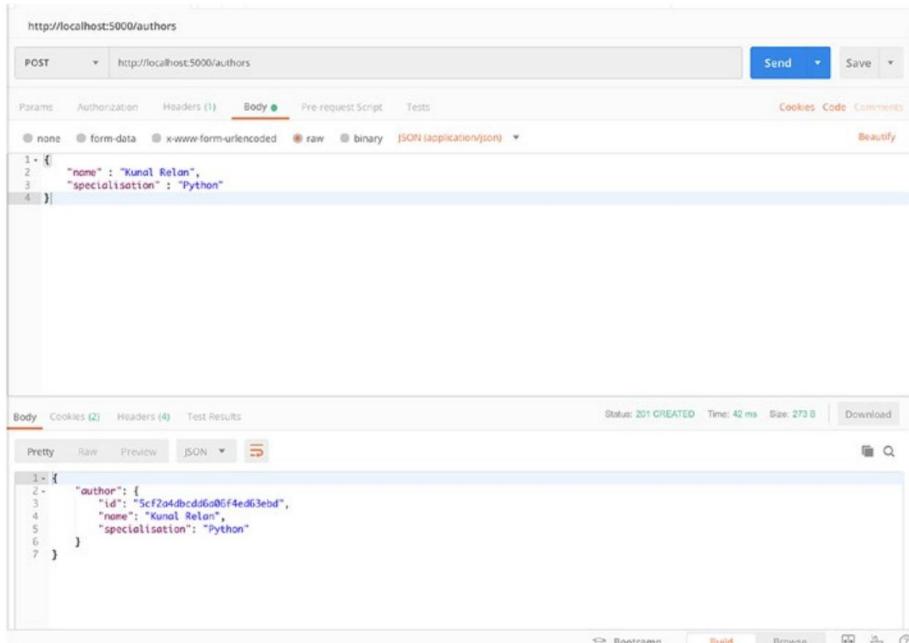


Figura 2-8. Solicitando POST/autores

Capítulo 2 Modelagem de banco de dados no Flask

Ao solicitar, você deverá obter uma saída semelhante à que vê na Figura 2.8 e agora, apenas para confirmar se nosso endpoint GET funciona bem, solicitaremos novamente para ver se ele retorna os dados.

The screenshot shows the Postman interface with a GET request to `http://localhost:5000/authors`. The response body is a JSON object:

```

1 - {
2 -   "authors": [
3 -     {
4 -       "id": "5c2a4d0cdde006f4ed63cb8",
5 -       "name": "Kunal Relan",
6 -       "specialisation": "Python"
7 -     }
8 -   ]
9 - }

```

Figura 2-9. Solicitando GET /autores

Como você pode ver na Figura 2-9, temos nosso autor recentemente registrado no ponto de extremidade GET /autores.

Em seguida, criaremos um endpoint para retornar autores usando o ID do autor e, em seguida, atualizaremos o endpoint para atualizar os detalhes do autor usando o ID do autor e o último endpoint para excluir um autor usando o ID do autor.

Para GET autor por ID, teremos uma rota como `/authors/<id>` que pegará o ID do autor do parâmetro de solicitação e encontrará o autor correspondente.

Adicione o seguinte código para o endpoint GET autor por ID abaixo do seu OBTERNA a rota de todos os autores.

```
@app.route('/authors/<id>', methods = ['GET']) def
get_author_by_id(id): get_author
    = Authors.objects.get_or_404(id=ObjectId(id)) author_schema =
AuthorsSchema(only=[ 'id', 'nome', 'especialização']) autor,
erro =
autor_schema.dump(get_autor) return
make_response(jsonify({"autor": autor}))
```

E agora, quando você solicitar o endpoint /authors/<id>, ele retornará o usuário com o ObjectId correspondente (Figura 2-10).

The screenshot shows a Postman interface with the following details:

- Request URL:** http://localhost:5000/authors/5cf2a4dbcd6a06f4ed63ebd
- Method:** GET
- Response Status:** 200 OK
- Response Time:** 14 ms
- Response Size:** 268 B
- Response Body (Pretty JSON):**

```
1 - [
2 -   "author": {
3 -     "id": "5cf2a4dbcd6a06f4ed63ebd",
4 -     "name": "Kunal Relan",
5 -     "specialisation": "Python"
6 -   }
7 - ]
```

Figura 2-10. OBTER autor por ID

Capítulo 2 Modelagem de banco de dados no Flask

A seguir, criaremos o endpoint PUT para atualizar as informações do autor usando o ID do autor. Adicione o código a seguir para o endpoint do autor PUT.

```
@app.route('/authors/<id>', methods = ['PUT']) def  
update_author_by_id(id): data =  
    request.get_json() get_author =  
    Authors.objects.get(id=ObjectId(id)) if data.get('especialização'):  
        get_author.specialisation =  
            dados['especialização'] if data.get('nome'): get_author.name =  
            dados['nome']  
            get_author.save() get_author.reload()  
    autor_schema =  
    AuthorsSchema(only=['id',  
    'nome', 'especialização']) autor, erro = autor_schema.dump(get_author)  
    return  
    make_response(jsonify({"autor": autor}))
```

Abra o Postman e siga a mesma rota que fizemos no outro módulo para atualizar as informações do autor, mas aqui use o ObjectId retornado no endpoint GET.

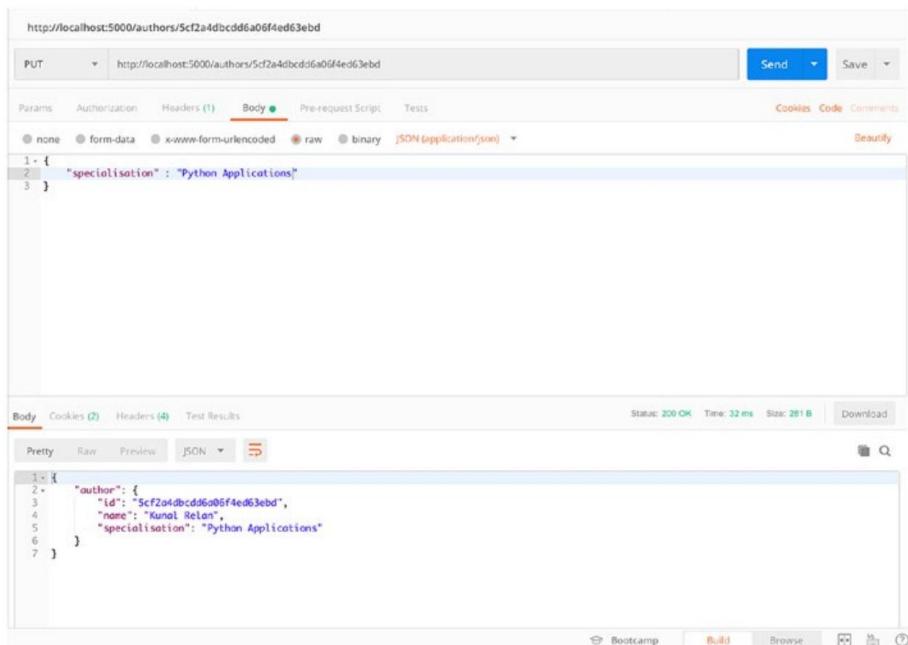


Figura 2-11. Ponto de extremidade do autor *PUT*

Como você pode ver na Figura 2.11, conseguimos atualizar a especialização do autor usando o endpoint `PUT`. A seguir, criaremos o endpoint `DELETE` para excluir um autor usando o ID do autor para concluir nosso aplicativo `CRUD`.

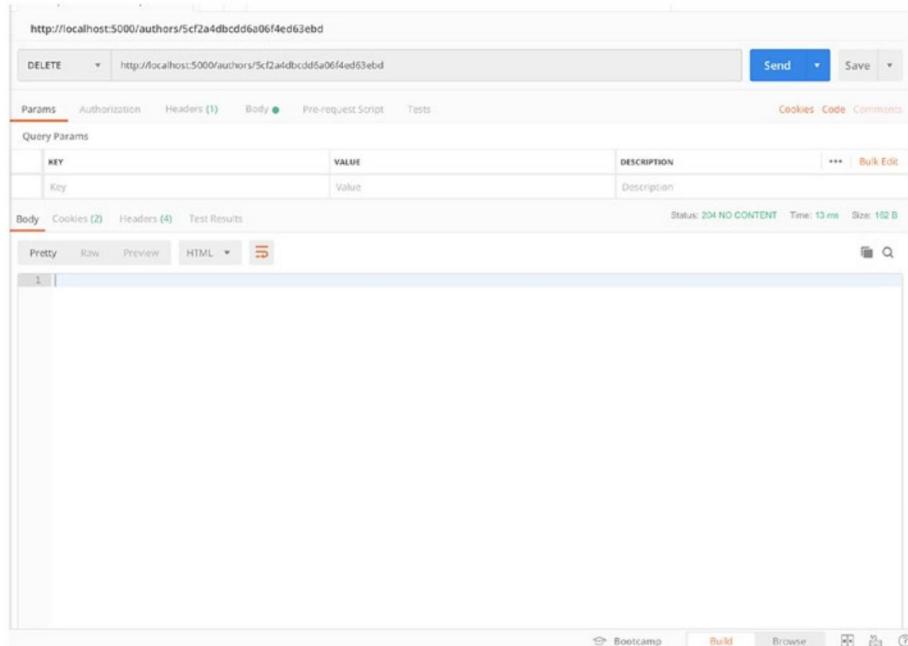
Adicione o código a seguir para criar o endpoint `DELETE` em nosso aplicativo.

```
@app.route('/autores/<id>', métodos = ['DELETE'])
def delete_author_by_id(id):
    Autores.objects(id=ObjectId(id)).delete()
    return make_response("",204)
```

Agora vamos excluir nosso autor recém-criado usando o ID do autor e, semelhante ao último aplicativo, este endpoint não retornará nenhum dado, mas um código de status 204.

Solicite o endpoint de exclusão usando o ID do autor que você fez anteriormente e ele retornará uma resposta semelhante à da Figura 2-12.

Capítulo 2 Modelagem de banco de dados no Flask

**Figura 2-12.** *DELETE endpoint do autor*

Então, isso conclui nosso aplicativo CRUD flask-mongo, e o código final em seu app.py deve ficar assim.

do flask importar Flask, request, jsonify, make_response do flask_mongoengine importar MongoEngine do marshmallow importar Esquema, campos, post_load do bson importar ObjectId

```
app = Flask(__name__)
app.config['MONGODB_DB'] = 'DB_NAME'
db = MongoEngine(app)
```

Esquema.TYPE_MAPPING[ObjectId] = campos.String

Autores da classe (db.Document):

```
nome = db.StringField()
```

```
especialização = db.StringField()

classe AuthorsSchema (Esquema):
    nome = campos.String (required = True)
    especialização = campos.String (required = True)

@app.route('/authors', methods = ['GET']) def
index():
    get_authors = Authors.objects.all()
    author_schema = AuthorsSchema(many=True, only=['id', 'name',
    'especialização']) autores, erro
    = autor_schema.dump(get_authors) return
    make_response(jsonify({"autores": autores}))

@app.route('/authors/<id>', methods = ['GET']) def
get_author_by_id(id): get_author
    = Authors.objects.get_or_404(id=ObjectId(id)) author_schema =
    AuthorsSchema(only=[ 'id', 'nome', 'especialização']) autor,
    erro =
    autor_schema.dump(get_autor) return
    make_response(jsonify({"autor": autor}))

@app.route('/authors/<id>', methods = ['PUT']) def
update_author_by_id(id): data =
    request.get_json() get_author
    = Authors.objects.get(id=ObjectId(id)) if data.get('especialização'):
    get_author.specialisation =
        dados['especialização'] if data.get('nome'): get_author.name =
        dados['nome']
    get_author.save() get_author.reload()
```

Capítulo 2 Modelagem de banco de dados no Flask

```
autor_schema = AuthorsSchema(only=['id', 'nome',
    'especialização'])
autor, erro = autor_schema.dump(get_autor)
return make_response(jsonify({"autor": autor}))

@app.route('/autores/<id>', métodos = ['DELETE'])
def delete_author_by_id(id):
    Autores.objects(id=ObjectId(id)).delete()
    retornar make_response("",204)

@app.route('/autores', métodos = ['POST'])
def criar_autor():
    dados=solicitação.get_json()
    autor = Autores(nome=dados['nome'],especialização=dados
        ['especialização'])
    autor.save()
    autor_schema = AuthorsSchema(only=['id','nome',
        'especialização'])
    autores, erro = autor_schema.dump(autor)
    return make_response(jsonify({"autor": autores}),201)

se __nome__ == "__main__":
    app.run(depuração=Verdadeiro)
```

Conclusão

Agora cobrimos a introdução ao SQLAlchemy e ao MongoEngine e criamos exemplos de aplicativos CRUD usando-os. No próximo capítulo, discutiremos detalhadamente a arquitetura de APIs REST e configuraremos a base para nosso aplicativo Flask REST API.

CAPÍTULO 3

Aplicação CRUD com Flask (Parte 1)

No último capítulo, discutimos sobre bancos de dados e implementamos exemplos baseados em NoSQL e SQL. Neste capítulo, criaremos um aplicativo RESTful Flask do zero. Aqui manteremos um banco de dados de objetos Autor junto com os livros que eles escreveram. Este aplicativo terá um mecanismo de autenticação de usuário para permitir que apenas usuários logados executem determinadas funções. Agora criaremos os seguintes endpoints de API para nossos aplicativos REST:

1. GET /authors: Obtém uma lista de autores ao lado de seus livros.
2. GET /authors/<id>: Obtém o autor com o ID especificado ao lado de seus livros.
3. POST /authors: Isso cria um novo objeto Autor.
4. PUT /authors/<id>: Isto irá editar o objeto autor com o ID fornecido.
5. DELETE /authors/<id>: Isso excluirá o autor com o ID fornecido.
6. GET /books: Isso retornará todos os livros.
7. GET /books/<id>: obtém o livro com o ID especificado.

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

8. POST /books: Isso cria um novo objeto livro.
9. PUT / books/<id>: Isto irá editar o objeto livro com o ID fornecido.
10. DELETE /book/<id>: Isso excluirá o livro com o ID fornecido.

Vamos direto ao assunto e começaremos criando um novo projeto e nomeá-lo-gerente-autor. Portanto, crie um novo diretório e comece criando um novo ambiente virtual.

```
$ mkdir gerenciador de autor && cd gerenciador de autor
```

```
$ virtualenv venv
```

E agora teremos nosso ambiente virtual configurado; em seguida precisamos ativar o ambiente e instale as dependências como fizemos no capítulo anterior.

Começaremos instalando as seguintes dependências para começar e adicione mais conforme precisarmos deles.

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

Também usaremos plantas nesta aplicação. Flask usa o conceito de projetos para criar componentes de aplicativos e oferecer suporte a padrões comuns em todo o aplicativo. Os blueprints ajudam a criar módulos menores para o aplicativo, facilitando o gerenciamento. O Blueprint é altamente valioso para aplicações maiores e simplifica o funcionamento de aplicações grandes.

Estruturaremos o aplicativo em pequenos módulos e manteremos todo o código do aplicativo na pasta /src dentro da pasta app. Então, vá em frente e crie uma pasta src dentro do seu diretório de trabalho atual e crie o arquivo run.py dentro dela.

```
(venv) $ mkdir src && cd src
```

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

Na pasta src teremos nosso arquivo run.py e outro diretório chamado api que irá exportar nossos módulos, então vá em frente e crie uma pasta api dentro de src. Inicializaremos nosso aplicativo Flask no arquivo main.py dentro de src e, em seguida, criaremos outro arquivo run.py que importará main.py, arquivo de configuração e executará o aplicativo.

Vamos começar com main.py.

Adicione o código a seguir para importar as bibliotecas necessárias e initialize o objeto do aplicativo. Aqui definiremos uma função que aceitará a configuração do aplicativo e então inicializará nosso aplicativo.

```
importar sistema operacional  
do frasco importar frasco  
do frasco importar jsonify  
  
app = Frasco(__nome__)  
  
se os.environ.get('WORK_ENV') == 'PROD':  
    app_config=ProduçãoConfig  
elif os.environ.get('WORK_ENV') == 'TESTE':  
    app_config=TestingConfig  
outro:  
    app_config=DesenvolvimentoConfig  
  
app.config.from_object(app_config)  
  
se __name__ == "__main__":  
    app.run(porta=5000, host="0.0.0.0", use_reloader=False)
```

Então esse é o nosso esqueleto de main.py; a seguir, criaremos run.py para chamar app e executar o aplicativo. Mais tarde adicionaremos rotas, inicializaremos nosso objeto db e configuraremos o log em main.py.

Adicione o seguinte código a run.py para importar create_app e executar o aplicativo.

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

do aplicativo de importação principal como aplicativo

```
se __name__ == "__main__":
    aplicação.run()
```

Aqui definimos a configuração, importamos create_app e inicializamos a aplicação. A seguir, moveremos a configuração para um diretório separado e especificaremos a configuração específica do ambiente. Criaremos outro diretório /api dentro de src e exportaremos configurações, modelos e rotas do diretório api, então agora crie um diretório dentro de src chamado api e então crie outro diretório chamado config dentro de api.

Nota Crie um arquivo vazio chamado `__init__.py` dentro de cada diretório para que o Python saiba que contém módulos.

Agora crie config.py dentro do diretório config e também `__init__.py`. Próximo adicione o seguinte código em config.py

classe Configuração (objeto):

```
DEBUGAR = Falso
```

```
TESTE = Falso
```

```
SQLALCHEMY_TRACK_MODIFICATIONS = Falso
```

classe ProduçãoConfig(Config):

```
SQLALCHEMY_DATABASE_URI = <URL do banco de dados de produção>
```

classe DesenvolvimentoConfig(Config):

```
DEBUGAR = Verdadeiro
```

```
SQLALCHEMY_DATABASE_URI = <URL do banco de dados de desenvolvimento>
```

```
SQLALCHEMY_ECHO = Falso
```

classe TestingConfig(Config):

```
TESTE = Verdadeiro
```

```
SQLALCHEMY_DATABASE_URI = <URL do banco de dados de teste>
```

```
SQLALCHEMY_ECHO = Falso
```

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

O código anterior define a configuração básica que fizemos em main.py e depois adiciona a configuração específica do ambiente na parte superior.

Assim, junto com main, importamos configurações de desenvolvimento, teste e produção do módulo de configuração e importamos o módulo do sistema operacional para ler os módulos do ambiente. Depois disso verificamos se a variável de ambiente WORK_ENV foi fornecida para iniciar a aplicação adequadamente; caso contrário, iniciamos o aplicativo usando a configuração de desenvolvimento por padrão.

Portanto, já fornecemos a configuração do banco de dados, mas não inicializamos o banco de dados em nosso aplicativo; a seguir, vamos fazer isso agora.

Agora crie outro diretório dentro da API chamado utils que conterá nossos módulos utilitários; por enquanto iniciaremos nosso objeto db lá.

Crie database.py dentro do utilitário e adicione o seguinte código nele.

```
de flask_sqlalchemy importar SQLAlchemy
banco de dados = SQLAlchemy()
```

E isso iniciará a criação de nosso objeto db; a seguir importaremos o banco de dados objeto em main.py e initialize-o.

Adicione o seguinte código onde importamos bibliotecas para importar o objeto db.

```
de api.utils.database importar banco de dados

def create_app(config):
    app = Frasco(__nome__)
    app.config.from_object(config)
    db.init_app(aplicativo)
    com app.app_context():
        db.create_all()
    aplicativo de devolução
```

E atualize create_app para inicializar o objeto db.

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

Agora temos a base do nosso aplicativo REST e seu aplicativo estrutura deve ficar assim.

```
venv/
  fonte
    API/
      __init__.py
      utilitários
        __init__.py
        banco de dados.py
      configuração
        __init__.py
        banco de dados.py
run.py
main.py
requisitos.txt
```

A seguir, vamos definir nosso esquema de banco de dados. Aqui trataremos de dois recursos, a saber, autor e livro. Então, vamos criar o esquema do livro primeiro. Colocaremos todo o esquema dentro de um diretório chamado models no diretório api, então vá em frente e inicie o módulo models e crie books.py

Adicione o seguinte código a books.py para criar o modelo de livros.

```
de api.utils.database importar banco de dados
de marshmallow_sqlalchemy importar ModelSchema
de campos de importação de marshmallow
```

livro de classe (db.Model):

```
__tablename__ = 'livros'
```

```
id = db.Column(db.Integer, chave_primária = Verdadeiro,
incremento automático = Verdadeiro)
título = db.Column(db.String(50))
ano = db.Column(db.Integer)
```

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

```
autor_id = db.Column(db.Integer, db.ForeignKey('autores.id'))\n\n    def __init__(self, título, ano, autor_id=Nenhum):\n        self.title = título\n        self.year = ano\n        self.author_id = autor_id\n\n    def criar(self):\n        db.session.add(self)\n        db.session.commit()\n        return self\n\nclasse BookSchema(ModelSchema):\n    class Meta (ModelSchema.Meta): modelo\n        = Livro\n        sqla_session=db.sessão\n\n        id = campos.Number(dump_only=True) título =\n        campos.String(required=True) ano =\n        campos.Integer(required=True) autor_id =\n        campos.Integer()
```

Aqui estamos importando o módulo db, marshmallow, como fizemos anteriormente para mapear os campos e nos ajude a retornar objetos JSON.

Observe que temos um campo aqui author_id que é uma chave estrangeira para o campo ID no modelo de autores. A seguir, criaremos o modeloauthors.py e criaremos os autores.

```
de api.utils.database importar banco de dados\nde marshmallow_sqlalchemy importar ModelSchema de\nmarshmallow importar campos de\napi.models.books importar BookSchema\n\nclasse Autor (db.Model):\n    __tablename__ = 'autores'
```

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

```

id = db.Column(db.Integer, primary_key=True,
incremento automático = Verdadeiro)
primeiro_nome = db.Column(db.String(20))
sobrenome = db.Column(db.String(20))
criado = db.Column(db.DateTime, server_default=db.func.now())
livros = db.relationship('Livro', backref='Autor', cascata="all,
delete-orphan")

def __init__(self, primeiro_nome, sobrenome, livros=[]):
    self.first_name = primeiro_nome
    self.last_name = last_name
    self.books = livros

def criar(auto):
    db.session.add(self)
    db.sessão.commit()
    retornar a si mesmo

classe AutorSchema (ModelSchema):
    classe Meta(ModelSchema.Meta):
        modelo = Autor
        sqla_session=db.sessão

    id = campos.Number(dump_only=True)
    primeiro_nome = campos.String(required=True)
    último_nome = campos.String(required=True)
    criado = campos.String(dump_only=True)
    livros = campos.Nested(BookSchema, muitos=True,
    somente=['título','ano','id'])

```

O código anterior criará nosso modelo de autores. Observe que também importamos o modelo de livros aqui e criamos o relacionamento entre o autor e seus livros para que, quando recuperarmos o objeto autor, possamos também obter os livros associados ao seu ID e, assim, configurarmos um para muitos relação entre autor e livros neste modelo.

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

Agora, uma vez que temos nosso esquema de banco de dados instalado, precisamos começar a criar nossas rotas, mas antes de começarmos a escrever as rotas, há mais uma coisa que devemos fazer como parte da modularização de nosso aplicativo e criar outro módulo de `respostas.py` para criar uma classe padrão de HTTP respostas.

Depois disso, criaremos configurações HTTP globais em `main.py`

Crie `respostas.py` dentro de `api/utils`, e aqui usaremos `jsonify` e `make_response` da biblioteca Flask para criar respostas padrão para nossas APIs.

Portanto, escreva o seguinte código em `respostas.py` para iniciar o módulo.

```
do frasco importar make_response, jsonify
```

```
def resposta_com(resposta, valor=Nenhum, mensagem=Nenhum,  
erro=Nenhum, cabeçalhos={}, paginação=Nenhum):
```

```
    resultado = {}
```

```
    se o valor não for Nenhum:
```

```
        resultado.atualização(valor)
```

```
    se response.get('message', None) não for None:
```

```
        resultado.update({'mensagem': resposta['mensagem']})
```

```
    resultado.update({'código': resposta['código']})
```

```
    se o erro não for Nenhum:
```

```
        resultado.update({'erros': erro})
```

```
    se a paginação não for Nenhuma:
```

```
        resultado.update({'paginação': paginação})
```

```
    headers.update({'Access-Control-Allow-Origin': '*'})
```

```
    headers.update({'server': 'API REST do Flask'})
```

```
    retornar make_response(jsonify(resultado), resposta['http_  
código']), cabeçalhos)
```

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

O código anterior expõe uma função `response_with` para nossa API endpoints para usar e responder; junto com isso, também criaremos mensagens e códigos de resposta padrão.

Então aqui está uma lista de respostas que nosso aplicativo suportará.

A Tabela 3-1 fornece as respostas HTTP que usaremos em nossa aplicação. Adicionar o código a seguir acima de `response_with` para defini-los em `respostas.py`.

Tabela 3-1. Respostas HTTP

200 200 Ok	Resposta padrão para solicitações HTTP
201 201 Criado	Implica que a solicitação foi atendida e um novo recurso foi criado
204 204 Sem conteúdo	Solicitação bem-sucedida e nenhum dado foi retornado
400 400 Solicitação incorreta	Implica que o servidor não pode processar a solicitação devido a um erro do cliente
403 403 Não Autorizado	Solicitação válida, mas o cliente solicitante não está autorizado a obter o recurso
404 404 não encontrado	O recurso solicitado não existe no servidor
422 422 Solicitação de entidade não processável	não pode ser processada devido a erro semântico
500 500 Erro interno do servidor	Erro genérico que implica uma condição inesperada no servidor

```
INVALID_FIELD_NAME_SENT_422 = {
    "http_código": 422,
    "código": "campoinválido",
    "message": "Campos inválidos encontrados"
}
```

```
INVALID_INPUT_422 = {
    "http_código": 422,
```

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

```
"code": "invalidInput",
"message": "Entrada inválida"
}

MISSING_PARAMETERS_422
= { "http_code":
422, "code": "missingParameter",
"message": "Parâmetros ausentes."
}

BAD_REQUEST_400
= { "http_code": 400,
"code": "badRequest",
"message": "Solicitação incorreta"
}

SERVER_ERROR_500
= { "http_code":
500, "code": "serverError",
"message": "Erro do servidor"
}

SERVER_ERROR_404
= { "http_code":
404, "code":
"notFound", "message": "Recurso não encontrado"
}

UNAUTHORIZED_403
= { "http_code":
403, "code": "notAuthorized",
"message": "Você não está autorizado a executar isto."
}
```

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

```
SUCCESS_200 =
    { 'http_code': 200,
      'código': 'sucesso'
    }

SUCCESS_201 =
    { 'http_code': 201,
      'código': 'sucesso'
    }

SUCCESS_204 =
    { 'http_code': 204,
      'código': 'sucesso'
    }
```

E agora teremos nosso módulo responds.py funcional; a seguir adicionaremos configurações HTTP globais para lidar com erros.

Em seguida, importe o status e a função response_with em main.py. Adicione as seguintes linhas na seção superior da importação main.py.

```
de api.utils.responses importar resposta_com importar
api.utils.responses como resp
```

E logo acima da função db.init_app, adicione o seguinte código a configurar configurações HTTP globais.

```
@app.after_request def
add_header (resposta): resposta
    de retorno

@app.errorhandler(400) def
bad_request(e):
    logging.error(e)
    return response_with(resp.BAD_REQUEST_400)
```

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

```
@app.errorhandler(500)
def server_error(e):
    logging.error(e)
    return response_with(resp.SERVER_ERROR_500)

@app.errorhandler(404)
def not_found(e):
    logging.error(e)
    return response_with(resp.SERVICE_ERROR_404)
```

O código a seguir adiciona respostas globais em situações de erro. Agora seu main.py deve ficar assim.

```
do frasco importar Frasco
do frasco importar jsonify de
api.utils.database importar db de
api.utils.responses importar resposta_com importar
api.utils.responses como resp

app = Frasco(__nome__)

se os.environ.get('WORK_ENV') == 'PROD':
    app_config = ProductionConfig elif
os.environ.get('WORK_ENV') == 'TEST': app_config
    = TestingConfig
outro:
    app_config=DesenvolvimentoConfig

app.config.from_object(app_config)

db.init_app(app)
com app.app_context():
    db.create_all()

# INICIAR CONFIGURAÇÕES HTTP GLOBAIS

@app.after_request
```

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

```
def add_header(resposta):
    resposta.headers['Content-Type'] = 'application/json'

@app.errorhandler(400) def
bad_request(e):
    logging.error(e)
    return response_with(resp.BAD_REQUEST_400)

@app.errorhandler(500) def
server_error(e):
    logging.error(e)
    return response_with(resp.SERVER_ERROR_500)

@app.errorhandler(404) def
not_found(e):
    logging.error(e)
    return response_with(resp.SERVER_ERROR_404)

db.init_app(app) com
app.app_context(): db.create_all()

se __nome__ == "__main__":
    app.run(porta=5000, host="0.0.0.0", use_reloader=False)
```

Em seguida, precisamos criar nossos endpoints de API e incluí-los em nosso main.py usando Blueprints.

Colocaremos nossas rotas dentro de um diretório chamado rotas na API, então vá em frente e crie a pasta; em seguida, adicione authors.py para criar a rota dos livros.

Em seguida, importe os módulos necessários usando o código a seguir.

```
de importação de frasco Blueprint de
solicitação de importação de frasco
de api.utils.responses importar resposta_com de api.utils
importar respostas como resp
```

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

```
de api.models.authors importar Autor, AuthorSchema  
de api.utils.database importar banco de dados
```

Aqui importamos o Blueprint e solicitamos módulos do Flask, response com e resp método do utilitário de respostas, esquema do autor e o objeto db.

A seguir configuraremos o Blueprint.

```
autor_rotas = Blueprint("autor_rotas", __name__)
```

Uma vez feito isso, podemos começar com nossa rota de autor POST e adicionar o seguinte código abaixo de book_routes.

```
@author_routes.route('/', métodos=['POST'])  
def criar_autor():  
    tentar:  
        dados=solicitação.get_json()  
        autor_schema = AutorSchema()  
        autor, erro = autor_schema.load(dados)  
        resultado = autor_schema.dump(autor.create()).data  
        retornar resposta_com(resp.SUCCESS_201, valor={"autor": resultado})
```

exceto Exceção como e:

```
    imprimir e  
    retornar resposta_com(resp.INVALID_INPUT_422)
```

Portanto, o código anterior pegará os dados JSON da solicitação e execute o método create no esquema Author e, em seguida, retorne o resposta usando o método response_with, fornecendo o tipo de resposta que é 201 para este endpoint e o valor dos dados que é um objeto JSON com o autor recém-criado.

Agora, antes de configurarmos todas as outras rotas, vamos registrar as rotas do autor Blueprint no aplicativo e executar o aplicativo para testar se está tudo bem.

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

Portanto, em seu main.py, importe as rotas do autor e registre o blueprint.

de api.routes.authors importar autor_routes

E adicione a seguinte linha logo acima de @app.after_request.

```
app.register_blueprint(author_routes, url_prefix='/api/autores')
```

Agora execute o aplicativo usando o comando Python run.py e nosso Flask o servidor deve estar instalado e funcionando.

Vamos experimentar o endpoint dos autores POST, então abra a solicitação postman em <http://localhost:5000/api/authors/> com os seguintes dados JSON.

```
{
    "primeiro_nome": "kunal",
    "sobrenome": "Relan"
}
```

The screenshot shows a Postman interface with the following details:

- Method:** POST
- URL:** http://localhost:5000/api/authors/
- Body:** JSON (checkbox checked)
- Body Content:**

```
1 - {
2     "first_name": "Kunal",
3     "last_name": "Relan"
4 }
```
- Status:** 201 CREATED
- Response Body:**

```
1 - {
2     "author": [
3         {
4             "book": [],
5             "created": "2019-06-01 18:57:07",
6             "first_name": "Kunal",
7             "id": 1,
8             "last_name": "Relan"
9         }
10    ],
11    "code": "success"
12 }
```

Figura 3-1. Ponto de extremidade dos autores do POST

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

Como você pode ver, books é um array vazio já que não criamos nenhum livro ainda; a seguir, vamos adicionar o endpoint de autores GET (Figura 3-1).

```
@author_routes.route('/', métodos=['GET'])
def get_author_list():
    buscado = Author.query.all()
    autor_schema = AuthorSchema(muitos=True, somente=['primeiro_nome',
    'sobrenome','id'])
    autores, erro = autor_schema.dump (buscado)
    retornar resposta_com(resp.SUCCESS_200, valor={"autores": autores})
```

O código anterior adicionará a rota GET todos os autores e aqui responderemos com uma matriz de autores contendo apenas seu ID, nome e sobrenome. Então vamos em frente e testá-lo.

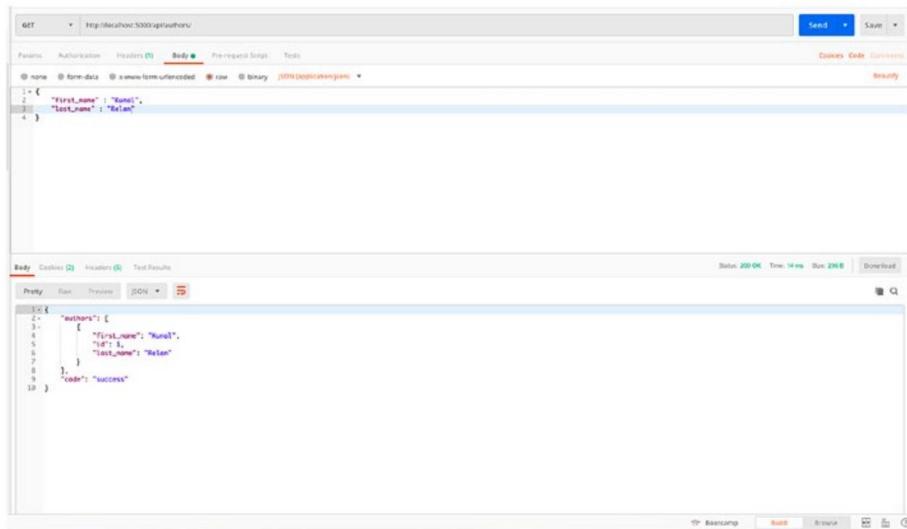


Figura 3-2. GET rota dos autores

Como você pode ver na Figura 3-2, o endpoint respondeu com uma série de autores.

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

A seguir, vamos adicionar outra rota GET para buscar um autor específico usando seu ID e adicione o seguinte código para adicionar a rota.

```
@author_routes.route('/<int:author_id>', métodos=['GET'])
def get_author_detail(autor_id):
    buscado = Author.query.get_or_404(autor_id)
    autor_schema = AutorSchema()
    autor, erro = autor_schema.dump(buscado)
    retornar resposta_com(resp.SUCCESS_200, valor={"autor": autor})
```

O código anterior pega um número inteiro dos parâmetros de rota e encontra o autor com o respectivo ID e retorna o objeto autor.

Então, vamos tentar buscar o autor com ID 1 (Figura 3-3).

```
HTTP/1.1 200 OK
Content-Type: application/json
Date: Mon, 03 Jun 2013 16:57:47 GMT
Server: Werkzeug/0.8.3 Python/2.7.3

{
  "author": {
    "id": 1,
    "first_name": "Kunel",
    "last_name": "Belan",
    "created": "2013-06-03 16:57:47"
  },
  "code": "success"
}
```

Figura 3-3. Buscando autor com ID 1

Se existir o autor com o ID, receberemos a resposta com o código de status 200 e o objeto do autor, caso contrário 404 como na imagem a seguir. Como você pode ver, não há autor com ID 2, e o get_or_404

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

O método gera um erro 404 no endpoint, que é então tratado pelo aplicativo. errorhandler(404) conforme mencionado em nosso main.py (Figura 3-4).

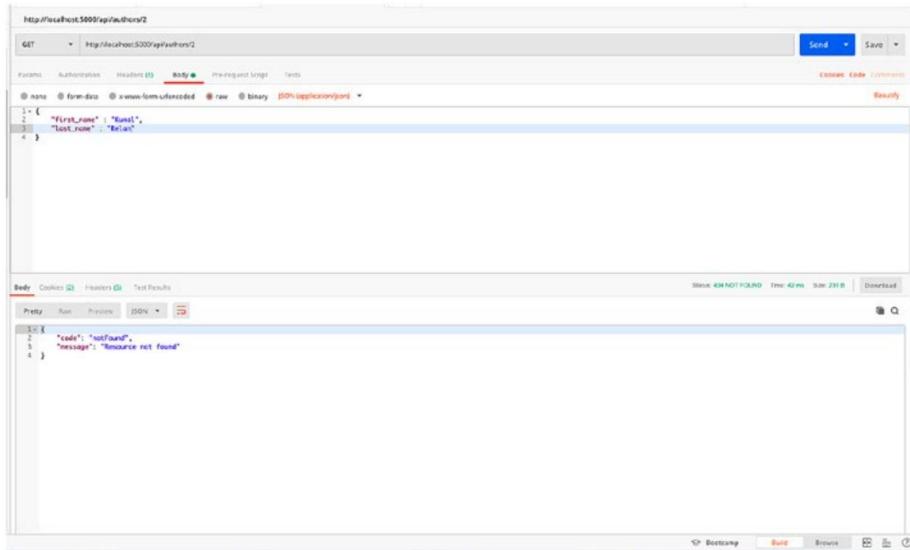


Figura 3-4. Nenhum autor foi encontrado com ID 2

Antes de prosseguirmos com a criação de endpoints PUT e DELETE para o objeto autor, vamos iniciar as rotas do livro. Crie books.py na mesma pasta de rotas e adicione o código a seguir para iniciar a rota.

```

do flask importar Blueprint, solicitação de
api.utils.responses importar resposta_com de api.utils importar
respostas como resp de api.models.books importar
livro, BookSchema de api.utils.database importar banco de
dados
  
```

```
book_routes = Blueprint("book_routes", __name__)
```

E então registre as rotas do livro em main.py como fizemos para as rotas do autor. Adicione o seguinte código logo abaixo de onde você importou as rotas do autor.

```
de api.routes.books importar book_routes
```

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

Logo abaixo de onde você adicionou o plano de rota do autor registro, adicione o seguinte código.

```
app.register_blueprint(book_routes, url_prefix='/api/books')
```

Agora seu main.py deve ter o seguinte código.

```
importar registro
importar sys
importar api.utils.responses como resp do
frasco importar Flask, jsonify de
api.utils.database importar db de
api.utils.responses importar resposta_com de
api.routes.authors importar autor_routes de
api.routes.books importar book_routes

def create_app(config):
    app = Frasco(__name__)
    app.config.from_object(config)
    db.init_app(app)
    com app.app_context():
        db.create_all()
    app.register_blueprint(author_routes, url_prefix='/api/authors')

    app.register_blueprint(book_routes, url_prefix='/api/books')

    @app.after_request
    def add_header(resposta):
        resposta de retorno

    @app.errorhandler(400)
    def bad_request(e):
        logging.error(e)
        return response_with(resp.BAD_REQUEST_400)
```

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

```

@app.errorhandler(500) def
server_error(e):
    logging.error(e) return
    response_with(resp.SERVER_ERROR_500)

@app.errorhandler(404) def
not_found(e):
    logging.error(e) return
    response_with(resp.SERVER_ERROR_404)

db.init_app(app) com
app.app_context(): db.create_all()

logging.basicConfig(stream=sys.stdout, format='%
(asctime)s%(levelname)s%(filename)s:%
(linenos)s%(message)s', level=logging.DEBUG)

```

aplicativo de devolução

A seguir, vamos começar criando o endpoint do livro POST; abra books.py dentro pasta de rotas e adicione o seguinte código abaixo de book_routes.

```

@book_routes.route('/', métodos=['POST']) def
create_book(): try: data
=
request.get_json() book_schema =
BookSchema() livro, erro =
book_schema.load(data) resultado =
book_schema.dump(book.create()).data return
response_with(resp.SUCCESS_201, value={"book": result}) exceto
Exceção
como e: print e return

response_with(resp.INVALID_INPUT_422)

```

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

O código anterior pegará os dados do usuário e, em seguida, executará o método create() no esquema do livro, assim como fizemos no objeto autor; vamos salvar o arquivo e testar o endpoint.

```
{
    "title": "Teste de penetração no iOS",
    "ano": 2016,
    "autor_id": 1
}
```

Usaremos os dados JSON anteriores para POST no endpoint e devemos obter uma resposta com o código de status 200 e o objeto de livro recém-criado. Além disso, como discutimos anteriormente, estabelecemos um relacionamento entre autores e livros e, no exemplo anterior, especificamos o autor com ID 1 para o novo livro, portanto, assim que esta API for bem-sucedida, poderemos buscar o autor com ID 1, e o array books em resposta deverá ter este livro como objeto (Figura 3-5).

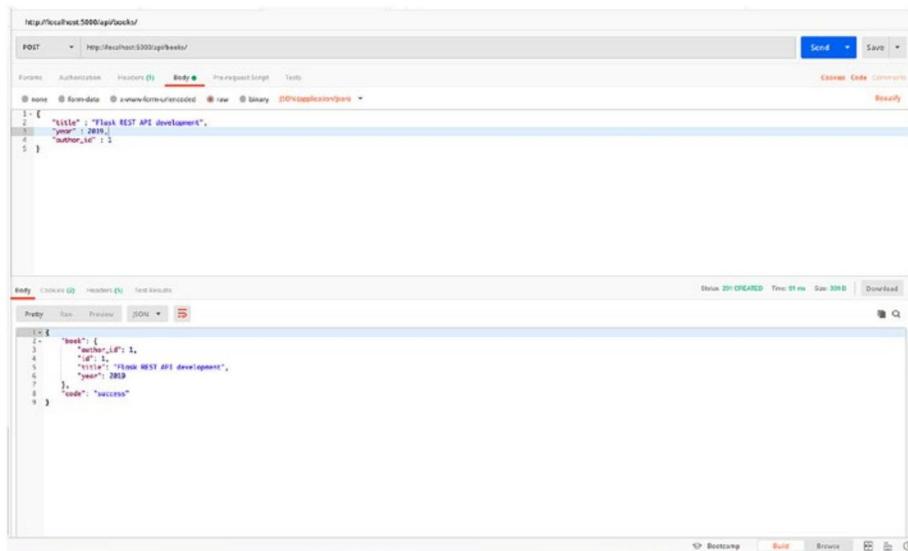


Figura 3-5. Buscar autor com ID 1

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

E como você pode ver na Figura 3-6, quando solicitamos o endpoint /authors/1, junto com os detalhes do autor, também obtemos o array books com a lista de livros ao qual o autor está vinculado.

```

1: {
2:   "author": {
3:     "books": [
4:       {
5:         "id": 3,
6:         "title": "Flask REST API development",
7:         "year": 2019
8:       }
9:     ],
10:    "created": "2020-09-03 18:57:07",
11:    "first_name": "Kevolt",
12:    "id": 1,
13:    "last_name": "Reizen"
14:  },
15:  "code": "success"
16: }

```

Figura 3-6. *GET endpoint do autor*

Portanto, nosso relacionamento modelo está funcionando bem; agora podemos prosseguir para a criação do restante dos endpoints para rotas de autor. Vá em frente e adicione o código a seguir para obter o endpoint PUT da rota do autor para atualizar o objeto do autor.

```

@author_routes.route('/<int:id>', métodos=['PUT'])
def update_author_detail(id):
    dados=solicitação.get_json()
    get_author = Autor.query.get_or_404(id)
    get_author.first_name = dados['first_name']
    get_author.last_name = dados['last_name']
    db.session.add(get_author)
    db.sessão.commit()
    autor_schema = AutorSchema()

```

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

```
autor, erro = autor_schema.dump(get_autor)
retornar resposta_com(resp.SUCCESS_200, valor={"autor": autor})
```

O código anterior criará nosso endpoint PUT para atualizar o objeto autor. No código anterior, pegamos um JSON de solicitação na variável de dados e então buscamos o autor com o ID fornecido no parâmetro de solicitação. Se o autor com esse ID não for encontrado, a solicitação termina com o código de status 404, ou então get_author contém o objeto autor e então atualizamos o first_name e last_name com os dados fornecidos na solicitação JSON e então salvamos a sessão.

Então vamos em frente e atualizar o nome e o sobrenome do autor criados há algum tempo (Figura 3-7).

The screenshot shows a Postman interface with the following details:

- Request URL:** http://localhost:5000/api/authors/1
- Method:** PUT
- Body:** application/json


```
1: {
2:   "first_name": "Jane",
3:   "last_name": "Doe"
4: }
```
- Response:**

```
1: {
2:   "author": {
3:     "book": [
4:       {
5:         "title": "Flask REST API development",
6:         "year": 2019
7:       }
8:     ],
9:     "create": "2019-01-03 13:57:07",
10:    "first_name": "Jane",
11:    "id": 1,
12:    "last_name": "Doe"
13:  },
14:  "code": "success"
15: }
```

Figura 3-7. Ponto de extremidade do autor PUT

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

Então aqui atualizamos o nome e o sobrenome do autor.

No entanto, em PUT precisamos enviar todo o corpo da solicitação do objeto conforme discutimos no segundo capítulo, então a seguir criaremos um endpoint PATCH para atualizar apenas uma parte do objeto autor. Adicione o código a seguir para o endpoint PATCH.

```
@author_routes.route('/<int:id>', métodos=['PATCH'])
def modificar_autor_detalhe(id):
    dados=solicitação.get_json()
    get_autor = Autor.query.get(id)
    if data.get('primeiro_nome'):
        get_author.first_name = dados['first_name']
    if data.get('sobrenome'):
        get_author.last_name = dados['last_name']
    db.session.add(get_author)
    db.sessão.commit()
    autor_schema = AutorSchema()
    autor, erro = autor_schema.dump(get_autor)
    retornar resposta_com(resp.SUCCESS_200, valor={"autor": autor})
```

O código anterior obtém o JSON da solicitação como o outro endpoint, mas não espera todo o corpo da solicitação, mas apenas o campo que precisa ser atualizado no corpo da solicitação e, da mesma forma, atualiza o objeto do autor e salva a sessão. Vamos tentar fazer isso e desta vez alteraremos apenas o primeiro nome do objeto autor.

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

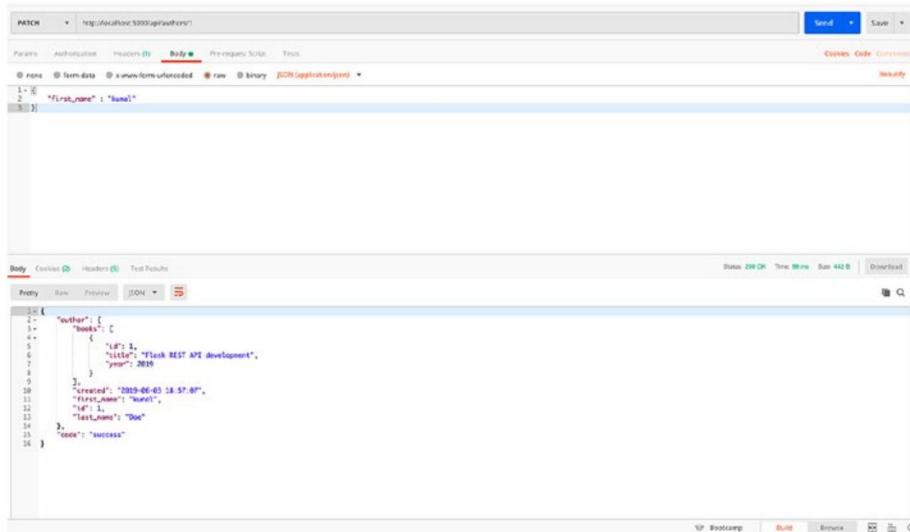


Figura 3-8. Alterar o primeiro nome do objeto autor

Como você pode ver na Figura 3-8, fornecemos apenas o primeiro nome no corpo da solicitação e ele foi atualizado. Então, a seguir, finalmente criaremos nosso endpoint DELETE autor, que pegará o ID do autor do parâmetro de solicitação e excluirá o objeto autor. Observe que neste responderemos com o código de status 204 e nenhum conteúdo.

```

@author_routes.route('/<int:id>', métodos=['DELETE'])
def delete_autor(id):
    get_author = Autor.query.get_or_404(id)
    db.session.delete(get_author)
    db.sessão.commit()
    retornar resposta_com(resp.SUCCESS_204)
      
```

Adicione o código anterior e agora isso criará nosso endpoint DELETE. Vamos tentar excluir nosso autor com ID 1 (Figura 3-9).

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

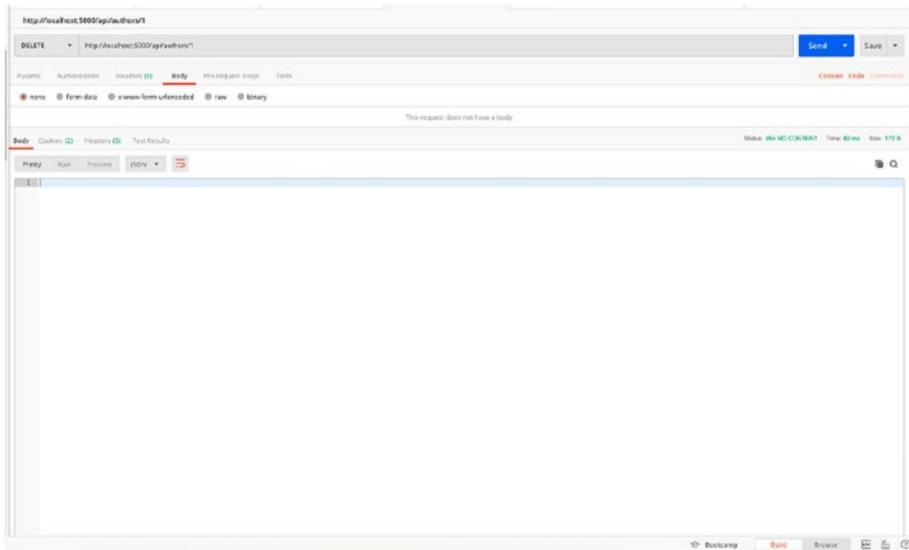


Figura 3-9. *DELETE endpoint do autor*

Com este endpoint, nosso objeto autor deve ser excluído do banco de dados e, ao criar o esquema do autor, configuramos toda a cascata no relacionamento do livro. Assim, todos os livros relacionados ao ID de autor 1 também serão excluídos garantindo que não tenhamos nenhum livro sem ID de autor.

Então é isso para as nossas rotas de autor e a seguir trabalharemos no resto do nosso pontos finais do livro. Em seguida, adicione o seguinte código em books.py para criar o endpoint GET books.

```
@book_routes.route('/', métodos=['GET'])
def get_book_list():
    buscado = Book.query.all()
    book_schema = BookSchema(muitos=True, somente=['author_id',
        'title', 'year'])
    livros, erro = book_schema.dump (buscado)
    return response_with(resp.SUCCESS_200, value={"livros": livros})
```

Salve o arquivo e teste o endpoint; por enquanto você obterá um array vazio já que o livro com ID de autor 1 foi excluído quando excluímos o autor.

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

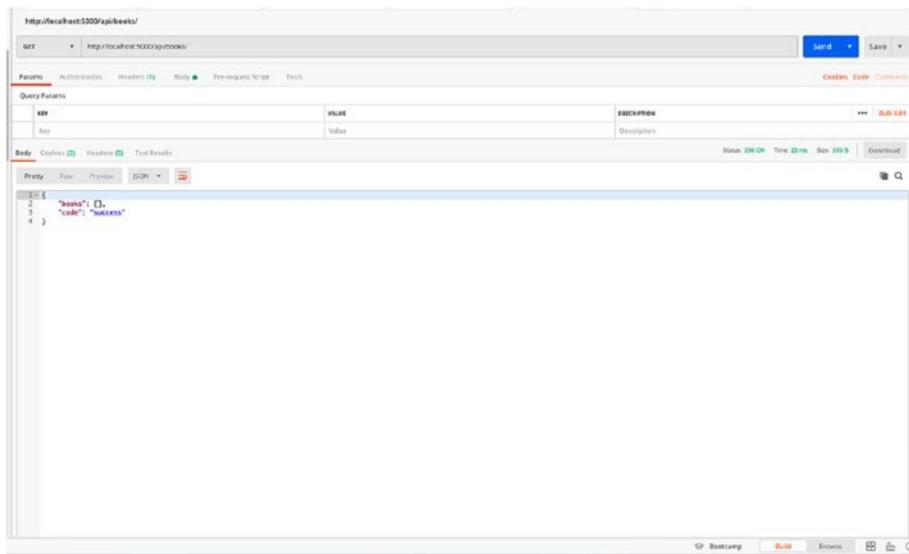


Figura 3-10. Ponto de extremidade GET livros

Como você pode ver na Figura 3-10, não há livros na tabela no momento, então vá em frente e crie um autor e, em seguida, adicione alguns livros com esse ID de autor, já que não podemos adicionar um livro sem um autor, caso contrário terminará em erro de entidade 422 não processável.

A seguir, criaremos o endpoint GET Book by ID.

```
@book_routes.route('/<int:id>', métodos=['GET'])

def get_book_detail(id):
    buscado = Book.query.get_or_404(id)
    esquema_livro = esquema_livro()
    livros, erro = book_schema.dump (buscado)
    return response_with(resp.SUCCESS_200, value={"livros": livros})
```

O código a seguir criará o endpoint GET Book by ID; a seguir, criaremos endpoints PUT, PATCH e DELETE e adicionaremos o código a seguir para os mesmos.

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

```
book_routes.route('/<int:id>', métodos=['PUT']) def
update_book_detail(id):
    dados = request.get_json()
    get_book = Book.query.get_or_404(id)
    get_book.title = dados['title']
    get_book.year = dados['ano']
    db.session.add(get_book)
    db.session.commit()
    book_schema = BookSchema()
    livro, erro = book_schema.dump(get_book) return
    response_with(resp.SUCCESS_200, value={"book": livro})

@book_routes.route('/<int:id>', métodos=['PATCH']) def
modificar_book_detail(id):
    dados = request.get_json()
    get_book = Book.query.get_or_404(id) if
    data.get('title'):
        get_book.title = data['title'] if
    data.get('year'):
        get_book.year = dados['ano']
    db.session.add(get_book)
    db.session.commit()
    book_schema = BookSchema()
    livro, erro = book_schema.dump(get_book) return
    response_with(resp.SUCCESS_200, value={"book" : livro})

@book_routes.route('/<int:id>', métodos=['DELETE']) def
delete_book(id):
    get_book = Book.query.get_or_404(id)
    db.session.delete(get_book)
    db.session.commit()
    retorna resposta_com(resp.SUCCESS_204)
```

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

Portanto, isso encerrará nossas rotas de livros e autores, e agora temos um aplicativo REST funcional. Agora você pode tentar executar o CRUD no autor e reservar rotas.

Autenticação de usuário

Depois de definir todas as nossas rotas, precisaremos adicionar a autenticação do usuário para garantir que apenas usuários logados possam acessar determinadas rotas. Agora adicionaremos o login do usuário e as rotas de inscrição, mas antes disso precisamos adicionar o usuário esquema.

Crie users.py dentro de modelos. No esquema adicionaremos dois métodos estáticos para criptografar a senha e verificar a senha, e para o mesmo precisaremos de uma biblioteca Python chamada passlib, então antes de criarmos o esquema, vamos instalar o passlib usando PIP.

```
(venv)$ pip instalar passlib
```

Uma vez feito isso, adicione o seguinte código para adicionar o esquema do usuário e os métodos.

```
de api.utils.database importar banco de dados
de passlib.hash importar pbkdf2_sha256 como sha256
de marshmallow_sqlalchemy importar ModelSchema
de campos de importação de marshmallow
```

classe Usuário (db.Model):

```
    __tablename__ = 'usuários'
    id = db.Column(db.Integer, chave_primária = True)
    nome de usuário = db.Column(db.String(120), exclusivo = Verdadeiro,
    anulável = Falso)
    senha = db.Column(db.String(120), anulável = Falso)
```

```
def criar(auto):
    db.session.add(self)
```

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

```
db.session.commit()
retornar a si mesmo

@classmethod
def find_by_username(cls, nome de usuário):
    retornar cls.query.filter_by(nome de usuário = nome de usuário).first()

@staticmethod
def gerar_hash(senha):
    retornar sha256.hash (senha)

@staticmethod
def verificar_hash(senha, hash):
    retornar sha256.verify(senha, hash)

classe UserSchema(ModelSchema):
    classe Meta(ModelSchema.Meta):
        modelo = usuário
        sqla_session=db.session

        id = campos.Number(dump_only=True)
        nome de usuário = campos.String (required=True)
```

Então aqui adicionamos um método de classe para encontrar um usuário pelo nome de usuário, e crie um usuário e depois dois métodos estáticos para gerar o hash e verificá-lo. Usaremos esses métodos quando criarmos as rotas do usuário.

Em seguida, crie users.py no diretório de rotas, e é aqui que adicionaremos nosso login de usuário e rotas de inscrição.

Para autenticação do usuário em todo o aplicativo, usaremos a autenticação JWT (JSON Web Tokens). JWT é um padrão aberto que define uma maneira compacta e independente de transmitir informações com segurança como um objeto JSON. JWT é uma forma popular de autorização de usuário no mundo REST.

No Flask existe uma extensão de código aberto chamada Flask-JWT-Extended que fornece suporte JWT e outros métodos úteis.

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

Vamos instalar o Flask-JWT-Extended.

```
(venv)$ pip install flask-jwt-extended
```

A seguir, inicializaremos o módulo JWT no aplicativo em main.py, então importe a biblioteca em main.py.

```
de flask_jwt_extended importar JWTManager
```

Em seguida, initialize o JWTManager com o seguinte código logo acima de db.init_app().

```
jwt = JWTManager(aplicativo)
```

Depois de instalado e inicializado, vamos importar os módulos necessários para nosso arquivo de rotas de usuário.

```
do frasco importar Blueprint, solicitar  
de api.utils.responses importar resposta_com  
de api.utils importa respostas como resp  
de api.models.users importar usuário, UserSchema  
de api.utils.database importar banco de dados  
de flask_jwt_extended importação create_access_token
```

Estes são os módulos que precisaremos para as rotas do usuário; a seguir vamos configurar a rota usando Blueprint com o código a seguir.

```
user_routes = Blueprint("user_routes", __name__)
```

A seguir, importaremos e registraremos as rotas /users em nosso arquivo main.py, então adicione o seguinte código em main.py para importar as rotas do usuário.

```
de api.routes.users importar user_routes
```

E agora logo abaixo de onde declaramos as outras rotas, adicione a seguinte linha de código.

```
app.register_blueprint(user_routes, url_prefix='/api/users')
```

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

A seguir, criaremos nossa rota de usuário POST para criar um novo usuário e adicionaremos o seguinte código em users.py dentro de rotas.

```
@user_routes.route('/', métodos=['POST']) def create_user():
    try: data = request.get_json()

        data['password'] =
            User.generate_hash(data['password']) user_schmea = UserSchema() usuário,
            erro = user_schmea.load(data)
            resultado = user_schmea.dump(user.create()).data
            return response_with(resp.SUCCESS_201) exceto Exceção como
            e: print e return response_with(resp.INVALID_INPUT_422)
```

Aqui estamos pegando os dados da solicitação do usuário em uma variável e então executando a função generate_hash() na senha e criando o usuário. Quando terminar, retornaremos uma resposta 201.

A seguir, criaremos uma rota de login para os usuários inscritos fazerem login. Adicione o seguinte código para o mesmo.

```
@user_routes.route('/login', métodos=['POST']) def authenticate_user():
    tente: data = request.get_json()

    current_user =
        User.find_by_username(data['username']) se não for current_user: return

        resposta_com(resp.SERVER_ERROR_404) if
        User.verify_hash(dados['senha'], usuário_atual. senha): access_token =

        create_access_token(identidade = dados['nome de usuário'])
```

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

```

        return response_with(resp.SUCCESS_201,
value={'message': 'Logado como {}'.format(current_
usuário.nomedeusuário), "access_token": access_token})
outro:
    retornar resposta_com(resp.UNAUTHORIZED_401)
exceto Exceção como e:
    imprimir e
    retornar resposta_com(resp.INVALID_INPUT_422)

```

O código a seguir pegará o nome de usuário e a senha da solicitação data e verifique se o usuário com o nome de usuário fornecido existe usando o método `find_by_username()` que criamos no esquema. A seguir, se o usuário não existir, responderemos com 404, ou então verificaremos a senha usando a função `verify_hash()` no esquema. Se o usuário existir, geraremos um Token JWT e responderemos com 200; caso contrário, responda com 401. Agora temos nosso login de usuário configurado. Em seguida, precisamos adicionar o decorador necessário `jwt` às rotas que queremos proteger. Portanto, navegue até `authors.py` nas rotas e importe o decorador usando o código a seguir.

de `flask_jwt_extended importar jwt_required`

E antes da definição do endpoint, adicione o decorador usando o código a seguir.

`@jwt_required`

Adicionaremos o decorador ao `DELETE`, `PUT`, `POST` e `PATCH` endpoints de `authors.py` e `books.py`, e as funções agora devem ficar assim.

```

@author_routes.route('/', métodos=['POST'])
@jwt_required
def criar_autor():
    ....Código de função

```

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

Vamos testar nossos endpoints de usuário. Abra o Postman e solicite o endpoint dos usuários POST com um nome de usuário e senha. Usaremos os seguintes dados de exemplo.

```
{  
    "nome de usuário": "admin",  
    "senha": "flask2019"  
}
```

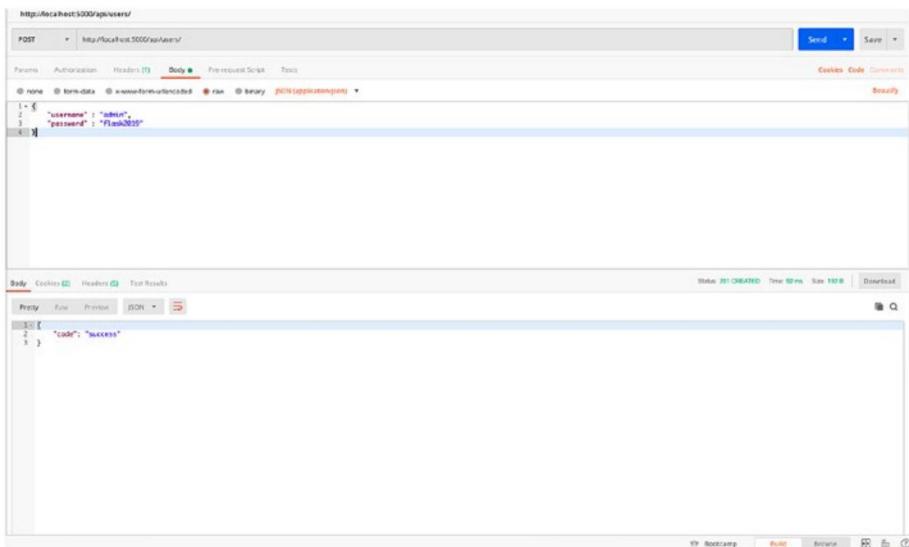


Figura 3-11. Ponto final de inscrição do usuário

Então nosso novo usuário foi criado (Figura 3-11); em seguida, tentaremos registrar com as mesmas credenciais e obtenha o JWT.

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

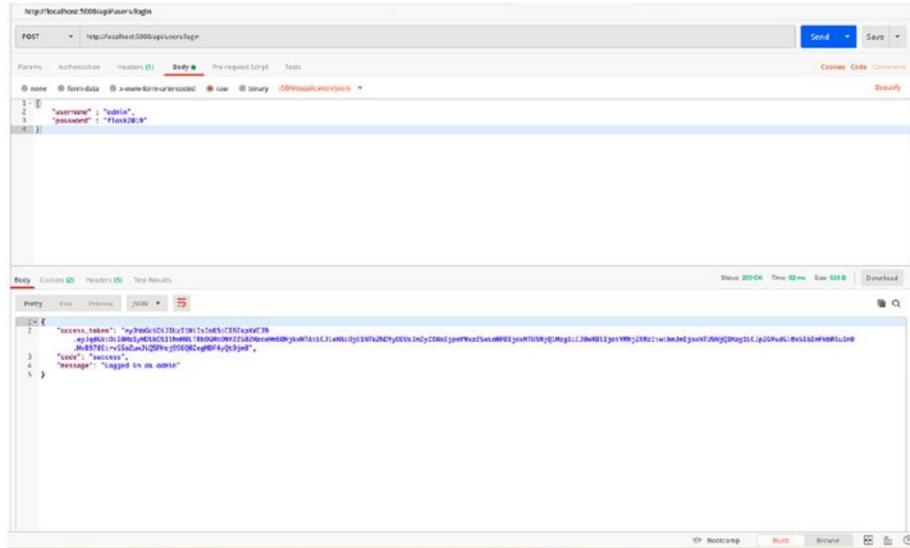


Figura 3-12. Ponto de extremidade de login do usuário

Como você pode ver na Figura 3-12, efetuamos login com sucesso usando os usuários recém-criados. Agora vamos tentar acessar a rota do autor POST à qual adicionamos recentemente o decorador `jwt_required` (Figura 3-13).

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

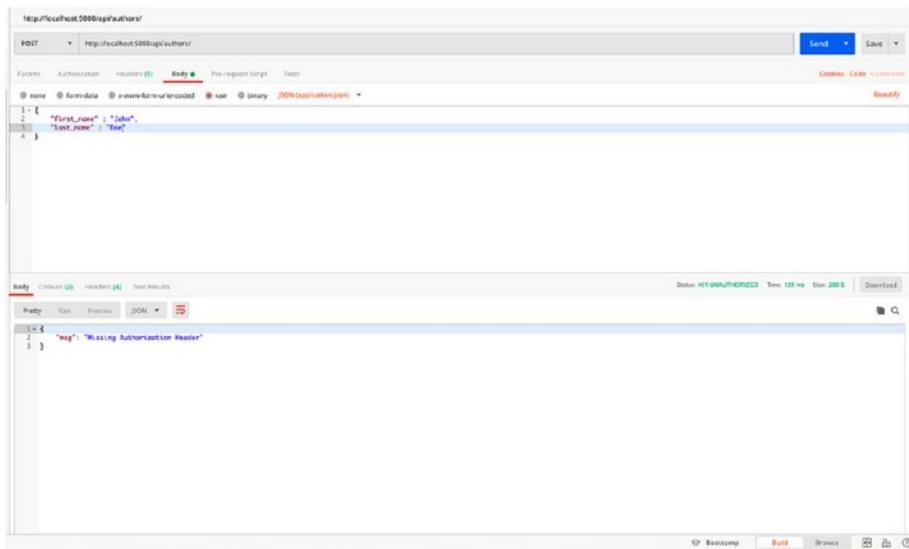


Figura 3-13. Rota do autor POST sem token JWT

Como você pode ver na Figura 3-14, não conseguimos acessar o autor do POST rota mais, e o decorador jwt_required respondeu com erro 401. Agora vamos tentar acessar a mesma rota fornecendo o JWT no cabeçalho. Na seção de cabeçalho da solicitação no Postman, adicione o token com uma chave chamada Authorization e, em seguida, no valor adicione Bearer <token> para fornecer o token JWT como na Figura 3-14.

Capítulo 3 Aplicação CRUD com Flask (Parte 1)

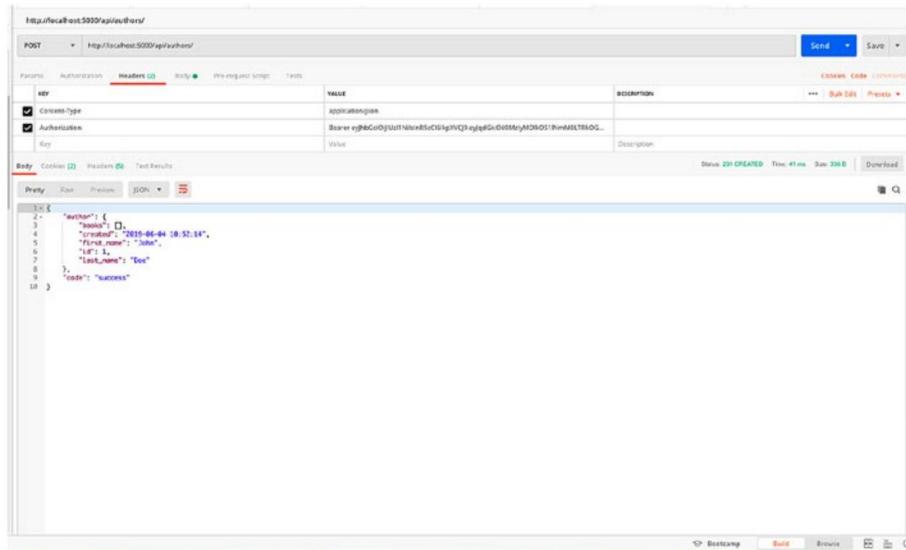


Figura 3-14. Rota do autor POST com JWT

Como você pode ver, após adicionar o Token JWT, podemos acessar o endpoint novamente, e é assim que podemos proteger nossos endpoints REST.

Portanto, no cenário a seguir, permitimos que qualquer pessoa fizesse login na plataforma e depois acessasse as rotas. No entanto, em aplicações do mundo real, também podemos ter verificação de e-mail e inscrição restrita de usuários, além de permitir o controle de acesso baseado em usuário, no qual diferentes tipos de usuários podem acessar determinadas APIs.

Conclusão

Isso conclui este capítulo e criamos com sucesso um aplicativo REST com autenticação de usuário. No próximo capítulo, trabalharemos na documentação de APIs REST, na integração de testes de unidade e na implantação de nosso aplicativo.

CAPÍTULO 4

Aplicação CRUD com Flask (Parte 2)

No último capítulo, criamos APIs REST usando Flask e agora temos um aplicativo CRUD funcional. Neste capítulo, discutiremos e implementaremos recursos para oferecer suporte e estender nossas APIs REST. Embora tenhamos tudo pronto para implantar, aqui estão mais algumas coisas que discutiremos antes de implantar o aplicativo.

1. Verificação de e-mail
2. Carregamento de arquivo
3. Discuta a documentação da API
4. Integre o Swagger

Introdução

No último capítulo, criamos uma aplicação REST usando Flask e MySQL. Neste capítulo discutiremos como estender o aplicativo para recursos adicionais. Começaremos adicionando a verificação de e-mail ao nosso modelo de usuários. A seguir, também adicionaremos endpoint de upload de arquivo ao objeto de usuários e também discutiremos sobre a necessidade de documentação de API, práticas recomendadas para documentar APIs e usar o Swagger como uma ferramenta de documentação de API.

Capítulo 4 Aplicação CRUD com Flask (Parte 2)

verificação de e-mail

No último capítulo, criamos a inscrição e o login do usuário usando um nome de usuário e senha exclusivos. Neste capítulo, estenderemos a autenticação do usuário adicionando inscrição de e-mail ao modelo de usuário e também adicionaremos verificação de e-mail. Para fazer o mesmo, adicionaremos o campo email ao modelo e, assim que um novo objeto de usuário for criado usando a API de inscrição, criaremos um token de verificação e enviaremos um email ao usuário com o link para verificar a conta. Também desabilitaremos o login do usuário até que o e-mail seja verificado. Primeiro vamos adicionar os campos obrigatórios no modelo do usuário.

Navegue até users.py em models e adicione as seguintes linhas abaixo de classe User.

```
isVerified = db.Column(db.Boolean, nullable=False, default=False)
```

```
email = db.Column(db.String(120), único = Verdadeiro, anulável = Falso)
```

E adicione a seguinte linha abaixo do nome de usuário na classe UserSchema.

```
email = db.Column(db.String(120), único = Verdadeiro, anulável = Falso)
```

Além disso, como agora temos e-mails de usuários, atualizaremos o método da classe find_by_username para localizar por e-mail. Portanto, atualize o método find_by_username para o seguinte.

```
@classmethod
def find_by_email(cls, email):
    retornar cls.query.filter_by(email = email).first()
```

Agora sua classe User deve ter o seguinte código.

classe Usuário (db.Model):

```
__tablename__ = 'usuários'
```

Capítulo 4 Aplicação CRUD com Flask (Parte 2)

```
id = db.Column(db.Integer, primary_key = True) nome de
usuário = db.Column(db.String(120), exclusivo = True, anulável
= False) senha =
db.Column(db.String(120), anulável = Falso) isVerified =
db.Column(db.Boolean, nullable=False, default=False) email =
db.Column(db.String(120), unique = True, nullable = False) def create(self):
    db.session .add(self)
    db.session.commit()
    retornar self

@classmethod
def find_by_email(cls, email): return
    cls.query.filter_by(email = email).first()

@classmethod
def find_by_username(cls, email): return
    cls.query.filter_by(username = nome de usuário).first()

@staticmethod
def generate_hash(senha):
    retornar sha256.hash (senha)

@staticmethod
def verify_hash(senha, hash): return
    sha256.verify(senha, hash)
```

E UserSchema deve ter o seguinte código.

```
classe UserSchema(ModelSchema):
    classe Meta (ModelSchema.Meta):
        modelo = Usuário
        sqla_session=db.sessão
```

Capítulo 4 Aplicação CRUD com Flask (Parte 2)

```
id = campos.Number(dump_only=True)
nome de usuário = campos.String(required=True)
email = campos.String(required=True)
```

Observe aqui que o campo isVerified é definido como False por padrão e quando o usuário verificar o e-mail, iremos defini-lo como True, permitindo que o usuário faça login.

A seguir adicionaremos um utilitário chamado token.py que conterá métodos para gerar o token de verificação e confirmar o token de verificação.

O link de verificação no e-mail conterá um URL exclusivo com o token de verificação que deve ser semelhante a `http://host/api/users/confirm/<verification_token>` e o token aqui deve ser sempre único. Usaremos seu pacote perigoso para codificar o e-mail do usuário junto com um carimbo de data/hora, então vamos criar token.py em api/utils.

Antes de escrevermos o código para gerar o token, precisamos adicionar mais algumas variáveis à configuração do aplicativo, já que seu perigo precisa de uma chave secreta e uma senha salt para funcionar, que forneceremos em nosso config.py. Adicione as seguintes linhas em config/config.py em Desenvolvimento, Teste e configurações de produção, garantindo que todas as chaves e sais sejam diferentes.

```
SECRET_KEY= 'sua_chave_segura_aqui'
SECURITY_PASSWORD_SALT= 'sua_senha_de_segurança_aqui'
```

A seguir, em token.py adicione o seguinte código para importar os requisitos.

```
de sua importação perigosa URLSafeTimedSerializer
do frasco importar current_app
```

E então adicione o seguinte código para gerar o token.

```
def generate_verification_token(e-mail):
    serializador = URLSafeTimedSerializer(atual_app.
    configuração['SECRET_KEY'])
    return serializador.dumps(email,salt=current_app.
    configuração['SECURITY_PASSWORD_SALT'])
```

Capítulo 4 Aplicação CRUD com Flask (Parte 2)

No método anterior, usamos URLSafeTimedSerializer para gerar um token usando endereço de e-mail, e o e-mail é codificado no token. A seguir criaremos outro método para validar o token e a expiração, e enquanto o token for válido e não expirado, retornaremos o email e verificaremos o email do usuário.

```
def confirm_verification_token(token, expiração=3600):
    serializador = URLSafeTimedSerializer(atual_app.
    configuração['SECRET_KEY'])

    tentar:
        email=serializer.loads(
            símbolo,
            sal = current_app.config['SECURITY_PASSWORD_SALT'],
            max_age=expiração
        )

    exceto exceção como e :
        retornar e
    retornar e-mail
```

Assim que tivermos nosso utilitário de token instalado, podemos modificar as rotas do usuário. Vamos começar desabilitando o login do usuário antes da verificação do email. Atualize a rota de login para ter o seguinte código; aqui mudamos find_by_nome de usuário para find_by_email, e agora esperaremos que o usuário envie o endereço de e-mail nos dados JSON do endpoint de login e, se o usuário não for verificado, retornaremos a solicitação com um código 400 incorreto sem o token.

Agora seu método de login deve conter o seguinte código.

```
@user_routes.route('/login', métodos=['POST'])
def autenticar_user():

    tentar:
        dados=solicitação.get_json()
        se data.get('email') :
            usuário_atual = Usuário.find_by_email(dados['email'])
```

Capítulo 4 Aplicação CRUD com Flask (Parte 2)

```

elif data.get('nome de usuário'):
    current_user = User.find_by_username(data['username']) se não
    current_user: retornar
        response_with(resp.SERVER_ERROR_404) se
    current_user e não current_user.isVerified: retornar
        response_with(resp.BAD_REQUEST_400) se
    User.verify_hash(data['password'], usuário_atual.senha):
    access_token
        = create_access_token(identity = current_user.username)
    return
    response_with(resp.SUCCESS_201,
    value={'message': 'Logado como {}'.format(current_user.username),
    "access_token": token de acesso})
outro:
    retorne resposta_com(resp.UNAUTHORIZED_401) exceto
Exceção como e:
    retornar resposta_com(resp.INVALID_INPUT_422)

```

Agora vamos criar um endpoint para verificar o token de email.

Começaremos importando os métodos criados recentemente em user.py

de api.utils.token importar generate_verification_token, confirm_verification_token

Em seguida, adicione o seguinte endpoint GET para lidar com a validação de e-mail corretamente abaixo do método de inscrição do usuário.

```

@user_routes.route('/confirm/<token>', methods=['GET'])
def verify_email(token): try: email =
    confirm_verification_token(token) exceto: return
    response_with(resp.SERVER_ERROR_401)

```

Capítulo 4 Aplicação CRUD com Flask (Parte 2)

```
usuário = User.query.filter_by(email=email).first_or_404() se user.isVerified:
```

```
    retornar resposta_com (resp. INVALID_INPUT_422)
outro:
    user.isVerified = True
    db.session.add(user)
    db.session.commit() return
    response_with(resp.SUCCESS_200, value={'message': 'E-mail verificado, você
    pode prosseguir com o login agora.'})
```

A próxima etapa é atualizar o método de inscrição do usuário para gerar o token e enviar o e-mail para o endereço especificado para verificação, então aqui começaremos criando um utilitário de e-mail em nossos utilitários para enviar e-mails.

Para fazer isso, precisaremos de uma biblioteca flask-mail; vamos começar instalando o mesmo. Certificando-se de que você ainda está no ambiente virtual, use a linha a seguir para instalar o flask-mail em seu terminal.

```
(venv) $ pip instalar Flask-Mail
```

Depois de instalado, vamos iniciar e configurar o flask-mail. Adicione as seguintes variáveis em config.py para configurar o email.

```
MAIL_DEFAULT_SENDER= 'seu_endereço_de_e-mail'
MAIL_SERVER= 'email_providers_smtp_address'
MAIL_PORT= <mail_server_port>
MAIL_USERNAME= 'seu_endereço_de_email'
MAIL_PASSWORD= 'seu_e-mail_senha'
MAIL_USE_TLS= Falso
MAIL_USE_SSL= Verdadeiro
```

Em seguida, crie email.py em utils e adicione o código a seguir.

```
from flask_mail import Message,Mail from flask
import current_app mail = Mail()
```

Capítulo 4 Aplicação CRUD com Flask (Parte 2)

A seguir, vamos importar e-mails em nosso main.py e iniciá-lo com a configuração do aplicativo.

```
de api.utils.email importar correio
```

Adicione isso entre outras importações em main.py e logo abaixo de onde iniciamos nosso JWTManager dentro de create_app, adicione o seguinte código.

```
mail.init_app(aplicativo)
```

E agora nosso objeto mail deve ser iniciado com a configuração do aplicativo; a seguir, em email.py, vamos escrever um método para enviar e-mails.

Adicione o seguinte código em email.py para criar um método send_email que receberá o endereço, o assunto e o modelo de email do remetente para enviar.

```
def send_email(para, assunto, modelo):
```

```
    mensagem = mensagem(  
        assunto,  
        destinatários=[para],  
        html=modelo,  
        sender=current_app.config['MAIL_DEFAULT_SENDER'])  
)  
    mail.send(msg)
```

Então isso é tudo que precisamos fazer para enviar o e-mail de verificação; vamos voltar para users.py e atualizar o método de inscrição do usuário para incorporar as alterações.

Vamos começar importando send_email, url_for e render_template_método string em users.py usando a seguinte linha.

```
de api.utils.email importar send_email  
do flask importar url_for, render_template_string
```

Atualize o seguinte código para o método create_user() em users.py, certo antes da função de retorno.

Capítulo 4 Aplicação CRUD com Flask (Parte 2)

tentar:

```

dados=solicitação.get_json()
if(User.find_by_email(data['email']) não é Nenhum ou
User.find_by_username(data['username']) não é Nenhum):
    retornar resposta_com(resp.INVALID_INPUT_422)
dados['senha'] = User.generate_hash(dados['senha'])
user_schmea = UserSchema()
usuário, erro = user_schmea.load(dados)
token = generate_verification_token(dados['e-mail'])
verificação_email = url_for('user_routes.verify_
email', token=token, _external=True)
html = render_template_string("<p>Bem-vindo! Obrigado por se inscrever.
Siga este link para ativar sua conta:</p> <p><a
href='{{ verify_email }}'>{{ verify_email }}</a></p> <br> <p>Obrigado!</p>",
verify_email=verification_email)

subject = "Por favor, verifique seu e-mail"
send_email(usuário.email, assunto, html)
resultado = user_schmea.dump(user.create()).data
retornar resposta_com(resp.SUCCESS_201)
exceto Exceção como e:
    imprimir e
    retornar resposta_com(resp.INVALID_INPUT_422)

```

Aqui estamos fornecendo o email para generate_verification_token e recebendo o token em troca. Em seguida, usamos url_for do Flask para gerar o URL de verificação usando a rota de verificação que acabamos de criar e o token. Depois disso, renderizamos o modelo HTML usando render_template_string do Jinja2, onde fornecemos a string HTML e a variável de verificação e, em seguida, fornecemos todo o e-mail, assunto e HTML fornecidos pelo usuário para send_método de e-mail para enviar o e-mail de verificação.

Capítulo 4 Aplicação CRUD com Flask (Parte 2)

Então, isso é tudo que precisamos para configurar a verificação de e-mail. Vamos começar a testar as rotas de inscrição, login e verificação para verificar se tudo está funcionando.

Vamos começar pelo endpoint de inscrição; abra seu Postman e solicite a API POST /users; entretanto, no corpo JSON, adicione um endereço de e-mail válido.

```
{
    "nome de usuário": "kunalrelan",
    "senha": "olá mundo",
    "e-mail": "kunal.relan@hotmail.com"
}
```

Usaremos o seguinte JSON nos dados da solicitação e acessaremos o endpoint; a resposta deve ser semelhante à anterior; entretanto, você deve receber um e-mail de verificação do seu endereço de e-mail configurado no e-mail especificado nos dados JSON com o token (Figura 4-1).

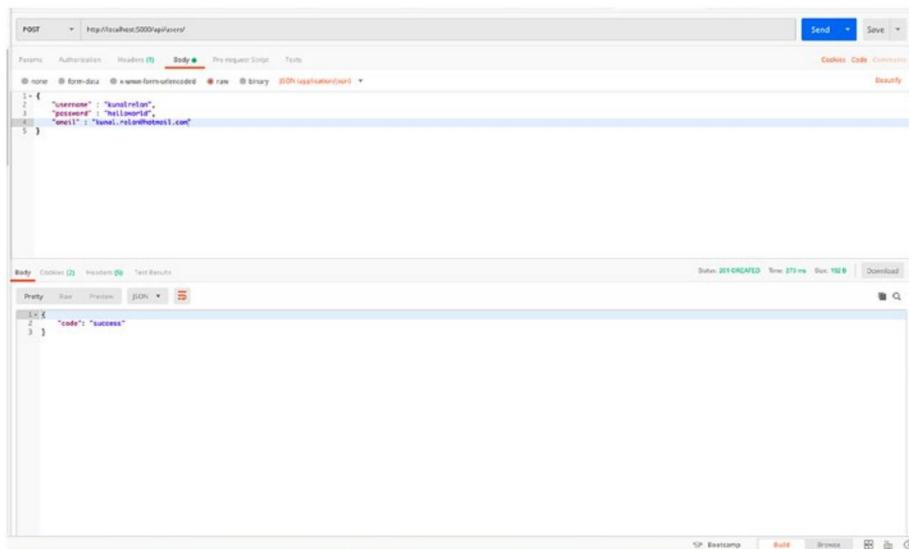
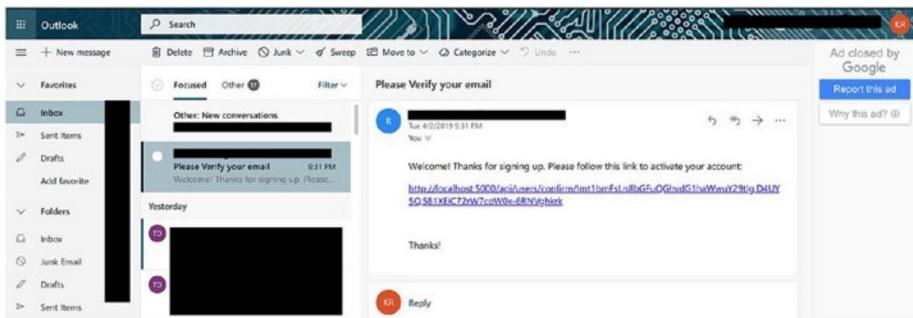


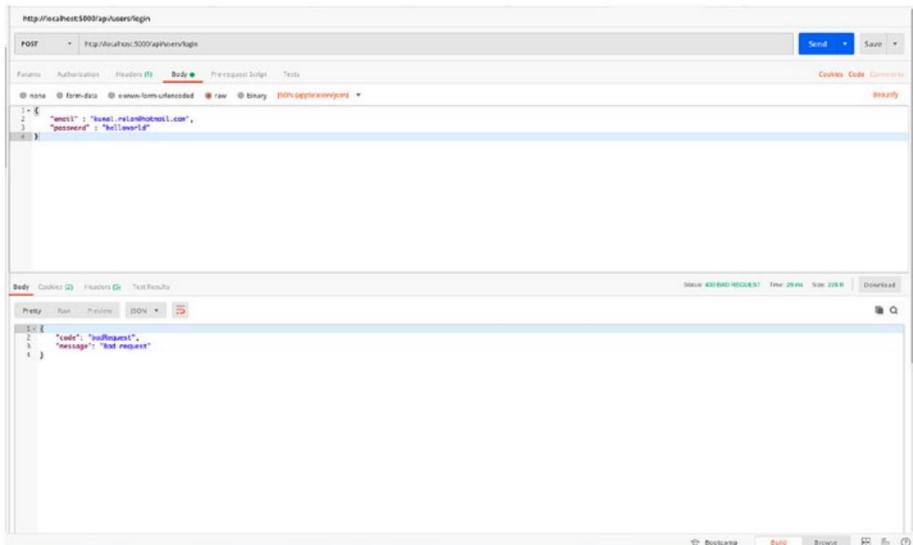
Figura 4-1. API de inscrição de usuário

A seguir, vamos verificar a caixa de entrada do email para verificar se o email chegou e verificar o usuário.

Capítulo 4 Aplicação CRUD com Flask (Parte 2)

**Figura 4-2.**

Como você pode ver na Figura 4-2, o e-mail de verificação chegou com o link para validar a conta do usuário. Antes de ativarmos a conta do usuário, vamos tentar fazer login com as credenciais do usuário para verificar se a validação do email funciona bem (Figura 4-3).

**Figura 4-3.** Login do usuário sem verificação

Capítulo 4 Aplicação CRUD com Flask (Parte 2)

Como você pode ver na Figura 4-4, o usuário não foi verificado e, portanto, não pode fazer login. Agora vamos abrir o link fornecido no e-mail para verificar o usuário, o que permitirá que o usuário faça login e obtenha o token JWT.



Figura 4-4. Verificação de e-mail do usuário

Depois que o usuário for verificado, vamos tentar fazer login novamente e agora deveremos conseguir fazer login e obter o token JWT.

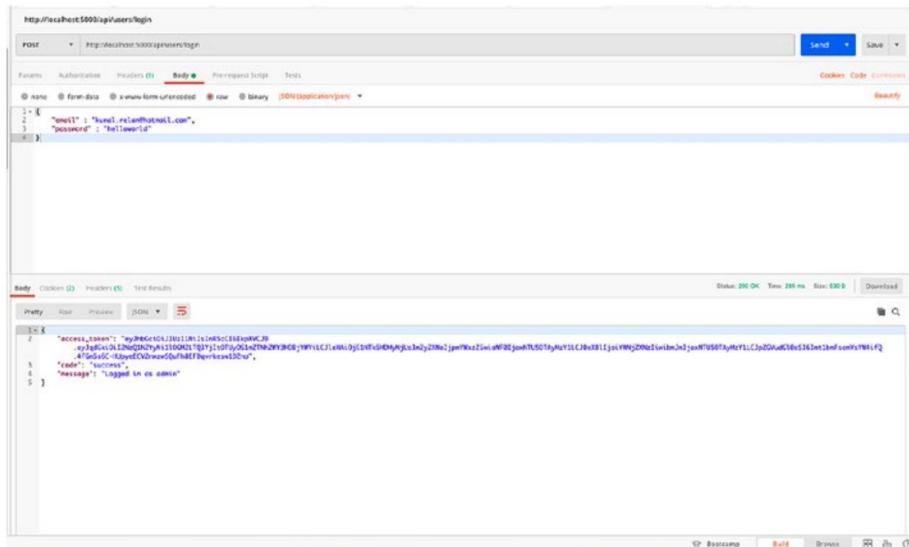


Figura 4-5. Login do usuário após verificação

Capítulo 4 Aplicação CRUD com Flask (Parte 2)

Como você pode ver na Figura 4-5, agora podemos fazer login novamente na conta após verificar o endereço de e-mail.

Então é isso nesta seção. Implementamos com sucesso verificações de e-mail de usuários, e o que fizemos aqui foi apenas um caso de uso de verificação de e-mail; há várias maneiras de usar a verificação de e-mail. Em muitos aplicativos, os usuários conseguem fazer login antes mesmo da verificação do e-mail; no entanto, existem certas funções que estão desabilitadas para usuários não verificados e que podem ser replicadas de forma semelhante à alteração que fizemos no endpoint de login. Na próxima seção implementaremos o upload e o manuseio de arquivos.

Upload de arquivo

Os uploads de arquivos são outro caso de uso comum em APIs REST. Nesta seção implementaremos o upload do avatar para o modelo do autor e um endpoint para acessar o avatar. A ideia é bastante direta aqui; atualizaremos o modelo do autor para armazenar o URL do avatar, criaremos outro endpoint para um usuário conectado para carregar o avatar de um autor usando o ID do autor, salvaremos o arquivo no sistema de arquivos e criaremos outro endpoint para lidar com arquivos de imagem estática.

Antes de começarmos a desenvolver o recurso, vamos falar um pouco mais sobre como lidar com uploads de arquivos no Flask. Aqui usaremos o tipo de conteúdo multipart/form-data que indica o tipo de mídia do recurso de solicitação para o cliente e usaremos `request.files`. Também definiremos um conjunto de extensões de arquivo permitidas, uma vez que não precisamos de nenhum outro tipo de arquivo, exceto imagens para serem carregadas, o que de outra forma pode levar a uma grande vulnerabilidade de segurança. Em seguida, escaparemos do nome do arquivo enviado com `werkzeug.secure_filename()` que gira em torno do princípio “nunca confie na entrada do usuário” e, portanto, o nome do arquivo pode conter código malicioso que pode levar à exploração de vulnerabilidades de segurança. Consequentemente, o método escapará dos caracteres especiais do nome do arquivo.

Para começar, vamos atualizar o modelo do autor para adicionar o campo `avatar`. Portanto, abra `autores.py` em `modelos` e, na declaração do modelo, adicione a seguinte linha na classe `Autor`

```
avatar = db.Column(db.String(20), anulável=True)
```

Capítulo 4 Aplicação CRUD com Flask (Parte 2)

e a seguinte linha na classe AuthorSchema

```
avatar = campos.String(dump_only=True)
```

Depois disso, crie uma nova pasta em /src e nomeie-a como imagens, e adicione a configuração da pasta de upload na configuração do aplicativo que usaremos mais tarde para salvar e buscar os avatares carregados.

Portanto, abra config.py em config e adicione o seguinte parâmetro.

```
UPLOAD_FOLDER= 'imagens'
```

Agora importaremos werkzeug.secure_filename() e url_for do Flask que precisaremos no endpoint que iremos criar, então adicione as seguintes linhas de código abaixo das outras importações em autores.py nas rotas.

```
de werkzeug.utils importar secure_filename
```

Em seguida, importamos o Blueprint e solicitamos do Flask, adicione url_for como o seguinte.

```
do flask importar Blueprint, request, url_for, current_app
```

Logo após a importação, declare Allow_extensions que conterá um conjunto de extensões de arquivo permitidas.

```
extensões_permitidas = set(['imagem/jpeg', 'imagem/png', 'jpeg'])
```

Assim que tivermos definido, vamos criar um método para verificar se o upload a extensão do arquivo é a de uma imagem.

Adicione o seguinte código logo abaixo de Allow_extensions.

```
def arquivo_permitido(nome do arquivo):
```

```
    retornar tipo de arquivo em extensões_permitidas
```

A função acima pegará o nome do arquivo e verificará se a extensão é válida e retornará.

Capítulo 4 Aplicação CRUD com Flask (Parte 2)

Agora adicione o seguinte endpoint para adicionar o endpoint de upload de avatar.

```
@author_routes.route('/avatar/<int:author_id>', métodos=['POST'])
```

```
@jwt_required
```

```
def upsert_author_avatar(autor_id):
```

```
tentar:
```

```
    arquivo = request.files['avatar']
```

```
    get_author = Autor.query.get_or_404(autor_id)
```

```
    se arquivo e arquivo_permitido(file.content_type):
```

```
        nome do arquivo = nome_do_arquivo_seguro(arquivo.nomedoarquivo)
```

```
        arquivo.save(os.path.join(current_app.config['UPLOAD_
```

```
            PASTA'], nome do arquivo))
```

```
        get_author.avatar=url_for('uploaded_file', nome do
```

```
            arquivo=nome do arquivo, _external=True)
```

```
        db.session.add(get_author)
```

```
        db.sessão.commit()
```

```
        autor_schema = AutorSchema()
```

```
        autor, erro = autor_schema.dump(get_autor)
```

```
        retornar resposta_com(resp.SUCCESS_200, valor={"autor": autor})
```

exceto Exceção como e:

imprimir e

```
retornar resposta_com(resp.INVALID_INPUT_422)
```

No código a seguir, procuramos o campo avatar em request.files e depois procuramos o usuário com o ID de usuário fornecido. Assim que tivermos isso, verificaremos se um arquivo foi carregado e escaparemos do nome do arquivo usando a função secure_filename que acabamos de importar. Então usaremos arquivo.save e salve o arquivo na pasta de imagens fornecendo o caminho concatenando UPLOAD_FOLDER da configuração e do nome do arquivo. Agora, uma vez salvo o arquivo, usaremos o método url_for para criar uma URL de acesso ao arquivo enviado, para isso criaremos uma rota com um método uploaded_file que aceita um nome de arquivo e o veicula a partir da pasta de upload configurada que

Capítulo 4 Aplicação CRUD com Flask (Parte 2)

criaremos a seguir. Feito isso, atualizaremos o modelo do autor e o campo do avatar com a URL do avatar enviado.

Em seguida, vá para main.py e adicione a seguinte rota logo após o Blueprint declarações para as rotas na função create_app.

```
@app.route('/avatar/<nome do arquivo>')
def arquivo_carregado(nome do arquivo):
    retornar send_from_directory(app.config['UPLOAD_PASTA'], nome do arquivo)
```

Portanto, esta função aceitará o nome do arquivo e retornará o arquivo do configurado UPLOAD_FOLDER na resposta.

Então é isso para upload de arquivo, e agora devemos ser capazes de fazer upload de um avatar para um autor e recuperá-lo. Vamos voltar ao Postman e experimentar.

Então agora solicite o endpoint de atualização do avatar com dados de formulário e especifique o avatar principal, selecione a imagem que deseja enviar e envie-a. Obteremos 200 respostas de sucesso com o objeto do usuário em resposta; agora observe o campo avatar com o link para o arquivo (Figura 4-6).

The screenshot shows the Postman interface with the following details:

- Request URL:** http://localhost:5000/api/authors/avatar/1
- Method:** POST
- Body:** form-data (selected)
 - Key:** avatar
 - Value:** DsverSec2019.jpg
- Response Status:** 200 OK
- Response Body (Pretty JSON):**

```
1: {
2:     "author": {
3:         "avatar": "http://localhost:5000/avatar/DsverSec2019.jpg",
4:         "created": "2020-08-08 15:12:58",
5:         "first_name": "John",
6:         "id": 1,
7:         "last_name": "Doe"
8:     },
9:     "code": "success"
10}
11}
```

Figura 4-6. Ponto de extremidade de upload do avatar do autor

Capítulo 4 Aplicação CRUD com Flask (Parte 2)

Em seguida, clique no link do avatar para buscar a imagem que você acabou de criar e verificar se isso existe.

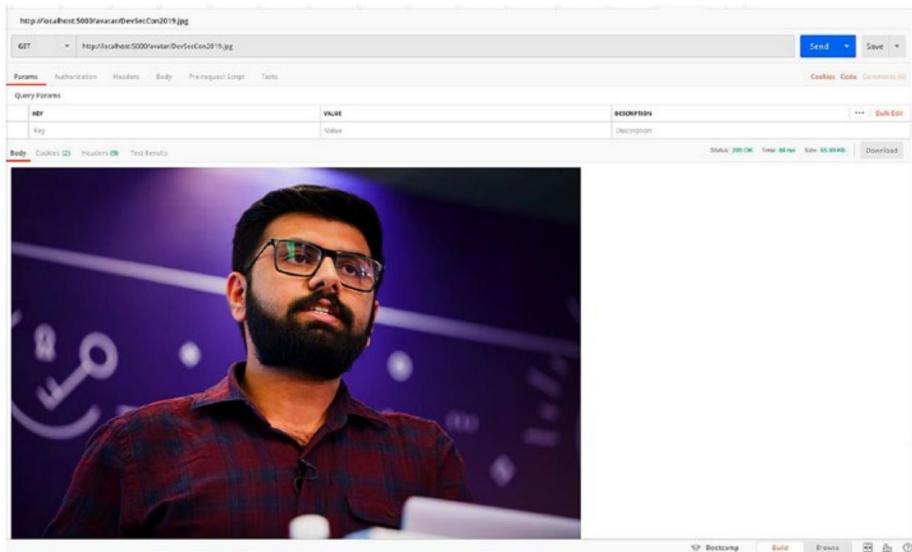


Figura 4-7. Buscar endpoint de avatar

Como você pode ver na Figura 4-7, podemos buscar a imagem usando o rota que criamos. A seguir, vamos tentar fazer upload de um arquivo HTML para verificar se a verificação de extensão permitida funciona bem. Para isso basta criar um arquivo HTML com qualquer texto ou usar qualquer arquivo HTML que você tenha e tentar carregá-lo.

Agora, como você pode ver na Figura 4-8, recebemos um erro ao tentar fazer upload de um arquivo HTML que não é permitido neste endpoint, garantindo que a verificação de extensão esteja funcionando bem para nós.

Capítulo 4 Aplicação CRUD com Flask (Parte 2)

The screenshot shows a POST request to the endpoint `http://localhost:5000/api/authors/avatar/1`. The 'Body' tab is selected, showing a JSON object with a single key-value pair: `{ "avatar": "test.html" }`. The response tab shows a status of `422 UNPROCESSABLE ENTITY` with the message `{ "error": "invalidInput", "message": "invalid input" }`.

Figura 4-8. Carregar endpoint de avatar com tipo de arquivo inválido

Documentação da API

O processo de desenvolvimento de APIs não termina logo após programá-las. Como as APIs REST são usadas por uma variedade de clientes e, portanto, por outros desenvolvedores que as acessam diretamente usando um cliente REST ou se integram a algum tipo de cliente REST, a documentação da API fornece uma maneira fácil de entender o funcionamento dos terminais REST, o que torna a documentação da API é uma parte essencial do desenvolvimento de um aplicativo baseado em REST.

Nesta seção discutiremos os fundamentos da documentação da API, especificação OpenAPI e Swagger, gerando documentos de API usando especificações OpenAPI, publicando documentos de API e testando APIs usando Swagger UI.

Capítulo 4 Aplicação CRUD com Flask (Parte 2)

Elementos básicos da documentação da API

Na documentação de referência da API REST, há cinco seções nas quais a documentação se baseia, a saber:

1. Descrição do recurso: conforme discutido anteriormente, os recursos referem-se às informações retornadas da API; no contexto deste livro, autor, livros e usuários são recursos. A descrição do recurso é geralmente breve, variando de uma a duas frases. Cada recurso possui certos verbos que podem ser acessados.
2. Terminais e métodos: os terminais definem como os recursos fornecidos podem ser acessados e os métodos indicam as interações ou verbos permitidos no recurso, por exemplo, GET, POST, PUT, DELETE e assim por diante. Qualquer recurso terá endpoints relacionados com caminhos diferentes e métodos, mas girarão em torno do mesmo recurso.
3. Parâmetros: Parâmetros são as partes variáveis do endpoint que especifica os dados nos quais você está trabalhando.
4. Exemplo de solicitação: o exemplo de solicitação inclui uma solicitação de amostra contendo os campos obrigatórios, campos opcionais e seu valor de amostra. O exemplo de solicitação geralmente deve ser o mais rico possível e conter todos os campos aceitáveis.
5. Exemplo de resposta e esquema: como o nome sugere, exemplo de resposta contém um exemplo elaborado da resposta da API de acordo com a solicitação. O esquema, por outro lado, define como a resposta é formatada e rotulada. A descrição da resposta é geralmente chamada de esquema de resposta, que é um documento complexo que descreve todos os parâmetros e tipos de resposta possíveis.

Capítulo 4 Aplicação CRUD com Flask (Parte 2)

Especificação OpenAPI

A Especificação OpenAPI (OAS) define uma interface padrão independente de linguagem para API REST, permitindo que humanos e computadores entendam os recursos do aplicativo sem olhar o código-fonte ou fazer inspeção de rede, permitindo que os consumidores da API entendam o funcionamento do aplicativo sem conhecer a lógica de implementação.

As definições OpenAPI podem ter vários casos de uso, incluindo geração de documentação para exibir APIs, ferramentas de teste e assim por diante.

Para o contexto deste livro, usaremos a especificação OpenAPI com Swagger UI para gerar e exibir a documentação de referência da API.

OpenAPI define um conjunto padrão que é então usado para descrever cada parte da API; ao fazer isso, ferramentas de publicação como o Swagger UI podem analisar informações de maneira programática e exibi-las com estilo personalizado e recurso de interatividade. Um documento de especificação OpenAPI pode ser expresso em YAML (YAML Ain't Markup Language) ou JSON, mas, em última análise, o arquivo de especificação será um documento JSON. Como o YAML é mais legível e tem um formato mais comum, usaremos o YAML para criar aqui o documento de especificação OpenAPI, que será publicado usando a UI do Swagger.

Portanto, antes de começarmos a escrever especificações OpenAPI para nossos endpoints, vamos entender o básico das especificações OpenAPI. Um documento de especificação OpenAPI possui três componentes necessários, a saber, openapi que define o número de versão semântica da especificação OpenAPI que é essencial para que os usuários entendam como o documento está formatado e para que as ferramentas de análise analisem o documento adequadamente; Informações que contêm os metadados da API que essencialmente possuem título e versão da API como campos obrigatórios, juntamente com campos adicionais como descrição, informações legais e contato; e Paths que contém informações sobre os endpoints e suas operações disponíveis.

O objeto Paths é o coração do documento de especificação OpenAPI que contém os detalhes dos endpoints disponíveis, que são basicamente os cinco componentes que discutimos na seção anterior.

Capítulo 4 Aplicação CRUD com Flask (Parte 2)

A especificação OpenAPI 3.0 é a versão mais recente; a versão mais antiga dele é a especificação Swagger 2.0 que foi atualizada e transformada em especificação OpenAPI posteriormente. Neste livro usaremos a especificação Swagger 2.0 e definiremos a documentação da API; para fazer isso você pode usar o Inspector do Swagger ou gerá-los usando a ferramenta de geração Swagger em tempo de construção. Vamos verificar ambas as abordagens. Começaremos verificando o Swagger Inspector e prosseguiremos para construir o gerador de tempo que integraremos em nosso aplicativo.

Para começar, abra <https://inspector.swagger.io> na janela do seu navegador (navegador Chrome) e faça login/inscreva-se com o modem de sua preferência (Figura 4-9).

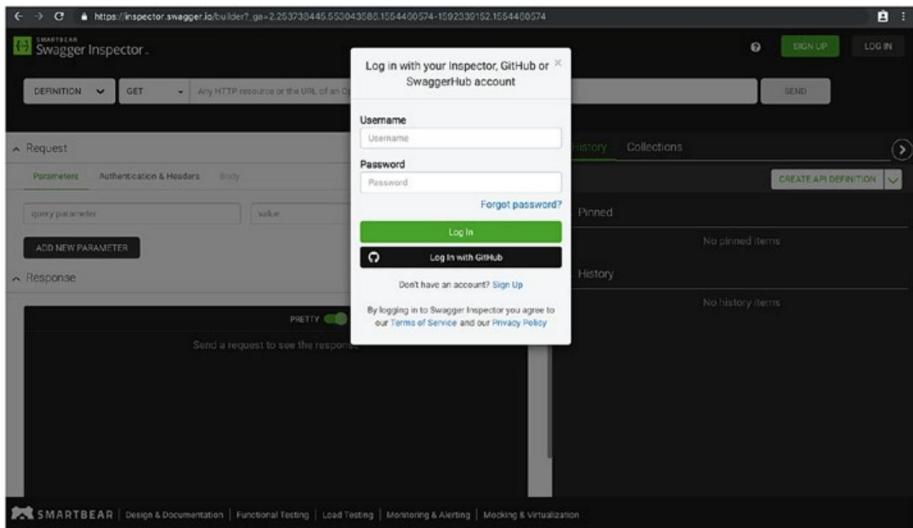


Figura 4-9. Inspector de arrogância

Depois de fazer login, você poderá usar todos os recursos do Inspector de arrogância; em seguida, precisaremos acessar nossos recursos de API usando seu cliente REST, e assim que fizermos isso, ele aparecerá no histórico e poderemos convertê-lo em um arquivo de especificação OpenAPI, mas antes de podermos acessar nosso aplicativo em execução em nosso servidor local, precisaremos adicionar Arrogância Inspector Chrome Extension e, para fazer isso, adicione a extensão usando

Capítulo 4 Aplicação CRUD com Flask (Parte 2)

<https://chrome.google.com/webstore/detail/swagger-inspector-extensi/biemppheiofggogojnfpkngdkchelik>. Depois de instalar a extensão, o Swagger Inspector também poderá executar solicitações em seu servidor local.

Uma vez feito isso, vamos começar acessando nosso endpoint Criar usuário. Então vai adiante e semelhante ao que fizemos no Postman, adicione a URL e escolha o método POST, e no corpo adicione os dados do corpo JSON e clique em enviar (Figura 4-10).

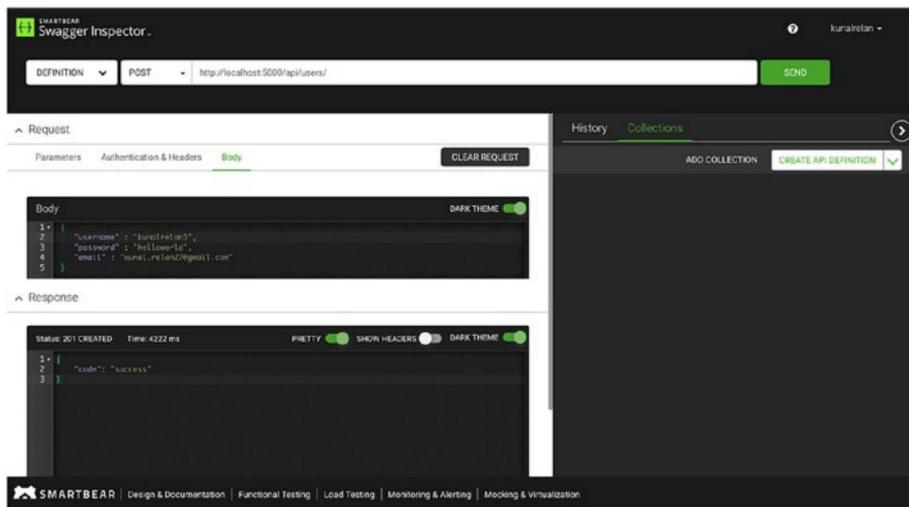


Figura 4-10. Criar endpoint de usuário Swagger Inspector

E semelhante ao Postman, você poderá verificar a resposta da API na janela de resposta, como você vê na figura anterior. Em seguida, você pode verificar o e-mail e acessar o endpoint de login (Figura 4-11).

Capítulo 4 Aplicação CRUD com Flask (Parte 2)

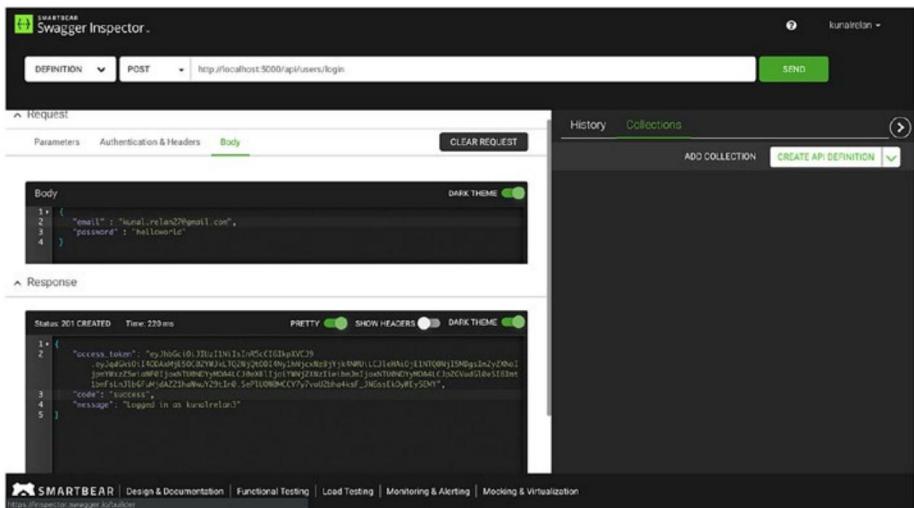


Figura 4-11. Ponto de extremidade de login

Depois de solicitar todos os endpoints, você deseja que a API documentação a ser gerada, basta clicar na guia Histórico e escolher os endpoints que você deseja que o documento de especificação gere e fixá-los. Depois de fixá-los, clique na pequena seta ao lado do botão Criar definição de API e selecione OAS 2.0 para usar a versão 2.0 do especificação (Figura 4-12).

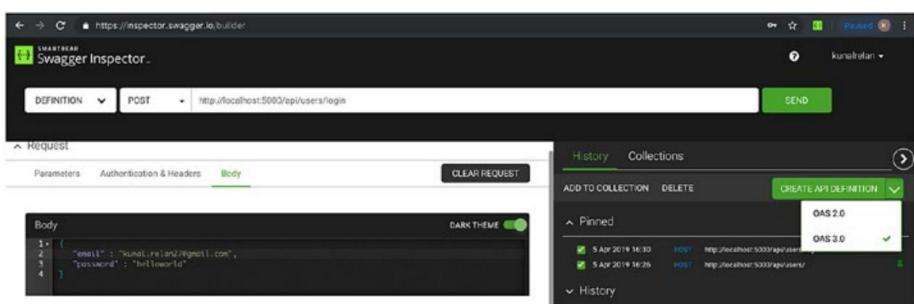


Figura 4-12. Solicitações fixadas

Capítulo 4 Aplicação CRUD com Flask (Parte 2)

Agora clique em Criar definição que, uma vez concluído, abrirá um pop-up com link para abrir o SwaggerHub, onde você pode importar a especificação OpenAPI e visualizar os documentos da API (Figura 4-13).

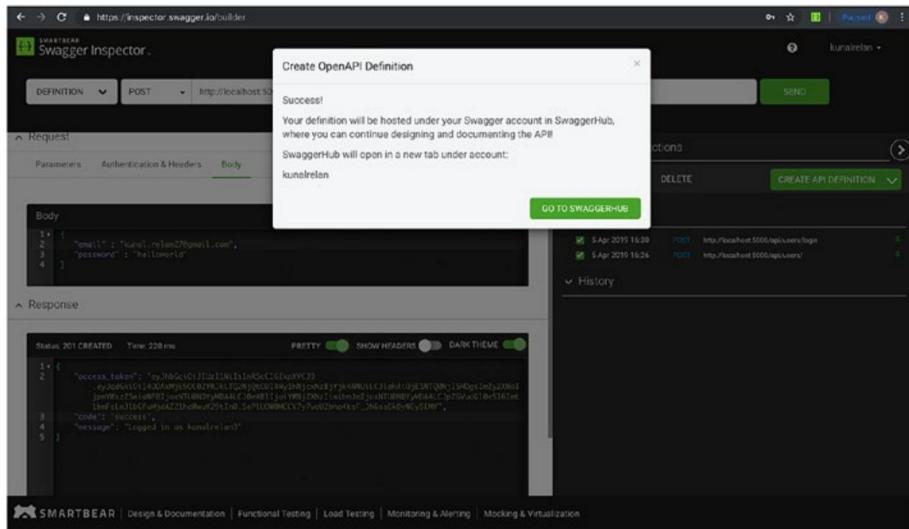


Figura 4-13. Geração de especificações OpenAPI

Agora siga o link e o SwaggerHub será aberto, solicitando que você insira o título e a versão de suas APIs. Aqui adicionaremos o banco de dados do autor e deixaremos a versão padrão 0.1, tornaremos a visibilidade privada e clicaremos em Importar API como na Figura 4-14.

Capítulo 4 Aplicação CRUD com Flask (Parte 2)

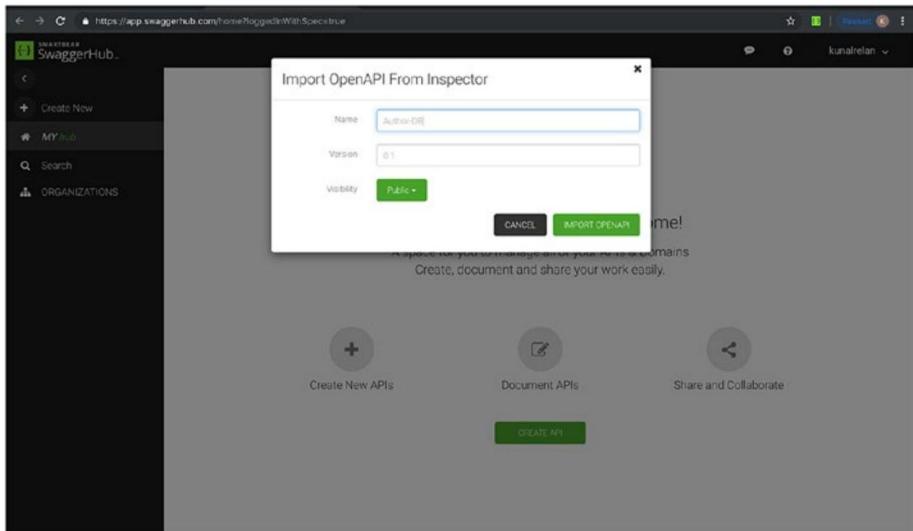


Figura 4-14. Importando OpenAPI do Inspector

Feito isso, você poderá verificar a documentação de suas APIs como na figura a seguir. Aqui para este tutorial, selecionei apenas dois endpoints, mas você pode ter todos os seus endpoints documentados aqui.

Na Figura 4-15 você pode ver que o servidor selecionado é o endereço do nosso servidor local servidor, e então temos nossos endpoints selecionados.

Capítulo 4 Aplicação CRUD com Flask (Parte 2)

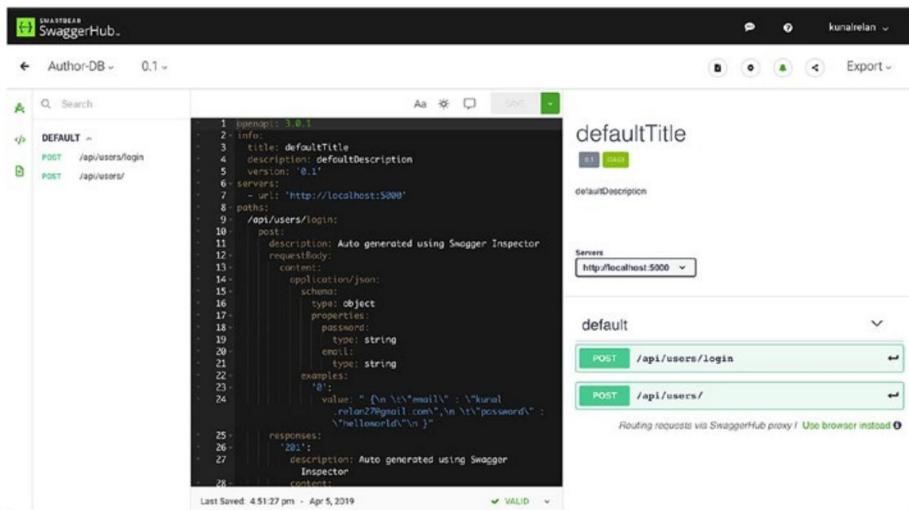


Figura 4-15. Swagger Hub

A seguir, vamos verificar a documentação da API clicando no ícone de papel na barra superior, como nas Figuras 4-16 e 4-17.

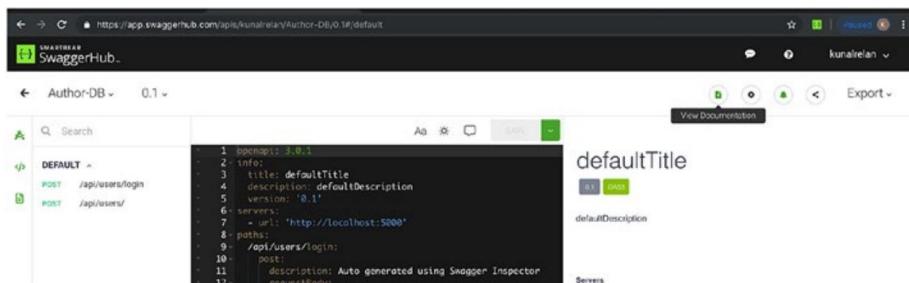


Figura 4-16. Ver documentação

Capítulo 4 Aplicação CRUD com Flask (Parte 2)

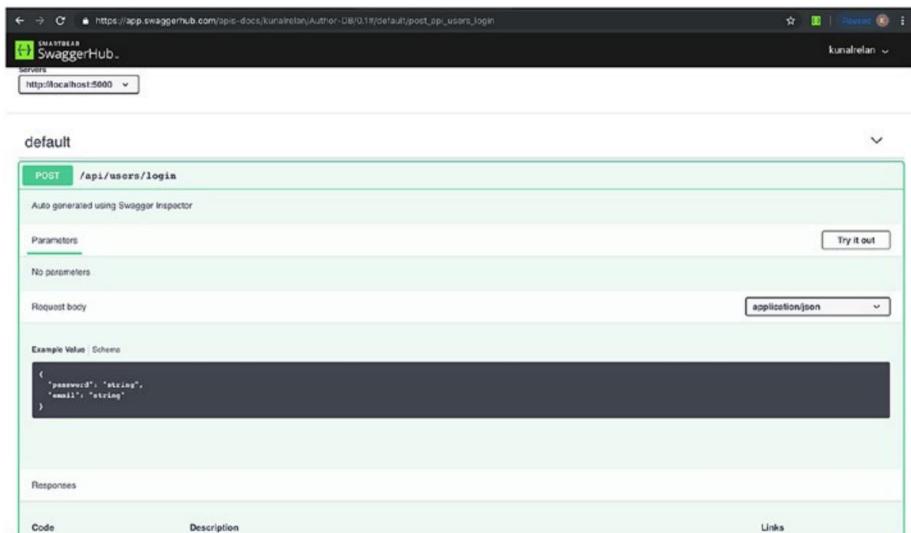


Figura 4-17. Ver página de documentação

Depois que a página for carregada, você estará no modo interativo da sua API documentação onde você pode ver os endpoints, parâmetros, solicitação de amostra e resposta de amostra. Em seguida, clique em Try It Out e a janela do corpo da solicitação se tornará editável, onde você poderá preencher os dados do corpo da solicitação, como na Figura 4-18. Abaixo você também pode ver as respostas e seus formatos.

Capítulo 4 Aplicação CRUD com Flask (Parte 2)

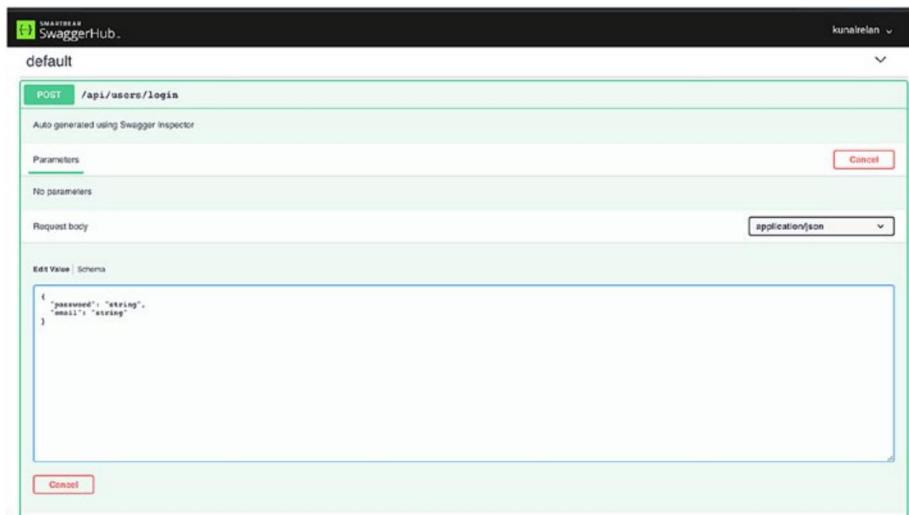


Figura 4-18. Modo de solicitação de API

Então vá em frente e edite o e-mail e a senha e clique em executar para solicitação para acessar a API.

Em seguida, você também pode exportar a versão YAML/JSON do documento de especificação para usá-lo com sua versão do Swagger UI.

Seguindo em frente, agora integraremos a documentação da API usando nosso próprio instalação da UI do Swagger e especificação do tempo de construção.

Para o mesmo usaremos as extensões `flask_swagger` e `flask_swagger_ui`; vamos instalar os dois usando PIP.

```
(venv)$ pip instalar flask_swagger flask_swagger_ui
```

Depois de instalado vamos integrá-lo em nossa aplicação; para fazer isso, abra `main.py` e importe ambas as bibliotecas usando as seguintes linhas.

```
de flask_swagger importar arrogância
de flask_swagger_ui importar get_swaggerui_blueprint
```

Serviremos a UI do Swagger no endpoint `/api/docs`.

Capítulo 4 Aplicação CRUD com Flask (Parte 2)

Agora criaremos um endpoint para atender às especificações de API definidas usando Swagger 2.0

Portanto, adicione o seguinte código abaixo das funções do manipulador de erros onde definiremos a rota /api/spec e iniciaremos nossa definição Swagger e retornaremos o arquivo JSON gerado.

```
@app.route("/api/spec")
especificação de definição():

    swag = swagger(app, prefix='/api')
    swag['info'][‘base’] = "http://localhost:5000"
    swag['info'][‘versão’] = "1.0"
    swag['info'][‘title’] = "Banco de dados do autor do frasco"
    retornar jsonify (ganhos)
```

Agora iniciaremos flask_swagger_ui para buscar este arquivo JSON e renderizar a UI do Swagger usando-o. Adicione o código a seguir abaixo da nova rota para iniciar o método get_swagger_blueprint que acabamos de importar de flask_swagger_ui, e aqui forneceremos a rota de documentos, roteador de arquivo JSON e app_name na variável de configuração e, em seguida, registraremos o Blueprint.

```
swaggerui_blueprint = get_swaggerui_blueprint('/api/docs', '/api/spec',
config={'app_name': "Flask Author DB"})
app.register_blueprint(swaggerui_blueprint,url_
prefixo=SWAGGER_URL)
```

E agora, quando você tentar acessar <http://localhost:5000/api/docs>, você poderá ver a UI do Swagger (Figura 4-19).



Figura 4-19. /U arrogante

Capítulo 4 Aplicação CRUD com Flask (Parte 2)

Na barra de URL anterior, você também pode fornecer o URL para o arquivo JSON exportado do SwaggerHub para explorar suas APIs.

Documentação de tempo de construção

A seguir, documentaremos as APIs usando a documentação de tempo de construção e geraremos o arquivo de documentação JSON; no entanto, usaremos YAML ao descrever os endpoints.

O Flask Swagger coletará automaticamente a documentação YAML de definições de método usando `request` sob método seguido pela descrição.

Aprenderemos isso usando uma definição de exemplo em nosso endpoint Criar usuário. Portanto, adicione as seguintes linhas após `def create_user()` nas rotas `users.py`.

```
"""
```

Criar endpoint de usuário

```
---
```

parâmetros:

- no corpo

- nome: corpo

- esquema:

- id: UserSignup

- obrigatório:

- nome de usuário

- senha

- e-mail

- propriedades:

- nome de usuário:

- tipo: string

- description: Nome de usuário exclusivo do usuário

- padrão: "Johndoe"

- senha:

- tipo: string

- descrição: Senha do usuário

Capítulo 4 Aplicação CRUD com Flask (Parte 2)

padrão: e-mail "algo forte":

tipo: string

descrição: email do usuário padrão:
"someemail@provider.com"

respostas:

201:

descrição: Esquema criado pelo usuário com
sucesso:

id: propriedades

UserSignUpSchema: código:

tipo: string

422:

descrição: Esquema de argumentos de entrada
inválidos:

id: propriedades

invalidInput:

código:

tipo: string

mensagem:

tipo: string

""""

Aqui estamos usando YAML para definir parâmetros e respostas conforme você podemos ver no exemplo anterior; definimos o tipo de parâmetro e, em nosso caso, é um parâmetro de corpo e, em seguida, definimos o esquema dos parâmetros necessários com dados de amostra e nomes de campos. Nas respostas definimos os diferentes tipos de respostas esperadas e seu esquema (Figura 4-20).

Capítulo 4 Aplicação CRUD com Flask (Parte 2)

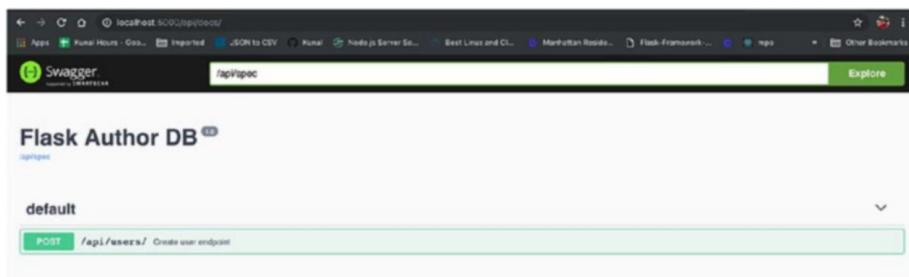


Figura 4-20. Geração de documento em tempo de construção

Agora, se você recarregar seu aplicativo e visitar a UI do Swagger, você deverá estar capaz de ver seu endpoint de usuário Criar e acessá-lo usando a UI do Swagger.

Observe aqui como a descrição, os parâmetros e as respostas foram interpretados e colocados na UI do Swagger.

Em seguida, adicione isto ao método de login para gerar documentos para o endpoint de login.

Login de usuário

parâmetros: -

in: nome do

corpo:

esquema

do corpo: id:

UserLogin

obrigatório: -

senha - email

propriedades:

e-mail:

tipo: string

descrição: e-mail do usuário padrão:

"someemail@provider.com" senha: tipo: string

Capítulo 4 Aplicação CRUD com Flask (Parte 2)

description: Senha do usuário padrão:
"somethingstrong"

respostas:

200:

descrição: Usuário logado com sucesso no esquema:

id: propriedades
UserLoggedIn:
código:
 tipo: string
mensagem:
 tipo: string
valor:
 esquema:
 id: UserToken
 propriedades:
 access_token:
 tipo: string
 código:
 tipo: string
 mensagem:
 tipo: string

401:

descrição: Esquema de argumentos de entrada
inválidos:

id: propriedades
invalidInput:
código:
 tipo: string
mensagem:
 tipo: string

....

Capítulo 4 Aplicação CRUD com Flask (Parte 2)

Em seguida, passaremos para o arquivo de rotaauthors.py e criaremos o documento para Criar autor, e se você se lembra, esta rota precisa que o usuário esteja logado e aqui adicionaremos um parâmetro de cabeçalho extra que aceitará autorização cabeçalho.

Criar endpoint do autor

parâmetros:

-in: nome do

corpo:

esquema do corpo:

id: Autor

obrigatório:

- primeiro_nome

- sobrenome -

livros

propriedades:

first_name:

tipo: string

descrição: Nome do autor padrão: "John"

last_name:

tipo: string

descrição: Sobrenome do autor padrão: "Doe" - in:
header

nome: autorização

tipo: string

obrigatório:

verdadeiro segurança:

- O portador: []

respostas:

Capítulo 4 Aplicação CRUD com Flask (Parte 2)

200:

descrição: Esquema criado com sucesso pelo autor:

id: AutorCriado

propriedades:

código:

tipo: string

mensagem:

tipo: string

valor:

esquema:

id: Autor Completo

propriedades:

nome_nome:

tipo: string

sobrenome:

tipo: string livros:

tipo: itens da

matriz:

esquema:

id: BookSchema

422:

descrição: Esquema de argumentos de entrada
inválidos:

id: propriedades

invalidInput:

código:

tipo: string

mensagem:

tipo: string

"""

Capítulo 4 Aplicação CRUD com Flask (Parte 2)

Em seguida, adicione as seguintes linhas para o endpoint do avatar do autor Upsert; perceber neste caso adicionaremos um parâmetro para o ID do autor ser uma variável no caminho.

....

Insira o avatar do autor

parâmetros:

-in: nome do

corpo:

esquema do corpo:

id: Autor

obrigatório:

- avatar

propriedades:

avatar:

tipo:

descrição do arquivo: Arquivo

de imagem - nome:

autor_id

em: descrição do caminho: ID do

autor obrigatório:

esquema verdadeiro:

tipo: inteiro

respostas:

200:

descrição: O avatar do autor alterou o esquema com sucesso:

id: AutorCriado

propriedades:

código:

tipo: string

mensagem:

Capítulo 4 Aplicação CRUD com Flask (Parte 2)

```
    tipo: string
    valor:
        esquema:
            id: Autor Completo
        propriedades:
            nome_nome:
                tipo: string
            sobrenome:
                tipo: string livros:
                    tipo: itens da
                    matriz:
                        esquema:
                            id: BookSchema
422:
    descrição: Esquema de argumentos de entrada
    inválidos:
        id: propriedades
        invalidInput:
            código:
                tipo: string
            mensagem:
                tipo: string
    """

```

E agora você pode recarregar sua IU do Swagger e deverá ser capaz de veja todos os endpoints (Figura 4-21) documentados.

Capítulo 4 Aplicação CRUD com Flask (Parte 2)



Figura 4-21. Recarregue a UI do Swagger para ver os endpoints

Conclusão

Para este capítulo, criaremos documentação apenas para os endpoints fornecidos, e você poderá desenvolver a partir dela usando as mesmas metodologias, o que o ajudará a criar documentação completa para seus endpoints REST.

No próximo capítulo, discutiremos o teste de nossos endpoints REST e abordaremos tópicos como testes de unidade, simulações, cobertura de código e assim por diante.

CAPÍTULO 5

Teste em frasco

Algo que não foi testado está quebrado.

Esta citação vem de uma fonte desconhecida; no entanto, não é totalmente verdade, mas a maior parte está certa. Aplicativos não testados são sempre uma aposta insegura. Embora os desenvolvedores estejam confiantes em seu trabalho, no mundo real as coisas funcionam de maneira diferente; portanto, é sempre uma boa ideia testar o aplicativo por completo. Aplicativos não testados também dificultam o aprimoramento do código existente. No entanto, com testes automatizados, é sempre fácil fazer alterações e saber instantaneamente quando algo quebra. Portanto, os testes não apenas garantem se o aplicativo está se comportando da maneira esperada, mas também facilitam o desenvolvimento contínuo.

Este capítulo aborda testes unitários automatizados de APIs REST e, antes de entrarmos na implementação real, veremos o que é teste unitário e os princípios por trás dele.

Introdução

A maioria dos desenvolvedores de software geralmente já está familiarizada com o termo “teste unitário”, mas para aqueles que não estão, o teste unitário gira em torno do conceito de quebrar um grande conjunto de código em unidades individuais para serem testadas isoladamente. Normalmente, nesse caso, um conjunto maior de código é o software e os componentes individuais são as unidades a serem testadas isoladamente. Assim, no nosso caso, uma única solicitação de API é uma unidade a ser testada. O teste unitário é o primeiro nível de desenvolvimento de software e geralmente é feito por desenvolvedores de software.

Capítulo 5 Teste em Flask

Vejamos alguns benefícios do teste de unidade:

1. Os testes unitários são testes simples para um bloco de código muito restrito, servindo como um bloco de construção do espectro maior de testes de aplicativos.
2. Com escopo restrito, os testes unitários são os mais fáceis de escrever e implementar.
3. Os testes de unidade aumentam a confiança na modificação do código e também são o primeiro ponto de falha se implementados corretamente, alertando o desenvolvedor sobre partes da lógica que quebram o aplicativo.
4. Escrever testes unitários torna o processo de desenvolvimento mais rápido, pois faz com que os desenvolvedores façam menos testes confusos e os ajuda a detectar os bugs mais cedo.
5. Detectar e corrigir bugs no desenvolvimento usando testes de unidade é mais fácil e mais barato do que fazer isso depois que o código é implantado na produção.
6. Os testes unitários também são uma forma mais confiável de testar em contraste com testes fuzz manuais.

Configurando testes unitários

Portanto, nesta seção, iremos direto à ação e começaremos a implementar os testes; para o mesmo, usaremos uma biblioteca chamada unittest2 que é uma extensão da estrutura de teste de unidade original do Python chamada unittest.

Vamos instalar a biblioteca primeiro.

```
(venv)$ pip instalar unittest2
```

Capítulo 5 Teste em Flask

Isso instalará o unittest2 para nós; a seguir configuraremos uma classe de teste base que importaremos em todos os nossos arquivos de teste. Esta classe base configurará a base para os testes e iniciará o cliente de teste como o nome sugere. Então vá em frente e crie um arquivo chamado test_base.py na pasta utils.

Agora vamos configurar nosso ambiente de testes, então abra seu config.py e adicione o seguinte código para adicionar configuração de teste.

```
classe TestingConfig(Config):
    TESTE = Verdadeiro
    SQLALCHEMY_ECHO = Falso
    JWT_SECRET_KEY = 'JWT-SECRET'
    SECRET_KEY= 'SECRET-KEY'
    SECURITY_PASSWORD_SALT= 'SENHA-SAL'
    MAIL_DEFAULT_SENDER=
    MAIL_SERVER= 'smtp.gmail.com'
    MAIL_PORT=465
    MAIL_USERNAME=
    MAIL_PASSWORD=
    MAIL_USE_TLS= Falso
    MAIL_USE_SSL= Verdadeiro
    UPLOAD_FOLDER= 'imagens'
```

Observe que não configuraremos o URI SQLAlchemy aqui, o que faremos em test_base.py

Em seguida, adicione as seguintes linhas para importar as dependências necessárias em test_base.py

```
importar unittest2 como unittest
da importação principal create_app
de api.utils.database importar banco de dados
de api.config.config importar TestingConfig
importar arquivo temporário
```

Capítulo 5 Teste em Flask

Em seguida, adicione a classe BaseTestCase com o código a seguir.

classe BaseTestCase(unittest.TestCase):

 """Um caso de teste básico"""

 def configuração(self):

 app = create_app(TestingConfig)

 self.test_db_file=tempfile.mkstemp()[1]

 app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:/// ' + self.test_db_file

 com app.app_context():

 db.create_all()

 app.app_context().push()

 self.app = app.test_client()

 def tearDown(self):

 db.session.close_all()

 db.drop_all()

Aqui estamos criando o banco de dados SQLAlchemy sqlite dinamicamente usando tempfile.

O que acabamos de criar anteriormente é chamado de stub, que é um módulo que atua como um substituto temporário para um módulo chamado, fornecendo a mesma saída do produto real.

Portanto, o método anterior será executado antes de cada teste ser executado e gerará um novo cliente de teste. Importaremos este método em todos os testes que criarmos. Um teste é reconhecido por todos os métodos da classe que começa com o prefixo test_. Aqui teremos um URL de banco de dados exclusivo sempre que configurarmos o arquivo temporário, e iremos corrigi-lo com um carimbo de data e hora e então configuraremos TESTING= True na configuração do aplicativo que desabilitará a captura de erros para permitir melhores testes e, finalmente, executaremos db.create_all() para criar as tabelas de banco de dados para o aplicativo.

A seguir definimos outro método tearDown que removerá

o arquivo de banco de dados atual e use um novo arquivo de banco de dados para cada teste.

Endpoints de usuário de teste de unidade

Então agora vamos começar a escrever os testes, e o primeiro passo é criar uma pasta chamada testes no diretório api onde criaremos todos os nossos arquivos de teste. Então vá em frente e crie a pasta de testes e crie nosso primeiro arquivo de teste chamado test_users.py.

Agora adicione as seguintes importações em test_users.py

```
importar JSON
de api.utils.test_base importar BaseTestCase
de api.models.users importar usuário
de data e hora importar data e hora
importar unittest2 como unittest
de api.utils.token importar generate_verification_token, confirm_verification_token
```

Feito isso, definiremos outro método para criar usuários usando o modelo SQLAlchemy para facilitar os testes.

Adicione isso ao arquivo a seguir.

```
def criar_usuários():
    usuário1 = Usuário(email="kunal.relan12@gmail.com",
                       nomedeusuario='kunalreln12',
                       senha=User.generate_hash('helloworld'), isVerified=True).create()

    usuário2 = Usuário(email="kunal.relan123@gmail.com",
                       nomedeusuario='kunalreln125',
                       senha=User.generate_hash('helloworld')).create()
```

Agora temos nossas importações e o método para criar usuários; a seguir vamos definir a classe TestUsers para realizar todos os nossos testes.

```
classe TestUsers (BaseTestCase):
    def configuração(self):
        super(TestUsers, self).setUp()
```

Capítulo 5 Teste em Flask

```
criar_usuários()

se __nome__ == '__principal__':
    unittest.main()
```

Adicione este código ao arquivo que irá importar nossa classe de teste base e configurar o cliente de teste e chamar o método `create_users()` para criar os usuários. Observe que no método `create_users()`, criamos um usuário verificado e um não verificado para que possamos cobrir todos os casos de teste. Agora podemos começar a escrever nossos testes unitários. Adicione o seguinte código dentro da classe `TestUsers()`.

Começaremos testando o endpoint de login e, como acabamos de criar um usuário verificado, devemos ter permissão para fazer login com um conjunto válido de credenciais.

```
def test_login_user(self):
    usuário = {
        "e-mail": "kunal.relan12@gmail.com",
        "senha": "olá mundo"
    }
    resposta = self.app.post(
        '/api/usuários/login',
        dados=json.dumps(usuário),
        content_type='aplicativo/json'
    )
    dados = json.loads(response.data)
    self.assertEqual(200, resposta.status_code)
    self.assertTrue('access_token' in dados)
```

Adicione o seguinte código dentro da classe `TestUsers` e devemos ter nosso primeiro teste de unidade no qual criamos um objeto de usuário e postamos o usuário no endpoint de login. Assim que recebemos a resposta, usaremos a asserção para verificar se obtivemos o código de status esperado e o `access_token` na resposta. Uma afirmação é uma expressão booleana que será verdadeira a menos que haja um bug ou a instrução condicional não corresponda. O teste unitário fornece uma lista de métodos de asserção que podemos usar para validar nossos testes.

Capítulo 5 Teste em Flask

Mas `assertEqual()`, `assertNotEqual()`, `assertTrue()` e `assertFalse()` cobrir a maior parte.

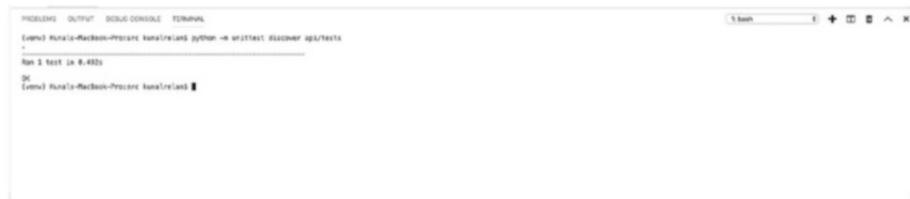
Aqui `assertEqual()` e `assertNotEqual()` correspondem aos valores, e `assertTrue()` e `assertFalse()` verificam se o valor da variável passada é um booleano.

Agora vamos fazer nosso primeiro teste, então basta abrir seu terminal e ativar seu ambiente virtual.

Em seu terminal execute o seguinte comando para executar os testes.

```
(venv)$ python -m unittest descobrir api/testes
```

O comando anterior executará todos os arquivos de teste dentro do diretório de testes; como temos apenas um teste por enquanto, podemos ver o resultado de nossos testes na figura a seguir.



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
[venv] Hunal@MacBook-Pro:~/Projetos/narizrelax$ python -m unittest discover api/testes
1 test in 0.482s
[OK]
[venv] Hunal@MacBook-Pro:~/Projetos/narizrelax$
```

Figura 5-1. Executando testes unitários

Então essa foi uma forma de executar nossos testes unitários, e antes de processarmos Além de escrever mais testes, gostaria de apresentar outra extensão para a biblioteca `unittest` chamada `nose`, que torna os testes mais fáceis, então vamos em frente e instalar o `nose`.

Use o código a seguir para instalar o `nose`.

```
(venv)$ pip instalar nariz
```

E agora que tivermos o `nose`, vamos ver como podemos usá-lo para executar nossos testes, já que, seguindo em frente, usaremos o `nose` para executar todos os nossos testes.

Capítulo 5 Teste em Flask

Por padrão, o nose encontrará todos os arquivos de teste usando uma expressão regular (? : \b|_|[Tt]est); entretanto, você também pode especificar o nome do arquivo a ser testado. Vamos fazer o mesmo teste novamente usando o nariz.

```
(venv)$ testes de nariz
```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
(venv) Kunal's-MacBook-Pro:src kunal$ nose test
.
Ran 1 test in 0.523s
OK
(venv) Kunal's-MacBook-Pro:src kunal$
```

Figura 5-2. Executando testes de unidade com nariz

Como você pode ver na figura anterior, podemos executar nossos testes usando um simples comando nosetests. A seguir, vamos escrever testes unitários para o modelo do usuário novamente.

Portanto nosso objetivo aqui é cobrir todos os cenários e verificar o comportamento da aplicação em cada um dos cenários; a seguir testaremos a API de login quando o usuário não for verificado e quando credenciais erradas forem enviadas.

Adicione o seguinte código para os respectivos testes.

```
def test_login_user_wrong_credentials(self):
    user = { "email": "kunal.relan12@gmail.com", "senha": "helloworld12" }

    resposta = self.app.post( '/api/
        users/login',
        data=json.dumps(usuario),
        content_type='application/json'

    ) dados = json.loads(response.data)
    self.assertEqual(401, response.status_code)
```

```
def test_login_unverified_user (self): usuário =
    { "email":
        "kunal.relan123@gmail.com", "senha": "helloworld"

    } resposta = self.app.post( '/api/
        users/login',
        data=json.dumps(usuário),
        content_type='application/json'

    ) dados = json.loads(response.data)
    self.assertEqual(400, response.status_code)
```

No código anterior, no método `test_login_user_wrong_credentials`, verificamos o código de status 401 na resposta, pois estamos fornecendo credenciais erradas e, no método `test_login_unverified_user()`, estamos tentando fazer login com um usuário não verificado, o que gerará o erro 400.

A seguir, vamos testar o endpoint `create_user` e começar criando um teste para crie um usuário com campos corretos para criar um novo usuário.

```
def test_create_user (self): usuário
    = { "nome
        de usuário": "kunalrelan2", "senha":
        "helloworld", "email":
        "kunal.relan12@hotmail.com"

    }

resposta = self.app.post( '/api/
    users/',
    data=json.dumps(usuário),
    content_type='application/json'

)
```

Capítulo 5 Teste em Flask

```

dados = json.loads(response.data)
self.assertEqual(201, resposta.status_code)
self.assertTrue('sucesso' in dados['codigo'])

```

O código anterior solicitará o endpoint Criar usuário com um novo objeto de usuário e poderá fazê-lo e responder com um código de status 201.

A seguir, adicionaremos outro teste quando o nome de usuário não for fornecido ao endpoint Criar usuário e, neste caso, obteremos uma resposta 422. Aqui está o código para isso.

```

def test_create_user_without_username(self):
    usuário = {
        "senha": "olá mundo",
        "e-mail": "kunal.relan12@hotmail.com"
    }

    resposta = self.app.post(
        '/api/usuários/',
        dados=json.dumps(usuário),
        content_type='aplicativo/json'
    )
    dados = json.loads(response.data)
    self.assertEqual(422, resposta.status_code)

```

Agora podemos prosseguir para testar nosso endpoint de confirmação de e-mail, e aqui primeiro criaremos um teste de unidade com e-mail válido, então você notou que tivemos um usuário não verificado criado no método `create_users()`, e aqui primeiro geraremos uma validação token, pois não estamos lendo o email usando os testes de unidade e, em seguida, enviamos o token para confirmar o endpoint do email.

```

def test_confirm_email(self):
    token = generate_verification_token('kunal.relan123@'
                                         'gmail.com')

```

```
resposta = self.app.get( '/api/
    users/confirm/'+token

) dados = json.loads(response.data)
self.assertEqual(200, response.status_code)
self.assertTrue('sucesso' em dados['código'])
```

A seguir, escreveremos outro teste com e-mail de um usuário já verificado para testar se obtemos 422 no código de status de resposta.

```
def test_confirm_email_for_verified_user(self):
    token = generate_verification_token('kunal.relan12@gmail.com')

    resposta = self.app.get( '/api/
        users/confirm/'+token

) dados = json.loads(response.data)
self.assertEqual(422, response.status_code)
```

E a última para este endpoint é que forneceremos um e-mail incorreto e devemos obter um código de status de resposta 404.

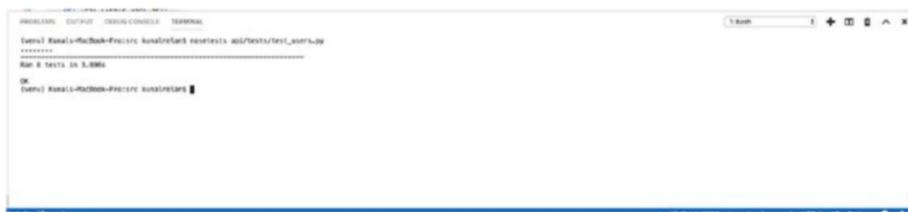
```
def test_confirm_email_with_incorrect_email(self):
    token = generate_verification_token('kunal.relan43@gmail.com')

    resposta = self.app.get( '/api/
        users/confirm/'+token

) dados = json.loads(response.data)
self.assertEqual(404, response.status_code)
```

Assim que tivermos nossos testes implementados, é hora de testar todos eles, então vá em frente e use nosetests e execute os testes.

Capítulo 5 Teste em Flask

**Figura 5-3.** Nosetests em *test_users.py*

Esses são todos os testes que queremos cobrir com o modelo de usuário; a seguir podemos passe para autores e livros.

A seguir vamos criar *test_authors.py* e adicionaremos as dependências com algumas alterações, então adicione as linhas a seguir para importar as dependências necessárias.

```
importar JSON
de api.utils.test_base importar TestCase
de api.models.authors importar Autor
de api.models.books importar livro
de data e hora importar data e hora
de flask_jwt_extended importação create_access_token
importar unittest2 como unittest
importar io
```

A seguir definiremos dois métodos auxiliares, a saber, *create_authors* e *faz login* e adicione o seguinte código para o mesmo.

```
def criar_autores():
    autor1 = Autor(primeiro_nome="John", último_nome="Doe").create()
    autor2 = Autor(primeiro_nome="Jane", último_nome="Doe").create()
```

Criaremos dois autores para o teste usando o método definido anteriormente, e o método *login* irá gerar um token de *login* e retornar apenas para rotas autorizadas.

```
def login():
    access_token = create_access_token(identity = 'kunal.relan@ hotmail.com') return
    access_token
```

A seguir, vamos definir nossa classe de teste como fizemos anteriormente e iniciá-la.

```
classe TestAuthors(BaseTestCase): def
    setUp(self):
        super(TestAuthors, self).setUp()
        create_authors()

if __name__ == '__main__':
    unittest.main()
```

Agora temos a base dos nossos testes unitários de autor e podemos adicionar o seguintes casos de teste que devem ser autoexplicativos.

Aqui criaremos um novo autor usando o endpoint do autor POST com o token JWT que geramos usando o método de login e esperaremos o objeto do autor com o código de status 201 em resposta.

```
def test_create_author(self): token =
    login() autor =
        { 'primeiro_nome': 'Johny',
          'sobrenome': 'Doe' }

    } resposta = self.app.post( '/api/
        authors/',
        data=json.dumps(autor),
        content_type='application/json', headers=
        { 'Autorização': 'Bearer '+token }
    )
```

Capítulo 5 Teste em Flask

```
dados = json.loads(response.data)
self.assertEqual(201, response.status_code)
self.assertTrue('autor' in dados)
```

Aqui tentaremos criar um autor com cabeçalho de autorização, e isso deve retornar 401 no código de status de resposta.

```
def test_create_author_no_authorization(self):
    autor = {
        'primeiro_nome': 'Johny',
        'sobrenome': 'Doe'
    }

    resposta = self.app.post('/api/
        autores/',
        data=json.dumps(autor),
        content_type='application/json',

    )
```

dados = json.loads(response.data)
self.assertEqual(401, response.status_code)

Neste caso de teste, tentaremos criar um autor sem o campo last_name, e deve responder com o código de status 422.

```
def test_create_author_no_name(self):
    token = login()
    autor = {
        'first_name':
            'Johny'
    }

    resposta = self.app.post('/api/
        autores/',
        data=json.dumps(autor),
        content_type='application/json',
```

```
headers= { 'Autorização': 'Bearer '+token }
```

```
) dados = json.loads(response.data)
self.assertEqual(422, response.status_code)
```

Neste testaremos o endpoint de upload de avatar e usaremos io para criar um arquivo de imagem temporário e enviá-lo como multipart/form-data para fazer upload da imagem.

```
def test_upload_avatar(self): token =
    login() resposta =
    self.app.post(
        '/api/authors/avatar/2',
        data=dict(avatar=(io.BytesIO(b'test'),
        'test_file.jpg')),
        content_type='multipart/form-data', headers=
        { ' Autorização': 'Portador' + token }

) self.assertEqual(200, resposta.status_code)
```

Aqui, testaremos o avatar de upload fornecendo um arquivo CSV e, como esperado, ele não deverá responder com o código de status 200.

```
def test_upload_avatar_with_csv_file(self): token =
    login() resposta =
    self.app.post(
        '/api/authors/avatar/2',
        data=dict(file=(io.BytesIO(b'test'), 'test_file.csv')), content_type='multipart/
        form-data', headers= { 'Autorização ': 'Portador'
        + token }

) self.assertEqual(422, resposta.status_code)
```

Capítulo 5 Teste em Flask

Neste teste, obteremos todos os autores usando o endpoint GET todos os autores.

```
def test_get_authors(self):
    resposta = self.app.get( '/api/
        authors',
        content_type='application/json'

    ) dados = json.loads(response.data)
    self.assertEqual(200, response.status_code)
    self.assertTrue('autores' em dados)
```

Aqui temos um teste de unidade para GET autor por endpoint de ID e ele retornará 200 códigos de status de resposta e objeto de autor.

```
def test_get_author_detail(self): resposta
    = self.app.get( '/api/authors/
        2',
        content_type='application/json' )

    dados = json.loads(response.data)
    self.assertEqual(200, response.status_code)
    self.assertTrue('autor' em dados)
```

Neste teste atualizaremos o objeto autor no recém-criado autor, e também deve retornar o código de status 200 na resposta.

```
def test_update_author(self): token
    = login() autor =
    { 'first_name':
        'Joseph'

    } resposta = self.app.put( '/api/
        authors/2',
        data=json.dumps(autor),
```

```
content_type='application/json', headers=
{ 'Autorização': 'Bearer '+token }

) self.assertEqual(200, resposta.status_code)

Neste teste, excluiremos o objeto autor e esperaremos o código de status de resposta 204.

def test_delete_author(self): token =
    login() resposta =
        self.app.delete( '/api/authors/2',
            headers= { 'Autorização':
            'Bearer '+token }

) self.assertEqual(204, resposta.status_code)
```



Figura 5-4. Teste de autores

Então agora você pode executar o teste dos autores como na figura anterior, e tudo deve passar como naquela figura; a seguir passaremos para o teste de modelo de livros.

Para testes de modelo de livros, podemos modificar os testes de autor e configurar testes unitários para livros no mesmo módulo, então vamos atualizar o método `create_authors` para criar alguns livros também; vá em frente e atualize o método com o código a seguir.

```
def create_authors(): autor1
    = Autor(first_name="John", last_name="Doe").criar()
```

Capítulo 5 Teste em Flask

```
Livro(title="Livro de Teste 1", ano=datahora(1976, 1, 1),
autor_id=autor1.id).create()
Livro(title="Livro de Teste 2", ano=datahora(1992, 12, 1),
autor_id=autor1.id).create()

autor2 = Autor(first_name="Jane", last_name="Doe").criar()

Livro(title="Livro de Teste 3", ano=datahora(1986, 1, 3),
autor_id=autor2.id).create()
Livro(title="Livro de Teste 4", ano=datahora(1992, 12, 1),
autor_id=autor2.id).create()
```

E aqui estão os testes unitários para rotas de livros.

```
def test_create_book(self): token
    = login() autor =
    { 'título':
        'Alice no país das maravilhas', 'ano':
        1982, 'autor_id':
        2
    }

    resposta = self.app.post( '/api/
        books/',
        data=json.dumps(autor),
        content_type='application/json',
        headers= { 'Autorização': 'Bearer '+token }

    ) dados = json.loads(response.data)
    self.assertEqual(201, response.status_code)
    self.assertTrue('livro' em dados)

def test_create_book_no_author(self): token
    = login()
```

```
autor =  
    { 'título': 'Alice no país das  
    maravilhas', 'ano': 1982  
}  
  
resposta = self.app.post( '/  
    api/books/',  
    data=json.dumps(autor),  
    content_type='application/json',  
    headers= { 'Autorização': 'Bearer '+token }  
  
) dados = json.loads(response.data)  
self.assertEqual(422, response.status_code)  
  
def test_create_book_no_authorization(self):  
    autor =  
        { 'título': 'Alice no país das  
        maravilhas',  
        'ano': 1982, 'autor_id': 2  
}  
  
    resposta = self.app.post( '/  
        api/books/',  
        data=json.dumps(autor),  
        content_type='application/json'  
  
) dados = json.loads(response.data)  
self.assertEqual(401, response.status_code)  
  
def test_get_books(self):  
    resposta = self.app.get( '/  
        api/books/',  
        content_type='application/json'  
)
```

Capítulo 5 Teste em Flask

```
dados = json.loads(response.data)
self.assertEqual(200, response.status_code)
self.assertTrue('livros' in dados)

def test_get_book_details(self): resposta
    = self.app.get( '/api/books/2',
        content_type='application/json' )

    dados = json.loads(response.data)
    self.assertEqual(200, response.status_code)
    self.assertTrue('livros' in dados)

def test_update_book(self): token
    = login() autor =
    { 'ano': 1992,
        'título': 'Alice'

    } resposta = self.app.put( '/api/
        books/2',
        data=json.dumps(autor),
        content_type='application/json',
        headers= { 'Autorização': 'Bearer '+token }

    ) self.assertEqual(200, resposta.status_code)

def test_delete_book(self): token
    = login() resposta
    = self.app.delete(
        '/api/livros/2',
```

```
        headers= { 'Autorização': 'Bearer '+token }
    )
self.assertEqual(204, resposta.status_code)
```

Cobertura de teste

Agora aprendemos a escrever casos de teste para nossa aplicação, e o objetivo dos testes de unidade é testar o máximo de código possível, por isso temos que garantir que todas as funções com todas as suas ramificações sejam cobertas, e quanto mais perto você chegar 100%, mais confortável você ficará antes de fazer alterações.

A cobertura de testes é uma ferramenta importante para uso no desenvolvimento; no entanto, 100% de cobertura não garante a ausência de bugs.

Você pode instalar ocoverge.py usando PIP com o seguinte comando.

```
(venv)$ cobertura de instalação do pip
```

A biblioteca Nose possui um plugin integrado que funciona com o módulo de cobertura, portanto, para executar a cobertura de teste, você precisa adicionar mais dois parâmetros ao terminal enquanto executa os testes.

Use o seguinte comando para executar nosetests com a cobertura de teste habilitada.

```
(venv)$ nosetests --with-coverage --cover-package=api.routes
```

Então, aqui estamos habilitando a cobertura usando o sinalizador --with-coverage e especificando para cobrir apenas o módulo de rotas, ou então, por padrão, também cobrirá os módulos instalados.

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
[Terminal] Macintosh-MacBook-Pro:mcsej:~/Desktop/nose$ nosetests --with-coverage --cover-package=api.routes
.....coverage.py warning: Nose's application has previously reported, but not measured (decide-not-measured)
-----
api.routes._init__.py     8    8 100%
api.routes.oauth2.py     6/2   5  99%
api.routes.books.py      5/2   3  60%
api.routes.users.py      5/2   3  60%
-----TOTAL----- 388 13 96%
Ran 24 tests in 5.792s
ok
[Terminal] Macintosh-MacBook-Pro:mcsej:~/Desktop/nose$
```

Figura 5-5. Cobertura de teste

Capítulo 5 Teste em Flask

Como você pode ver, temos uma cobertura significativa de testes de código, e você pode cobrir todos os outros casos extremos para obter 100% de cobertura de teste.

Em seguida, você também pode ativar o sinalizador `--cover-html` para gerar informações em formato HTML que é mais legível e predefinido.

```
(venv)$ nosetests --with-coverage --cover-package=api.routes --cover-html
```

O comando anterior irá gerar o resultado do teste em formato HTML cobertura, e agora você deverá ver uma pasta chamada cobertura em seu diretório de trabalho; abra a pasta e abra `index.html` usando seu navegador para ver o relatório de cobertura de teste em HTML.

Como você pode ver na figura anterior, temos a versão HTML do nosso relatório de cobertura de teste.

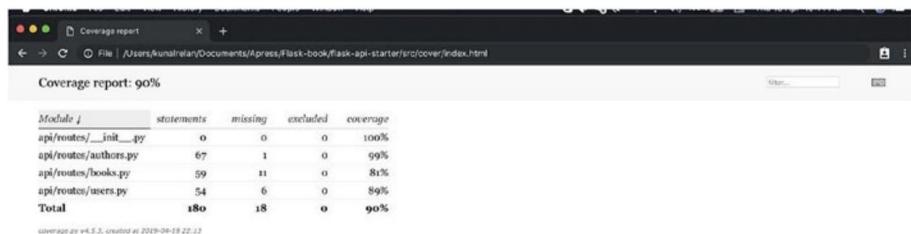


Figura 5-6. Relatório de cobertura de teste em HTML

Conclusão

Então é isso neste capítulo; aprendemos os fundamentos dos testes de unidade, implementamos casos de teste para nosso aplicativo e cobrimos testes de unidade para todas as nossas rotas e cobertura de teste integrada usando a biblioteca de testes de nariz. Isso abrange nossa jornada de desenvolvimento deste aplicativo. No próximo capítulo, discutiremos sobre a implantação e implantação de nosso aplicativo em vários provedores de serviços de nuvem.

CAPÍTULO 6

Implantando Flask Formulários

Portanto, até agora neste livro, nos concentramos inteiramente no desenvolvimento do aplicativo e, neste capítulo, discutiremos a próxima etapa que é implantar nosso aplicativo e gerenciar o aplicativo após a implantação, que é uma parte crucial do desenvolvimento do aplicativo. Neste capítulo discutiremos principalmente várias maneiras de implantar um aplicativo Flask com segurança. Pode haver várias maneiras de implantar um aplicativo Flask, e cada forma tem seus prós e contras, portanto, iremos avaliá-los e discutir sua relação custo-benefício, bem como formas de segurança e desempenho para implantar nosso aplicativo. Como mencionei anteriormente, o servidor Flask não é adequado para implantação de produção e destina-se apenas ao desenvolvimento e depuração, portanto, examinaremos várias opções.

Neste capítulo abordaremos os seguintes tópicos:

1. Implantando Flask com uWSGI e Nginx no Alibaba ECS na nuvem
2. Implantando Flask com Gunicorn no Alibaba Cloud ECS
3. Implantando Flask no Heroku
4. Implantando Flask no AWS Elastic Beanstalk
5. Implantando Flask no Google App Engine

Capítulo 6 Implantando aplicativos Flask

Portanto, neste capítulo, nos concentraremos inteiramente na implantação de nosso aplicativo em todas essas plataformas e discuta os prós e os contras de cada uma delas. Embora todas sejam ótimas opções, são inteiramente o caso de uso de negócios e os recursos que definem onde implantamos o aplicativo.

Implantando Flask com uWSGI e Nginx no Alibaba Cloud ECS

A implantação de aplicativos dessa forma costuma ser chamada de hospedagem tradicional, onde as dependências são instaladas manualmente ou por meio de um instalador com script, o que envolve a instalação manual do aplicativo e suas dependências e sua proteção. Nesta seção, instalaremos e executaremos nosso aplicativo em produção usando uWSGI e Nginx em um sistema operacional Linux hospedado no Alibaba Cloud Elastic Compute Service.

uWSGI é um servidor HTTP completo e um protocolo capaz de executar aplicativos de produção. uWSGI é um servidor uwsgi (protocolo) popular, enquanto Nginx é um servidor HTTP gratuito, de código aberto e de alto desempenho e um proxy reverso. Em nosso caso, usaremos o Nginx para fazer proxy reverso de nossas chamadas HTTP de e para o servidor uwsgi que implantaremos no sistema operacional Ubuntu.

Então, vamos direto ao assunto e implantar nosso aplicativo, mas antes de fazer isso, temos que congelar nossas bibliotecas em requisitos.txt usando pip freeze. Execute os comandos a seguir para garantir que o arquivo contenha a lista de todas as dependências necessárias.

```
(venv)$ pip congelar > requisitos.txt
```

Então aqui o pip freeze produzirá todos os pacotes instalados necessários no formato de requisitos. Em seguida, precisamos enviar nossa base de código para um sistema de gerenciamento de versão como o GitHub, que extrairemos mais tarde em nossa instância do Linux. Para isso criaremos uma instância do Ubuntu no Alibaba Cloud para a qual você pode se inscrever em www.alibabacloud.com ou você pode usar sua instância do Ubuntu em qualquer outro provedor de nuvem ou até mesmo usar um virtual.

Capítulo 6 Implantando aplicativos Flask

Portanto, antes de começarmos a implantar, também precisamos ter um servidor MySQL e, como se trata de implantar o aplicativo Flask, não abordaremos a implantação do servidor MySQL. No entanto, você pode implantar um na mesma instância ou usar um serviço de servidor MySQL gerenciado e editar os detalhes de configuração do banco de dados em config.py.

Depois de configurar sua conta na nuvem, crie uma instância do Ubuntu de preferência versão 16.04 ou superior. Aqui temos Alibaba Cloud ECS (Elastic Compute Service) e, assim que tivermos nossa instância, usaremos SSH para usar um par de chaves ou uma senha.

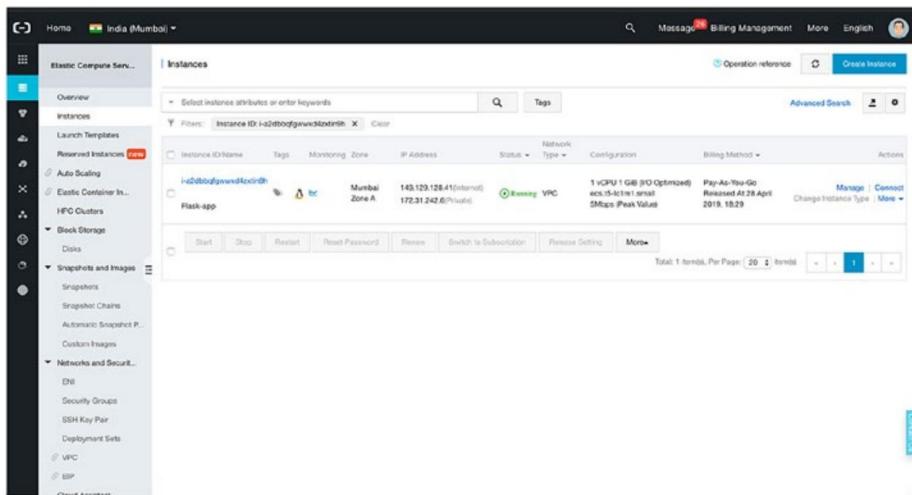


Figura 6-1. Console Alibaba Cloud ECS

Depois de ter sua instância do Ubuntu instalada e funcionando, faça SSH nela e extraia a base de código do seu sistema de gerenciamento de versão preferido.

Capítulo 6 Implantando aplicativos Flask



```

kunalrelan — root@iZt4nbqeosngqf68owp80aZ: ~ — ssh root@161.117.82.73 -P...
Kunals-MacBook-Pro:~ kunalrelan$ ssh root@161.117.82.73 -P...
The authenticity of host '161.117.82.73 (161.117.82.73)' can't be established.
ECDSA key fingerprint is SHA256:cs6oLt9NfwzSBw+DNchFFmJVTZODbLOxyMPfy9xjKIY.
Are you sure you want to continue connecting (yes/no)? yes
[Warning: Permanently added '161.117.82.73' (ECDSA) to the list of known hosts.
root@161.117.82.73's password:
Welcome to Ubuntu 16.04.6 LTS (GNU/Linux 4.4.0-146-generic x86_64)

 * Documentation: https://help.ubuntu.com
 * Management: https://landscape.canonical.com
 * Support: https://ubuntu.com/advantage

Welcome to Alibaba Cloud Elastic Compute Service !
root@iZt4nbqeosngqf68owp80aZ:~# █

```

Figura 6-2. SSH na instância do Ubuntu

Como você pode ver por padrão, efetuamos login como root, portanto, antes de mover em diante, criaremos outro usuário sudo chamado Flask, que é uma boa medida de segurança. É uma boa ideia executar cada aplicativo com sua própria conta de usuário, para limitar os danos que as vulnerabilidades de segurança no aplicativo podem causar.

\$ adduser frasco

Em seguida, você será solicitado a definir uma senha para o novo usuário e inserir alguns detalhes; você pode simplesmente inserir a senha e deixar os outros campos vazios se desejar e então executar o seguinte comando para adicionar o usuário à lista de sudoers.

\$ usermod -aG sudo frasco

Agora que tivermos nosso novo usuário, vamos fazer login com esse usuário no shell usando o seguinte comando.

\$ su - frasco

Em seguida, extrairemos nosso aplicativo de nosso repositório GitHub, portanto, certifique-se de ter o cliente git instalado e, caso não tenha, use o seguinte comando para fazer isso.

\$ sudo apt-get install git

Capítulo 6 Implantando aplicativos Flask

Use o seguinte comando para clonar o repositório do aplicativo.

```
$ sudo git clone <repo_name>
```

Em seguida, altere seu diretório atual para o código-fonte do aplicativo e instale virtualenv e uwsgi já que não os teremos em nosso requirements.txt.

```
$ sudo pip install virtualenv uwsgi
```

Crie um ambiente virtual como fizemos no capítulo anterior e instale as dependências após ativar o ambiente virtual com o seguinte comando.

```
$ pip install -r requisitos.txt
```

Instalaremos todas as dependências necessárias para configurar o aplicativo a partir de Reppositórios Ubuntu, e começaremos instalando python-pip que é um gerenciador de pacotes para Python e python-dev que contém os arquivos de cabeçalho necessários para compilar extensões Python.

```
$ sudo apt-get install python-pip python-dev
```

Assim que tivermos nossas dependências instaladas, criaremos um arquivo de configuração uWSGI que será chamado flask-app.ini, então vá em frente e crie um arquivo chamado flask-app.ini em seu diretório atual e adicione as seguintes linhas a ele.

```
[uwsgi]
módulo = executar:aplicativo
mestre = verdadeiro
processos = 5
soquete=flask-app.sock
soquete chmod = 660
vácuo = verdadeiro
morrer a termo = verdadeiro
```

Capítulo 6 Implantando aplicativos Flask

Este arquivo começa com o cabeçalho [uwsgi] para que o WSGI saiba como aplicar as configurações. Também especificamos o módulo e o chamável que é run.py em nosso caso menos a extensão e o chamável que é application.

Em seguida, instruímos o uwsgi a iniciar o processo como mestre e gerar cinco processos de trabalho para lidar com as solicitações.

A seguir, forneceremos o arquivo de soquete Unix para Nginx seguir as solicitações uWSGI para nosso aplicativo. Vamos também alterar as permissões no soquete. Daremos ao grupo Nginx a propriedade do processo uWSGI mais tarde, portanto, precisamos ter certeza de que o proprietário do grupo do soquete pode ler informações dele e gravar nele. Também limparemos o soquete quando o processo parar, adicionando a opção de vácuo.

A última coisa que faremos é definir a opção de morte a prazo. Isso pode ajudar a garantir que o sistema init e o uWSGI tenham as mesmas suposições sobre o significado de cada sinal de processo.

A seguir, criaremos um arquivo de unidade de serviço systemd que permitirá o Ubuntu init para iniciar nosso aplicativo automaticamente sempre que o servidor inicializar.

Este arquivo será chamado flask-app.service e será colocado em /etc/systemd/system

Diretório.

```
$ sudo nano /etc/systemd/system/flask-app.service
```

E cole as seguintes linhas no arquivo.

```
#Seção de metadados e dependências
[Unit]
Description=Serviço de aplicativo Flask
Depends=network.target
#Define usuários e diretório de trabalho do aplicativo
[Service]
User=frasco
Group=www-dados
```

```
WorkingDirectory=/home/flask/flask-api-app/src  
Ambiente="WORK_ENV=PROD"  
ExecStart=/home/flask/flask-api-app/src/venv/bin/uwsgi --ini flask-app.ini
```

#Vincule o serviço para iniciar no sistema multiusuário

[Instalar]

WantedBy = multiusuário.target

Depois disso, execute o seguinte comando para habilitar e iniciar nosso novo serviço.

```
$ sudo systemctl start flask-app  
$ sudo systemctl enable flask-app
```

Nosso servidor uWSGI agora deve estar funcionando, aguardando solicitações no arquivo de soquete que fizemos anteriormente. Agora instalaremos e configuraremos o Nginx para passar e processar as solicitações usando o protocolo uwsgi.

```
$ sudo apt-get install nginx
```

Agora devemos ter um servidor Nginx instalado e funcionando e começaremos criando um novo arquivo de configuração de bloco de servidor em /etc/nginx/sites-available , e o chamaremos de flask-app.

```
$ sudo nano /etc/nginx/sites-available/flask-app
```

Abriremos um bloco de servidor e o instruiremos a escutar na porta 80 e definiremos o nome do servidor que deverá ser o nome de domínio do seu serviço. A seguir definiremos um bloco location dentro do bloco server para definir o

localização base e importe cabeçalhos uwsgi_params internos que especificam alguns parâmetros uWSGI gerais que precisam ser definidos. Em seguida, passaremos as solicitações para o soquete que definimos usando a diretiva uwsgi_pass.

```
servidor {  
    ouça 80;
```

Capítulo 6 Implantando aplicativos Flask

```
nome_servidor flaskapp;  
  
localização {  
    inclua uwsgi_params;  
    uwsgi_pass unix:/home/flask/flask-api-app/src/flask-app.sock;  
  
}  
}
```

As linhas anteriores devem configurar nosso bloco server para escutar as solicitações do servidor no soquete. Assim que tivermos tudo pronto, criaremos um link simbólico para o diretório habilitado para sites.

```
$ sudo ln -s /etc/nginx/sites-available/flask-app /etc/nginx/  
habilitado para sites
```

Com isso implementado, agora podemos testar nossas alterações em busca de erros de sintaxe usando o seguinte comando.

```
$ sudo nginx -t
```

Se não houve erros de sintaxe, você deverá ver isso impresso em seu terminal.

```
$ nginx: a sintaxe do arquivo de configuração /etc/nginx/nginx.conf está ok
```

```
$ nginx: arquivo de configuração /etc/nginx/nginx.conf teste foi bem-sucedido
```

E agora, quando você acessar seu domínio, ele deverá estar funcionando. Caso você não tenha um domínio e queira acessar o aplicativo, você deve editar o bloco de servidor padrão em sites habilitados em vez de criar um novo, ou excluir o aplicativo padrão e você pode apenas acessar o aplicativo com o IP endereço.

Capítulo 6 Implantando aplicativos Flask

Certifique-se também de que, se você estiver executando o servidor Ubuntu em um serviço de nuvem, a porta 80 seja permitida no firewall e, no nosso caso, esteja configurada usando o grupo de segurança.

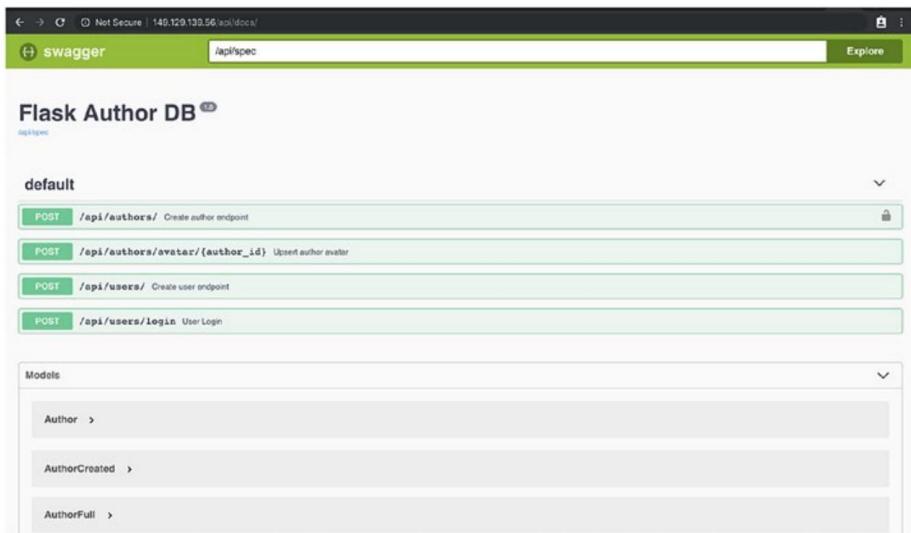


Figura 6-3. Aplicativo implantado

Implantando Flask no Gunicorn com Apache no Alibaba Cloud ECS

Agora instalaremos nosso aplicativo Flask usando Gunicorn, que é um servidor Python WSGI HTTP para Unix que executará nosso aplicativo, e então reverteremos o proxy das solicitações usando o servidor Apache.

Para seguir esta seção, você precisará do seguinte:

1. Servidor Ubuntu com usuário não root com privilégios sudo
2. Servidor Apache instalado
3. Uma cópia do nosso aplicativo Flask no diretório inicial

Capítulo 6 Implantando aplicativos Flask

Como mencionei, usaremos o Gunicorn para executar nosso aplicativo, então vamos instalar o Gunicorn usando PIP.

```
$ pip install gunicorn
```

A seguir criaremos um serviço de sistema como fizemos na seção anterior, então vá em frente e crie nosso novo serviço com o seguinte comando.

```
$ sudo nano /etc/systemd/system/flask-app.service
```

Em seguida, adicione as seguintes linhas em seu editor nano.

[Unidade]

Descrição = serviço do aplicativo Flask

Depois=network.target

[Serviço]

Usuário=frasco

Grupo=www-dados

Reiniciar = em caso de falha

Ambiente="WORK_ENV=PROD"

WorkingDirectory=/home/flask/flask-api-app/src

ExecStart=/home/flask/flask-api-app/src/venv/bin/gunicorn -c /home/flask/flask-api-app/src/gunicorn.conf -b 0.0.0.0:5000 wsgi:application

[Instalar]

WantedBy = multiusuário.target

Agora salve o arquivo e saia, e devemos ter nosso serviço de sistema. Próximo habilitaremos e iniciaremos nosso serviço usando o seguinte comando.

```
$ sudo systemctl start flask-app
```

```
$ sudo systemctl enable flask-app
```

Então agora nosso aplicativo deve estar rodando na porta 5000; em seguida precisamos configurar o Apache para fazer proxy reverso em nosso aplicativo.

Capítulo 6 Implantando aplicativos Flask

Por padrão, o módulo de proxy reverso está desabilitado no Apache e, para habilitá-lo, digite o seguinte comando.

```
$a2enmod
```

E solicitará a ativação dos módulos; entre nos seguintes módulos a ser ativado:

```
$ proxy proxy_ajp proxy_http reescrever deflacionar cabeçalhos proxy_
balanceador proxy_connect proxy_html
```

Agora adicione nosso aplicativo ao arquivo de configuração do servidor web Apache. Adicione as seguintes linhas (dentro do bloco VirtualHost) a /etc/apache2/sites-available/000-default.conf

```
<Proxy*>
    Ordem negar, permitir
    Permitir de todos
</Proxy>
ProxyPreserveHost ativado
<Local "/>
    ProxyPass "http://127.0.0.1:5000/"
    ProxyPassReverse "http://127.0.0.1:5000/"
</Local>
```

Agora ele deve estar fazendo proxy de nosso aplicativo na rota raiz e em seu servidor final bloco deve ficar assim.

```
<HostVirtual *:80>
    # A diretiva ServerName define o esquema de solicitação, nome do
    host e porta que
    # que o servidor usa para se identificar. Isto é usado quando
    criando
    # URLs de redirecionamento. No contexto de hosts virtuais, o
    Nome do servidor
```

Capítulo 6 Implantando aplicativos Flask

```
# especifica qual nome de host deve aparecer na solicitação
    Host: cabeçalho para

# corresponde a este host virtual. Para o host virtual padrão (este arquivo),
    este

# valor não é decisivo, pois é usado como host de último recurso

    sem considerar.

# No entanto, você deve configura-lo para qualquer outro host virtual
    explicitamente.

#ServerName www.example.com

Webmaster ServerAdmin@localhost
DocumentRoot /var/www/html

# Níveis de log disponíveis: trace8, ..., trace1, debug, info, warning, warning, #
    error, crit, alert,
emerg.

# Também é possível configurar o nível de log para # módulos específicos, por
exemplo

#LogLevel informações ssl:aviso

ErrorLog ${APACHE_LOG_DIR}/error.log
CustomLog ${APACHE_LOG_DIR}/access.log combinado

<Proxy*>
    Ordem negar, permitir
    Permitir de todos

</Proxy>
ProxyPreserveHost ativado
<Local "/">
    ProxyPass "http://127.0.0.1:5000/"
    ProxyPassReverse "http://127.0.0.1:5000/"
</Local>

# Para a maioria dos arquivos de configuração de conf-available/, que estão #
habilitados ou desabilitados em nível global, é possível
```

Capítulo 6 Implantando aplicativos Flask

```
# inclui uma linha para apenas um host virtual específico. Para
  exemplo, a #
linha a seguir habilita a configuração CGI para este
  apenas anfitrião
# depois de ter sido desabilitado globalmente com "a2disconf".
#Incluir conf-available/serve-cgi-bin.conf
</VirtualHost>
```

vim: syntax=apache ts=4 sw=4 sts=4 sr noet

Salve o arquivo e saia, e nós cobrimos tudo. Basta reiniciar o servidor com o seguinte comando.

```
$ sudo serviço apache2 reiniciar
```

Agora visite o aplicativo em seu navegador usando o endereço IP.

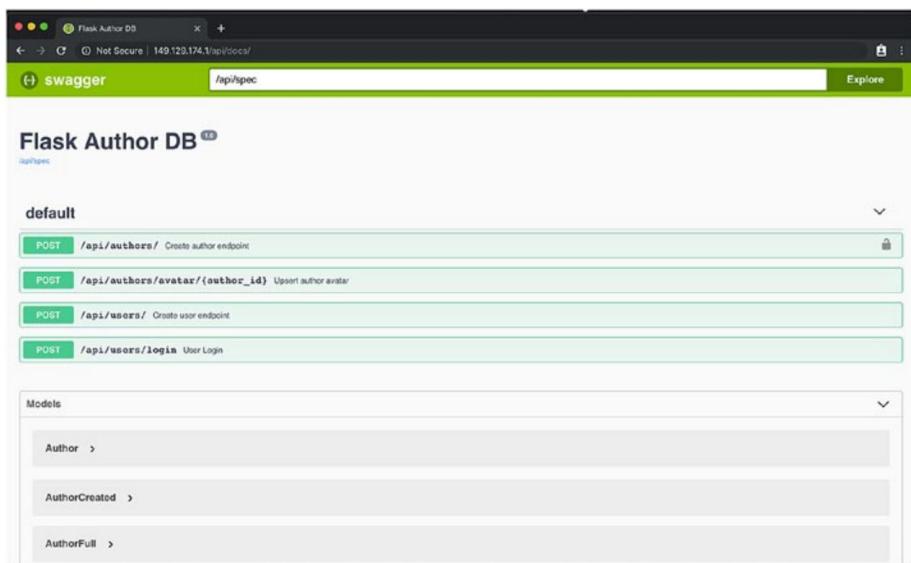


Figura 6-4. Aplicativo Flask implantado no Gunicorn

Capítulo 6 Implantando aplicativos Flask

Implantando Flask no AWS Elastic Beanstalk

Nesta seção, implantaremos nosso aplicativo Flask usando AWS Elastic Beanstalk. O AWS Elastic Beanstalk é um serviço fácil de usar para implantar e dimensionar aplicações e serviços web.

Presumiremos que você já tenha uma conta AWS ativa e uma configuração AWS CLI em sua máquina de desenvolvimento ou então você pode usar os documentos da AWS sobre isso.

Para criar o ambiente do aplicativo e implantá-lo, initialize seu repositório EB CLI com o comando eb init.

```
$ eb init -p python-2.7 flask-app --region <sua_região>
```

Nota Para obter a lista de regiões, consulte este guia: <https://docs.aws.amazon.com/general/latest/gr/rande.html>

Você deverá ver a seguinte resposta em seu terminal.

```
$ O aplicativo flask-app foi criado.
```

O comando anterior cria um novo aplicativo chamado flask-app e configura seu repositório local para criar ambientes com o Python 2.7 mais recente.

Em seguida, execute eb init novamente para configurar um par de chaves para login SSH.

A seguir, criaremos um ambiente e implantaremos seu aplicativo nele com eb criar.

```
$ eb criar
```

Em seguida, insira o nome do ambiente, o prefixo DNS e o tipo de平衡ador de carga ou que eliminará a necessidade de expor o servidor web ao mundo.

```
MacBook-Pro:Flask-App kundrel@eian: ~ eb create
Creating application version archive "app-190428_235315".
Uploading [ooooooooooooooo...oooooooooooo] 100% Done...
Environment details for: flask-app-dev
Application name: flask-app
Region: us-west-2
Deployed Version: app-190428_235315
Platform: aws.lambda.elasticbeanstalk.ap-south-1::platform:Python 2.7 running on 64bit Amazon Linux/2.8.2
Tier: WebServer-Standard-1
CPU: m1.small
Logs: /aws/lambda/app-190428_235315.log
Updated: 2018-04-28 19:30:00+00:00
Printing Status
INFO: Creating environment is starting.
INFO: Using elasticbeanshell-ap-south-1:548270294496 as Amazon S3 storage bucket for environment data.
INFO: Environment health has transitioned to Pending. Initialization in progress (waited 11 seconds). There are no instances.
INFO: Created target group named: awseb-tg-14839406-targetgroup
INFO: Created security group named: sg-03bd52321ce1c64
INFO: Created CloudWatch Metrics MetricsStream named: awseb-metrics-stream-138918219027
INFO: Created Auto Scaling launch configuration named: awseb-launchconfig-slack-AWSLambdaHelloWorld-launchConfiguration-TOLAY23PQX2
INFO: Created Auto Scaling group named: awseb-launchconfig-slack-AWSLambdaHelloWorld-autoScalingGroup-C07TQZ4QWQ
INFO: Created CloudWatch Metrics MetricsStream named: awseb-metrics-stream-154077094495
INFO: Creating Auto Scaling group policy named: awseb-launchconfig-slack-AWSLambdaHelloWorld-autoScalingPolicy-6496748F-410e-4d55-aabb-51081840bf99
INFO: Group-C07TQZ4QWQ created with policy: awseb-launchconfig-slack-AWSLambdaHelloWorld-autoScalingPolicy-6496748F-410e-4d55-aabb-51081840bf99
Group-C07TQZ4QWQ policy: awseb-launchconfig-slack-AWSLambdaHelloWorld-autoScalingPolicy-6496748F-410e-4d55-aabb-51081840bf99
INFO: Created CloudWatch Metrics MetricsStream named: awseb-metrics-stream-14839406
INFO: Created CloudWatch Metrics MetricsStream named: awseb-metrics-stream-154077094495
INFO: Created CloudWatch Metrics MetricsStream named: awseb-metrics-stream-154077094496
INFO: Created CloudWatch Metrics MetricsStream named: awseb-metrics-stream-154077094497
INFO: Created CloudWatch Metrics MetricsStream named: awseb-metrics-stream-154077094498
INFO: Created CloudWatch Metrics MetricsStream named: awseb-metrics-stream-154077094499
INFO: Created CloudWatch Metrics MetricsStream named: awseb-metrics-stream-154077094496
INFO: Created CloudWatch Metrics MetricsStream named: awseb-metrics-stream-154077094497
INFO: Created CloudWatch Metrics MetricsStream named: awseb-metrics-stream-154077094498
INFO: Created CloudWatch Metrics MetricsStream named: awseb-metrics-stream-154077094499
INFO: Your WebPath refers to a file that does not exist.
ERROR: Application available at flask-app-190428_235315.elasticbeanstalk.com.
INFO: Successfully launcher environment flask-app-dev
```

Figura 6-5. eb criar

Agora levará cerca de 5 minutos para ser implantado. Depois de implantado, só precisamos configurar mais alguns detalhes, o que faremos diretamente no console web da AWS.

```
Creating application version archive "app-190428_235315".
Uploading [ooooooooooooooo...oooooooooooo] 100% Done...
Environment details for: flask-app-dev
Application name: flask-app
Region: us-west-2
Deployed Version: app-190428_235315
Platform: aws.lambda.elasticbeanstalk.ap-south-1::platform:Python 2.7 running on 64bit Amazon Linux/2.8.2
Tier: WebServer-Standard-1
CPU: m1.small
Logs: /aws/lambda/app-190428_235315.log
Updated: 2018-04-28 19:30:00+00:00
Printing Status
INFO: Creating environment is starting.
INFO: Using elasticbeanshell-ap-south-1:548270294496 as Amazon S3 storage bucket for environment data.
INFO: Environment health has transitioned to Pending. Initialization in progress (waited 11 seconds). There are no instances.
INFO: Created target group named: awseb-tg-14839406-targetgroup
INFO: Created security group named: sg-03bd52321ce1c64
INFO: Created CloudWatch Metrics MetricsStream named: awseb-metrics-stream-138918219027
INFO: Created Auto Scaling launch configuration named: awseb-launchconfig-slack-AWSLambdaHelloWorld-launchConfiguration-TOLAY23PQX2
INFO: Created Auto Scaling group named: awseb-launchconfig-slack-AWSLambdaHelloWorld-autoScalingGroup-C07TQZ4QWQ
INFO: Created CloudWatch Metrics MetricsStream named: awseb-metrics-stream-154077094495
INFO: Creating Auto Scaling group policy named: awseb-launchconfig-slack-AWSLambdaHelloWorld-autoScalingPolicy-6496748F-410e-4d55-aabb-51081840bf99
INFO: Group-C07TQZ4QWQ created with policy: awseb-launchconfig-slack-AWSLambdaHelloWorld-autoScalingPolicy-6496748F-410e-4d55-aabb-51081840bf99
Group-C07TQZ4QWQ policy: awseb-launchconfig-slack-AWSLambdaHelloWorld-autoScalingPolicy-6496748F-410e-4d55-aabb-51081840bf99
INFO: Created CloudWatch Metrics MetricsStream named: awseb-metrics-stream-14839406
INFO: Created CloudWatch Metrics MetricsStream named: awseb-metrics-stream-154077094495
INFO: Created CloudWatch Metrics MetricsStream named: awseb-metrics-stream-154077094496
INFO: Created CloudWatch Metrics MetricsStream named: awseb-metrics-stream-154077094497
INFO: Created CloudWatch Metrics MetricsStream named: awseb-metrics-stream-154077094498
INFO: Created CloudWatch Metrics MetricsStream named: awseb-metrics-stream-154077094499
INFO: Your WebPath refers to a file that does not exist.
ERROR: Application available at flask-app-190428_235315.elasticbeanstalk.com.
INFO: Successfully launcher environment flask-app-dev
```

Figura 6-6. eb criar sucesso

Depois que a implantação do aplicativo for concluída, você deverá ver uma saída semelhante à da figura anterior. Em seguida, faça login no console web da AWS e abra a guia Configuração do Elastic Beanstalk.

Capítulo 6 Implantando aplicativos Flask

The screenshot shows the AWS Elastic Beanstalk configuration overview for an application named 'flask-app'. The interface is divided into several sections:

- Software:** AWS X-Ray disabled, Rotate logs disabled (default), Log streaming disabled (default), Status files: 1, Environment properties: 0.
- Instances:** EC2 instance type: t2.micro, EC2 image ID: ami-020a427b62fb724f, Monitoring interval: 5 minute, Root volume type: container default, Root volume size (GB): container default, Root volume IOPS: container default, Security groups: sg-0541a5c2d4d4370d8.
- Capacity:** Environment type: load balancing, auto scaling, Availability Zones: Any, Instances: 1-4.
- Load balancer:** Load balancer type: application, Listeners: 1, Processes: 1, Rules: 1.
- Rolling updates and deployments:** Deployment policy: All at once, Rolling updates: Health, Health check: enabled.
- Monitoring:** Health reporting system: Enhanced, Ignore HTTP 4xx: disabled.
- Managed updates:** Managed updates: disabled.
- Security:** Service role: arn:aws:elasticbeanstalk:service-role:Virtual machine key pair: aws-vb, Virtual machine instance profile: arn:aws:elasticbeanstalk:ec2-role.
- Notifications:** Email address: --.

Buttons at the top right include 'Cancel', 'Review changes', and 'Apply configuration'.

Figura 6-7. Configuração do aplicativo Elastic Beanstalk

Em seguida clique em modificar; na guia software e dentro das opções do contêiner, atualize o WSGIPath para run.py.

The screenshot shows the 'Modify software' configuration page for the 'flask-app' application. It includes the following settings:

- Container Options:** WSGIPath: run.py (The file that contains the WSGI application).
- Process Groups:** NumProcesses: 1 (The number of daemon processes that should be started for the process group), NumThreads: 15 (The number of threads to create to handle requests in each daemon process).
- AWS X-Ray:** X-Ray daemon: Enabled (Service charges may apply).
- S3 log storage:** Configure the instances in your environment to upload rotated logs to Amazon S3. Learn more.

At the bottom, there are 'Delete' and 'Finish' buttons.

Figura 6-8. Conjunto de caminho WSGI

Capítulo 6 Implantando aplicativos Flask

Agora role para baixo e, na variável de ambiente, forneça WORK_ENV e defina-o como Prod para que nosso aplicativo seja executado em modo de produção. Em seguida, clique em aplicar e o aplicativo deverá recarregar e começar a funcionar.

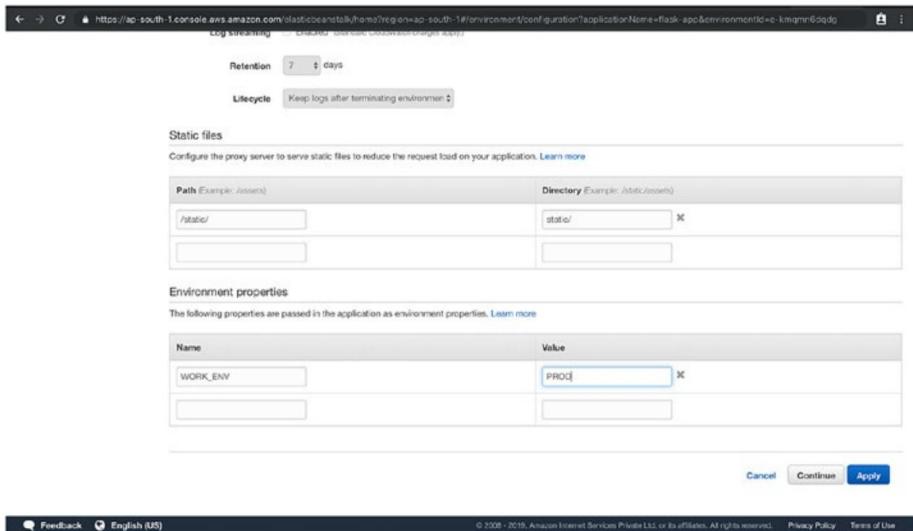


Figura 6-9. Variáveis de ambiente do Elastic Beanstalk

Agora você pode voltar ao painel para encontrar a URL do aplicativo, e deve estar instalado e funcionando.

Nota No Elastic Beanstalk você também pode configurar e iniciar um MySQL Servidor RDS conectado ao ambiente para execução com o aplicativo, mas isso está fora do escopo deste livro.

Capítulo 6 Implantando aplicativos Flask

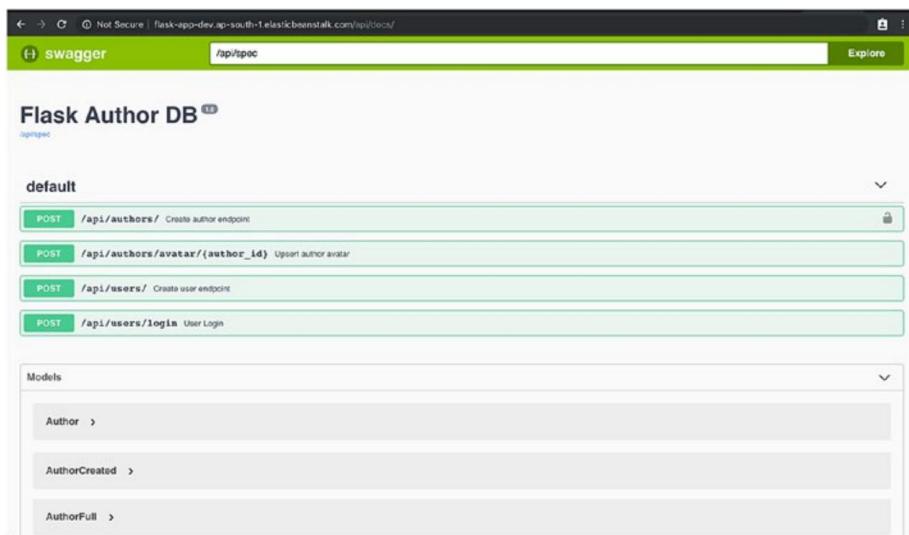


Figura 6-10. Aplicativo Flask implantado no Elastic Beanstalk

Implantando o aplicativo Flask no Heroku

Heroku é uma plataforma como serviço (PaaS) que oferece suporte a uma variedade de aplicativos modernos, fornecendo um ambiente baseado em contêiner para implantação e gerenciamento de aplicativos em escala na nuvem. A implantação de aplicativos no Heroku é bastante simples e instantânea. Você pode usar o Heroku git, conectar-se à sua conta atual do GitHub ou usar o registro de contêiner.

Aqui iremos implantar nosso aplicativo usando Heroku CLI; portanto, certifique-se de ter uma conta Heroku ativa em <https://signup.heroku.com/> que permite implantar até cinco aplicativos gratuitamente.

Você também precisará do Heroku CLI, que está disponível para download em <https://devcenter.heroku.com/articles/heroku-command-line>. Assim que tiver a CLI, faça login no Heroku CLI usando o seguinte comando, que solicitará que você forneça suas credenciais de login.

```
$ login do heroku
```

Adicionando um perfil

Para que possamos implantar com sucesso nosso aplicativo no Heroku, teremos que adicionar Procfile a esse aplicativo que define o comando a ser executado para que o aplicativo seja executado.

Com o Heroku usaremos um servidor web chamado Gunicorn, então antes de começarmos crie Procfile, instale o Gunicorn usando o seguinte comando.

```
(venv)$ pip instalar gunicorn
```

Agora atualize seu arquivo requisitos.txt usando o comando pip freeze.

```
(venv)$ pip congelar > requisitos.txt
```

Agora vamos primeiro testar se o Gunicorn está funcionando bem com nosso aplicativo; execute o seguinte comando para iniciar localmente um servidor Gunicorn.

```
(venv)$ gunicorn run:aplicativo
```

Ao executar, você deverá obter a seguinte saída em seu terminal o que implica que o servidor está funcionando bem.

```
[2019-04-29 22:54:41 +0530] [37191] [INFO] Iniciando o gunicorn  
19.9.0  
[2019-04-29 22:54:41 +0530] [37191] [INFO] Ouvindo em: http://  
127.0.0.1:8000 (37191)  
[2019-04-29 22:54:41 +0530] [37191] [INFO] Usando trabalhador: sincronização  
[2019-04-29 22:54:41 +0530] [37194] [INFO] Inicializando trabalhador com pid:  
37194
```

Nota Por padrão, o Gunicorn inicia na porta 8000.

Capítulo 6 Implantando aplicativos Flask

A seguir, crie um arquivo chamado Procfile dentro do diretório src e adicione a seguinte linha nele.

```
web: gunicorn run:aplicativo
```

Aqui Web é especificado para o Heroku iniciar um servidor web para o aplicativo. Agora só falta mais uma coisa antes de criarmos e implantarmos nosso aplicativo no Heroku. Como o Heroku por padrão usa o tempo de execução do Python 3.6, teremos que criar outro arquivo chamado runtime.txt e adicionar a seguinte linha para que o Heroku use a versão correta do Python para nosso aplicativo.

```
python-2.7.16
```

Agora estamos prontos para implantar nosso aplicativo; no diretório src do seu aplicativo, execute o seguinte comando para criar um novo aplicativo Heroku.

```
$ heroku criar <app_name>
```

Deve levar alguns segundos e você verá uma saída semelhante em seu terminal.

Criando flask-app-2019... concluído

<https://flask-app-2019.herokuapp.com/> | <https://git.heroku.com/flask-app-2019.git>

Em seguida, inicialize um novo repositório git Heroku com o seguinte comando.

```
$ git inicialização
```

```
$ heroku git:remote -a flask-app-2019
```

Agora adicione todos os arquivos e confirme o código com o seguinte comando.

```
$ git add $ .  
git commit -m "init"
```

Capítulo 6 Implantando aplicativos Flask

Antes de enviar e implantar o código, uma última coisa que precisamos fazer é definir a variável de ambiente WORK_ENV, então use o seguinte comando para fazer isso.

```
$ configuração do heroku:set WORK_ENV=PROD
```

Em seguida, precisamos enviar o código para o Heroku git e ele será implantado automaticamente.

```
$ git push mestre heroku
```

Em alguns minutos, seu aplicativo deverá estar implantado e em execução. A URL do seu aplicativo é https://<app_name>.herokuapp.com

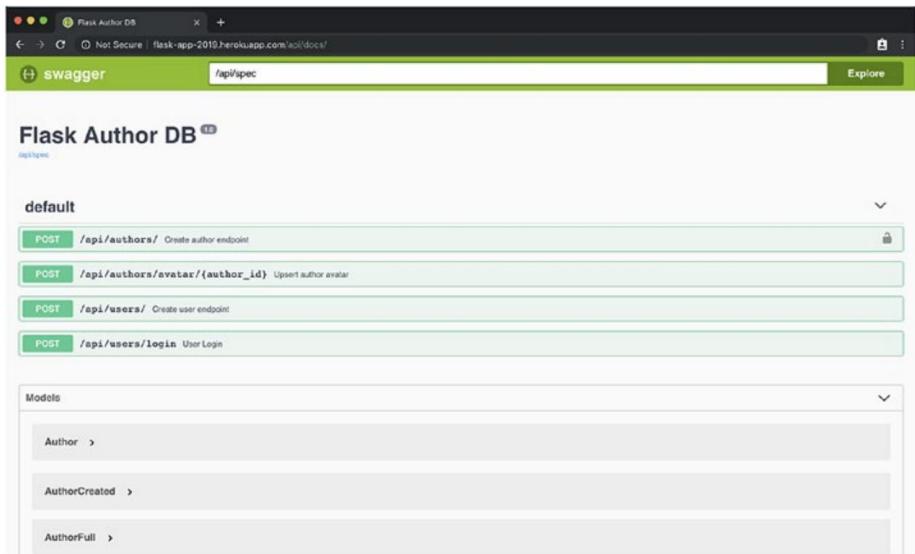


Figura 6-11. Aplicativo Flask no Heroku

Então foi isso para implantar nosso aplicativo no Heroku; você pode aprender mais sobre Heroku em <https://devcenter.heroku.com>

Capítulo 6 Implantando aplicativos Flask

Implantando o aplicativo Flask no Google App Engine

Nesta seção, implantaremos nosso aplicativo no Google Cloud App Engine, que é uma plataforma sem servidor totalmente gerenciada para implantação e escalonamento de aplicativos na nuvem. O App Engine oferece suporte a diversas plataformas, incluindo Python, e fornece um serviço totalmente gerenciado para implantação de serviços de back-end. Portanto, antes de começarmos, certifique-se de ter uma conta ativa do Google Cloud ou inscreva-se em https://cloud.google.com/products/pesquisar_aplicar/. O Google Cloud também oferece créditos de US\$ 300 por um ano.

Em seguida, instale o Google Cloud CLI usando o seguinte guia:

<https://cloud.google.com/sdk/docs/quickstarts>. Depois de configurado, execute o seguinte comando para fazer login em sua conta do Google Cloud.

```
$ login de autenticação gcloud
```

Quando tivermos sucesso, podemos começar a criar nosso aplicativo Google App Engine, mas antes de fazer isso, precisamos criar alguns arquivos de configuração.

Primeiro crie app.yaml no diretório src com o código a seguir para configurar os conceitos básicos do aplicativo para o Google App Engine.

tempo de execução: python27

versão_api: 1

threadsafe: verdadeiro

manipuladores:

-url: /avatar

static_dir: imagens

-url: /*

script: wsgi.application

bibliotecas:

-nome:ssl

versão: mais recente

variáveis_env:

WORK_ENV: PROD

Capítulo 6 Implantando aplicativos Flask

Em seguida, crie `appengine_config.py` que obterá os módulos instalados de nosso ambiente virtual para o app engine saber onde os módulos de terceiros estão instalados.

do fornecedor de importação `google.appengine.ext`

```
fornecedor.add('venv/lib/python2.7/site-packages/')
```

Agora estamos prontos para inicializar nosso aplicativo; execute o seguinte comando para o mesmo:

```
$ inicialização do gcloud
```

que solicitará que você crie o aplicativo, o nome e o projeto e selecione a região, então insira apropriadamente.

Uma vez feito isso, execute o seguinte comando para implantar seu aplicativo em Google Cloud App Engine.

Dentro de alguns minutos, ele deve estar implantado e funcionando.

Agora você pode executar o seguinte comando em seu terminal para abrir o aplicativo em seu navegador padrão.

```
$ navegação no aplicativo gcloud
```

Capítulo 6 Implantando aplicativos Flask

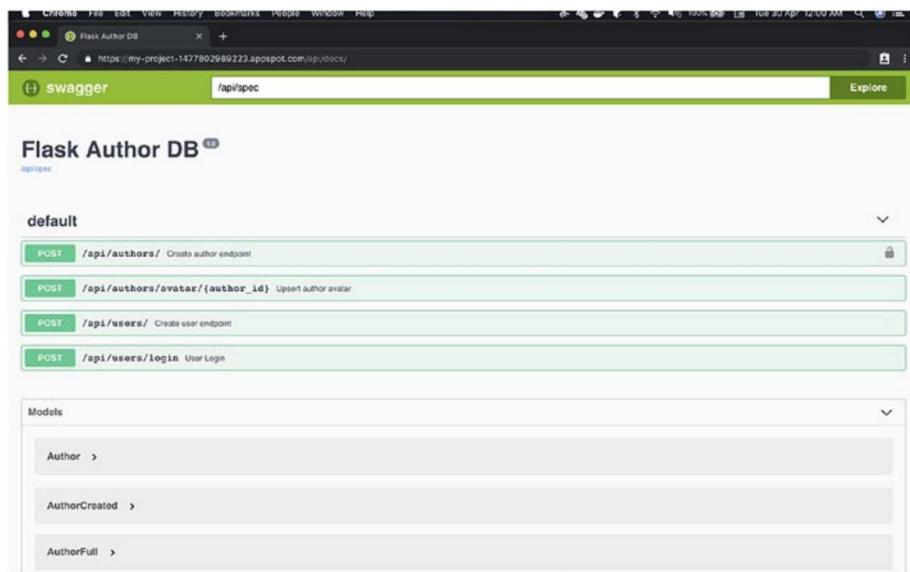


Figura 6-12. Aplicativo Flask no Google Cloud App Engine

Conclusão

Portanto, neste capítulo, implantamos nosso aplicativo em várias plataformas de nuvem usando diferentes metodologias, o que deveria ter fornecido a você uma introdução ou várias opções de implantação e escalonamento para implantar seus aplicativos Flask. No próximo capítulo, discutiremos as etapas pós-implantação para gerenciar e depurar seu aplicativo implantado.

CAPÍTULO 7

Frasco de monitoramento Formulários

Até agora, cobrimos o desenvolvimento, o teste e a implantação de nosso aplicativo flask. Neste capítulo, discutiremos alguns complementos para gerenciar e oferecer suporte ao seu aplicativo Flask e os passos a seguir a partir daqui.

Monitoramento de aplicativos

Mesmo depois de realizar vários tipos de testes em nossa aplicação, no mundo real sempre existem alguns cenários e gargalos excepcionais que desconhecemos durante o desenvolvimento, e que surgem como bugs e erros na produção, à medida que as pessoas começam a usar a aplicação. É quando precisamos de monitoramento de aplicativos que monitore o comportamento de seu aplicativo em produção, incluindo verificação de tempo de inatividade, erros de endpoint, falhas, exceções e problemas relacionados ao desempenho. O monitoramento é crucial para os aplicativos, uma vez que os arquivos de log brutos são difíceis de interpretar e tornam-se volumosos com o tempo para os desenvolvedores entenderem. Os arquivos de log podem detectar erros funcionais na maioria das vezes, mas não dizem muito sobre os problemas relacionados ao desempenho, o que também é crucial para o seu aplicativo, já que o negócio depende da capacidade do aplicativo de atender seus clientes em tempo hábil. Portanto, o monitoramento proativo de aplicativos é fundamental para lidar com estabilidade, desempenho e erros em seu a

Capítulo 7 Monitorando Aplicações Flask

Todos os principais provedores de serviços em nuvem oferecem suporte a recursos de monitoramento máquinas virtuais. Implantamos nosso aplicativo para utilização de CPU, utilização de memória e rede no nível do sistema operacional. No entanto, o monitoramento de aplicativos geralmente abrange o seguinte:

1. Erros e avisos de aplicativos
2. Desempenho do aplicativo em cada transação
3. Desempenho de consulta de integração de banco de dados e terceiros
4. Monitoramento e métricas básicas de servidor
5. Dados de registro do aplicativo

Existem muitas ferramentas excelentes de monitoramento de aplicativos no mercado e, aqui neste capítulo, abordaremos a integração de algumas delas, pois todas vêm com diferentes conjuntos de recursos, opções de pagamento e assim por diante.

Aqui está uma lista de alguns projetos de monitoramento de código aberto:

1. Sentinel
2. Sensual
3. Canário de serviço
4. Painel de monitoramento de frascos

Existem alguns serviços de monitoramento disponíveis no mercado, como New Relic, Sentry.io, Scout e assim por diante, que eliminam o fardo de implantar o software de monitoramento, mas têm um preço. Examinaremos os tipos de ferramentas de monitoramento de aplicativos.

Sentinela

Sentry é um sistema de monitoramento de aplicativos de código aberto desenvolvido em Python, mas está disponível para todas as principais plataformas. O Sentry também possui um serviço hospedado em nuvem que possui diversos modelos de assinatura que vão desde uma versão gratuita para desenvolvedores até versões empresariais que custam cerca de US\$ 80 por mês. Nisto iremos verificar a versão gratuita e integrá-la à nossa, o que é bastante simples.

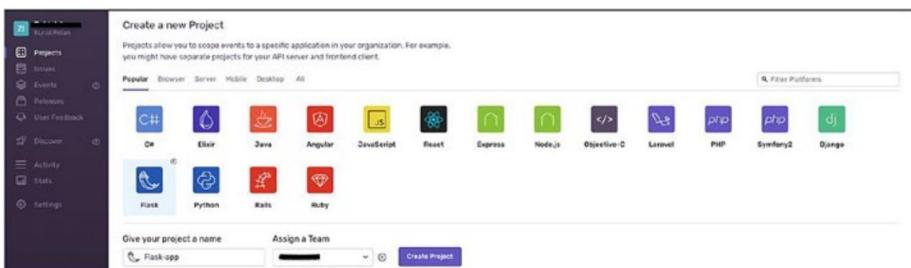


Figura 7-1. Criar novo projeto

Para começar, inscreva-se no sentinel em <https://sentry.io/signup/>, e depois de se inscrever com sucesso, faça login no seu painel. Uma vez feito isso, clique no botão adicionar projeto e selecione Flask como estrutura, insira o nome e envie ao criar o projeto.

Feito isso, nos levará para a próxima página que contém os detalhes da integração, basta instalar o sentry SDK usando PIP com o comando fornecido e copiar e colar o código de integração em seu main.py.

```
(venv)$pip instalar sentry-sdk[frasco]
```

Em seguida, adicione o seguinte código antes de app = Flask(__name__).

```
sentry_sdk.init(  
    dsn=<seu_dsn_aqui>  
    integrações=[FlaskIntegration()])  
)
```

Capítulo 7 Monitorando Aplicações Flask

Uma vez feito isso, você pode implantar o aplicativo e o sentinel cuidará do resto.

Vamos testar atingindo um endpoint 404 fazendo um evento no sentinel painel.

Em seu navegador, solicite qualquer endpoint que não exista e o sentry SDK disparará um evento do aplicativo, como na captura de tela a seguir.

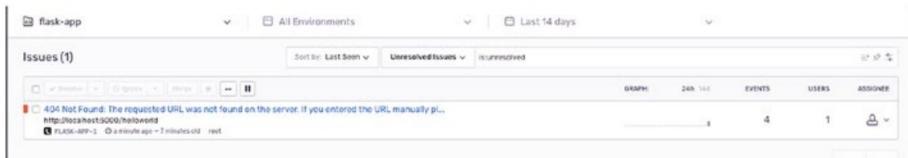


Figura 7-2. Listagem de problemas de sentinel

Clicar no problema fornecerá informações sobre o problema e detalhes para ajudá-lo a resolver o problema como na captura de tela a seguir.

This screenshot provides a detailed view of the 404 Not Found issue from Figure 7-2. The top navigation bar includes 'flask-app', 'All Environments', and a date range 'Last 14 days'. The main header for the issue is '404 Not Found: The requested URL was not found on the server. If you entered the URL ...' with URL '<http://localhost:5000/helloworld>'. Below the header, there are tabs for 'Issue #', 'Events', 'Users', and 'Assignee'. The 'Events' tab shows 4 events, all from 'FLASK-APP-1'. The 'Details' tab is selected, showing the event ID '4e465b1020232a25249d53258cb45e0244' and timestamp 'May 7, 2019 6:24:05 PM UTC JSON (3.3 KB)'. The 'Logs' tab shows a trace file with log entries. The 'Message' tab contains the error message '404 Not Found: The requested URL was not found on the server. If you entered the URL manually please check your spelling and try again.' The 'Headers' tab lists 'Accept: */*', 'Accept-Encoding: gzip, deflate', and 'Cache-Control: no-cache'. The 'Environment' tab shows 'SERVER_NAME: 0.0.0.0' and 'SERVER_PORT: 5000'. On the right side, there are sections for 'Ownership Rules', 'All Environments', 'Last 24 Hours', 'Last 30 Days', 'First Seen', 'Last Seen', 'Linked Issues', and 'Tags' (with a progress bar at 100% error).

Figura 7-3. Detalhes do problema do Sentinel

O Sentry tem muito mais recursos para você explorar e ajudar seu aplicação estável e livre de erros.

Painel de monitoramento de frascos

Flask Monitoring Dashboard é uma extensão para aplicativos Flask. Ele monitora completamente o desempenho e a utilização do aplicativo, traça o perfil das solicitações e também pode ser configurado para executar tarefas específicas para gerenciar seu aplicativo. É de código aberto e gratuito, então vamos começar.

Instale o Flask Monitoring Dashboard usando PIP com o código a seguir.

```
(venv)$pip instalar flask_monitoringdashboard
```

A seguir, em seu arquivo main.py, importe a extensão usando o código a seguir, onde temos outras importações de biblioteca.

```
importar flask_monitoringdashboard como painel
```

Agora, logo abaixo de onde iniciamos nosso objeto de aplicativo, adicione o seguinte código.

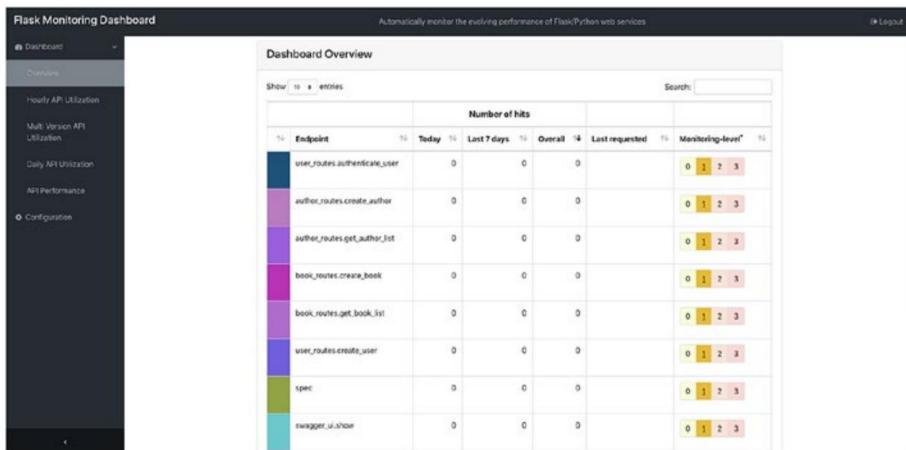
```
painel.bind(aplicativo)
```

E é basicamente isso; agora reinicie seu aplicativo e visite `http://<host>:<port>/dashboard`, e a página de login será aberta. As credenciais padrão são admin e admin, que você deve alterar logo em seguida.

Capítulo 7 Monitorando Aplicações Flask

**Figura 7-4.** Painel de monitoramento de frascos

Depois de fazer login, você será redirecionado para o painel que terá uma visão geral de seus endpoints, e você pode clicar em cada um deles para obter uma visão mais profunda e também pode definir o nível de monitoramento para cada um deles de acordo com suas preferências.

**Figura 7-5.** Visão geral do painel de monitoramento do Flask

Clicar nos endpoints irá redirecioná-lo para a página de insights de cada endpoint onde você pode produzir dados gráficos para cada solicitação e obter informações mais detalhadas sobre o uso do endpoint.



Figura 7-6. Insights de utilização de API

Nova Relíquia

New Relic é uma das ferramentas de monitoramento de aplicações mais confiáveis e competitivas do mercado; eles fornecem um banco de dados abrangente com serviços de monitoramento em tempo real. New Relic é um serviço pago baseado em assinatura, mas eles têm uma avaliação de 14 dias que usaremos para conferir. Então vá em frente e inscreva-se no mesmo em <https://newrelic.com/signup>

Depois de se inscrever, você será solicitado a escolher a plataforma; vá em frente e selecione Python como plataforma.



Figura 7-7. Configuração da Nova Relíquia

Capítulo 7 Monitorando Aplicações Flask

No diretório do seu aplicativo, vá em frente e instale o agente New Relic usando o seguinte comando.

```
(venv)$ pip install newrelic
```

Em seguida, precisamos gerar a configuração usando nossa chave New Relic, então clique no botão Revelar chave de licença na tela que exibirá sua chave privada.

Agora execute o seguinte comando com sua chave.

```
(venv)$ newrelic-admin generate-config <sua-chave-vai-aqui> newrelic.ini
```

Então agora nosso aplicativo deve estar configurado; em seguida use o seguinte comando para executar seu aplicativo e testar a configuração.

```
(venv)$ NEW_RELIC_CONFIG_FILE=newrelic.ini newrelic-admin programa de execução python run.py
```

Depois de fazer isso, clique no botão Ouvir meu aplicativo como na captura de tela a seguir, que começará a ouvir as solicitações de seu aplicativo para configurar seu painel e, assim que receber os dados com sucesso, você verá um botão para redirecionar para seu New Relic painel.

5 See data in 5 minutes

In a few minutes, your application will send data to New Relic and you'll be able to start monitoring your application's performance. You will also be automatically upgraded to New Relic PRO for a limited time.

You won't see any data in your dashboard until restart has completed.

[Listen for my application](#)

► Deployed and still not seeing your data?

Figura 7-8. New Relic Ouça o aplicativo

Capítulo 7 Monitorando Aplicações Flask

Agora você verá o painel do aplicativo que lista seus aplicativos; clique em Aplicativo Python, pois temos apenas um aplicativo aqui.



Figura 7-9. Painel do aplicativo New Relic

Depois de clicar, você será redirecionado para o painel do aplicativo, que parece bastante abrangente, mas fornece muitos insights sobre o estado do aplicativo. Consulte a captura de tela a seguir como exemplo.

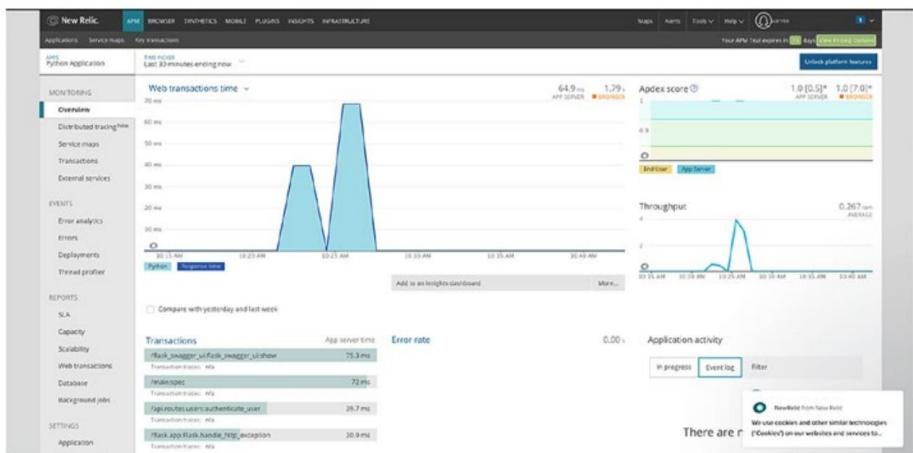


Figura 7-10. Painel do aplicativo Python

Como você pode ver, podemos verificar as transações mais recentes e o gráfico de tempo de transação e, em seguida, temos o gráfico de rendimento e a pontuação apdex que pontua o aplicativo com base em um valor definido de tempo de resposta de 0,5 segundos. Em eventos, você pode verificar análises de erros e erros que fornecem detalhes detalhados dos erros gerados no sistema.

Capítulo 7 Monitorando Aplicações Flask

Você pode aprender mais sobre o New Relic em sua documentação oficial em <https://docs.newrelic.com/docs/apm/new-relic-apm/guides>.

Serviços de bônus

Portanto, cobrimos todos os tópicos para você começar a desenvolver aplicativos REST usando Flask. No entanto, este é apenas um guia para iniciantes e há muito mais coisas a serem abordadas em casos reais de uso de negócios, incluindo integração de serviços de terceiros, como pesquisa, cache, pub-sub, comunicação em tempo real e assim por diante. Este livro cobre os fundamentos do desenvolvimento da API REST do Flask, que pode ser usado como base para construir seus aplicativos REST. Neste módulo abordaremos o básico de algumas bibliotecas e ferramentas adicionais que pode agregar valor à sua aplicação.

Pesquisa de texto completo com Flask

O Flask é compatível com algumas bibliotecas que fornecem integração com aplicativos de pesquisa de texto completo, como o Elasticsearch.

Pesquisa Pielástica

Pyelasticsearch é uma biblioteca limpa para integrar o Elasticsearch, que é um mecanismo de pesquisa muito popular e poderoso em seu aplicativo flask. No entanto, o Elasticsearch fornece endpoints REST para conexão com o mecanismo de pesquisa, mas o pyelasticsearch facilita a comunicação com os endpoints.

Você pode aprender mais sobre pyelasticsearch em <https://pyelasticsearch.readthedocs.io> e verifique este link para entender mais sobre o elasticsearch www.elastic.co/guide/en/elasticsearch/reference/current/first-steps.html

Frasco-WhooshAlquimia

Whoosh é uma biblioteca de mecanismo de pesquisa rápida construída em Python e é altamente flexível e oferece suporte a pesquisas de dados complexas com base em texto estruturado ou de formato livre. Flask-WhooshAlchemy é uma extensão do Flask para integrar a biblioteca do mecanismo de busca Whoosh com SQLAlchemy no Flask. É muito simples integrar e obter sua pesquisa de texto completa sem qualquer aplicativo de terceiros, ao contrário do pyelasticsearch. Você pode aprender mais sobre isso em <https://pythonhosted.org/Flask-WhooshAlchemy/>

E-mail

Usamos o Flask mail para verificar os e-mails dos usuários. Aqui está uma lista de outras bibliotecas que você pode usar para uma integração eficiente de email em seu aplicativo.

Flask-SendGrid

Flask-SendGrid é uma extensão do Flask para simplificar o envio de e-mail usando sendgrid, que é um serviço de e-mail renomado; você pode se inscrever no SendGrid em <https://sendgrid.com/>; eles fornecem uma assinatura gratuita que inclui 40.000 e-mails nos primeiros 30 dias e 100/dia para sempre. Você pode conferir o Flask-SendGrid em <https://github.com/frankv/flask-sendgrid> o que torna o envio de e-mail muito fácil.

AWS SNS usando Boto3

Boto3 é um AWS SDK para Python, e você pode usar o SNS (Simple Notification Service) da AWS para enviar e-mail e mensagens de texto usando o Boto. Aqui está um guia para enviar e-mail usando Python com Boto <https://docs.aws.amazon.com/ses/latest/DeveloperGuide/send-using-sdk-python.html>. Você precisará de uma conta AWS ativa para isso e do Boto instalado em <https://boto3.readthedocs.io/en/latest/guide/quickstart.html#instalação>.

Capítulo 7 Monitorando Aplicações Flask

Armazenamento de arquivo

O armazenamento de arquivos é um aspecto importante de um aplicativo; armazenamos avatares de usuários em um sistema de arquivos de servidor de aplicativos, o que não é uma boa abordagem para aplicativos de produção e, nesses casos, você gostaria de armazenar e acessar seus arquivos usando um serviço de armazenamento de arquivos. Aqui estão algumas sugestões para o mesmo.

AWS S3 usando Boto3

Você pode aproveitar o Boto3 para gerenciar seus arquivos no Amazon AWS S3, que é um poderoso sistema de gerenciamento de arquivos. Este guia fornecerá tudo que você precisa para gerenciar seus arquivos usando Flask <https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/s3.html>

Alibaba Nuvem OSS

Alibaba Cloud fornece uma plataforma sofisticada de armazenamento de arquivos chamada Object Storage Service; você pode usar o Python SDK em <https://github.com/aliyun/aliyun-oss-python-sdk> e configure facilmente o gerenciamento de arquivos usando o guia OSS, disponível em https://help.aliyun.com/document_detail/32026.html

Conclusão

Isso marca o fim deste capítulo e do livro. Há muito mais para explorar na otimização e atualização de seu aplicativo, mas isso servirá como a base certa para crescer; se você tiver problemas na integração de qualquer um dos serviços mencionados, verifique seus documentos oficiais, entre em contato com o suporte ou entre em contato com Stack Overflow para resolvê-los. Você também pode usar este link para fazer ou ler perguntas sobre o Flask <https://stackoverflow.com/pesquisar?q=frasco>.

Índice

A, B

Console Alibaba Cloud ECS, [161](#)
 Amazon AWS S3, [194](#)
 Métodos de afirmação, [140](#)

C

Cache, [13](#)
 Criar, ler, atualizar e excluir
 (CRUD), [4](#)
 O aplicativo CRUD
 adiciona respostas globais, [71, 72](#)
 Documentação da API
 blocos de construção, [115](#)
 OEA *veja* (OpenAPI
 Especificação (OEA)),
 estrutura de aplicação, [64](#)
 registro de plano de rota do
 autor, [78](#)
 blueprints, [60](#)
 modelo de criação de autores, [65,](#)
 [66](#) modelo de criação de
 livros, [64](#) criação de
 __init__.py, [62](#) criação de
 respostas.py
 dentro de api/utils, [67](#)
 criação de
 endpoint de livro POST, [79](#) função db.init_app, [70](#)

DELETE endpoints, [77, 84, 85](#)
 verificação de email
 método create_user(), [104,](#)
 [105](#)
 instalação da biblioteca
 flask-mail, [103](#) GET
 endpoint, [102](#) método
 de login, [101](#) token.py,
 [100](#)
 URLSafeTimedSerializer, [101](#) código
 de classe do usuário, [98, 99](#)
 email do usuário verificação, [108](#)
 Login do usuário após
 verificação, [108](#)
 Login de usuário sem
 verificação, [107](#) API
 de inscrição de usuário, [106,](#)
 [107](#) users.py em modelos, [98](#)
 buscar ID do autor, [76, 80](#)
 upload de
 arquivo adicionar campo de
 avatar, [109, 111](#) endpoint de avatar
 com tipo de arquivo inválido, [114](#)
 endpoint do campo avatar, [112](#)
 buscar endpoint do avatar, [113](#)
 GET endpoint do autor, [81, 82](#)
 Rota dos autores GET, [75](#)
 Ponto final GET livros, [86](#)

ÍNDICE

- Aplicação CRUD (cont.)
- Respostas HTTP, 68
 - importação create_app, 61
 - endpoint PATCH, 83 rota
 - de autor POST, 73 endpoint
 - de autores POST, 74 endpoint de
 - autor PUT, 82 autenticação de
 - usuário
 - criar rota de usuário POST, 91
 - criar esquema, 88, 89
 - Flask-JWT-Extended, ponto de extremidade de login 89 , 94
 - Rota do autor POST, 95, 96
 - Endpoint de usuários POST,
 - endpoint de inscrição de usuário 93 , 93
- D**
- Modelagem de banco de dados criando banco de dados de autor, 34 db.create_all(), 34 DELETE autor por ID, 43–45 GET todos os autores, 39 GET autor por ID endpoint, 40 GET/resposta dos autores, 36, 37 POST/authors endpoint, 38 PUT endpoint , 41, 42 APIs RESTful CRUD, 33 Flask-SQLAlchemy, 30–32 aplicativo MongoEngine
 - criar instância de banco de dados, 47, 48 criar endpoint PUT, 54, 55
- E**
- Aplicativo CRUD, 56–58 endpoint de exclusão, 55 endpoint GET, 48, 50 instalação, 47 dados JSON, 51 solicitando POST / autores, 51, 52 MySQL vs MongoDB, 29 bancos de dados NoSQL, 28 SQLAlchemy, criação de banco de dados de autor, 34 bancos de dados SQL, 28 Função db.init_app, 70 Implantando aplicativos Alibaba Cloud ECS
- Repositório GitHub, 162 Gunicórnio, 167–169, 171 Nginx, 160, 162, 164–167 erros de sintaxe, 166 Instância do Ubuntu, 160, 162 uWSGI, 160, 162, 164–167 AWS Elastic Beanstalk, 172 Google Cloud App Engine, 180–182 Heroku
 - Gunicórnio, 177-179
 - PaaS, 176
 - Perfil, 177–179
- Aplicativo Elastic Beanstalk AWS, configuração 172 , 174

aplicativo flask implantado, 176
 eb create, 173
 variáveis de ambiente, 175
 Conjunto de caminho WSGI, 174

F

método find_by_username(), 92
Frasco
 aplicativo, 2, 3
 componentes, 1–4
 instalação, 25
 ambiente de desenvolvimento
 python
 Configuração IDE, 18, 19
 Instalação PIP, 16–18 aplicativo
 virtual em execução, 23, 24 estrutura
 de
 diretório virtual, 21, 22 virtualenv, 20

SQLAlchemy, 4

Painel de monitoramento de frascos, 187
 Flask-SQLAlchemy, 30
 Frasco-WhooshAlquimia, 193

G, H

Google Cloud App Engine, 180–182

__init__.py, 62

J, K, L

Tokens JSON Web (JWT), 89

M

Monitoramento de serviços de bônus de
 aplicativos de frasco
 e-mails, 193, 194
 Frasco-WhooshAlquimia, 193
 Pyelasticsearch, painel de
 monitoramento de 192 frascos, 187, 188
 Nova Relíquia, 189–191
 Sentinel, 185–187

N

Nova Relíquia, 189

Ó

Mapeador de documentos de objetos
 (ODM), 28
 Mapeador Relacional de Objetos
 (ORM), 28
 Serviço de armazenamento de objetos (OSS), 194
 Especificação OpenAPI (OEA), 116
 Modo de solicitação de API,
 documentação de tempo de construção 124
 ponto final do avatar do autor,
 132–134
 criar endpoint do autor, 130, 131

»

ÍNDICE

Especificação OpenAPI (OAS)
 (cont.) criar
 endpoint de usuário, 126,
 127
 Swagger UI, 128
 usando YAML, 127
 definição, 116
 importando OpenAPI do Inspector,
 121 endpoint de
 login, 119 solicitações
 fixadas, 119 geração de
 especificações, 120
 SwaggerHub, 122
 Swagger Inspector , 117, 118 UI
 Swagger, 125
 visualizar documentação, 122, 123

técnica de cache, 13
 mensagens, 9–12
 planejamento, 14, 15
 representação, 8
 recursos, 12
 apatridia, 14 interface
 uniforme, 7 verbos HTTP
 usados, 8
 SOAP, 6
 resposta_com função, 70

S

Sentinela, 185
 Protocolo de acesso a objetos simples
 (SABÃO), 6
 Linguagem de consulta estruturada
 (SQL), 28

P, Q

Plataforma como serviço (PaaS), 176
 Pesquisa Pielástica, 192
 Índice de pacotes Python (PPI), 2

R

método render_template_string,
 104
 Transferência de Estado Representacional
 (DESCANSAR)
 Design de API,
 arquitetura 15, 16 ,
 6 definições, 4
 serviços

T

Método test_login_unverified_user(),
 143 Método
 test_login_user_wrong_credentials, 143

você

Benefícios de
 testes de unidade,
 configuração de 136 ,
 cobertura de testes de 136
 a 138 , endpoints de 155 usuários

- métodos de asserção, teste de [140](#) autores, [151](#), [153](#), [155](#)
- criar arquivo de imagem temporário, [149](#) criar `test_authors.py`, [146](#)
- endpoint
- `create_user`, [143](#), [144](#) método
- `create_users()`, [144](#) criar autor sem campo `last_name`, [148](#)
- arquivo CSV, [149](#) excluir objeto de autor , [151](#) gerar token de login, [146](#)
- GET ID do autor, [150](#) endpoint do autor POST, [147](#)(WSGI), [1](#)
- executando com nariz, [142](#)
- modelo SQLAlchemy, [139](#) API de login de teste, [142](#)
- classe `TestUsers()`, [140](#)
- `test_users.py`, [139](#)
- objeto de autor de atualização, [150](#)

V

`Virtualenv`, [20](#)

W X Y Z

Interface de gateway do servidor web

GET ID do autor, [150](#) endpoint do autor POST, [147](#)(WSGI), [1](#)