Manual sobre Postman e Teste de API

Propósito do documento

O Propósito deste documento é explicar como funciona a ferramenta postman e como funciona os testes de API.

Historico do Documento

Versão	Data	Autor	Descrição
1.0	06/07/2020	Kleyfferson e Tayna	Criação dos conteúdos do postman e de teste de API

Sumario

POSTMAN

1.1 HISTORIA DO POSTMAN

Os Cofundadores do postman , Abhinav Asthana e Ankit Sobti trabalhavam no Yahoo Os dois estavam construindo uma arquitetura front-end de um aplicativo, e foi aí que eles tiveram que trabalhar com APIs. Foi aí que eles perceberam que faltavam ferramentas para dar suporte ao desenvolvimento da API e sentaram-se para escrever códigos para resolver o problema e publicaram na Web Store do Chrome - e foi aí que as coisas decolaram.Porém eles encontraram um barreira nessa empreitada que era acompanhar as atualizações da API - por exemplo, um recurso adicional em uma API exigia uma atualização em todas as outras APIs relacionadas O código que Abhinav e Ankit escreveram para aliviar as dores do desenvolvimento da API logo passou de uma ideia para um negócio completo e uma empresa de gerenciamento de API com clientes como Netflix, Cisco, Microsoft, Sony e PayPal.

Fundado em Bangalore, o Postman é descrito como o conjunto completo de ferramentas para desenvolvedores de API.É uma ferramenta elegante e flexível usada para criar software conectado via APIs - de maneira rápida, eficiente e precisa.Na rodada inicial, Postman recebeu US $ 1 milhão em financiamento da Nexus Venture Partners, que também investiu outros US $ 7 milhões na rodada da Série A para a partida. "Temos um grande interesse desde que começamos como empresa, mas achamos que a Nexus era o melhor parceiro para nós", diz Asthana.

Sua Base de cliente são enormes hoje podemos citar Twiter,Pay Pal Here, Foursquare, Hero e mais 2520 mais cliente pelo mundo todo.

1.2 COMO FUNCIONA A FERRAMENTA

O Postman, é possível realizar as requisições conforme as especificações, sem termos implementado ao menos uma linha de código, desta forma, auxiliando nos testes de API's

disponibilizadas por terceiros, e para que possamos também realizar testes em nossas API's para verificar se o comportamento dela está dentro do esperado inclusive é possível realizar exportações destes testes para incluirmos essa execução em software de integração contínua para garantirmos a maior qualidade das Api's disponibilizadas.Iniciaremos com o processo de instalação do POSTMAN, alguns questionamentos que podem aparecer no ato da instalação.

- Preciso ter o Chrome para conseguir instalar?

  Não, atualmente o Postman conta com instalações nativas para Mac, Windows e Linux, link onde é possivel realizar o download https://www.getpostman.com/

- Qual a vantagem de instalar o Postman nativo ao invés do complemento do Chrome?

  Ele conta com recursos que não são disponíveis pelo complemento do Chrome:

- **Cookies** – permitem que você trabalhe com cookies diretamente;
- **Built-in proxy** – vêm com um proxy interno que você pode usar para capturar o tráfego da rede;
- **Menu de barra** – uma barra de menu mais completa, onde é possível criar coleções, alterar para o histórico de requisições entre outras funcionalidades;
- **Restricted headers** – é possível enviar nos headers da requisição Origin e User-Agent personalizados;
- **Postman Console** – tem um console embutido, que permite exibir os detalhes do pedido de rede para chamadas de API.

Algumas destas vantagens como o uso do Postman console é disponível para o aplicativo/extensão do chrome porém para isso é necessário a instalação de um outro plugin, e ainda realizar configurações para liberação do console para aplicativos/complementos do chrome.

Após a instalação do POSTMAN a tela inicial/usabilidade do POSTMAN será mostrada como na Figura 1:

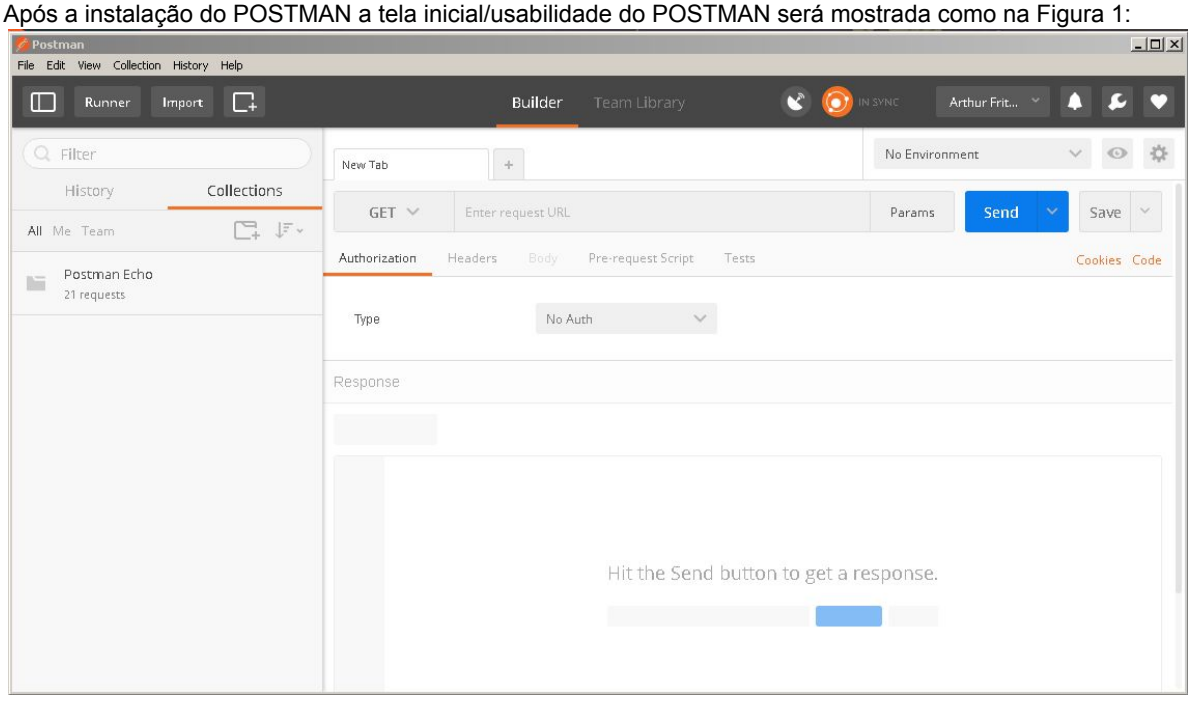

Figura 1 – Tela Inicial Postman

Como realizar minha primeira operação, podemos observar na Figura 2:

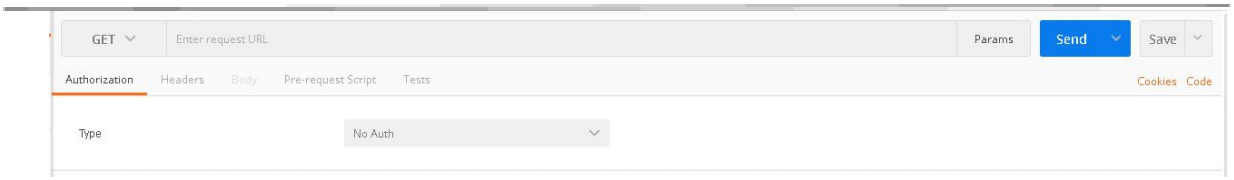

Figura 2 – Menu de requisições

No campo GET, é possível mudar o tipo da requisição que será feita para a URL que será passada no campo posterior, tendo várias operações como POST, DELETE, PUT, OPTIONS, entre outras. Antes do botão salvar no params, é possível informar parâmetros que serão passados no formato de requisições get sendo url?param1=valor1&param2=valor2, porém tendo um preenchimento mais amigável.

Abaixo temos 5 guias sendo elas:

Authorization : onde informamos o tipo de autenticação necessária e como será feita a obtenção desse token para que possa passar para a API

Headers : onde passamos todos os cabeçalhos da request, ele contém um auto-complete nas chaves.

Body: aqui colocamos o corpo da request, quando é uma requisição que tenha corpo como por exemplo um POST, informamos o tipo da informação e o seu corpo, caso seja XML, JSON, HTML,Text selecionar o tipo raw, que será acionado uma compo para que consiga passar o body corretamente.

Pre-request Script: São comandos em JS que serão executados, antes da requisição.

Tests: são validações para que realize teste na request do valor retornado e manipulação de dados.

Ao final das abas temos 2 termos em laranja sendo eles:

Cookies – onde iremos manipular caso necessário o cookie da request

Code – onde o Postman tem um gerador de código fonte em algumas linguagens referente a esta request que você está criando.

Para criamos algumas requisições válidas vamos utilizar os exemplos que estão junto com a instalação do POSTMAN onde contém algumas requisições de exemplo para verificar o comportamento do software conforme Figura 3 :

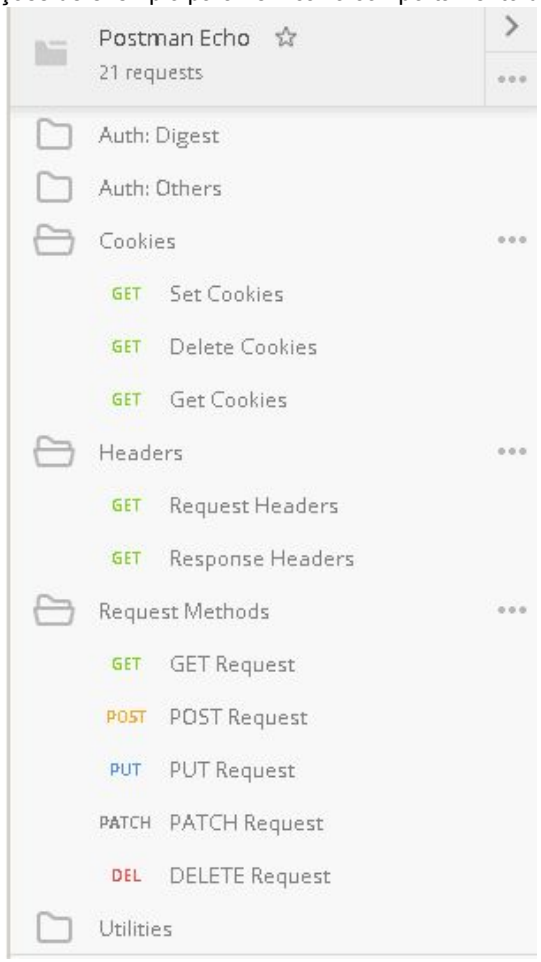

Figura 3 – Painel de requisições de exemplo POSTMAN

Para iniciarmos com o Postman vamos usar a requisição “Request Headers” da própria coletânea do Postman, e adicionarmos o nosso header, conforme Figura 4.

Figura 4 – Painel headers da requisição

A resposta do serviço do POSTMAN irá nos retornar um json com atributo headers, e tendo uma propriedade com nosso header e valor. Também é possível observar na parte superior da resposta uma barra onde contém todas as informações referente a este request, conforme Figura 5.

Figura 5 – Response de exemplo da requisição header

Agora iremos fazer uma requisição “POST” utilizando o exemplo “POST Request”, nesta requisição iremos alterar na guia body, o corpo da requisição que desejamos enviar e avaliar o seu retorno, conforme Figura 6.

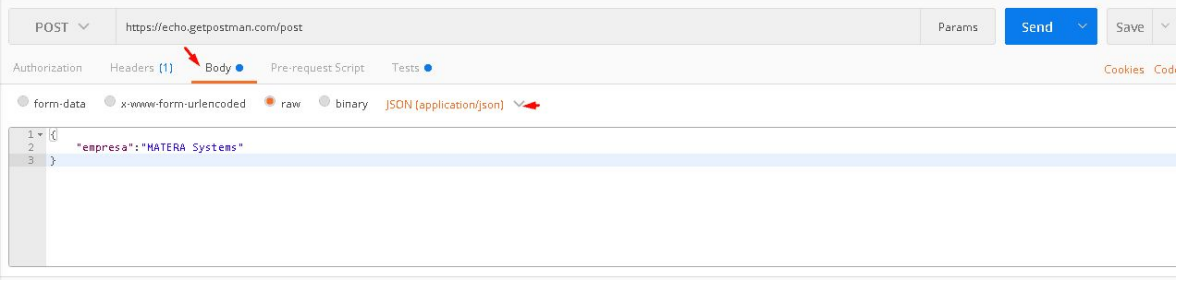

Figura 6 – Painel de alteração do body da requisição

Na resposta, temo como foi enviado a informação para o serviço e depois uma propriedade com o tipo de dado da requisição e as informações enviadas, conforme Figura 6.

Figura 7 – Response de exemplo da requisição de alteração do body

Com este post é possível, ter o conhecimento inicial do POSTMAN, e como fazer a comunicação dele com suas api’s ou de terceiros. Nos próximos post’s iremos nos aprofundar mais sobre essa ferramenta, criando collections, testes automáticos, e ainda iremos utilizarmos recursos npm para executar os testes em nossos serviços, assim podendo integrar em uma ferramenta de integração contínua.

1.3 Outras ferramentas similares

Além do Postman existe outras ferramentas que testam API’s como por exemplo Insomnia e pode-se testar a API pelo terminal do Linux que também fornece teste de API

TESTE DE API'S

1.4 O QUE É UMA API?

API é um conjunto de rotinas e padrões de programação para acesso a um aplicativo de software ou plataforma baseado na Web. O termo API refere-se ao termo em inglês "Application Programming Interface" que significa em tradução para o português "Interface de Programação de Aplicativos".

Uma API é criada quando uma empresa de software tem a intenção de que outras aplicações de software desenvolvam produtos associados ao seu serviço. Existem vários deles que disponibilizam seus códigos e instruções para serem usados em outros sites da maneira mais conveniente para seus usuários. O Google Maps é um dos grandes exemplos na área de APIs. Por meio de seu código original, muitos outros sites e aplicações utilizam os dados do Google Maps adaptando-o da melhor forma a fim de utilizar esse serviço.

1.5 TESTE EM API

Hoje estamos na era dos microsserviços e com o surgimento dessa nova arquitetura, os serviços monolíticos estão sendo abandonados. Porém com tantos serviços desse tipo, surge uma questão preocupante: os testes. Como realizar testes unitários de um recurso presente em uma API REST? Como realizar testes de integração dos microsserviços? Essas são algumas dúvidas que surgem no dia a dia e tornam a construção dos testes algo desafiador para o desenvolvedor. Então, com o objetivo de ajudá-los, vou compartilhar alguns frameworks, ferramentas e dicas de como realizar tais testes.

Para mostrar como é fácil a utilização do Postman iremos construir um passo a passo para testar uma API. Essa API é muito simples ela é um CRUD de Tarefas, onde temos os seguintes recursos.

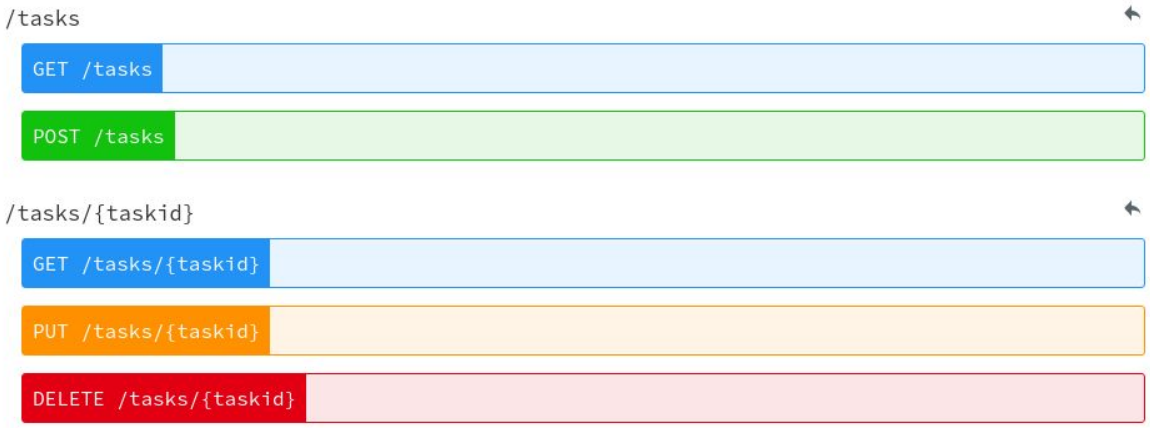

Figura 8 – guia de recurso do postman

Regra	Resultado esperado
POST/ tasks com valores faltando retornar status da requisição inválida.	HTTP Status 400
POST/ tasks ao executar um retornar o status de criação e com location no header.	HTTP Status 201
GET/ tasks ao executar retornar status de Ok.	HTTP Status 200
GET/ tasks {taskid} com taskid válido retornar o objeto task e o status OK.	Objeto task e HTTP Status 200
GET/ tasks {taskid} com o ID inválido retorna status não encontrado.	HTTP Status 404
DELET/tasks {taskid} com o ID válido retorna o status de nenhum conteúdo.	HTTP Status 204

REQUESTS

Como começar a usar Requests. Ela assume que você já tem Requests instalado. Se você não tem, o Requests é desenvolvido ativamente no GitHub, onde o código está sempre disponível.

Você pode clonar o repositório público:

```
git clone git://github.com/kennethreitz/requests.git
```

Baixar o arquivo tar:

```
$ curl -OL https://github.com/kennethreitz/requests/tarball/master
```

Ou, baixar o arquivo zip:

```
$ curl -OL https://github.com/kennethreitz/requests/zipball/master
```

Uma vez que você tiver uma cópia do código, você pode incluí-lo no seu pacote Pytohn, ou instalá-lo no seu diretório site-packages facilmente:

```
$ python setup.py instal
```

- Requests está instalado
- Requests está atualizado

Vamos começar com alguns exemplos simples.

Fazer uma requisição

Para fazer uma requisição com Requests é muito simples.Comece importando o módulo Requests:

```
>>> import requests
```

Agora, vamos tentar baixar uma página da web. Para este exemplo, vamos baixar a timeline pública do GitHub:

```
>>> r = requests.get('https://github.com/timeline.json')
```

Veja que agora, nós temos um objeto Response chamado r. Nós podemos pegar todas as informações que precisamos desse objeto. A API simples de Requests significa que todas as formas de requisição HTTP são óbvias. Por exemplo, este é o jeito que se faz uma requisição HTTP POST:

```
>>> r = requests.post("http://httpbin.org/post")
```

Legal, né? Que tal outros tipos de requisições HTTP: PUT, DELETE, HEAD e OPTIONS? São todas igualmente simples:

```
>>> r = requests.put("http://httpbin.org/put")
>>> r = requests.delete("http://httpbin.org/delete")
>>> r = requests.head("http://httpbin.org/get")
>>> r = requests.options("http://httpbin.org/get")
```

Está tudo muito bem, mas também é só o começo do que Requests pode fazer.

Passando parâmetros em URLs

Você geralmente quer mandar algum tipo de dado na query string da URL. Se você estivesse construindo a URL manualmente, este dado seria dado como pares chave/valor na URL após um ponto de interrogação, por exemplo `httpbin.org/get?key=val`. Requests permite que você forneça estes argumentos como um dicionário, usando o argumento `params`. Por exemplo, se você quisesse passar `key1=vaue1` e `key2=value2` para `httpbin.org/get`, você usaria o seguinte código:

```
>>> payload = {'key1': 'value1', 'key2': 'value2'}
```

```
>>> r = requests.get("http://httpbin.org/get", params=payload)
```

Você pode ver que a URL foi corretamente gerada imprimindo a URL:

```
>>> print r.url
u'http://httpbin.org/get?key2=value2&key1=value1'
```

Conteúdo da resposta

Nós podemos estar atentos a leitura do conteúdo da resposta do servidor. Considerando a timeline do GitHub novamente:

```
>>> import requests
>>> r = requests.get('https://github.com/timeline.json')
>>> r.text
'[{"repository":{"open_issues":0,"url":"https://github.com/...
```

Requests irá automaticamente decodificar o conteúdo do servidor. A maioria dos conjuntos de caracteres unicode são decodificados nativamente. Quando você faz uma requisição, Requests advinha qual a codificação da reposta baseada nos cabeçalhos HTTP. A codificação de texto adivinhada por Requests é usada quando você acessar `r.text`. Você pode descobrir qual codificação Requests está usando, e mudá-la, utilizando a propriedade `r.enconding`:

```
>>> r.encoding
'utf-8'
>>> r.encoding = 'ISO-8859-1'
```

Se você mudar a codificação, Requests irá usar o novo valor de `r.enconding` sempre que você chamar `r.text`.

Requests também usará codificações personalizadas na ocasião de você precisar delas. Se você criou sua própria codificação e a registrou com o módulo `codecs`, você pode simplesmente usar o nome do codec como valor de ``r.enconding `` e Requests irá cuidar da decodificação para você.

Resposta binária

Você pode acessar o corpo da resposta como bytes, para requisições que não são textos:

```
>>> r.content
b'[{"repository":{"open_issues":0,"url":"https://github.com/...
```

As codificações de transferências `gzip` e `deflate` são decodificadas automaticamente para você.

Por exemplo, para criar uma imagem de dados binários retornados por uma requisição, você pode usar o seguinte código:

```
>>> from PIL import Image
>>> from StringIO import StringIO
>>> i = Image.open(StringIO(r.content))
```

Resposta JSON

Também existe um decodificador JSON integrado, no caso de você lidar com dados JSON:

```
>>> import requests
>>> r = requests.get('https://github.com/timeline.json')
>>> r.json()
[{u'repository': {u'open_issues': 0, u'url': 'https://github.com/..
```

No caso da decodificação do JSON falhar, `r.json` levanta uma exceção. Por exemplo, se a resposta tem código 401, tentar usar `r.json` levanta `Value Error: No JSON objecto could be decoded`

Resposta crua

No caso raro de você querer recuperar a resposta crua do socket vinda do servidor, você pode acessar `r.raw`. Se você quiser fazer isso, certifique-se de que você definiu `stream=True` na sua requisição inicial. Uma vez que você fizer, poderá utilizar:

```
>>> r = requests.get('https://github.com/timeline.json', stream=True)
>>> r.raw
<requests.packages.urllib3.response.HTTPResponse object at 0x101194810>
>>> r.raw.read(10)
'\x1f\x8b\x08\x00\x00\x00\x00\x00\x00\x03'
```

Cabeçalhos personalizados

Se você quiser adicionar cabeçalhos HTTP para uma requisição, simplesmente passe-os em um `dict` para o parâmetro `headers`.

Por exemplo, nós não especificamos o content-type no exemplo anterior:

```
>>> import json
>>> url = 'https://api.github.com/some/endpoint'
>>> payload = {'some': 'data'}
>>> headers = {'content-type': 'application/json'}
>>> r = requests.post(url, data=json.dumps(payload), headers=headers)
```

Requisições POST mais complicadas

Tipicamente, você quer enviar algum dado em forma de formulário - assim como um formulário HTML. Para fazer isto, simplesmente passe um dicionário para o argumento **`data`.** O seu dicionário de dados será automaticamente formatado como formulário e a requisição é feita:

```
>>> payload = {'key1': 'value1', 'key2': 'value2'}
```

```
>>> r = requests.post("http://httpbin.org/post", data=payload)
>>> print r.text
{
  ...
  "form": {
    "key2": "value2",
    "key1": "value1"
  },
  ...
}
```

Existem várias ocasiões que você quer enviar dados que não estejam como formulário. Se você passar uma `string` em vez de um `dict`, o dado será enviado diretamente. Por exemplo, a API v3 do GitHub aceita dados codificados como JSON em POST/PATCH:

```
>>> import json
>>> url = 'https://api.github.com/some/endpoint'
>>> payload = {'some': 'data'}
>>> r = requests.post(url, data=json.dumps(payload))
```

POST de arquivo Multipart

Requests simplifica o upload de arquivos codificados Multipart:

```
>>> url = 'http://httpbin.org/post'
>>> files = {'file': open('report.xls', 'rb')}
>>> r = requests.post(url, files=files)
>>> r.text
{
  ...
  "files": {
    "file": "<dados...binários...censurados>"
  },
  ...
}
```

Você pode definir o nome do arquivo explicitamente:

```
>>> url = 'http://httpbin.org/post'
>>> files = {'file': ('report.xls', open('report.xls', 'rb'))}
>>> r = requests.post(url, files=files)
>>> r.text
{
  ...
  "files": {
    "file": "<dados...binários...censurados>"
  },
  ...
}
```

Se você quiser, você pode enviar strings para serem recebidas como arquivos:

```
>>> url = 'http://httpbin.org/post'
>>>files={'file':('report.csv',,data,to,send\nanother,row,to,send\n')}
>>> r = requests.post(url, files=files)
>>> r.text
{
  ...
  "files": {
    "file": "algum,dado,para,enviar\\noutra,linha,para,enviar\\n"
  },
  ...
}
```

Código do status da resposta

Você pode verificar o código do status da resposta:

```
>>> r = requests.get('http://httpbin.org/get')
>>> r.status_code
200
```

Requests também vem com um objeto de referência de códigos de status integrado:

```
>>> r.status_code == requests.codes.ok
True
```

Se nós fizermos uma requisição mal-feita (resposta que não tenha código 200), podemos levantar uma exceção usando Response.raise_for_status():

```
>>> bad_r = requests.get('http://httpbin.org/status/404')
>>> bad_r.status_code
404
>>> bad_r.raise_for_status()
Traceback (most recent call last):
  File "requests/models.py", line 832, in raise_for_status
    raise http_error
requests.exceptions.HTTPError: 404 Client Error
```

Mas, já que nosso `status_code` para `r` foi `200`, quando nós chamamos `raise_for_status()`, recebemos:

```
>>> r.raise_for_status()
None
```

Está tudo bem

Cabeçalhos da resposta

Nós podemos visualizar os cabeçalhos da resposta do servidor usando um dicionário Python:

```
>>> r.headers
{
    'content-encoding': 'gzip',
    'transfer-encoding': 'chunked',
    'connection': 'close',
    'server': 'nginx/1.0.4',
    'x-runtime': '148ms',
    'etag': '"e1ca502697e5c9317743dc078f67693f"',
    'content-type': 'application/json; charset=utf-8'
}
```

No entanto, o dicionário é especial: ele é feito sobretudo para cabeçalhos HTTP. De acordo com a RFC 2616, cabeçalhos HTTP são case-insensitive.Assim, nós podemos acessar os cabeçalhos usando qualquer capitalização que quisermos:

```
>>> r.headers['Content-Type']
'application/json; charset=utf-8'
>>> r.headers.get('content-type')
'application/json; charset=utf-8'
```

Se um cabeçalho não existe no objeto Response, seu valor padrão é `None`:

```
>>> r.headers['X-Random']
None
```

Cookies

Se uma resposta contém alguns cookies, você tem acesso rápido a eles:

```
>>> url = 'http://example.com/some/cookie/setting/url'
>>> r = requests.get(url)
>>> r.cookies['example_cookie_name']
'example_cookie_value'
```

Para enviar seus próprios cookies para o servidor, você pode usar o parâmetrO `cookies`:

```
>>> url = 'http://httpbin.org/cookies'
>>> cookies = dict(cookies_are='working')
>>> r = requests.get(url, cookies=cookies)
>>> r.text
'{"cookies": {"cookies_are": "working"}}'
```

Redirecionamento e Histórico

Requests irá automáticamente realizar redireccionamientos quando utilizados os verbos GET e OPTIONS.

GitHub redireciona todas as requisições HTTP para HTTPS. Nós podemos usar o método `history` do objeto Response para acompanhar o redirecionamento. Vamos ver o que o GitHub faz:

```
>>> r = requests.get('http://github.com')
>>> r.url
'https://github.com/'
>>> r.status_code
200
>>> r.history
[<Response [301]>]
```

A lista Response.history contém uma lista de objetos Request que foram criados para completar a requisição. A lista é ordenada da requisição mais antiga para a mais nova.

Se você estiver usando GET ou OPTIONS, você pode desabilitar o redirecionamento com o parâmetro `allow_redirects`:

```
>>> r = requests.get('http://github.com', allow_redirects=False)
>>> r.status_code
301
>>> r.history
[]
```

Se você estiver usando POST, PUT, PATCH, DELETE ou HEAD, você pode habilitar o redirecionamento também:

```
>>> r = requests.post('http://github.com', allow_redirects=True)
>>> r.url
'https://github.com/'
>>> r.history
[<Response [301]>]
```

Timeouts

Você pode dizer para as requisições pararem de esperar por uma resposta depois de um dado número de segundos com o parâmetro `timeout`:

```
>>> requests.get('http://github.com', timeout=0.001)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
requests.exceptions.Timeout:         HTTPConnectionPool(host='github.com',
port=80): Request timed out. (timeout=0.001)
```

Notas:timeout somente afeta o processo da conexão, não o download do corpo da resposta.

Erros e Exceções

Na ocasião de um problema de rede (por exemplo, falha de DNS, conexão recusada, etc.), Requests irá levantar uma exceção ConnectionError.Na ocasião de uma rara resposta HTTP inválida, Requests irá levantar uma exceção HTTPError.Se uma requisição excede o tempo limite, uma exceção Timeout é levantada.

Se uma requisição excede o número máximo de redirecionamentos configurado, uma exceção TooManyRedirects é levantada. E todas as exceções levantadas explicitamente pelo Requests são herdadas de requests.exceptions.RequestException.

REFERÊNCIAS

1. Autor desconhecido.Test ONAP API with Postman,2019.Disponível em:<https://docs.onap.org/en/dublin/submodules/integration.git/docs/docs_postman.html>.Acessado em: 6.jul.2020.
2. Autor desconhecido.Título desconhecido.Disponível em: <https://www.postman.com/>.Acessado em:6.jul.2020.
3. Santiago,Arthur Fritz.Iniciando com o Postman,2017.Disponível em:<http://www.matera.com/blog/post/iniciando-com-o-postman>.Acessado em: 6.jul.2020.
4. Autor desconhecido.Título desconhecido.Disponível em:<https://learning.postman.com/docs/postman/launching-postman/introduction/>. Acessado em:6.jul.2020.
5. Autor desconhecido.Quickstar.Disponível em:<https://requests.readthedocs.io/en/master/user/quickstart/>.Acessado em: 7.jul.2020.
6. Redação.O que é API?,2016.Disponível em:<https://canaltech.com.br/software/o-que-e-api/>.Acessado em: 7.jul.2020.
7. Autor desconhecido.Guia de início rápido-Requests 1.2.0.Disponível em:<https://requests.readthedocs.io/pt_BR/latest/user/quickstart.html>.Acessado em: 7.jul.2020.
8. Borpuzari;Pranbihanga, Bhattacharya;Ashutos.Postman:From a side project to a million-dollar startup, Postman has come a long way,2018.Disponível em:<https://economictimes.indiatimes.com/small-biz/startups/features/from-a-side-project-to-a-million-dollar-startup-postman-has-come-a-long-way/articleshow/64759580.cms?utm_source=contentofinterest&utm_medium=text&utm_campaign=cppst>.Acessado em: 7.jul.2020.