

# |Requests|

مشتاق شروع هستید؟ این صفحه مقدمه خوبی در مورد نحوه شروع کار با **Requests** ارائه می‌دهد.

ابتدا مطمئن شوید که:

- درخواست‌ها نصب شده است

- درخواست‌ها به روز هستند

باید با چند مثال ساده شروع کنیم.

## درخواست دهید

ایجاد درخواست با **Requests** بسیار ساده است.

با وارد کردن مازول **Requests** شروع کنید:

**import requests**

حالا، باید سعی کنیم یک صفحه وب را دریافت کنیم.  
برای این مثال، باید جدول زمانی عمومی GitHub را دریافت کنیم:

```
r = requests.get('https://api.github.com/events')
```

حالا، ما یک **Response** شیء به نام داریم **r**.  
می‌توانیم تمام اطلاعات مورد نیازمان را از این شیء دریافت کنیم.

به این معنی است که تمام API ساده‌ی **Requests** به همان اندازه واضح HTTP اشکال درخواست هستند. برای مثال، نحوه ارسال یک درخواست HTTP POST به این صورت است:

```
r = requests.post('https://httpbin.org/post', data={'key': 'value'})
```

خوب، درست است؟ در مورد انواع دیگر درخواست چطور؟ HTTP **PUT**, **DELETE**, **HEAD** و **OPTIONS** همه اینها به همین سادگی هستند:

```
r = requests.put('https://httpbin.org/put', data={'key': 'value'}) r =  
requests.delete('https://httpbin.org/delete') r = requests.head('https://  
httpbin.org/get') r =  
requests.options('https://httpbin.org/get')
```

همه اینها خوب و عالی است، اما این تنها شروع کاری است که Requests می‌تواند انجام دهد.

## ارسال پارامترها در URL‌ها

شما اغلب می‌خواهید نوعی داده را در رشته پرس‌وجوی URL ارسال کنید. اگر URL را به صورت دستی می‌سازید، این داده‌ها به صورت جفت‌های کلید/مقدار در URL پس از علامت سوال ارائه می‌شوند، مثلاً `httpbin.org/get?key=val`. Requests به شما امکان می‌دهد این آرگومان‌ها را به عنوان یک دیکشنری از رشته‌ها، با استفاده از `params` آرگومان کلمه کلیدی ارائه دهید. به عنوان مثال، اگر می‌خواهید `key2=value2` و `pass key1=value1` را ارسال کنید `httpbin.org/get` کرد:

```
payload = {'key1': 'value1', 'key2': 'value2'}  
r = requests.get('https://httpbin.org/get',  
params=payload)
```

با چاپ URL می‌توانید ببینید که URL به درستی کدگذاری شده است:

```
print(r.url) https://httpbin.org/get?  
key2=value2&key1=value1
```

توجه داشته باشید که هر کلید دیکشنری که مقدار آن باشد، `None` به رشته‌ی جستجوی URL اضافه نخواهد شد.

همچنین می‌توانید لیستی از آیتم‌ها را به عنوان مقدار ارسال کنید:

```
payload = {'key1': 'value1', 'key2': ['value2',  
'value3']} r = requests.get('https://  
httpbin.org/get', params=payload)  
print(r.url) https://httpbin.org/get?  
key1=value1&key2=value2&key2=value3
```

## محتوای پاسخ

می‌توانیم محتوای پاسخ سرور را بخوانیم. دوباره جدول زمانی گیت‌هاب را در نظر بگیرید:

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

درخواست‌ها به طور خودکار محتوا را از سرور رمزگشایی می‌کنند. اکثر مجموعه کاراکترهای یونیکد به طور یکپارچه رمزگشایی می‌شوند.

وقتی درخواستی ارسال می‌کنید، **Requests** بر اساس هدرهای **HTTP** حدهای آگاهانه‌ای در مورد کدگذاری پاسخ می‌زند. کدگذاری متنی که توسط **Requests** حده زده می‌شود، هنگام دسترسی شما استفاده می‌شود. می‌توانید با استفاده از **r.text** می‌توانید با استفاده از ویژگی

زیر بفهمید که Requests از چه کدگذاری استفاده می‌کند و آن را تغییر دهید :r.encoding

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

اگر کدگذاری را تغییر دهید، Requests r.encoding از مقدار جدید استفاده خواهد کرد r.text. شما می‌توانید این کار را در هر موقعیتی انجام دهید که بتوانید منطق خاصی را برای تعیین کدگذاری محتوا اعمال کنید. به عنوان مثال، XML و HTML این قابلیت را دارند که کدگذاری خود را در بدنه خود مشخص کنند. در موقعیت‌هایی مانند این، باید r.content از برای یافتن کدگذاری و سپس تنظیم آن استفاده کنید r.encoding. این به شما امکان می‌دهد r.text از کدگذاری صحیح استفاده کنید.

درخواست‌ها همچنین در صورت نیاز شما از کدگذاری‌های سفارشی استفاده خواهند کرد. اگر کدگذاری خودتان را ایجاد کرده و آن را در codecs مازول ثبت کرده‌اید، می‌توانید به سادگی از نام کدک به عنوان مقدار استفاده کنید r.encoding و درخواست‌ها رمزگشایی را برای شما انجام خواهند داد.

## محتوای پاسخ دودویی

همچنین می‌توانید برای درخواست‌های غیر متنی، به صورت بایت به بدنه پاسخ دسترسی داشته باشید:

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

کدگذاری‌های deflate و gzip انتقال به طور خودکار برای شما رمزگشایی می‌شوند.

برای اگر یک کتابخانه **Brotli** نصب شده باشد، کدگذاری **brotlicffi** یا **brotli** مانند انتقال به طور خودکار برای شما رمزگشایی می‌شود.

برای مثال، برای ایجاد یک تصویر از داده‌های دودویی برگردانده شده توسط یک درخواست، می‌توانید از کد زیر استفاده کنید:

```
from PIL import Image from io import  
BytesIO i =  
Image.open(BytesIO(r.content))
```

## محتوای پاسخ JSON

همچنین یک رمزگشای JSON داخلی نیز وجود دارد، در صورتی که با داده‌های JSON سرو کار دارید:

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

در صورتی که رمزگشایی JSON با شکست مواجه شود، `r.json()` یک استثنای ایجاد می‌شود. برای مثال، اگر پاسخ خطای 204 (بدون محتوا) دریافت کند، یا اگر پاسخ حاوی JSON نامعتبر باشد، تلاش برای `r.json()` رمزگشایی منجر به `requests.exceptions.JSONDecodeError` خطا می‌شود. این استثنای پوششی، قابلیت همکاری را برای چندین استثنای ممکن است توسط نسخه‌های مختلف پایتون و کتابخانه‌های سریال‌سازی json ایجاد شوند، فراهم می‌کند.

لازم به ذکر است که موفقیت فراخوانی `to` نشان دهنده موفقیت پاسخ نیست. برخی از سرورها ممکن است در یک پاسخ ناموفق یک شیء JSON برگردانند (مثلاً جزئیات خطا با HTTP 500). چنین JSON رمزگشایی و برگردانده می‌شود. برای بررسی موفقیت‌آمیز بودن یک درخواست، از دستور `checkr.raise_for_status()` یا `r.json()` اسکنید. `r.status_code` تفاده کنید.

## محتوای پاسخ خام

در موارد نادری که می‌خواهید پاسخ خام سوکت را از سرور دریافت کنید، می‌توانید به این دسترسی داشته باشید `r.raw`. اگر می‌خواهید این کار را انجام دهید، مطمئن شوید که `stream=True` درخواست اولیه خود را تنظیم کرده‌اید. پس از انجام این کار، می‌توانید این کار را انجام دهید:

```
r = requests.get('https://api.github.com/')
```

```
events', stream=True) r.raw
<urllib3.response.HTTPResponse object
at 0x101194810> r.raw.read(10)
b'\x1f\x8b\x08\x00\x00\x00\x00\x00\x00
\x03'
```

با این حال، به طور کلی، شما باید از الگویی مانند این برای ذخیره آنچه که در حال پخش شدن در یک فایل استفاده کنید:

```
with open(filename, 'wb') as fd: for chunk
in r.iter_content(chunk_size=128):
    fd.write(chunk)
```

استفاده از این روش **Response.iter\_content** بسیاری از کارهایی را که در حالت عادی باید **Response.raw** مستقیماً انجام می‌دادید، انجام می‌دهد. هنگام پخش یک فایل دانلودی، روش فوق روش ترجیحی و توصیه شده برای بازیابی محتوا است. توجه داشته باشید که **chunk\_size** می‌توان آن را آزادانه به عددی تنظیم کرد که ممکن است برای موارد استفاده شما مناسب‌تر باشد.

توجه داشته باشید

یک نکته مهم در مورد استفاده از **Response.iter\_content** مقابله با **Response.raw**. **Response.iter\_content** به طور خودکار رمزگشایی می‌کند **deflate** و **gzip** انتقال-

رمزگذاری‌ها. Response.raw یک جریان خام از بایت‌ها است - محتوای پاسخ را تغییر نمی‌دهد. اگر واقعاً به دسترسی به بایت‌ها به همان شکلی که برگردانده شده‌اند نیاز دارید، از .استفاده Response.raw کنید.

## سربرگ‌های سفارشی

اگر می‌خواهید هدرهای HTTP را به یک درخواست اضافه کنید، کافیست a به dict پارامتر headers را ارسال کنید.

برای مثال، ما در مثال قبلی user-agent خود را مشخص نکردیم:

```
url = 'https://api.github.com/some/endpoint' headers = {'user-agent': 'my-app/0.0.1'} r = requests.get(url, headers=headers)
```

توجه: به هدرهای سفارشی نسبت به منابع اطلاعاتی خاص‌تر، اولویت کمتری داده می‌شود. برای مثال:

- هدرهای مجوز تنظیم شده با `=headers` در صورت مشخص شدن اعتبارنامه‌ها در لغو می‌شوند. netrc، که به نوبه خود توسط پارامتر netrc =auth لغو می‌شوند. درخواست‌ها فایل netrc را در `~/netrc` یا `~/_netrc` در مسیری که توسط متغیر محیطی NETRC مشخص شده است، جستجو می‌کند. جزئیات را در netrc

## authentication بررسی کنید .

- در صورت ریدایرکت شدن به خارج از میزبان، هدرهای مجوز حذف خواهند شد.
- هدرهای **Proxy-Authorization** توسط اعتبارنامه‌های پروکسی ارائه شده در URL لغو می‌شوند.
- هدرهای **Content-Length** زمانی که بتوانیم طول محتوا را تعیین کنیم، لغو می‌شوند.

علاوه بر این، **Requests** رفتار خود را بر اساس اینکه کدام هدرهای سفارشی مشخص شده‌اند، به هیچ وجه تغییر نمی‌دهد. هدرها به سادگی به درخواست نهایی منتقل می‌شوند.

نکته: تمام مقادیر هدر باید **a string**, **bytestring** یا **unicode** باشند. در حالی که مجاز است، توصیه می‌شود از ارسال مقادیر هدر یونیکد خودداری کنید.

## درخواست‌های POST پیچیده‌تر

معمولًاً، شما می‌خواهید مقداری داده‌ی کدگذاری شده توسط فرم ارسال کنید - دقیقاً مانند یک فرم **HTML**. برای انجام این کار، کافیست یک دیکشنری به **data** آرگومان ارسال کنید. دیکشنری داده‌های شما هنگام ارسال درخواست، به طور خودکار کدگذاری شده توسط فرم خواهد شد:

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

آرگومان **data** همچنین می‌تواند برای هر کلید چندین مقدار داشته باشد. این کار را می‌توان با ایجاد **data** لیستی از تاپل‌ها یا یک دیکشنری با لیست‌ها به عنوان مقادیر انجام داد. این امر به ویژه زمانی مفید است که فرم دارای چندین عنصر باشد که از یک کلید استفاده می‌کنند:

```
payload_tuples = [('key1', 'value1'), ('key1', 'value2')] r1 = requests.post('https://httpbin.org/post', data=payload_tuples) payload_dict = {'key1': ['value1', 'value2']} r2 = requests.post('https://httpbin.org/post', data=payload_dict) print(r1.text) { ... "form": { "key1": [ "value1", "value2" ] }, ... } r1.text == r2.text True
```

موقعی وجود دارد که ممکن است بخواهید داده‌هایی را ارسال کنید که به صورت فرم کدگذاری نشده‌اند. اگر به جای **a** از **a** استفاده کنید **dict**، آن داده‌ها مستقیماً ارسال می‌شوند.

برای مثال، GitHub API نسخه ۳ داده‌های POST کدگذاری شده با JSON را می‌پذیرد:

```
import json url = 'https://api.github.com/
```

```
some/endpoint' payload = {'some': 'data'}  
r = requests.post(url,  
data=json.dumps(payload))
```

لطفاً توجه داشته باشید که کد بالا-Content هدر را اضافه نمی‌کند (بنابراین به طور خاص آن Type را روی تنظیم نمی‌کند).(application/json

اگر به آن مجموعه هدر نیاز دارید و نمی‌خواهید dict خودتان آن را کدگذاری کنید، می‌توانید آن را مستقیماً با استفاده از json پارامتر (که در نسخه ۲.۴.۲ اضافه شده است) ارسال کنید و به طور خودکار کدگذاری خواهد شد:

```
url = 'https://api.github.com/some/  
endpoint' payload = {'some': 'data'}
```

```
r = requests.post(url, json=payload)
```

توجه داشته باشید که این data اگر یکی از files ارسال شود، پارامتر نادیده گرفته می‌شود.

ارسال یک فایل کدگذاری شده چند بخشی

در خواستها آپلود فایل‌های کدگذاری شده ی چندبخشی را ساده می‌کند:

```
url = 'https://httpbin.org/post' files = {'file':
```

```
open('report.xls', 'rb')} r =  
requests.post(url, files=files) r.text { ...  
"files": { "file": "<censored...binary...data>"  
}, ... }
```

می‌توانید نام فایل، نوع محتوا و هدرها را به طور  
صریح تنظیم کنید:

```
url = 'https://httpbin.org/post' files = {'file':  
('report.xls', open('report.xls', 'rb'),  
'application/vnd.ms-excel', {'Expires': '0'})}  
r = requests.post(url, files=files) r.text { ...  
"files": { "file": "<censored...binary...data>"  
}, ... }
```

اگر بخواهید، می‌توانید رشته‌ها را برای دریافت به  
صورت فایل ارسال کنید:

```
url = 'https://httpbin.org/post' files = {'file':  
('report.csv',  
'some,data,to,send\nanother,row,to,send\n'  
)} r = requests.post(url, files=files) r.text { ...  
"files": { "file": "some,data,to,send\  
\nanother,row,to,send\\n" }, ... }
```

در صورتی که یک فایل بسیار بزرگ را به  
عنوان **multipart/form-data** درخواست ارسال  
می‌کنید، ممکن است بخواهید درخواست را پخش  
کنید. به طور پیش‌فرض، **requests** از این پشتیبانی  
نمی‌کند، اما یک بسته جداگانه وجود دارد که این کار را

انجام می‌دهد - . برای جزئیات بیشتر در مورد نحوه استفاده از آن، requests-toolbelt باید مستندات requests-toolbelt را مطالعه کنید.

برای ارسال چندین فایل در یک درخواست، به بخش پیشرفتی مراجعه کنید .

## هشدار

اکیداً توصیه می‌شود که فایل‌ها را در حالت دودویی باز کنید . دلیل این امر این است که درخواست‌ها ممکن است سعی کنند هدر را برای شما ارائه دهند Content-Length و در این صورت، این مقدار به تعداد بایت‌های موجود در فایل تنظیم می‌شود. اگر فایل را در حالت متنی باز کنید، ممکن است خطاهایی رخ دهد .

## کدهای وضعیت پاسخ

می‌توانیم کد وضعیت پاسخ را بررسی کنیم:

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

همچنین دارای یک شیء جستجوی کد Requests: وضعیت داخلی برای ارجاع آسان است

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

اگر درخواست بدی ارسال کردیم (خطای کلاینت 4XX)

یا پاسخ خطای سرور 5XX)، می‌توانیم آن را با دستور `:()Response.raise_for_status` زیر مطرح کنیم:

```
bad_r = requests.get('https://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
```

اما، از آنجایی که ما `rwas status_code` for می‌کنیم، وقتی فراخوانی 200 بود، `()raise_for_status` به صورت زیر خواهیم داشت:

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

همه چیز خوب است.

## هدرهای پاسخ

می‌توانیم هدرهای پاسخ سرور را با استفاده از یک دیکشنری پایتون مشاهده کنیم:

```
r.headers { 'content-encoding': 'gzip',  
'transfer-encoding': 'chunked',  
'connection': 'close', 'server': 'nginx/1.0.4',  
'x-runtime': '148ms', 'etag':
```

```
"e1ca502697e5c9317743dc078f67693f",  
'content-type': 'application/json' }
```

با این حال، این دیکشنری خاص است: فقط برای هدرهای HTTP ساخته شده است. طبق RFC 7230، نامهای هدر HTTP به حروف کوچک و بزرگ حساس نیستند.

بنابراین، می‌توانیم با استفاده از هر نوع حروف بزرگ و کوچکی که می‌خواهیم به هدرها دسترسی پیدا کنیم:

```
r.headers['Content-Type'] 'application/  
json' r.headers.get('content-type')  
'application/json'
```

همچنین از این جهت خاص است که سرور می‌توانست چندین بار یک هدر مشابه را با مقادیر مختلف ارسال کند، اما درخواست‌ها آنها را ترکیب می‌کند تا بتوانند در یک نگاشت واحد در دیکشنری نمایش داده شوند، همانطور که در RFC 7230 آمده است :

یک گیرنده می‌تواند چندین فیلد سرآیند با نام فیلد یکسان را در یک جفت «نام فیلد: مقدار فیلد» ترکیب کند، بدون اینکه معنای پیام تغییر کند، و برای این کار، هر مقدار فیلد بعدی را به ترتیب به مقدار فیلد ترکیبی اضافه می‌کند و با کاما از هم جدا می‌کند.

## کوکی‌ها

اگر پاسخی حاوی کوکی باشد، می‌توانید به سرعت به

آنها دسترسی پیدا کنید:

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

برای ارسال کوکی‌های خودتان به سرور، می‌توانید از پارامتر زیر استفاده کنید:

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

کوکی‌ها در یک برگردانده

می‌شوند **RequestsCookieJar**، که مانند a عمل می‌کند **dict** اما رابط کاربری کامل‌تری نیز ارائه می‌دهد که برای استفاده در چندین دامنه یا مسیر مناسب است. فایل‌های کوکی را می‌توان به درخواست‌ها نیز ارسال کرد:

```
jar =  
requests.cookies.RequestsCookieJar()  
jar.set('tasty_cookie', 'yum',  
domain='httpbin.org', path='/cookies')  
jar.set('gross_cookie', 'blech',  
domain='httpbin.org', path='/elsewhere')  
url = 'https://httpbin.org/cookies' r =  
requests.get(url, cookies=jar) r.text
```

```
'{"cookies": {"tasty_cookie": "yum"}}'
```

## تغییر مسیر و تاریخچه

به طور پیش‌فرض، Requests برای همه افعال به جز HEAD، تغییر مسیر مکان را انجام می‌دهد.

ما می‌توانیم history از ویژگی شیء Response برای ردیابی تغییر مسیر استفاده کنیم.

این لیست Response.history شامل Response اشیایی است که برای تکمیل درخواست ایجاد شده‌اند. این لیست از قدیمی‌ترین تا جدیدترین پاسخ مرتب شده است.

برای مثال، گیت‌هاب تمام درخواست‌های HTTP را به HTTPS هدایت می‌کند:

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

اگر از GET، OPTIONS، POST، PUT، PATCH یا DELETE استفاده می‌کنید، می‌توانید مدیریت تغییر مسیر را با پارامتر زیر غیرفعال کنید:

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

اگر از **HEAD** استفاده می‌کنید، می‌توانید ریدایرکت را نیز فعال کنید:

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

## تایم اوت‌ها

شما می‌توانید با استفاده از این پارامتر به درخواست‌ها بگویید که پس از تعداد ثانیه‌های مشخص، دیگر منتظر پاسخ نباشند **timeout**. تقریباً تمام کدهای عملیاتی باید از این پارامتر در تقریباً تمام درخواست‌ها استفاده کنند. عدم انجام این کار می‌تواند باعث شود برنامه شما به طور نامحدود هنگ کند:

```
requests.get('https://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)
```

توجه داشته باشید

محدودیت زمانی برای کل دانلود پاسخ **timeout** چند ثانیه **timeout** نیست؛ بلکه اگر سرور برای به طور دقیق‌تر، اگر هیچ (پاسخی ارسال نکرد) باشد

بایتی برای چند ثانیه در سوکت اصلی دریافت نشده باشد (**timeout**)، یک استثنای ایجاد می‌شود. اگر هیچ مهلت زمانی به صراحت مشخص نشده باشد، درخواست‌ها مهلت زمانی ندارند.

## خطاهای و استثنایات

در صورت بروز مشکل در شبکه (مثلاً خرابی DNS، عدم اتصال و غیره)، یک استثنای **ConnectionError** ایجاد می‌کند.

اگر **Response.raise\_for\_status()** **HTTPError** کد وضعیت ناموفقی را HTTP درخواست برگرداند، خطای “” را ایجاد می‌کند.

اگر درخواستی به پایان برسد، یک **Timeout** استثنای ایجاد می‌شود.

اگر درخواستی از تعداد پیکربندی شده‌ی حداقل تغییر مسیرها فراتر رود، یک **TooManyRedirects** استثنای ایجاد می‌شود.

تمام استثنایاتی که **Requests** صریحاً ایجاد می‌کند، از به ارت

می‌رسند **requests.exceptions.RequestException**.

برای اطلاعات بیشتر آماده‌اید؟ به بخش پیشرفتیه مراجعه کنید.

# | Request/advance |

¶

این سند برخی از ویژگی‌های پیشرفته‌تر Requests را پوشش می‌دهد.

## اشیاء جلسه

شیء Session به شما امکان می‌دهد پارامترهای خاصی را در درخواست‌ها حفظ کنید. همچنین کوکی‌ها را در تمام درخواست‌های انجام شده از نمونه urllib3 از ادغام Session حفظ می‌کند و اتصال استفاده می‌کند. بنابراین اگر چندین درخواست را به یک میزبان ارسال می‌کنید، اتصال TCP زیربنایی دوباره استفاده می‌شود که می‌تواند منجر به افزایش قابل توجه عملکرد شود (به اتصال مداوم HTTP مراجعه کنید).

یک شیء Session تمام متدهای API درخواست‌های اصلی را دارد.

باید برخی از کوکی‌ها را در درخواست‌ها حفظ کنیم:

```
s = requests.Session()
s.get('https://httpbin.org/cookies/set/sessioncookie/123456789')
r = s.get('https://httpbin.org/cookies')
print(r.text) # '{"cookies": {"sessioncookie": "123456789"}'}
```

همچنین می‌توان از جلسات (Session) برای ارائه داده‌های پیش‌فرض به متدهای درخواست استفاده کرد. این کار با ارائه داده‌ها به ویژگی‌های (property) یک شیء جلسه (Session) انجام می‌شود:

```
s = requests.Session()
s.auth = ('user', 'pass')
s.headers.update({'x-test': 'true'}) # both 'x-test' and 'x-test2' are sent
s.get('https://httpbin.org/headers',
headers={'x-test2': 'true'})
```

هر دیکشنری که به یک متدد درخواست ارسال می‌کنید، با مقادیر سطح جلسه که تنظیم شده‌اند، ادغام می‌شود. پارامترهای سطح روش، پارامترهای جلسه را نادیده می‌گیرند.

با این حال، توجه داشته باشید که پارامترهای سطح متدد، حتی اگر از یک جلسه استفاده کنند، در درخواست‌های مختلف ذخیره نخواهند شد. این مثال فقط کوکی‌ها را با درخواست اول ارسال می‌کند، اما برای درخواست دوم ارسال نمی‌کند:

```
s = requests.Session()
r = s.get('https://httpbin.org/cookies',
cookies={'from-my':
```

```
'browser'}) print(r.text) # '{"cookies":  
{"from-my": "browser"}{' r = s.get('https://  
httpbin.org/cookies') print(r.text) #  
{"cookies": {}}
```

اگر می خواهید کوکی ها را به صورت دستی به جلسه خود اضافه کنید، از توابع کاربردی کوکی برای `.Session.cookies` استفاده کنید

جلسات همچنین می توانند به عنوان مدیران زمینه استفاده شوند:

```
with requests.Session() as s: s.get('https://  
httpbin.org/cookies/set/sessioncookie/  
123456789')
```

این کار تضمین می کند که به محض خروج از `with` بلوک، حتی اگر استثنایات مدیریت نشده ای رخ داده باشد، جلسه بسته شود.

## حذف یک مقدار از پارامتر `Dict`

گاهی اوقات می خواهید کلیدهای سطح جلسه را از یک پارامتر `dict` حذف کنید. برای انجام این کار، کافیست مقدار آن کلید را `None` در پارامتر سطح متند برابر با قرار دهید. این مقدار به طور خودکار حذف خواهد شد.

تمام مقادیری که در یک جلسه (`session`) وجود دارند، مستقیماً در دسترس شما هستند. برای کسب اطلاعات بیشتر به مستندات API جلسه (`Session`)

## اشیاء درخواست و پاسخ

هر زمان که با دوستان و آشنایان خود تماسی برقرار می‌شود (`requests.get()`)، شما دو کار مهم انجام می‌دهید. اول، شما در حال ساخت یک `Request` شیء هستید که برای درخواست یا پرس و جو از منبعی به سرور ارسال می‌شود. دوم، `Response` هنگامی که پاسخی از سرور دریافت می‌کند، یک شیء ایجاد می‌شود. این `Response` شیء شامل تمام اطلاعات برگردانده شده توسط سرور و همچنین شامل `Request` شیءی است که شما در ابتدا ایجاد کرده‌اید. در اینجا یک درخواست ساده برای دریافت برخی اطلاعات بسیار مهم از سرورهای ویکی‌پدیا آمده است:

```
r = requests.get('https://en.wikipedia.org/wiki/Monty_Python')
```

اگر بخواهیم به هدرهایی که سرور برای ما ارسال کرده است دسترسی پیدا کنیم، این کار را انجام می‌دهیم:

```
r.headers {'content-length': '56170', 'x-content-type-options': 'nosniff', 'x-cache': 'HIT from cp1006.eqiad.wmnet, MISS from cp1010.eqiad.wmnet', 'content-encoding': 'gzip', 'age': '3080', 'content-language': 'en', 'vary': 'Accept-Encoding,Cookie', 'server': 'Apache', 'last-modified': 'Wed, 13 Jun'}
```

```
2012 01:33:50 GMT', 'connection': 'close',
'cache-control': 'private, s-maxage=0,
max-age=0, must-revalidate', 'date': 'Thu,
14 Jun 2012 12:59:39 GMT', 'content-
type': 'text/html; charset=UTF-8', 'x-
cache-lookup': 'HIT from
cp1006.eqiad.wmnet:3128, MISS from
cp1010.eqiad.wmnet:80'}
```

با این حال، اگر بخواهیم هدرهایی را که به سرور ارسال کردہایم دریافت کنیم، به سادگی به درخواست و سپس هدرهای درخواست دسترسی پیدا می‌کنیم:

```
r.request.headers {'Accept-Encoding':
'identity, deflate, compress, gzip', 'Accept':
'*/*', 'User-Agent': 'python-requests/
1.2.0'}
```

## درخواست‌های آماده‌شده

یا API هر زمان که از یک فراخوانی Response یک شیء دریافت می‌کنید، فراخوانی Session در واقع آن ویژگی request، چیزی است که استفاده PreparedRequest همان شده است. در برخی موارد، ممکن است بخواهید قبل از ارسال درخواست، کارهای اضافی روی بدنه یا هدرها (یا هر چیز دیگری) انجام دهید. دستور العمل ساده برای این کار به شرح زیر است:

```
from requests import Request, Session s
```

```
= Session() req = Request('POST', url,
data=data, headers=headers) prepped =
req.prepare() # do something with
prepped.body prepped.body = 'No, I want
exactly this as the body.' # do something
with prepped.headers del
prepped.headers['Content-Type'] resp =
s.send(prepped, stream=stream,
verify=verify, proxies=proxies, cert=cert,
timeout=timeout ) print(resp.status_code)
```

از آنجایی که شما کار خاصی با شیء انجام نمی‌دهید، **Request** شیء را تغییر می‌دهید. سپس آن را به همراه پارامترهای دیگری که به **.requests**\* یا **.Session** می‌کردید، ارسال می‌کنید.

با این حال، کد بالا برخی از مزایای داشتن یک **Session** شیء **Requests** را از دست می‌دهد. به طور خاص، **Session** وضعیت سطح - مانند کوکی‌ها به درخواست شما اعمال نمی‌شود. برای اعمال آن وضعیت، فراخوانی **to** **PreparedRequest** با فراخوانی **to()** **Request.prepare** را جایگزین کنید **()** **Session.prepare\_request**، مانند این:

```
from requests import Request, Session s
= Session() req = Request('GET', url,
data=data, headers=headers) prepped =
s.prepare_request(req) # do something
with prepped.body prepped.body =
```

```
'Seriously, send exactly these bytes!' # do something with prepped.headers  
prepped.headers['Keep-Dead'] = 'parrot'  
resp = s.send(prepped, stream=stream,  
verify=verify, proxies=proxies, cert=cert,  
timeout=timeout ) print(resp.status_code)
```

وقتی از جریان درخواست آماده شده استفاده می کنید، به خاطر داشته باشید که محیط را در نظر نمی گیرد. اگر از متغیرهای محیطی برای تغییر رفتار درخواستها استفاده کنید، این می تواند مشکلاتی ایجاد کند. به عنوان مثال: گواهی های SSL خودامضا شده مشخص شده در در نظر REQUESTS\_CA\_BUNDLE گرفته نمی شوند. در نتیجه خطای an رخ می دهد. می توانید با ادغام صریح تنظیمات محیط در جلسه خود، این رفتار را دور

SSL: CERTIFICATE\_VERIFY\_FAILED: بزنید:

```
from requests import Request, Session  
s = Session() req = Request('GET', url)  
prepped = s.prepare_request(req) # Merge environment settings into session settings =  
s.merge_environment_settings(prepped.url, {}, None, None, None) resp =  
s.send(prepped, **settings)  
print(resp.status_code)
```

## تأیید گواهی SSL

درست مانند یک مرورگر وب، گواهی‌های SSL تأیید می‌کند. به HTTPS را برای درخواست‌های SSL فعال است و SSL طور پیش‌فرض، تأیید در صورت عدم توانایی در تأیید گواهی، خطای SSLError را نمایش می‌دهد:

```
requests.get('https://requestb.in')
requests.exceptions.SSLError: hostname
'requestb.in' doesn't match either of
'*.herokuapp.com', 'herokuapp.com'
```

من روی این دامنه SSL تنظیم نکرده‌ام، بنابراین یک استثنای ایجاد می‌کند. عالی. هرچند گیت‌هاب این کار را می‌کند:

```
requests.get('https://github.com')
<Response [200]>
```

می‌توانید verify مسیر را به یک فایل یا دایرکتوری CA\_BUNDLE که دارای گواهی‌های CA معتبر است، ارسال کنید:

```
requests.get('https://github.com', verify='/path/to/certfile')
```

یا مدام:

```
s = requests.Session()
s.verify = '/path/to/certfile'
```

توجه داشته باشید

اگر **verify** روی مسیری به یک دایرکتوری تنظیم شده باشد، دایرکتوری باید با استفاده از **c\_rehash** ابزار ارائه شده همراه **OpenSSL** پردازش شده باشد.

این لیست از CA های مورد اعتماد را می توان از طریق **REQUESTS\_CA\_BUNDLE** متغیر محیطی نیز مشخص کرد.

اگر **REQUESTS\_CA\_BUNDLE** تنظیم نشده باشد، **CURL\_CA\_BUNDLE** به عنوان جایگزین استفاده خواهد شد.

اگر روی **False** تنظیم کنید، درخواستها می توانند تأیید گواهی SSL را نیز نادیده بگیرند

```
requests.get('https://kennethreitz.org',  
verify=False) <Response [200]>
```

توجه داشته باشید که وقتی **verify** روی تنظیم شده باشد **False**، درخواستها هرگونه گواهی TLS ارائه شده توسط سرور را می پذیرند و عدم تطابق نام میزبان و/یا گواهی های منقضی شده را نادیده می گیرند، که برنامه شما را در برابر حملات مرد میانی **verify** (آسیب پذیر می کند. تنظیم **MitM**) روی **False** ممکن است در طول توسعه یا آزمایش محلی مفید باشد.

به طور پیش فرض، **True** روی **verify** تنظیم شده است. این گزینه **verify** فقط برای گواهی های میزبان

اعمال می‌شود.

## گواهی‌های سمت کلاینت

همچنین می‌توانید یک گواهی محلی را برای استفاده به عنوان گواهی سمت کلاینت، به عنوان یک فایل واحد (حاوی کلید خصوصی و گواهی) یا به عنوان یک چندتایی از مسیرهای هر دو فایل مشخص کنید:

```
requests.get('https://kennethreitz.org',
cert=('/path/client.cert', '/path/
client.key')) <Response [200]>
```

یا مدام:

```
s = requests.Session() s.cert = '/path/
client.cert'
```

اگر مسیر اشتباه یا گواهی نامعتبر را مشخص کنید، خطای **SSLError** دریافت خواهد کرد:

```
requests.get('https://kennethreitz.org',
cert='/wrong_path/client.pem') SSLError:
[Errno 336265225] _ssl.c:347:
error:140B0009:SSL
routines:SSL_CTX_use_PrivateKey_file:PE
M lib
```

هشدار

کلید خصوصی گواهی محلی شما باید رمزگذاری نشده باشد. در حال حاضر، Requests از استفاده از کلیدهای رمزگذاری شده پشتیبانی نمی‌کند.

## گواهینامه‌های CA

از گواهی‌های موجود در Requests استفاده می‌کند. این به کاربران اجازه certifi بسته می‌دهد تا گواهی‌های معتبر خود را بدون تغییر نسخه Requests به روزرسانی کنند.

قبل از نسخه ۲.۱۶ Requests مجموعه‌ای از گواهی‌های ریشه (root CA) را که به آنها اعتماد داشت، از فروشگاه اعتماد موزیلا (Mozilla trust) به صورت بسته‌ای ارائه می‌داد. این گواهی‌ها فقط برای هر نسخه Requests یک بار به روزرسانی می‌شدند. در صورت عدم نصب، این امر منجر به بسته‌های گواهی بسیار قدیمی هنگام استفاده از نسخه‌های بسیار قدیمی‌تر Requests می‌شد.

برای امنیت بیشتر، توصیه می‌کنیم مرتبأ گواهینامه خود را ارتقا دهید!

## گردش کار بدنی محتوا

به طور پیش‌فرض، وقتی درخواستی ارسال می‌کنید، بدنی پاسخ بلافاصله دانلود می‌شود. می‌توانید این رفتار را لغو کنید و دانلود بدنی پاسخ را تا زمانی که به Response.content پارامتر stream ویژگی با

دسترسی پیدا کنید، به تعویق بیندازید:

```
tarball_url = 'https://github.com/psf/  
requests/tarball/main' r =  
requests.get(tarball_url, stream=True)
```

در این مرحله فقط هدرهای پاسخ دانلود شده‌اند و اتصال باز می‌ماند، از این رو به ما اجازه می‌دهد بازیابی محتوا را مشروط کنیم:

```
if int(r.headers['content-length']) <  
TOO_LONG: content = r.content ...
```

شما می‌توانید با **Response.iter\_content()** گردش کار را بیشتر کنترل ، و استفاده از متدهای به عنوان یک **Response.iter\_lines()**. جایگزین، می‌توانید بدنه رمزگشایی نشده را از **urllib3.HTTPResponse** در **urllib3** زیرین بخوانید **Response.raw**.

اگر هنگام ارسال درخواست، آن را روی «درخواست‌ها» تنظیم کنید **stream=True**، نمی‌تواند اتصال را به مخزن بازگرداند، مگر اینکه تمام داده‌ها یا فراخوانی را مصرف کنید **Response.close**. این می‌تواند منجر به ناکارآمدی اتصالات شود. اگر هنگام استفاده از «درخواست‌ها»، متوجه شدید که بدنه‌های درخواست را تا حدی می‌خوانید (یا اصلاً نمی‌خوانید)، باید درخواست را در یک **with** دستور قرار دهید تا مطمئن شوید که همیشه بسته است:

```
with requests.get('https://httpbin.org/get',  
stream=True) as r: # Do things with the  
response here.
```

## زنده نگه داشتن

خبر عالی - به لطف `urllib3` keep-alive` در یک جلسه ۱۰۰٪ خودکار است! هر درخواستی که در یک جلسه ارسال کنید، به طور خودکار از اتصال مناسب استفاده مجدد می‌کند!

توجه داشته باشید که اتصالات فقط زمانی برای استفاده مجدد به مخزن آزاد می‌شوند که تمام داده‌های بدنی خوانده شده باشند؛ حتماً ویژگی شیء را `stream=False` تنظیم کنید یا `read_content()`

## آپلودهای استریمینگ

درخواست‌ها از آپلودهای استریمینگ پشتیبانی می‌کنند که به شما امکان می‌دهد استریم‌ها یا فایل‌های بزرگ را بدون خواندن آنها در حافظه ارسال کنید. برای استریم و آپلود، کافیست یک شیء فایل‌مانند برای بدنی خود ارائه دهید:

```
with open('massive-body', 'rb') as f:  
    requests.post('http://some.url/streamed',  
data=f)
```

## هشدار

اکیداً توصیه می‌شود که فایل‌ها را در حالت دودویی باز کنید. دلیل این امر این است که درخواست‌ها ممکن است سعی کنند هدر را برای شما ارائه دهند **Content-Length** و در این صورت، این مقدار به تعداد بایت‌های موجود در فایل تنظیم می‌شود. اگر فایل را در حالت متنی باز کنید، ممکن است خطاهایی رخ دهد.

## درخواست‌های کدگذاری‌شدهٔ تکه‌ای

درخواست‌ها همچنین از کدگذاری انتقال تکه‌ای برای درخواست‌های خروجی و ورودی پشتیبانی می‌کنند. برای ارسال یک درخواست کدگذاری شدهٔ تکه‌ای، کافیست یک مولد (یا هر تکرارکننده بدون طول) برای بدن‌هه خود ارائه دهید:

```
def gen(): yield 'hi' yield 'there'  
requests.post('http://some.url/chunked',  
data=gen())
```

برای پاسخ‌های کدگذاری‌شدهٔ قطعه‌بندی‌شده، بهتر است با استفاده از روی داده‌ها تکرار کنید **Response.iter\_content()**. در یک وضعیت ایده‌آل، شما روی درخواست تنظیم کرده‌اید، که در این صورت می‌توانید با فراخوانی با پارامتر **stream=True**، می‌خواهید حداکثر اندازهٔ قطعهٔ به قطعهٔ تکرار کنید. اگر

می‌توانید پارامتر را روی هر عدد صحیحی تنظیم کنید.  
`iter_contentchunk_size``None``chunk_size`

## فایل‌های کدگذاری شده چند بخشی POST چندگانه

شما می‌توانید چندین فایل را در یک درخواست ارسال کنید. برای مثال، فرض کنید می‌خواهید فایل‌های تصویری را در یک فرم `HTML` با فیلد چند فایلی آپلود کنید:

```
<input type="file" name="images"  
multiple="true" required="true"/>
```

برای انجام این کار، فقط فایل‌ها را روی لیستی از تاپل‌های زیر تنظیم کنید:  
(`form_field_name`, `file_info`)

```
url = 'https://httpbin.org/post'  
multiple_files = [ ('images', ('foo.png',  
open('foo.png', 'rb'), 'image/png')),  
('images', ('bar.png', open('bar.png', 'rb'),  
'image/png'))]  
r = requests.post(url,  
files=multiple_files)  
r.text { ... 'files':  
{'images': ' ...'} }  
'Content-Type':  
'multipart/form-data;  
boundary=3131623adb2043caaeb5538cc'
```

7aa0b3a', ... }

## هشدار

اکیداً توصیه می‌شود که فایل‌ها را در حالت دودویی باز کنید. دلیل این امر این است که درخواست‌ها ممکن است سعی کند هدر را برای شما ارائه دهند **Content-Length** و در این صورت، این مقدار به تعداد بایت‌های موجود در فایل تنظیم می‌شود. اگر فایل را در حالت متنی باز کنید، ممکن است خطاهایی رخ دهد.

## قلاب‌های رویداد

دارد که **Requests** (hook) یک سیستم قلاب می‌توانید از آن برای دستکاری بخش‌هایی از فرآیند درخواست یا ارسال سیگنال برای مدیریت رویدادها استفاده کنید.

قلاب‌های موجود:

**response:**

پاسخی که از یک درخواست (Request) تولید می‌شود.

شما می‌توانید با ارسال یک دیکشنری به پارامتر درخواست، یک تابع هوک را بر اساس هر درخواست اختصاص دهید:

**hooks{hook\_name: callback\_function}**

```
hooks={'response': print_url}
```

این تابع **callback\_function**، بخشی از داده‌ها را به عنوان اولین آرگومان خود دریافت خواهد کرد.

```
def print_url(r, *args, **kwargs):  
    print(r.url)
```

تابع فراخوانی شما باید استثنای خودش را مدیریت کند. هر استثنای مدیریت نشده‌ای به صورت بی‌صدا ارسال نمی‌شود و بنابراین باید توسط کدی که **Requests** را فراخوانی می‌کند، مدیریت شود.

اگر تابع فراخوانی مقداری را برگرداند، فرض بر این است که قرار است جایگزین داده‌های ارسالی شود. اگر تابع چیزی برنگرداند، هیچ چیز دیگری تحت تأثیر قرار نمی‌گیرد.

```
def record_hook(r, *args, **kwargs):  
    r.hook_called = True  
    return r
```

باید برخی از آرگومان‌های متدهای درخواست را در زمان اجرا چاپ کنیم:

```
requests.get('https://httpbin.org/',  
             hooks={'response': print_url}) https://  
httpbin.org/ <Response [200]>
```

شما می‌توانید چندین هوك را به یک درخواست واحد

اضافه کنید. بباید دو هوک را همزمان فراخوانی کنیم:

```
r = requests.get('https://httpbin.org/',  
hooks={'response': [print_url,  
record_hook]}) r.hook_called True
```

شما همچنین می‌توانید قلاب‌هایی را به یک Session نمونه اضافه کنید. هر قلابی که اضافه کنید، در هر درخواستی که به جلسه ارسال می‌شود، فراخوانی می‌شود. برای مثال:

```
s = requests.Session()  
s.hooks['response'].append(print_url)  
s.get('https://httpbin.org/') https://  
httpbin.org/ <Response [200]>
```

می‌تواند چندین قلاب داشته باشد که به ترتیب اضافه شدنشان فراخوانی می‌شوند.

## احراز هویت سفارشی

درخواست‌ها به شما امکان می‌دهد مکانیزم احراز هویت خود را مشخص کنید.

هر تابع فراخوانی که به عنوان auth آرگومان به متده درخواست ارسال شود، این امکان را خواهد داشت که درخواست را قبل از ارسال، تغییر دهد.

پیاده‌سازی‌های احراز هویت زیرکلاس‌هایی از هستند AuthBase و تعریف آنها آسان است.

دو پیاده‌سازی رایج از طرح احراز هویت Requests و ارائه requests.auth: HTTPBasicAuth و HTTPDigestAuth می‌دهد.

باید وانمود کنیم که یک سرویس وب داریم که فقط در صورتی پاسخ می‌دهد که **X-Pizza** هدر روی مقدار رمز عبور تنظیم شده باشد. بعید است، اما همین‌طور ادامه می‌دهیم.

```
from requests.auth import AuthBase class PizzaAuth(AuthBase): """Attaches HTTP Pizza Authentication to the given Request object.""" def __init__(self, username): # setup any auth-related data here self.username = username def __call__(self, r): # modify and return the request r.headers['X-Pizza'] = self.username return r
```

سپس، می‌توانیم با استفاده از Pizza Auth خود درخواستی ارسال کنیم:

```
requests.get('http://pizzabin.org/admin', auth=PizzaAuth('kenneth')) <Response [200]>
```

## درخواست‌های پخش

با استفاده از این `Response.iter_lines()` می‌توانید به راحتی روی API‌های استریمینگ

مانند Twitter Streaming API کار کنید . کافیست آن را روی `stream` تنظیم کنید `True` و با استفاده از `:iter_lines` دستور زیر روی پاسخ تکرار کنید

```
import json
import requests
r = requests.get('https://httpbin.org/stream/20', stream=True)
for line in r.iter_lines():
    # filter out keep-alive new lines if line:
    decoded_line = line.decode('utf-8')
    print(json.loads(decoded_line))
```

هنگام استفاده از `decode_unicode=True` به `Response.iter()` یا `Response.iter_lines()` همراه باشد، باید یک کدگذاری جایگزین در صورتی که سرور آن را ارائه نمی‌دهد، ارائه دهید:

```
r = requests.get('https://httpbin.org/stream/20', stream=True)
if r.encoding is None:
    r.encoding = 'utf-8'
for line in r.iter_lines(decode_unicode=True):
    print(json.loads(line))
```

## هشدار

برای ورود مجدد امن نیست. فرآخوانی `iter_lines` چندین باره این متده است از دست رفتن برخی از داده‌های دریافتی می‌شود. در صورتی که نیاز به فرآخوانی آن از چندین مکان دارد، به جای آن از شیء تکرارکننده حاصل استفاده کنید:

```
lines = r.iter_lines() # Save the first line  
for later or just skip it first_line =  
next(lines) for line in lines: print(line)
```

## پروکسی‌ها

اگر نیاز به استفاده از پروکسی دارید، می‌توانید درخواست‌های جداگانه را با آرگومان **proxies** هر متدهای درخواست پیکربندی کنید:

```
import requests proxies = { 'http': 'http://  
10.10.1.10:3128', 'https': 'http://  
10.10.1.10:1080', } requests.get('http://  
example.org', proxies=proxies)
```

روش دیگر این است که می‌توانید آن را یک بار برای کل زمان پیکربندی کنید:

```
import requests proxies = { 'http': 'http://  
10.10.1.10:3128', 'https': 'http://  
10.10.1.10:1080', } session =  
requests.Session()  
session.proxies.update(proxies)  
session.get('http://example.org')
```

## هشدار

تنظیمات ممکن است متفاوت از آنچه انتظار می‌رود عمل کنند. مقادیر ارائه شده توسط پروکسی‌های محیطی (آنها یعنی که

`urllib.request.getproxies` session.`proxies` برگردانده می‌شوند. برای اطمینان از استفاده از پروکسی‌ها در حضور پروکسی‌های محیطی، آرگومان را در تمام درخواست‌های جداگانه، همانطور که در ابتدا در بالا توضیح داده شد، به صراحت مشخص کنید.

برای جزئیات بیشتر به [#۲۰۱۸](#) مراجعه کنید.

وقتی پیکربندی پروکسی‌ها طبق آنچه در بالا نشان داده شده است، برای هر درخواست لغو نشود، به پیکربندی پروکسی تعریف شده توسط Requests متغیرهای محیطی استandard `http_proxy`, `https_proxy`, `no_proxy` و `all_proxy`. انواع حروف بزرگ این متغیرها نیز پشتیبانی می‌شوند. بنابراین می‌توانید آنها را برای پیکربندی Requests تنظیم کنید ( فقط مواردی را که مربوط به نیازهای شما هستند تنظیم کنید):

```
$ export HTTP_PROXY="http://  
10.10.1.10:3128"  
$ export  
HTTPS_PROXY="http://10.10.1.10:1080"  
$ export  
ALL_PROXY="socks5://10.10.1.10:3434"  
$ python >>> import requests >>>  
requests.get('http://example.org')
```

برای استفاده از **HTTP Basic Auth** با پروکسی خود، از دستور `http://`

در هر یک از ورودی‌های `user:password@host/` پیکربندی فوق استفاده کنید:

```
$ export HTTPS_PROXY="http://  
user:pass@10.10.1.10:1080" $ python >>>  
proxies = {'http': 'http://  
user:pass@10.10.1.10:3128/'}
```

## هشدار

ذخیره اطلاعات حساس نام کاربری و رمز عبور در یک متغیر محیطی یا یک فایل کنترل شده با نسخه، یک ریسک امنیتی است و اکیداً توصیه نمی‌شود.

برای ارائه پروکسی برای یک طرح و میزبان خاص، از فرم `scheme://hostname` برای کلید استفاده کنید. این برای هر درخواستی به طرح و نام میزبان دقیق داده شده مطابقت خواهد داشت.

```
proxies = {'http://10.20.1.128': 'http://  
10.10.1.10:5323'}
```

توجه داشته باشید که URL های پروکسی باید شامل این طرح باشند.

در نهایت، توجه داشته باشید که استفاده از پروکسی برای اتصالات https معمولاً مستلزم آن است که دستگاه محلی شما به گواهی ریشه پروکسی اعتماد کند. به طور پیش‌فرض، لیست گواهی‌های مورد اعتماد را می‌توانید در آدرس زیر بیابید:

```
from requests.utils import  
DEFAULT_CA_BUNDLE_PATH  
print(DEFAULT_CA_BUNDLE_PATH)
```

شما می‌توانید با `REQUESTS_CA_BUNDLE` (یا `CURL_CA_BUNDLE`) به مسیر فایل دیگری، این بسته گواهی پیش‌فرض را لغو کنید:

```
$ export REQUESTS_CA_BUNDLE="/usr/  
local/myproxy_info/cacert.pem" $ export  
https_proxy="http://10.10.1.10:1080" $  
python >>> import requests >>>  
requests.get('https://example.org')
```

## جواب

جدید در نسخه ۲.۱۰۰.

علاوه بر پروکسی‌های HTTP پایه، Requests از پروکسی‌هایی که از پروتکل SOCKS استفاده می‌کند نیز پشتیبانی می‌کند. این یک ویژگی اختیاری است که مستلزم نصب کتابخانه‌های شخص ثالث اضافی قبل از استفاده است.

می‌توانید وابستگی‌های این ویژگی را از موارد زیر دریافت کنید:

```
$ python -m pip install 'requests[socks]'
```

پس از نصب این وابستگی‌ها، استفاده از پروکسی **SOCKS** به آسانی استفاده از پروکسی **HTTP** خواهد بود:

```
proxies = { 'http': 'socks5://  
user:pass@host:port', 'https': 'socks5://  
user:pass@host:port' }
```

استفاده از این طرح **socks5** باعث می‌شود که حل و فصل **DNS** به جای سرور پروکسی، روی کلاینت انجام شود. این با **curl** مطابقت دارد که از این طرح برای تصمیم‌گیری در مورد انجام حل و فصل **DNS** روی کلاینت یا پروکسی استفاده می‌کند. اگر می‌خواهید دامنه‌ها را روی سرور پروکسی حل و فصل **socks5h** کنید، از این طرح استفاده کنید.

## انطباق

درخواست‌ها باید با تمام مشخصات و **RFC**‌های مربوطه مطابقت داشته باشند، تا جایی که این انطباق برای کاربران مشکلی ایجاد نکند. این توجه به مشخصات می‌تواند منجر به رفتارهایی شود که ممکن است برای کسانی که با مشخصات مربوطه آشنا نیستند، غیرمعمول به نظر برسد.

## کدگذاری‌ها

وقتی پاسخی دریافت می‌کنید، **Requests** حدسی در

مورد کدگذاری می‌زند تا هنگام دسترسی شما به ویژگی، از آن برای رمزگشایی پاسخ استفاده کند.

ابتدا کدگذاری موجود در هدر HTTP را بررسی می‌کند و اگر کدگذاری وجود نداشته باشد، از `chardet` یا `charset_normalizer` برای حدس زدن کدگذاری `Response.text` استفاده می‌کند.

اگر `chardet` نصب شده باشد، استفاده می‌کند، با این حال برای دیگر یک وابستگی اجباری نیست. این `chardet` کتابخانه یک وابستگی دارای مجوز LGPL است و برخی از کاربران درخواست‌ها نمی‌توانند به وابستگی‌های اجباری دارای مجوز LGPL وابسته باشند.

وقتی بدون مشخص کردن موارد `requests` اضافی نصب از `chardet` و `[use_chardet_on_py3]` می‌کنید برای (MIT دارای مجوز) قبل نصب نشده است، از استفاده می‌کند `requests` حدس زدن کدگذاری `.charset-normalizer`

تنها زمانی که Requests کدگذاری را حدس نمی‌زند، زمانی است که هیچ مجموعه کاراکتر صریحی در هدرهای HTTP وجود نداشته باشد و هدر-Content-Type شامل `.text` در این شرایط، RFC 2616 مشخص می‌کند که مجموعه کاراکتر پیش‌فرض باید ISO-8859-1 باشد. Requests در این مورد از مشخصات پیروی می‌کند. اگر به کدگذاری متفاوتی نیاز دارید، می‌توانید `Response.encoding` ویژگی

را به صورت دستی تنظیم کنید یا از `raw` . استفاده `.Response.content` کنید

## افعال HTTP

`Requests` تقریباً به طیف کاملی از افعال HTTP دسترسی می‌دهد: `GET`, `OPTIONS`, `HEAD`, `POST`, `PUT`, `PATCH` و `DELETE`. در ادامه مثال‌های مفصلی از استفاده از این افعال مختلف در `Requests` با استفاده از GitHub API ارائه شده است.

ما با فعلی که بیشتر استفاده می‌شود شروع می‌کنیم: `HTTP GET`. `HTTP GET` یک متده خودتوان است که منبع را از یک `URL` مشخص بر می‌گرداند. در نتیجه، فعلی است که باید هنگام تلاش برای بازیابی داده‌ها از یک مکان وب از آن استفاده کنید. یک نمونه از کاربرد آن، تلاش برای دریافت اطلاعات در مورد یک `commit` خاص از GitHub است. فرض کنید می‌خواهیم آن را در `Requests` انجام دهیم. آن را به این صورت دریافت می‌کنیم:

```
import requests
r = requests.get('https://api.github.com/repos/psf/requests/git/commits/a050faf084662f3a352dd1a941f2c7c9f886d4ad')
```

ما باید تأیید کنیم که گیت‌هاب به درستی پاسخ داده است. اگر پاسخ داده است، می‌خواهیم بفهمیم که چه

نوع محتوایی است. این کار را به این صورت انجام دهید:

```
if r.status_code == requests.codes.ok:  
    print(r.headers['content-type']) application/  
    json; charset=utf-8
```

بنابراین، گیتهاب **JSON** را برمی‌گرداند. این عالی است، می‌توانیم از این **r.json()** روش برای تجزیه آن به اشیاء پایتون استفاده کنیم.

```
commit_data = r.json()  
print(commit_data.keys()) ['committer',  
'author', 'url', 'tree', 'sha', 'parents',  
'message']  
print(commit_data['committer']) {'date':  
'2012-05-10T11:10:50-07:00', 'email':  
'me@kennethreitz.com', 'name': 'Kenneth  
Reitz'} print(commit_data['message'])  
makin' history
```

تا اینجا که خیلی ساده بود. خب، باید کمی **API** گیتهاب را بررسی کنیم. حالا می‌توانیم به مستندات **Requests** نگاهی بیندازیم، اما اگر به جای آن از استفاده کنیم، شاید کمی بیشتر سرگرم شویم. می‌توانیم از فعل **OPTIONS** استفاده کنیم تا بینیم چه نوع متدهای **HTTP** در آدرسی که استفاده کرده‌ایم پشتیبانی می‌شوند.

```
verbs = requests.options(r.url)
```

## verbs.status\_code 500

خب، چی؟ این کمکی نمی‌کند! معلوم شد که گیت‌هاب، مانند بسیاری از ارائه‌دهندگان API، در واقع متدهای **OPTIONS** را پیاده‌سازی نمی‌کند. این یک اشتباه آزاردهنده است، اما اشکالی ندارد، می‌توانیم از مستندات خسته‌کننده استفاده کنیم. با این حال، اگر گیت‌هاب **OPTIONS** را به درستی پیاده‌سازی کرده باشد، باید متدهای مجاز در هدرها را بر می‌گرداند، مثلاً

```
verbs = requests.options('http://a-good-website.com/api/cats')
print(verbs.headers['allow'])
GET,HEAD,POST,OPTIONS
```

با مراجعه به مستندات، می‌بینیم که تنها روش مجاز دیگر برای کامیت‌ها، **POST** است که یک کامیت جدید ایجاد می‌کند. از آنجایی که ما از مخزن **Requests** استفاده می‌کنیم، احتمالاً باید از انجام **POST**‌های دست و پا گیر برای آن خودداری کنیم. در عوض، باید با ویژگی **GitHub Issues** در **GitHub** کار کنیم.

این مستندات در پاسخ به مشکل شماره ۴۸۲ اضافه شده است. با توجه به اینکه این مشکل از قبل وجود دارد، از آن به عنوان مثال استفاده خواهیم کرد. باید با دریافت آن شروع کنیم.

```
r = requests.get('https://api.github.com/repos/psf/requests/issues/482')
r.status_code 200 issue =
```

```
json.loads(r.text) print(issue['title'])
```

Feature any http verb in docs

```
print(issue['comments']) 3
```

عالیه، ما سه تا نظر داریم. باید نگاهی به آخرین نظر بیندازیم.

```
r = requests.get(r.url + '/comments')
```

```
r.status_code 200 comments = r.json()
```

```
print(comments[0].keys()) ['body', 'url',
```

```
'created_at', 'updated_at', 'user', 'id']
```

```
print(comments[2]['body']) Probably in the  
"advanced" section
```

خب، به نظر جای احمقانه‌ای می‌اد. بیاین یه کامنت بذاریم و به ارسال‌کننده بگیم که احمقه. اصلاً ارسال‌کننده کیه؟

```
print(comments[2]['user']['login'])
```

```
kennethreitz
```

خب، پس باید به این آقای کنت بگوییم که به نظر ما این مثال باید در راهنمای شروع سریع قرار بگیرد. طبق مستندات API گیتهاب، روش انجام این کار ارسال کد POST به نخ است. باید این کار را انجام دهیم.

```
body = json.dumps({u"body": u"Sounds  
great! I'll get right on it!"}) url = u"https://  
api.github.com/repos/psf/requests/  
issues/482/comments" r =
```

```
requests.post(url=url, data=body)
```

```
r.status_code 404
```

ها، این عجیبه. احتمالاً باید احراز هویت کنیم. این کار در دسرساز میشه، نه؟ اشتباهه. Requests استفاده از انواع مختلف احراز هویت، از جمله احراز هویت پایه که خیلی هم رایجه، رو آسودن میکنه.

```
from requests.auth import  
HTTPBasicAuth auth =  
HTTPBasicAuth('fake@example.com',  
'not_a_real_password') r =  
requests.post(url=url, data=body,  
auth=auth) r.status_code 201 content =  
r.json() print(content['body']) Sounds  
great! I'll get right on it.
```

عالی. اوه، صبر کن، نه! منظورم این بود که کمی طول میکشد، چون باید بروم به گربه‌ام غذا بدهم. کاش میتوانستم این نظر را ویرایش کنم! خوشبختانه، گیتهاب به ما اجازه می‌دهد از یک فعل HTTP دیگر، یعنی PATCH، برای ویرایش این نظر استفاده کنیم. باید این کار را انجام دهیم.

```
print(content[u"id"]) 5804413 body =  
json.dumps({u"body": u"Sounds great! I'll  
get right on it once I feed my cat."}) url =  
u"https://api.github.com/repos/psf/  
requests/issues/comments/5804413" r =  
requests.patch(url=url, data=body,
```

```
auth=auth) r.status_code 200
```

عالی. حالا، فقط برای اینکه این یارو کنت رو اذیت کنم، تصمیم گرفتم بذارم عرق کنه و بهش نگم که دارم روش کار می‌کنم. یعنی می‌خوام این نظر را حذف کنم. گیت‌هاب به ما اجازه می‌ده نظرات را با استفاده از متد فوق‌العاده و با نام **DELETE** حذف کنیم. بیا از شرش خلاص شیم.

```
r = requests.delete(url=url, auth=auth)  
r.status_code 204 r.headers['status'] '204  
No Content'
```

عالی. همه چیز تمام شد. آخرین چیزی که می‌خواهم بدانم این است که چقدر از **ratelimit** خود را استفاده کرده‌ام. بیایید بفهمیم. گیت‌هاب این اطلاعات را در هدرها ارسال می‌کند، بنابراین به جای دانلود کل صفحه، یک درخواست **HEAD** برای دریافت هدرها ارسال می‌کنم.

```
r = requests.head(url=url, auth=auth)  
print(r.headers) 'x-ratelimit-remaining':  
'4995' 'x-ratelimit-limit': '5000' ...
```

عالی. وقتی‌شیه یه برنامه پایتون بنویسیم که از API گیت‌هاب به انواع روش‌های هیجان‌انگیز سوءاستفاده کنه، ۴۹۹۵ بار دیگه.

## افعال سفارشی

هر از گاهی ممکن است با سروری کار کنید که به هر دلیلی، اجازه استفاده یا حتی نیاز به استفاده از افعال HTTP که در بالا پوشش داده نشده‌اند را می‌دهد. یک نمونه از این موارد، متد **MKCOL** است که برخی از سرورهای **WEBDAV** از آن استفاده می‌کنند. نگران نباشید، این متدها هنوز هم می‌توانند با درخواست‌ها استفاده شوند. این متدها از **request** متد داخلی استفاده می‌کنند. به عنوان مثال:

```
r = requests.request('MKCOL', url,  
data=data) r.status_code 200 # Assuming  
your call was correct
```

با استفاده از این، می‌توانید از هر فعل متدهی که سرور شما اجازه می‌دهد استفاده کنید.

## سربرگ‌های لینک

بسیاری از API‌های HTTP دارای سرآیندهای لینک هستند. آن‌ها API‌ها را خودتوصیفتر و قابل کشفتر می‌کنند.

گیت‌هاب از این موارد برای صفحه‌بندی در API خود استفاده می‌کند، برای مثال:

```
url = 'https://api.github.com/users/  
kennethreitz/repos?  
page=1&per_page=10' r =  
requests.head(url=url) r.headers['link']  
'<https://api.github.com/users/
```

```
kennethreitz/repos?  
page=2&per_page=10>; rel="next",  
<https://api.github.com/users/kennethreitz/repos?  
page=6&per_page=10>; rel="last"
```

درخواست‌ها به طور خودکار این هدرهای لینک را تجزیه و تحلیل کرده و آنها را به راحتی قابل استفاده می‌کنند:

```
r.links["next"] {"url": 'https://api.github.com/users/kennethreitz/repos?  
page=2&per_page=10', 'rel': 'next'}  
r.links["last"] {"url": 'https://api.github.com/users/kennethreitz/repos?  
page=7&per_page=10', 'rel': 'last'}
```

## آدأپتورهای حمل و نقل

از نسخه ۱.۰.۰، **Transport** با استفاده از **Requests** به یک طراحی داخلی ماژولار منتقل شده است. این اشیاء مکانیزمی برای تعریف روش‌های تعاملی برای یک سرویس **HTTP** ارائه می‌دهند. به طور خاص، آنها به شما امکان می‌دهند پیکربندی را برای هر سرویس اعمال کنید.

درخواست‌ها با یک آدأپتور انتقال واحد، **HTTPAdapter** یعنی **HTTP** و **HTTPS** را با فرض درخواست‌ها را با **HTTP** و **HTTPS** پیش‌فرض با استفاده از کتابخانه **urllib3** فراهم می‌کند.

هر زمان که یک درخواست Session مقداردهی اولیه می‌شود، یکی از این‌ها برای HTTP و دیگری برای Session به شیء متصل می‌شود HTTPS.

درخواست‌ها کاربران را قادر می‌سازد تا آداتورهای انتقال خود را که قابلیت‌های خاصی را ارائه می‌دهند، ایجاد و استفاده کنند. پس از ایجاد، یک آداتور انتقال می‌تواند به یک شیء Session متصل شود، همراه با نشانه‌ای از اینکه به کدام سرویس‌های وب باید اعمال شود.

```
s = requests.Session()
s.mount('https://github.com/', MyAdapter())
```

فراخوانی Transport mount، یک نمونه خاص از mount را به یک پیشوند ثبت می‌کند. پس از mount شدن، هر درخواست HTTP که با استفاده از آن جلسه انجام شود و URL آن با پیشوند داده شده شروع شود، از Transport Adapter داده شده استفاده خواهد کرد.

توجه داشته باشید

آداتور بر اساس طولانی‌ترین تطابق پیشوند انتخاب خواهد شد. به پیشوندهایی مانند http://localhost will also match http://http://localhost.other.com localhost@other.com توجه داشته باشید. توصیه می‌شود نام‌های میزبان کامل را با /. خاتمه دهید.

بسیاری از جزئیات پیاده‌سازی یک آداتور انتقال فراتر از محدوده این مستندات است، اما برای یک مورد استفاده ساده **SSL** به مثال بعدی نگاهی بیندازید.  
برای اطلاعات بیشتر، می‌توانید به زیرکلاس‌سازی **BaseAdapter**. نگاهی بیندازید.

# مثال: نسخه خاص SSL

تیم درخواست‌ها تصمیم خاصی گرفته است که از هر نسخه SSL پیش‌فرض در کتابخانه اصلی (`urllib3`) استفاده کند. معمولاً این خوب است، اما هر از گاهی، ممکن است نیاز به اتصال به یک نقطه پایانی سرویس داشته باشد که از نسخه‌ای استفاده می‌کند که با نسخه پیش‌فرض سازگار نیست.

شما می‌توانید با استفاده از بخش عمده‌ی پیاده‌سازی موجود **HTTPAdapter** و اضافه کردن پارامتر `ssl_version` که به `urllib3` ارسال می‌شود، از **Transport Adapters** برای این کار استفاده کنید. ما یک **Transport Adapter** خواهیم ساخت که به کتابخانه دستور می‌دهد از **SSLv3** استفاده کند:

```
import ssl from urllib3.poolmanager
import PoolManager from
requests.adapters import HTTPAdapter
class Ssl3HttpAdapter(HTTPAdapter):
    """Transport adapter" that allows us to
use SSLv3."""
    def init_poolmanager(self,
connections, maxsize, block=False):
```

```
self.poolmanager = PoolManager(  
    num_pools=connections,  
    maxsize=maxsize, block=block,  
    ssl_version=ssl.PROTOCOL_SSLv3)
```

## مثال: تلاش‌های مجدد خودکار

به طور پیش‌فرض، **Requests** اتصالات ناموفق را دوباره امتحان نمی‌کند. با این حال، می‌توان **Session** با استفاده از کلاس **urllib3.util.Retry**، تلاش‌های مجدد خودکار را با مجموعه‌ای قدرتمند از ویژگی‌ها، از جمله **backoff** در یک **Requests** پیاده‌سازی کرد:

```
from urllib3.util import Retry from  
requests import Session from  
requests.adapters import HTTPAdapter s  
= Session() retries = Retry( total=3,  
backoff_factor=0.1,  
status_forcelist=[502, 503, 504],  
allowed_methods={'POST'}, )  
s.mount('https://',  
HTTPAdapter(max_retries=retries))
```

## مسدودکننده یا غیرمسدودکننده؟

با وجود آداپتور انتقال پیش‌فرض، **Requests** هیچ نوع ورودی/خروجی غیرمسدودکننده‌ای ارائه نمی‌دهد. این ویژگی **Response.content** تا زمانی

که کل پاسخ دانلود نشده باشد، مسدود خواهد ماند.  
اگر به جزئیات بیشتری نیاز دارید، ویژگی‌های پخش  
جريانی کتابخانه (به درخواست‌های پخش  
جريانی مراجعه کنید) به شما امکان می‌دهد مقادیر  
کمتری از پاسخ را در یک زمان بازیابی کنید. با این  
حال، این فراخوانی‌ها همچنان مسدود خواهند شد.

اگر نگران استفاده از مسدود کردن ورودی/خروجی  
(IO) هستید، پروژه‌های زیادی وجود دارند که  
پایتون ترکیب می‌کنند. برخی از نمونه‌های عالی  
**requests**-  
**threads** ، **grequests** ، **requests-**  
. **httpx** و **futures**

## ترتیب سربرگ

در شرایط غیرمعمول، ممکن است بخواهید هدرها را  
به صورت مرتب ارائه دهید.

اگر **headers** به **OrderedDict** آرگومان کلمه کلیدی،  
یک مقدار " ارسال کنید، این کار باعث می‌شود هدرها  
مرتب شوند. با این حال ، ترتیب هدرهای پیش‌فرض  
استفاده شده توسط **Requests** ترجیح داده می‌شود،  
به این معنی که اگر هدرهای پیش‌فرض را  
در **headers** آرگومان کلمه کلیدی لغو کنید، ممکن است  
در مقایسه با سایر هدرها در آن آرگومان کلمه کلیدی،  
نامرتب به نظر برسند.

اگر این مشکل‌ساز است، کاربران باید با تنظیم هدرهای  
پیش‌فرض روی یک **Session** شوند، آن

را Session.headers به صورت سفارشی تنظیم کند OrderedDict. این ترتیب همیشه ترجیح داده می‌شود.

## تایم اوت‌ها

اکثر درخواست‌ها به سرورهای خارجی باید دارای یک timeout باشند، در صورتی که سرور به موقع پاسخ timeout ندهد. به طور پیش‌فرض، درخواست‌ها نمی‌شوند مگر اینکه مقدار timeout به صراحة تنظیم شده باشد. بدون timeout، کد شما ممکن است برای چند دقیقه یا بیشتر هنگ کند.

زمان انتظار اتصال، تعداد ثانیه‌هایی است که درخواست‌ها منتظر می‌مانند تا کلاینت شما اتصالی به یک دستگاه از راه دور برقرار کند (مطابق با فرآخوانی `connect()`) در سوکت. بهتر است زمان انتظار اتصال را کمی بزرگ‌تر از مضربی از ۳ تنظیم کنید، که پنجره پیش‌فرض ارسال مجدد بسته TCP است.

پس از اتصال کلاینت به سرور و ارسال درخواست HTTP، زمان خواندن (read timeout) تعداد ثانیه‌هایی است که کلاینت منتظر ارسال پاسخ از سرور می‌ماند. (به طور خاص، این تعداد ثانیه‌هایی است که کلاینت بین بایت‌های ارسالی از سرور منتظر می‌ماند. در ۹۹.۹٪ موارد، این زمان قبل از ارسال اولین بایت توسط سرور است).

اگر یک مقدار واحد برای timeout تعیین کنید، مانند

این:

```
r = requests.get('https://github.com',  
timeout=5)
```

مقدار `timeout` هم برای `the connect` و هم برای `read` اعمال خواهد شد. اگر می خواهید مقادیر را جداگانه تنظیم کنید، یک تاپل مشخص کنید:

```
r = requests.get('https://github.com',  
timeout=(3.05, 27))
```

اگر سرور راه دور خیلی کند است، می توانید با ارسال مقدار `None` به عنوان مقدار `timeout` و سپس دریافت یک فنجان قهوه، به `Requests` بگویید که برای همیشه منتظر پاسخ بماند.

```
r = requests.get('https://github.com',  
timeout=None)
```

توجه داشته باشید

زمان انتظار اتصال برای هر تلاش اتصال به یک آدرس IP اعمال می شود. اگر چندین آدرس برای یک نام دامنه وجود داشته باشد، سیستم عامل `urllib3` هر آدرس را به ترتیب امتحان می کند تا زمانی که یکی با موفقیت متصل شود. این ممکن است منجر به یک زمان انتظار اتصال کلی مؤثر چندین برابر بیشتر از زمان مشخص شده شود، به عنوان مثال، یک سرور بدون پاسخ که

دارای هر دو آدرس IPv4 و IPv6 است، زمان انتظار درک شده آن دو برابر خواهد شد ، بنابراین هنگام تنظیم زمان انتظار اتصال این نکته را در نظر بگیرید.

## توجه داشته باشید

نه زمان‌های وقفه اتصال و نه زمان‌های وقفه خواندن، ساعت دیواری نیستند . این بدان معناست که اگر درخواستی را شروع کنید و به زمان آن نگاه کنید، و سپس به زمانی که درخواست تمام می‌شود یا زمان آن به پایان می‌رسد نگاه کنید، زمان واقعی ممکن است بیشتر از چیزی باشد که شما مشخص کرده‌اید.