Requests Documentation

Publicación 1.1.0

Kenneth Reitz

07 de April de 2016

Índice general

Versión v1.1.0. (*Installation*)

Requests es una librería para HTTP, ref:*licenciada bajo Apache2 <apache2>*, escrita en Python, para seres humanos.

El módulo urllib2 que se encuentra en el estándar de Python, ofrece la mayoría de las funcionalidades necesarias para HTTP, pero su api está completamente **rota**. Fue construida para otra época, - y una web diferente–. Requiere una gran cantidad de trabajo (incluso reimplementar métodos) para ejecutar las tareas más sencillas.

Las cosas no deberían ser así. No en Python.

```
>>> r = requests.get('https://api.github.com/user', auth=('user', 'pass'))
>>> r.status_code
200
>>> r.headers['content-type']
'application/json; charset=utf8'
>>> r.encoding
'utf-8'
>>> r.text
u'{"type":"User"...'
>>> r.json()
{u'private_gists': 419, u'total_private_repos': 77, ...}
```

Ver el mismo código, sin Requests.

Requests quita las complicaciones de trabajar HTTP/1.1 en Python - haciendo que la integración con servicios web sea transparente. No hay necesidad de agregar queries a tus URLs manualmente, o convertir tu información a formularios para hacer una petición POST. La reutilización de keep-alive y conexión HTTP se hace automáticamente, todo gracias a urllib3, el cual está integrado en Requests.

CAPÍTULO 1

Testimonios

El gobierno de su Majestad, Amazon, Google, Twilio, Mozilla, Heroku, PayPal, NPR, Obama for America, Transifex, Native Instruments, The Washington Post, Twitter, SoundClound, Kippt, Readability y algunas organizaciones Federales de los Estados Unidos de América utilizan Requests internamente. Ha sido descargado más de 8,000,000 de veces desde PyPI.

Armin Ronacher Requests es el ejemplo perfecto de qué tan hermosa pueder ser una API con el nivel correcto de abstracción.

Matt DeBoard Voy a tatuarme el módulo de Python Requests de @kennethreitz, en mi cuerpo, de alguna forma. Todo completo.

Daniel Greenfeld Eliminé una librería de 1200 líneas de código enredado, con unas 10 líneas de código gracias a la librería Requests de @kennethreitz. Hoy ha sido un día GENIAL.

Kenny Meyers Python HTTP: Cuando tengas dudas, o cuando no, usa Requests. Bonita, simple pytónica.

CAPÍTULO 2

Soporte Destacado

Request está listo para al web de hoy

- URLs y Dominios internacionales
- *Keep-Alive* y Agrupamiento de conexiones (*Connection Pooling*)
- Sesiones con Cookies persistentes
- Verificación SSL al estilo navegador
- Autenticación Básica y Digest
- Elegantes Cookies en pares Llave/Valor
- Descompresión automática
- Cuerpos de respuestas Unicode
- Subida de archivos Multiparte
- Tiempos de espera de conexión
- Soporte para *.netrc*
- Python 2.6 – 3.3
- Seguridad para programación en hilos (*Thread-safety*)

CAPÍTULO 3

Guía de Usuario

Esta parte de la documentación, la cual está compuesta de prosa en su mayoría, empieza dando información general acerca de Requests, luego se centra en instrucciones paso por paso de cómo utilizar la mayoría de funcionalidades que brinda Requests.

3.1 Introducción

3.1.1 Filosofía

Requests fue desarrollado al estilo **PEP 20**.

1. Hermoso es mejor que feo.
2. Explícito es mejor que implícito.
3. Simple es mejor que complejo.
4. Complejo es mejor que complicado.
5. La legibilidad cuenta.

Todas las contribuciones a Requests deben tener en cuenta estas reglas.

3.1.2 Licencia Apache2

Hoy en día, encuentras un buen número de proyectos open source que son licenciados bajo GPL. Aunque GPL tiene su tiempo y lugar, esta licencia no es la que deberías usar para tu próximo proyecto open source.

Un proyecto publicado como GPL no puede ser utilizado en ningún producto comercial que no sea también open source.

Las licencias MIT, BSD, ISC y Apache2 son buenas alternativas a la GPL, estas permiten que tu software open source sea utilizado libremente en software propietario y de fuentes cerradas (*closed-source*).

Requests está publicado bajo los términos de la Licencia Apache2.

3.1.3 Licencia de Requests

3.2 Instalación

Esta sección de la documentación cubre la instalación de Requests. El primer paso a la hora de usar cualquier paquete es instalarlo de manera apropiada.

3.2.1 Distribute & Pip

Instalar Requests es simple usando pip:

```
$ pip install requests
```

o easy_install:

```
$ easy_install requests
```

Pero en realidad no deberías hacer esto último.

3.2.2 Cheeseshop Mirror

Si el Cheeseshop está abajo, puedes instalar Requests desde uno de los mirrors. Crate.io es uno de ellos:

```
$ pip install -i http://simple.crate.io/ requests
```

3.2.3 Obtener el código fuente

Requests se está desarrollando activamente en GitHub, donde el código está siempre disponible.

Puedes elegir entre clonar el repositorio público:

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

Descargar el tarball:

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

O descargar el zipball:

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

Una vez obtengas una copia del código fuente, puedes incluirlo en tu instalación de Python, o instalarlo en site-packages fácilmente:

```
$ python setup.py install
```

3.3 Quickstart

¿Ansioso por empezar? Esta página brinda una buena introducción sobre como empezar a utilizar Requests. Esta guía asume que ya tengas instalado Requests. Si aún no lo has hecho, ve a la sección *Instalación*.

Primero, asegúrate que:

- Requests esté *instalado*
- Requests esté *actualizado*

Empecemos con algunos ejemplos sencillos.

3.3.1 Realizar un petición

Realizar una petición en Requests muy sencillo.

Comienza importando el módulo de Requests:

```
>>> import requests
```

Ahora, intentemos obtener un página web. Para este ejemplo, vamos a obtener el timeline público de GitHub.:

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

Ahora, tenemos un objeto `Response` llamado `r`. Podemos obtener toda la información que necesitamos a partir de este objeto.

En Requests, un API simple significa que todas las formas the peticiones HTTP son obvias. Por ejemplo, así es como realizas una petición HTTP POST:

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

¿Qué tal otros tipos de peticiones HTTP?: PUT, DELETE, HEAD y OPTIONS? Todos estos son igual de 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")
```

Todo esto está bien, pero es solo el comienzo de lo que Requests puede hacer.

3.3.2 Pasar parámetros en URLs

Con frecuencia, debes enviar algún tipo de información en el *query string* de la URL. Si estuvieses creando la URL a mano, esta información estaría en forma de pares llave/valor luego del signo de interrogación en la URL, por ejemplo `httpbin.org/get?key=val`. Requests te permite proveer estos argumentos en forma de diccionario, usando el parámetro en llave (*keyword argument*) `params`. Como ejemplo, si quisieras pasar `key1=value1` y `key2=value2` a `httpbin.org/get`, usarías algo como esto:

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

Puedes ver que la URL ha sido codificada correctamente imprimiéndola:

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

Nota que cualquier llama del diccionario cuyo valor es `None` no será agregada al *query string* del URL.

3.3.3 Contenido de respuesta

Podemos leer el contenido de la respuesta del servidor. Usemos el timeline de GitHub nuevamente:

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

Requests automáticamente decodificará el contenido que viene del servidor. La mayoría de caracteres unicode serán decodificados correctamente.

Cuando ejecutas una petición, Requests tratará de obtener la codificación de la respuesta basándose en las cabeceras HTTP. La codificación del texto que Requests halló (o supuso), será utilizada cuando se acceda a `r.text`. Puedes conocer la codificación que Requests está utilizando, y cambiarla, usando la propiedad `r.encoding`:

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

Si cambias la codificación, Requests utilizará este nuevo valor de `r.encoding` siempre que se invoque a `r.text`. Podrías querer hacer esto en cualquier situación donde puedas aplicar una lógica de trabajo especial para trabajar según la codificación que esté en el contenido. Por ejemplo, HTTP y XML tienen la habilidad de especificar su codificación en su cuerpo. En situaciones como esta, deberías usar `r.content` para encontrar la codificación y después configuar `r.encoding`. Esto te permitirá usar `r.text` con la codifiación correcta.

Requests también puede utilizar codificaciones del usuario, en caso de ser necesario. Si creaste tu propia codificación, y la has registrado usando el módulo `codecs`, puedes asignar el nombre de este codec como valor de `r.encoding` y Requests se encargará de la decodificación.

3.3.4 Contenidos de respuesta binarios

También puedes acceder al cuerpo de la respuesta como bytes, para peticiones que no sean de texto:

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

Las codificaciones de transferencia `gzip` y `deflate` serán decodificadas automáticamente.

Por ejemplo, para crear una imagen a partir de datos binarios en una respuesta, puedes usar el siguiente código:

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

3.3.5 Contenido de respuesta JSON

También hay un decodificador de JSON incorporado en Requests, en caso de que estés trabajando con datos JSON:

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

Si la decodificación falla, `r.json` levantará una excepción. Por ejemplo, si la respuesta obtiene un código 401 (No Autorizado/ Unauthorized), intentar `r.json` mandará una excepción `ValueError: No JSON object could be decoded`.

3.3.6 Contenido de respuesta en crudo

En el caso extraño que quieras obtener la respuesta en crudo a nivel socket, puedes acceder `r.raw`. Si quieres hacer esto, asegúrate de pasar `stream=True` en la petición inicial. Una vez que hagas esto, puedes hacer lo siguiente:

```
>>> 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'
```

De manera general, sin embargo, deberías usar un patrón como este para guardar lo que se recibe del *stream* a un archivo:

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

Al usar `Response.iter_content` se manejará mucho de lo que deberías haber manipulado a mano usando `Response.raw` directamente. Lo de arriba es la forma preferida y recomendad de obtener el contenido obtenido

3.3.7 Cabeceras personalizadas

Si quieres agregar cabeceras HTTP a una petición, simplemente pasa un `dict` al parámetro `headers`.

Por ejemplo, en el ejemplo anterior no especificamos la cabecera content-type:

```
>>> 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)
```

3.3.8 Peticiones POST más complicadas

Típicamente, quieres enviar información en forma de formulario, como un formulario HTML. Para hacerlo, pasa un diccionario al argumento *data*. Este diccionario será codificado automáticamente como formulario al momento de realizar la petición:

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

Existen ocasiones en las que quieres enviar datos en otra codificación. Si pasas un `string` en vez de un `dict`, la información será posteada directamente.

Por ejemplo, el API v3 de GitHub acepta información en forma de JSON POST/PATCH:

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

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

3.3.9 Pasar un Archivo Multiparte en POST

Requests hace que sea simple subir archivos Multiparte:

```
>>> url = 'http://httpbin.org/post'
>>> files = {'file': open('report.xls', 'rb')}

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

Puedes establecer explícitamente el nombre del archivo, _content_type_ y encabezados:

```
>>> url = 'http://httpbin.org/post'
>>> files = {'file': ('report.xls', open('report.xls', 'rb'), 'application/vnd.ms-excel', {'Expires'

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

Si quieres, puedes enviar cadenas de caracteres para ser recibidas como archivos:

```
>>> url = 'http://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"
  },
  ...
}
```

Si estás enviando un archivo muy larga como una petición `multipart/form-data`, puedes querer hacer un *stream* de la petición. Por defecto, `requests` no lo soporta, pero hay un paquete separado que sí lo hace - `requests-toolbelt`–. Deberías leer la documentación de toolbelt para más detalles de cómo usarlo.

3.3.10 Códigos de estado de respuesta

Podemos verificar el código de estado de la respuesta:

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

Requests también incluye un objeto para buscar estados de respuesta y pueden ser referenciados fácilmente:

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

Si hacemos una mala petición (respuesta diferente a 200), podemos levantar una excepción con `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
```

Pero, debido a que nuestro `status_code` para `r` fue `200`, cuando llamamos `raise_for_status()` obtenemos:

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

Todo está bien.

3.3.11 Cabeceras de respuesta

Podemos ver las cabeceras de respuesta del servidor utilizando un diccionario:

```
>>> r.headers
{
    'status': '200 OK',
    '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'
}
```

Este diccionario es especial: está hecho únicamente para las cabeceras HTTP. De acuerdo con el RFC 7230 , los nombres de las cabeceras HTTP no hacen distinción entre mayúsculas y minúsculas.

Así que podemos acceder a las cabeceras utilizando letras mayúsculas o minúsculas:

```
>>> r.headers['Content-Type']
'application/json; charset=utf-8'

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

3.3.12 Cookies

Si una respuesta contiene Cookies, puedes acceder a ellas rápidamente:

```
>>> url = 'http://example.com/some/cookie/setting/url'
>>> r = requests.get(url)

>>> r.cookies['example_cookie_name']
'example_cookie_value'
```

Para enviar tus propias *cookies* al servidor, puedes utilizar el parámetro `cookies`:

```
>>> url = 'http://httpbin.org/cookies'
>>> cookies = dict(cookies_are='working')

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

3.3.13 Historial y Redireccionamiento

Requests realizará redireccionamiento para peticiones par todos los verbos, excepto HEAD.

GitHub redirecciona todas las peticiones HTTP hacia HTTPS. Podemos usar el método `history` del objeto `Response` para rastrear las redirecciones. Veamos que hace GitHub:

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

La lista `Response.history` contiene una lista de objetos tipo `Request` que fueron creados con el fín de completar la petición. La lista está ordenada desde la petición más antigüa, hasta las más reciente.

Si estás utilizando GET u OPTIONS, puedes deshabilitar el redireccionamiento usando el parámetro `allow_redirects`:

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

Si estás utilizando HEAD, puedes habilitar el redireccionamento de la misma manera:

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

3.3.14 Timeouts

Con el parámetro `timeout` puedes indicarle a Requests que deje de esperar por una respuesta luego de un número determinado de segundos:

```
>>> 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. (time
```

Note:

`timeout` no es el tiempo límite que la respuesta completa se descargue; de lo contrario, una excepción se levanta si el servidor no ha dado una respuesta dentro de los seguntos establecidos por `timeout` (más precisamente, si no se han recibido bytes en el socket por `timeout` segundos)

3.3.15 Errores y excepciones:

En el caso de un problema de red (falla de DNS, conexión rechazada, etc), Requests levantará una excepción tipo `ConnectionError`.

En el caso de una respuesta HTTP inválida, Requests levantará una excepción tipo :class::*HTTPError*.

Si se cumple el tiempo de espera (*timeout*), se levantará una excepción tipo `Timeout`.

Si una petición excede el número configurado de redirecciones máximas, se levantará una excepción tipo `TooManyRedirects`.

Todas las excepciones levantadas por Requests, heredan de la clase `requests.exceptions.RequestException`.

¿Listo para más? Mira la sección *avanzado*.

3.4 Uso avanzado

Este documento explica el uso de algunas funcionalidades más avanzadas de Requests.

3.4.1 Objetos de sesión

El objeto de sesión o `Session`, permite mantener ciertos parámetros a través de múltiples peticiones. De igual forma, almacena las cookies generadas en todas las peticiones que hayan usado la misma instancia de `Session`.

Un objeto sesión tiene todos los métodos del API principal de Requests.

Utilicemos una sesión para almacenar las cookies de varias peticiones:

```
s = requests.Session()

s.get('http://httpbin.org/cookies/set/sessioncookie/123456789')
r = s.get("http://httpbin.org/cookies")

print r.text
# '{"cookies": {"sessioncookie": "123456789"}}'
```

Las sesiones también pueden ser utilizadas para proveer información por defecto a los métodos de peticiones. Esto se logra asignándole propiedades a un objeto tipo `Session`:

```
s = requests.Session()
s.auth = ('user', 'pass')
s.headers.update({'x-test': 'true'})

# se envían ambos: 'x-test' y 'x-test2'
s.get('http://httpbin.org/headers', headers={'x-test2': 'true'})
```

Cualquier diccionario que sea pasado en la petición, será unido con los valores que fueron asignados a nivel de sesión. Los parámetros asignados en la petición sobreescribirán los de la sesión.

Eliminar una entrada en un Diccionario de parámetro.

En ocasiones, querrás omitir valores asignados a nivel de sesión. Para hacer esto, simplemente asigna `None`, en la petición, a la llave que se desee omitir.

Todos los valores contenidos dentro de un objeto sesión están disponibles. Ver *Session API Docs* para más información.

3.4.2 Objetos de petición y de respuesta

Siempre que se hace un llamado a `requests.get()` y amigos, está ocurriendo dos cosas importantes. Primero, se está construyendo un objeto tipo `Request`, el cual será enviado a un servidor con el fín de obtener información de éste. Segundo, un objeto `Response` es generado una vez que `requests` obtenga una respuesta del servidor. El objeto respuesta contiene toda la información entregada por el servidor, así como el objecto `Request` que fue creado originalmente. A continuación una simple petición para obtener información importante de los servidores de Wikipedia:

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

Si queremos acceder a las cabeceras que el servidor envió de vuelta, hacemos lo siguiente:

```
>>> 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'}
```

Sin embargo, si queremos obtener las cabeceras que enviamos al servidor, simplemente accedemos a la petición, y de ahí, a las cabeceras:

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

3.4.3 Peticiones Preparadas

Cuando recibes un objeto `Response` desde una llamada API o un llamado de Session, el atributo `request` es en realidad un `PreparedRequest`. En algunos casos desearías hacer algún trabajo adicional al cuerpo o los encabezados (o a cualquier cosa en realidad) antes de enviar una petición. Lo siguiente es una sencilla receta para ello:

```
from requests import Request, Session

s = Session()
req = Request('GET', url,
    data=data,
    headers=header
)
prepped = req.prepare()

# hacer algo con prepped.body
# hacer algo con prepped.headers

resp = s.send(prepped,
    stream=stream,
    verify=verify,
    proxies=proxies,
    cert=cert,
    timeout=timeout
)

print(resp.status_code)
```

Debido a que no estás haciendo nada especial con el objeto `Requests`, lo prepara inmediatemente y modifica el objeto `PreparedRequest`. Entonces lo envía con el otro parámetro que habría sido enviado a `requests.*` o‘‘Sesssion.*‘‘.

Sin embargo, el código de arriba perderá algunas de las ventajas de tener un objeto Requests *`Session`*. En especial, el estado de nivel *`Session`* como lo es cookies no será aplicado a tu petición. Para obtener un `PreparedRequest` con ese estado aplicado, reemplace la llamada

`Request.prepare()` con una invocación a `Session.prepare_request()`, como lo siguiente:

```
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
# do something with prepped.headers

resp = s.send(prepped,
    stream=stream,
    verify=verify,
    proxies=proxies,
    cert=cert,
    timeout=timeout
)

print(resp.status_code)
```

3.4.4 Validación de Certificados SSL

Requests puede verificar certificados SSL para peticiones HTTPS, al igual que un navegador web. Para validar el certificado SSL de algún host, podemos utilizar el argumento `verify`:

```
>>> requests.get('https://kennethreitz.com', verify=True)
requests.exceptions.SSLError: hostname 'kennethreitz.com' doesn't match either of '*.herokuapp.com',
```

Debido a que no tengo SSL en este dominio, la petición falla. Intentemos ahora con GitHub:

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

Para utilizar certificados privados, puedes pasar la ruta a un archivo CA_BUNDLE en el parámetro `verify`, o asignar el valor en la variable de entorno `REQUESTS_CA_BUNDLE`.

Requests puede saltarse la verificación si pasas `verify=False` en la petición.:

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

Por defecto, `verify` tiene el valor `True`, y solo aplica para certificados del host.

También puedes especificar un certificado local para utilizar un certificado en el lado del cliente; existen dos maneras, la primera como un archivo que contenga la llave privada y el certificado, o como una tupla con las rutas de ambos archivos:

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

Si pasas una ruta inválida, o un certificado inválido:

```
>>> requests.get('https://kennethreitz.com', cert='/wrong_path/server.pem')
SSLError: [Errno 336265225] _ssl.c:347: error:140B0009:SSL routines:SSL_CTX_use_PrivateKey_file:PEM
```

3.4.5 Workflow del cuerpo del contenido

Por defecto, cuando realizas una petición, el cuerpo de la respuesta es descargado inmediatamente. Este comportamiento se puede cambiar, postergando la descarga al momento en el que se acceda el atributo `Response.content`, con el parámetro `stream=True`:

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

En este momento, únicamente las cabeceras de respuesta han sido descargadas, y la conexión permanece abierta, lo que nos permite realizar una descarga del contenido condicionada:

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

Puedes controlar aún más este *workflow* utilizando los métodos `Response.iter_content` y `Response.iter_lines`, o leyendo desde la clase de urllib3 subyacente `urllib3.HTTPResponse` en `Response.raw`.

Si condiguras `stream` a `True` mientras estás haciendo una petición, Requests no puede liberar la conexión al *pool* a menos de que consumas todos los datos o llames a `Response.close`. Esto puede llevarte a ineficiencia con las conexiones. Si te encuentras leyendo cuerpos de peticiones (o no leyéndolos del todo) mientras estás usando `stream=True`, deberías considerar el usar `contextlib.closing` (documentado aquí), así:

```
from contextlib import closing

with closing(requests.get('http://httpbin.org/get', stream=True)) as r:
    # hacer cosas con la respuesta.
```

3.4.6 Keep-Alive

Buenas noticias - gracias a urllib3, *keep-alive* es 100 % automático dentro de una sesión! Cualquier petición que se ejecute dentro de una sesión, reutilizará la conexión apropiada!

Note que las conexiones sólo son devueltas a la piscina *pool* una vez se haya leído toda la información en el cuerpo de la respuesta. Asegúrese de pasar `stream=False`, o de leer la propiedad `content` del objeto `Response`.

3.4.7 Subir por Streaming

Requests soporta subidas por *streaming*, lo cual permite enviar archivos pesados sin leerlos en memoria. Para usar esta funcionalidad, simplemente debes proveer un objeto tipo archivo para el cuerpo de la petición:

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

3.4.8 Peticiones Fragmentadas *Chunk-Encoded*

Requests también soporta transferencias fragmentadas para peticiones de entrada y salida. Para enviar una petición por fragmentos, simplemente debes proveer un objeto generador (o cualquier iterador sin tamaño) para el cuerpo de la petición:

```
def gen():
    yield 'hi'
    yield 'there'

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

3.4.9 Hooks de eventos

Requests tiene un sistem de *hooks* que puedes utilizar para manipular porciones del proceso de petición, o manipulación de señales.

Hooks disponibles:

response: La respuesta generada a partir de Request.

Puedes asignar una función a este *hook* en cada petición, pasando un diccionario `{hook_name: callback_funcion}` al parámetro `hooks` de la misma:

```
hooks=dict(response=print_url)
```

La función `callback_function` recibirá una porción de datos como su primer argumento.

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

Si ocurre algún error mientras se ejecuta el *callback*, se emitirá una advertencia.

Si la función *callback* regresa algún valor, es asumido que este valor reemplazará a los datos que le fueron pasados originalmente. Si la función no regresa ningún valor, nada más es afectado.

Imprimamos algunos argumentos de la petición en tiempo de ejecución:

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

3.4.10 Autenticación personalizada

Requests permite especificar tu propio mecanismo de autenticación.

Cualquier objeto invocable (*callable*) que se pase en el parámetro `auth` en una petición, podrá modificar esta petición antes de que sea ejecutada.

Las implementaciones de autenticación son clases heredadas de `requests.auth.AuthBase` y son fáciles de definir. Requests provee implementaciones de dos formas de autenticación comunes en `requests.auth`: `HTTPBasicAuth` y `HTTPDigestAuth`.

Supongamos que tenemos un servicio web que responderá únicamente si la cabecera `X-Pizza` contiene cierta contraseña. Es poco probable, pero es un buen ejemplo.

```
from requests.auth import AuthBase

class PizzaAuth(AuthBase):
    """Attaches HTTP Pizza Authentication to the given Request object."""
    def __init__(self, username):
        # configurar cualquier dato de auth-related aquí
        self.username = username

    def __call__(self, r):
        # modify and return the request
        r.headers['X-Pizza'] = self.username
        return r
```

Ahora, podemos crear una petición usando nuestra implementación de Pizza Auth:

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

3.4.11 Peticiones en streaming

Usando `requests.Response.iter_lines()` puedes iterar fácilmente sobre APIs de streaming como el API de Streaming de Twitter. Configura `stream` a `True` e itera con la respuesta usando *`iter_lines()`*:

```
import json
import requests

r = requests.get('http://httpbin.org/stream/20', stream=True)

for line in r.iter_lines():

    # filter out keep-alive new lines
    if line:
        print json.loads(line)
```

3.4.12 Proxies

Si necesitas utilizar un proxy, puedes configurar peticiones individuales usando el argumento `proxies` de la petición:

```
import requests

proxies = {
  "http": "http://10.10.1.10:3128",
  "https": "http://10.10.1.10:1080",
}

requests.get("http://example.org", proxies=proxies)
```

También puedes configurar proxies por medio de las variables de entorno `HTTP_PROXY` y `HTTPS_PROXY`.

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

Para usar HTTP Basi Auth con tu proxy, debe utilzar la sintáxis *http://user:password@host/*:

```
proxies = {
    "http": "http://user:pass@10.10.1.10:3128/",
}
```

3.4.13 Conformidad

Requests está pensado para que sea conforme con todas las especificaciones que apliquen, así como con RFCs, siempre y cuando esto no traiga consigo complicaciones para los usuarios. Estos cuidados con las especificaciones pueden llevar a comportamientos que para algunas personas que no estén familiarizadas con ellas.

Codificaciones

Cuando recibes una respuesta, Requests supone automáticamente la codificación a usar cuando accesas al atributo *`Response.text`*. Requests primero verificará alguna codificación en el encabezado HTTP, si no se ha especificado, se utilizará charade para intentar adivinar la codificación.

La única ocasión en la que Requests no intentará adivinar la codificación, es cuando no hay un *charset* explícito en las cabeceras HTTP **y** la cabecera `Content-Type` contiene `text`. En tal caso, el RFC 2616 especifica que el *charset* por defecto será `ISO-8859-1`. Requests obedecerá la especificación en este caso. Si se necesita una codificación diferente, puedes establecer manualmente el atributo *`Response.encoding`* o usar *`Response.content`*.

3.4.14 Verbos HTTP

Requests provee acceso a casi todo el rango de verbos HTTP: GET, OPTIONS, HEAD, POST, PUT, PATCH y DELETE. A continuación, se expondrán algunos ejemplos detallados de como usar estos verbos en Requests, usando el API de GitHub.

Comenzaremos con el verbo más común: GET. HTTP GET es un método idempotente el cual regresa un recurso a partir de una URL; por lo tanto, este verbo es utilizado cuando se quiere obtener información desde una ubicación web. Un ejemplo de uso, es el de obtener información acerca de un commit específico en GitHub. Supongamos que queremos obtener el commit `a050faf` de Requests. Lo hacemos de la siguiente manera:

```
>>> import requests
>>> r = requests.get('https://api.github.com/repos/kennethreitz/requests/git/commits/a050faf084662f3a
```

Debemos confirmar que GitHub respondió correctamente; en caso afirmativo, queremos conocer el tipo de contenido, así:

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

De tal manera que GitHub regresa JSON. Genial, podemos utilizar el método `r.json` para procesarlo en objetos de Python.

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

Hasta ahora todo ha sido sencillo. Pues bien, vamos a investigar el GitHub un poco más. Ahora podríamos ver la documentación, pero podríamos divertirnos un poco más si usáramos Requests. Podemos utilizar el verbo OPTIONS soportado por Requests para ver qué tipo de métodos HTTP están soportados en la URL que acabamos de utilizar.

```
>>> verbs = requests.options(r.url)
>>> verbs.status_code
500
```

¿Qué? ¡Esto no nos ayuda! Resulta que GitHub, al igual que muchos proveedores de APIs, no implementan el método OPTIONS. Esto es algo molesto, pero está bien, podemos utilizar la aburrida documentación. Si GitHub hubiese implementado correctamente el verbo OPTIONS, debería regresar los métodos permitidos en las cabeceras, por ejemplo:

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

Al observar la documentación, vemos que solo hay otro método permitido para commits, el cual es POST, y lo que hace es crear un nuevo commit. Debido a que estamos utilizando el repositorio de Requests, vamos a evitar crear POSTS manualmente. En lugar de esto, vamos a jugar un poco con la funcionalidad de Issues de GitHub.

Esta documentación fue agregada en respuesta al Issue #482. Dado que este reporte ya existe, vamos a utilizarlo como ejemplo. Vamos a empezar por obtener este recurso.

```
>>> r = requests.get('https://api.github.com/repos/kennethreitz/requests/issues/482')
>>> r.status_code
200
>>> issue = json.loads(r.text)
>>> print issue[u'title']
Feature any http verb in docs
>>> print issue[u'comments']
3
```

Bien, ahora tenemos tres comentarios. Ahora, miremos el último de los comentarios.

```
>>> r = requests.get(r.url + u'/comments')
>>> r.status_code
200
>>> comments = r.json()
```

```
>>> print comments[0].keys()
[u'body', u'url', u'created_at', u'updated_at', u'user', u'id']
>>> print comments[2][u'body']
Probably in the "advanced" section
```

Bueno, esto parece ser algo tonto. Vamos a postear un comentario diciéndole al posteador que es un tonto. Quién es el posteador?

```
>>> print comments[2][u'user'][u'login']
kennethreitz
```

Esta bien, vamos a decirle a este sujeto Kenneth que pensamos que este ejemplo debe ir en la sección *quickstart*. De acuerdo con la documentación del API de GitHub, la forma de hacer esto es haciendo un POST a la conversación (*thread*). Hagámoslo.

```
>>> body = json.dumps({u"body": u"Sounds great! I'll get right on it!"})
>>> url = u"https://api.github.com/repos/kennethreitz/requests/issues/482/comments"
>>> r = requests.post(url=url, data=body)
>>> r.status_code
404
```

¿?, Esto es extraño. Probablemente necesitemos autenticarnos. Esto será problemático, verdad? Pues no, Requests hace que usar varios métodos de autenticación sea fácil, incluyendo Basic Auth.

```
>>> 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[u'body']
Sounds great! I'll get right on it.
```

Genial. Mmm ¡No, espera! Quería agregar que me tomará un tiempo, ya que tengo que alimentar a mi gato. Si tan solo pudiera editar este comentario! Por fortuna, GitHub nos permite usar el verbo HTTP PATCH para editar este comentario. Vamos a hacerlo.

```
>>> 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/kennethreitz/requests/issues/comments/5804413"
>>> r = requests.patch(url=url, data=body, auth=auth)
>>> r.status_code
200
```

Excelente. Ahora, solo por hacerle la vida imposible a este sujeto Kenneth, he decidido hacerle preocupar al no informarle que estoy trabajando en esto. Esto quiere decir que quiero eliminar este comentario. GitHub nos permite eliminar comentarios utilizando el método DELETE. Vamos a deshacernos de este comentario.

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

Excelente. Se ha ido. Por último, quiero saber qué tanto he utilizado el API. GitHub envía esta información en las cabeceras, así que en vez de descargar la página completa, voy a enviar una petición tipo HEAD, para obtener los encabezados.

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

Excelente. Es hora de escribir una aplicación en Python que abuse del API de GitHub otras 4995 veces.

3.4.15 Link Headers

Muchas APIs soportan *Link headers*. Estas cabeceras hacen que las APIs sean más auto-descriptivas y detectables.

GitHub las utiliza para paginación en su API, por ejemplo:

```
>>> 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.githu
```

Requests procesará automáticamente estas cabeceras y hará que sean fácilmente utilizables:

```
>>> 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'}
```

3.4.16 Transport Adapters

A partir de v1.0.0, Requests se movió a un diseño interno de tipo modular. Parte de las razones de ello fue el implementar los Adaptores de Transporte (*Transport Adapters*), originalmente **'descritas aquí'_**. Los Transport Adapters proveen un mecanismo para definir métodos de interacción para un servicio HTTP.En especial, permiten aplicar configuración por cada servicio (*per-service*).

Request viene con un Transpor Adapter sencillo, el `HTTPAdapter`. Este adaptador provee la interacción por defecto de Request con HTTP y HTTPS usando la poderosa biblioteca

urllib3. En cualquier momento en que alguna clase Request se

inicializa, uno de estos es adjuntada al objeto *`Session`* para HTTP y otra para HTTPS.

Requests permite a los usuarios crear y usar sus propios Transport Adapters que proveean funcionalidad específica. Una vez creados, un Transport Adapter puede ser montado en un objeto `Session`, junto con una indicación de cuáles servicios web debería aplicar.

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

El montaje llama a registros de instancias específicas de un Transport Adapter a un prefijo. Una vez montada, cualquier petición HTTP hecha con esa sesión cuya URL inicie con el prefijo dado usará dicho Transport Adapter.

Muchos de los detalles de implementar un Transport Adapter está más allá del alcance de esta documentación, pero mira en el siguiente ejemplo para caso de uso sencillo de SSL. Para más sobre el asunto, deberías mirar en la subclase `requests.adapters.BaseAdapter`.

Ejemplo: Versión Específica de SSL

El equipo de Requests ha hecho una elección específica en usar cualquier versión SSL por defecto en la biblioteca (urllib3). Normalmente esto está bien, pero de vez en vez, podrías encontrarte en la necesidada de conectarte a un *service-endpoint* que usa una versión que no es compatible con la default.

Puedes usar Transport Adapters para esto al tomar la mayoría de la implementación existente de HTTPAdaptar, y agregando un parámetro *ssl_version* que obtiene pasando por *urllib3*. Haremos un TA que instruya a la biblioteca a usar SSLv3:

```
import ssl

from requests.adapters import HTTPAdapter
from requests.packages.urllib3.poolmanager import PoolManager


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)
```

3.4.17 Bloqueante o no-Bloqueante

Con el Transport Adapter puesto en su sitio, Requests no provee ningún tipo de IO no-bloqueante. La propiedad `Response.content` bloquerá hasta que la respuesta completa haya sido descargada. Si se requiere más granularidad, la característica de *streaming* de la biblioteca (vea streaming-requests) permite obtener pequeñas cantidades de una respuesta a la vez. Sin embargo, esas llamadas serán aún bloqueantes.

Si tienes preocupación sobre el uso de IO bloqueante, hay muchos proyectos por ahí que combinan Requests con alguno de los Framework asincrónicos de Python. Dos excelentes ejemplos son grequests y requests-futures.

3.4.18 Timeouts

La mayoría de las peticiones externas deben tener un timeout anexo, en caso de que el servidor no esté respondiendo a tiempo.

El timeout **connect** es el número de segundos que Request esperará para que tu cliente establesca una conexión a una máquina remota (correspondiente al método connect()) en el socket. Es una buena práctica establecer tiempos de conexión a algo un poco más grande que un múltiplo de 3, para permitir el tiempo por defecto TCP retransmission window.

Una vez que tu cliente se ha conectado al servidor y enviado la petición HTTP, el timeout **read** es el número de segundos que el cliente esperará para que el servidor envie una respuesta. (Específicamente, es el número de segundos que el cliente esperará *entre* los bytes enviados desde el servidor. En la práctica, esto es el tiempo antes de que el servidor envíe el primer byte).

Si especificas un solo volor para el timeout, como esto:

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

El valor de timeout será aplicado a ambos timeouts: `connect` y `read`. Especifique una tupla si deseas establecer el valor separadamente:

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

Si el servidor remoto es demasiado lente, puedes decirle a Request que espere por siempre la respuesta, pasando None como el valor de timeout.

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

3.5 Autenticación

En este documento se discuten varios métodos de autenticación con Requests.

Muchos servicios web requieren autenticación, y existen muchas formas de hacerlo. Abajo, se expondrán las nociones generales de varias formas de autenticación presentes en Requests, desde lo simple hasta lo complejo.

3.5.1 Autenticación Básica

Muchos servicios web aceptan autenticación mediante HTTP Basic Auth. Esta es la forma más sencilla, y es soportada por Requests.

Ejecutar peticiones con autenticación básica HTTP es sencillo:

```
>>> from requests.auth import HTTPBasicAuth
>>> requests.get('https://api.github.com/user', auth=HTTPBasicAuth('user', 'pass'))
<Response [200]>
```

De hecho, HTTP Basic Auth es tan común, que Requests provee una forma más fácil de usarla:

```
>>> requests.get('https://api.github.com/user', auth=('user', 'pass'))
<Response [200]>
```

Pasar las credenciales en un tupla de esta manera, es exactamente igual a utilizar `HTTPBasicAuth` como en el primer ejemplo.

Autenticación netrc

Si no se ha dado ningún método de autenticación en el argumento `auth`, Requests intentará obtener las credenciales de autenticación para el hostname del URL del archivo de usuario netrc.

Si se encuentran las credenciales para el hostname, la petición es enviada con HTTP Basic Auth.

3.5.2 Autenticación Digest

Otra forma popular de autenticación HTTP es Digest Authenticacion, y Requests la soporta de manera similar:

```
>>> from requests.auth import HTTPDigestAuth
>>> url = 'http://httpbin.org/digest-auth/auth/user/pass'
>>> requests.get(url, auth=HTTPDigestAuth('user', 'pass'))
<Response [200]>
```

3.5.3 Autenticación OAuth 1

Una forma común de autenticación para varios API Web es OAuth. La biblioteca `requests-oauthlib` permite a los usuarios de Requests el hacer peticiones de autentición OAuth con facilidad:

```
>>> import requests
>>> from requests_oauthlib import OAuth1

>>> url = 'https://api.twitter.com/1.1/account/verify_credentials.json'
>>> auth = OAuth1('YOUR_APP_KEY', 'YOUR_APP_SECRET',
                  'USER_OAUTH_TOKEN', 'USER_OAUTH_TOKEN_SECRET')

>>> requests.get(url, auth=auth)
<Response [200]>
```

Para más información en cómo funciona OAuth, por favor vea el website oficial de OAuth website. Para ejemplos y documentación con requests-oauthlib, por favor vea el repositorio requests_oauthlib en GitHub.

3.5.4 Otras formas de autenticación

Requests está diseñado para permitir que otras formas de autenticación puedan ser incluidas fácilmente. Los miembros de la comunidad open-source con frecuencia escriben *handlers* de autenticación para formas de autenticación más complicadas o que son usadas con menos frecuencia. Algunos de estos handlers han sido recopilados por la Requests organization, entre los que se incluyen:

- Kerberos
- NTLM

Si quieres usar alguna de estas formas de autenticación, dirígete a su página en GitHub y sigue las instrucciones.

3.5.5 Nuevas formas de autenticación

Si no logras encontrar una buena implementación del método de autenticación que deseas utilizar, puedes implementarlo tu mismo. Requests hace que sea fácil agregar tu propia forma de autenticación.

Para hacer esto, escribe una clase que herede de `requests.auth.AuthBase` e implementa el método `__call__()`:

```
>>> import requests
>>> class MyAuth(requests.auth.AuthBase):
...     def __call__(self, r):
...         # Implementar mi autenticación
...         return r
...
>>> url = 'http://httpbin.org/get'
>>> requests.get(url, auth=MyAuth())
<Response [200]>
```

Cuando un handler de autenticación es añadido a una petición, éste es llamado durante la preparación de la misma. Por lo tanto. El método `__call__()` debe hacer todo lo que se necesite para que la autenticación funcione. Algunos métodos de autenticación adicionarán hooks para brindar más funcionalidad.

Algunos ejemplos se encuentran en Requests organization y en el archivo `auth.py`.

CAPÍTULO 4

Guía para la comunidad

Esta parte de la documentación, la cual está compuesta de prosa en su mayoría, detalla el ecosistema alrededor de Requests y su comunidad.

4.1 Preguntas Frecuentes

Esta parte de la documentación responde preguntas comunes acerca de Requests.

4.1.1 ¿Datos Codificados?

Requests descomprime automáticamente respuestas codificadas con gzip, y hace su mejor esfuerzo por decodificar el contenido de la respuesta a Unicode, cuando es posible.

También puedes tener acceso directo a la respuesta en crudo (e incluso al socket) de ser necesario.

4.1.2 ¿Agentes de usuario propios?

Requests te permite reimplementar las cadenas del Agente de usuario, al igual que cualquier otra cabecera HTTP.

4.1.3 ¿Por qué no Httplib2?

Chris Adams dió un excelente resumen en Hacker News:

> httplib2 es parte del porque debes usar requests: es mucho más respetable como cliente pero no está tan bien documentado y aún requiere de mucho código para realizar las operaciones básicas. Aprecio lo que httplib2 intenta realizar, existe un buen número de molestias de bajo nivel a la hora de crear un cliente HTTP moderno, pero en serio, simplemente usa requests. Kenneth Reitz está motivado y entiende el grado al que las cosas simples deben ser simples, mientras que httplib2 se siente más como un ejercicio académico en lugar de algo que las personas deberían utilizar para desarrollar sistemas de producción[1]
>
> Divulgación: Estoy en la lista de autores de requests, pero solamente me puedo dar crédito por alrededor del 0.0001 % de su genialidad.
>
> 1. http://code.google.com/p/httplib2/issues/detail?id=96 es un buen ejemplo: un molesto bug que afectaba a muchas personas, para el cual había una solución desde hace meses, esta solución funcionó estupendamente cuando la apliqué a un *fork* y pude moler un par de TB de información con ella, pero tardó más de un año para que llegara a *trunk* y aún más para que llegara a PyPI donde cualquier otro proyecto que requería "httplib2" obtendría la versión funcional.

4.1.4 ¿Soporte para Python 3?

¡Sí! Esta es una lista de plataformas de Python que están soportadas oficialmente:

- Python 2.6
- Python 2.7
- Python 3.1
- Python 3.2
- Python 3.3
- PyPy 1.9

4.2 Integraciones

4.2.1 ScraperWiki

ScraperWiki es un excelente servicio que permite ejecutar en la web *scrappers* escritos en Python, Ruby y PHP. Ahora, Requests v0.6.1 está disponible para usar en tus *scrappers*!

Para probar, simplemente:

```
import requests
```

4.2.2 Python para iOS

Requests está incorporado en Python para iOS

Para probar, simplemente:

```
import requests
```

4.3 Artículos y Charlas

- Python para la web enseña como usar Python para interactuar con la web usando Requests.
- Reseña de Requests por Daniel Greenfield
- Mi charla: 'Python para Humanos' (audio)
- Charla 'consumiendo Web APIs' por Issac Kelly
- Entrada de Blog acerca de Requests via Yum
- Entrada de Blog en Ruso presentando Requests

4.4 Soporte

Si tienes alguna pregunta o problemas relacionados con Requests, existen algunas opciones:

4.4.1 Envía un Tweet

Si tu pregunta tiene menos de 140 caracteres, no dudes en enviar un tweet a @kennethreitz.

4.4.2 Reporta un problema

Si observas algún comportamiento inesperado en Requests, o quieres que alguna característica nueva sea soportada, reporta el problema en GitHub.

4.4.3 E-mail

Estoy encantado de responder cualquier pregunta de carácter personal, o preguntas a fondo acerca de Requests. No dude en escribir a requests@kennethreitz.com.

4.4.4 IRC

El canal oficial de Requests en Freenode es #python-requests

También estoy disponible como **kennethreitz** en Freenode.

4.5 Actualizaciones

Existen varias opciones para estar actualizado con la comunidad y el desarrollo de Requests:

4.5.1 GitHub

La mejor forma de seguir el desarrollo de Requests es en el repositorio en GitHub.

4.5.2 Twitter

Regularmente hago tweets acerca de nuevas funcionalidades y publicaciones de Requests.

Sigue a @kennethreitz para recibir actualizaciones.

4.5.3 Mailing List

Existe una lista de correos para Requests. Para subscribirse envía un correo a requests@librelist.org.

4.6 Software Updates

4.6.1 Release History

2.2.1 (2014-01-23)

Bugfixes

- Fixes incorrect parsing of proxy credentials that contain a literal or encoded '#' character.
- Assorted urllib3 fixes.

2.2.0 (2014-01-09)

API Changes

- New exception: `ContentDecodingError`. Raised instead of `urllib3` `DecodeError` exceptions.

Bugfixes

- Avoid many many exceptions from the buggy implementation of `proxy_bypass` on OS X in Python 2.6.
- Avoid crashing when attempting to get authentication credentials from ~/.netrc when running as a user without a home directory.
- Use the correct pool size for pools of connections to proxies.
- Fix iteration of `CookieJar` objects.
- Ensure that cookies are persisted over redirect.
- Switch back to using chardet, since it has merged with charade.

2.1.0 (2013-12-05)

- Updated CA Bundle, of course.
- Cookies set on individual Requests through a `Session` (e.g. via `Session.get()`) are no longer persisted to the `Session`.
- Clean up connections when we hit problems during chunked upload, rather than leaking them.
- Return connections to the pool when a chunked upload is successful, rather than leaking it.
- Match the HTTPbis recommendation for HTTP 301 redirects.
- Prevent hanging when using streaming uploads and Digest Auth when a 401 is received.
- Values of headers set by Requests are now always the native string type.
- Fix previously broken SNI support.
- Fix accessing HTTP proxies using proxy authentication.
- Unencode HTTP Basic usernames and passwords extracted from URLs.
- Support for IP address ranges for no_proxy environment variable
- Parse headers correctly when users override the default `Host:` header.
- Avoid munging the URL in case of case-sensitive servers.
- Looser URL handling for non-HTTP/HTTPS urls.
- Accept unicode methods in Python 2.6 and 2.7.
- More resilient cookie handling.
- Make `Response` objects pickleable.
- Actually added MD5-sess to Digest Auth instead of pretending to like last time.
- Updated internal urllib3.
- Fixed @Lukasa's lack of taste.

2.0.1 (2013-10-24)

- Updated included CA Bundle with new mistrusts and automated process for the future
- Added MD5-sess to Digest Auth
- Accept per-file headers in multipart file POST messages.
- Fixed: Don't send the full URL on CONNECT messages.
- Fixed: Correctly lowercase a redirect scheme.
- Fixed: Cookies not persisted when set via functional API.
- Fixed: Translate urllib3 ProxyError into a requests ProxyError derived from ConnectionError.
- Updated internal urllib3 and chardet.

2.0.0 (2013-09-24)

API Changes:

- Keys in the Headers dictionary are now native strings on all Python versions, i.e. bytestrings on Python 2, unicode on Python 3.
- Proxy URLs now *must* have an explicit scheme. A `MissingSchema` exception will be raised if they don't.
- Timeouts now apply to read time if `Stream=False`.
- `RequestException` is now a subclass of `IOError`, not `RuntimeError`.
- Added new method to `PreparedRequest` objects: `PreparedRequest.copy()`.
- Added new method to `Session` objects: `Session.update_request()`. This method updates a `Request` object with the data (e.g. cookies) stored on the `Session`.
- Added new method to `Session` objects: `Session.prepare_request()`. This method updates and prepares a `Request` object, and returns the corresponding `PreparedRequest` object.
- Added new method to `HTTPAdapter` objects: `HTTPAdapter.proxy_headers()`. This should not be called directly, but improves the subclass interface.
- `httplib.IncompleteRead` exceptions caused by incorrect chunked encoding will now raise a Requests `ChunkedEncodingError` instead.
- Invalid percent-escape sequences now cause a Requests `InvalidURL` exception to be raised.
- HTTP 208 no longer uses reason phrase `"im_used"`. Correctly uses `"already_reported"`.
- HTTP 226 reason added (`"im_used"`).

Bugfixes:

- Vastly improved proxy support, including the CONNECT verb. Special thanks to the many contributors who worked towards this improvement.
- Cookies are now properly managed when 401 authentication responses are received.
- Chunked encoding fixes.
- Support for mixed case schemes.
- Better handling of streaming downloads.
- Retrieve environment proxies from more locations.
- Minor cookies fixes.

- Improved redirect behaviour.
- Improved streaming behaviour, particularly for compressed data.
- Miscellaneous small Python 3 text encoding bugs.
- `.netrc` no longer overrides explicit auth.
- Cookies set by hooks are now correctly persisted on Sessions.
- Fix problem with cookies that specify port numbers in their host field.
- `BytesIO` can be used to perform streaming uploads.
- More generous parsing of the `no_proxy` environment variable.
- Non-string objects can be passed in data values alongside files.

1.2.3 (2013-05-25)

- Simple packaging fix

1.2.2 (2013-05-23)

- Simple packaging fix

1.2.1 (2013-05-20)

- Python 3.3.2 compatibility
- Always percent-encode location headers
- Fix connection adapter matching to be most-specific first
- new argument to the default connection adapter for passing a block argument
- prevent a KeyError when there's no link headers

1.2.0 (2013-03-31)

- Fixed cookies on sessions and on requests
- Significantly change how hooks are dispatched - hooks now receive all the arguments specified by the user when making a request so hooks can make a secondary request with the same parameters. This is especially necessary for authentication handler authors
- certifi support was removed
- Fixed bug where using OAuth 1 with body `signature_type` sent no data
- Major proxy work thanks to @Lukasa including parsing of proxy authentication from the proxy url
- Fix DigestAuth handling too many 401s
- Update vendored urllib3 to include SSL bug fixes
- Allow keyword arguments to be passed to `json.loads()` via the `Response.json()` method
- Don't send `Content-Length` header by default on `GET` or `HEAD` requests
- Add `elapsed` attribute to `Response` objects to time how long a request took.

- Fix `RequestsCookieJar`
- Sessions and Adapters are now picklable, i.e., can be used with the multiprocessing library
- Update charade to version 1.0.3

The change in how hooks are dispatched will likely cause a great deal of issues.

1.1.0 (2013-01-10)

- CHUNKED REQUESTS
- Support for iterable response bodies
- Assume servers persist redirect params
- Allow explicit content types to be specified for file data
- Make merge_kwargs case-insensitive when looking up keys

1.0.3 (2012-12-18)

- Fix file upload encoding bug
- Fix cookie behavior

1.0.2 (2012-12-17)

- Proxy fix for HTTPAdapter.

1.0.1 (2012-12-17)

- Cert verification exception bug.
- Proxy fix for HTTPAdapter.

1.0.0 (2012-12-17)

- Massive Refactor and Simplification
- Switch to Apache 2.0 license
- Swappable Connection Adapters
- Mountable Connection Adapters
- Mutable ProcessedRequest chain
- /s/prefetch/stream
- Removal of all configuration
- Standard library logging
- Make Response.json() callable, not property.
- Usage of new charade project, which provides python 2 and 3 simultaneous chardet.
- Removal of all hooks except 'response'

- Removal of all authentication helpers (OAuth, Kerberos)

This is not a backwards compatible change.

0.14.2 (2012-10-27)

- Improved mime-compatible JSON handling
- Proxy fixes
- Path hack fixes
- Case-Insensistive Content-Encoding headers
- Support for CJK parameters in form posts

0.14.1 (2012-10-01)

- Python 3.3 Compatibility
- Simply default accept-encoding
- Bugfixes

0.14.0 (2012-09-02)

- No more iter_content errors if already downloaded.

0.13.9 (2012-08-25)

- Fix for OAuth + POSTs
- Remove exception eating from dispatch_hook
- General bugfixes

0.13.8 (2012-08-21)

- Incredible Link header support :)

0.13.7 (2012-08-19)

- Support for (key, value) lists everywhere.
- Digest Authentication improvements.
- Ensure proxy exclusions work properly.
- Clearer UnicodeError exceptions.
- Automatic casting of URLs to tsrings (fURL and such)
- Bugfixes.

0.13.6 (2012-08-06)

- Long awaited fix for hanging connections!

0.13.5 (2012-07-27)

- Packaging fix

0.13.4 (2012-07-27)

- GSSAPI/Kerberos authentication!
- App Engine 2.7 Fixes!
- Fix leaking connections (from urllib3 update)
- OAuthlib path hack fix
- OAuthlib URL parameters fix.

0.13.3 (2012-07-12)

- Use simplejson if available.
- Do not hide SSLErrors behind Timeouts.
- Fixed param handling with urls containing fragments.
- Significantly improved information in User Agent.
- client certificates are ignored when verify=False

0.13.2 (2012-06-28)

- Zero dependencies (once again)!
- New: Response.reason
- Sign querystring parameters in OAuth 1.0
- Client certificates no longer ignored when verify=False
- Add openSUSE certificate support

0.13.1 (2012-06-07)

- Allow passing a file or file-like object as data.
- Allow hooks to return responses that indicate errors.
- Fix Response.text and Response.json for body-less responses.

0.13.0 (2012-05-29)

- Removal of Requests.async in favor of grequests
- Allow disabling of cookie persistiance.
- New implimentation of safe_mode
- cookies.get now supports default argument
- Session cookies not saved when Session.request is called with return_response=False
- Env: no_proxy support.
- RequestsCookieJar improvements.
- Various bug fixes.

0.12.1 (2012-05-08)

- New `Response.json` property.
- Ability to add string file uploads.
- Fix out-of-range issue with iter_lines.
- Fix iter_content default size.
- Fix POST redirects containing files.

0.12.0 (2012-05-02)

- EXPERIMENTAL OAUTH SUPPORT!
- Proper CookieJar-backed cookies interface with awesome dict-like interface.
- Speed fix for non-iterated content chunks.
- Move `pre_request` to a more usable place.
- New `pre_send` hook.
- Lazily encode data, params, files.
- Load system Certificate Bundle if `certify` isn't available.
- Cleanups, fixes.

0.11.2 (2012-04-22)

- Attempt to use the OS's certificate bundle if `certifi` isn't available.
- Infinite digest auth redirect fix.
- Multi-part file upload improvements.
- Fix decoding of invalid %encodings in URLs.
- If there is no content in a response don't throw an error the second time that content is attempted to be read.
- Upload data on redirects.

0.11.1 (2012-03-30)

- POST redirects now break RFC to do what browsers do: Follow up with a GET.
- New `strict_mode` configuration to disable new redirect behavior.

0.11.0 (2012-03-14)

- Private SSL Certificate support
- Remove select.poll from Gevent monkeypatching
- Remove redundant generator for chunked transfer encoding
- Fix: Response.ok raises Timeout Exception in safe_mode

0.10.8 (2012-03-09)

- Generate chunked ValueError fix
- Proxy configuration by environment variables
- Simplification of iter_lines.
- New *trust_env* configuration for disabling system/environment hints.
- Suppress cookie errors.

0.10.7 (2012-03-07)

- *encode_uri* = False

0.10.6 (2012-02-25)

- Allow '=' in cookies.

0.10.5 (2012-02-25)

- Response body with 0 content-length fix.
- New async.imap.
- Don't fail on netrc.

0.10.4 (2012-02-20)

- Honor netrc.

0.10.3 (2012-02-20)

- HEAD requests don't follow redirects anymore.
- raise_for_status() doesn't raise for 3xx anymore.
- Make Session objects picklable.
- ValueError for invalid schema URLs.

0.10.2 (2012-01-15)

- Vastly improved URL quoting.
- Additional allowed cookie key values.
- Attempted fix for "Too many open files" Error
- Replace unicode errors on first pass, no need for second pass.
- Append '/' to bare-domain urls before query insertion.
- Exceptions now inherit from RuntimeError.
- Binary uploads + auth fix.
- Bugfixes.

0.10.1 (2012-01-23)

- PYTHON 3 SUPPORT!
- Dropped 2.5 Support. (*Backwards Incompatible*)

0.10.0 (2012-01-21)

- `Response.content` is now bytes-only. (*Backwards Incompatible*)
- New `Response.text` is unicode-only.
- If no `Response.encoding` is specified and `chardet` is available, `Respoonse.text` will guess an encoding.
- Default to ISO-8859-1 (Western) encoding for "text" subtypes.
- Removal of *decode_unicode*. (*Backwards Incompatible*)
- New multiple-hooks system.
- New `Response.register_hook` for registering hooks within the pipeline.
- `Response.url` is now Unicode.

0.9.3 (2012-01-18)

- SSL verify=False bugfix (apparent on windows machines).

0.9.2 (2012-01-18)

- Asynchronous async.send method.
- Support for proper chunk streams with boundaries.
- session argument for Session classes.
- Print entire hook tracebacks, not just exception instance.
- Fix response.iter_lines from pending next line.
- Fix but in HTTP-digest auth w/ URI having query strings.
- Fix in Event Hooks section.
- Urllib3 update.

0.9.1 (2012-01-06)

- danger_mode for automatic Response.raise_for_status()
- Response.iter_lines refactor

0.9.0 (2011-12-28)

- verify ssl is default.

0.8.9 (2011-12-28)

- Packaging fix.

0.8.8 (2011-12-28)

- SSL CERT VERIFICATION!
- Release of Cerifi: Mozilla's cert list.
- New 'verify' argument for SSL requests.
- Urllib3 update.

0.8.7 (2011-12-24)

- iter_lines last-line truncation fix
- Force safe_mode for async requests
- Handle safe_mode exceptions more consistently
- Fix iteration on null responses in safe_mode

0.8.6 (2011-12-18)

- Socket timeout fixes.
- Proxy Authorization support.

0.8.5 (2011-12-14)

- Response.iter_lines!

0.8.4 (2011-12-11)

- Prefetch bugfix.
- Added license to installed version.

0.8.3 (2011-11-27)

- Converted auth system to use simpler callable objects.
- New session parameter to API methods.
- Display full URL while logging.

0.8.2 (2011-11-19)

- New Unicode decoding system, based on over-ridable *Response.encoding*.
- Proper URL slash-quote handling.
- Cookies with `[`, `]`, and _ allowed.

0.8.1 (2011-11-15)

- URL Request path fix
- Proxy fix.
- Timeouts fix.

0.8.0 (2011-11-13)

- Keep-alive support!
- Complete removal of Urllib2
- Complete removal of Poster
- Complete removal of CookieJars
- New ConnectionError raising
- Safe_mode for error catching
- prefetch parameter for request methods
- OPTION method
- Async pool size throttling
- File uploads send real names
- Vendored in urllib3

0.7.6 (2011-11-07)

- Digest authentication bugfix (attach query data to path)

0.7.5 (2011-11-04)

- Response.content = None if there was an invalid repsonse.
- Redirection auth handling.

0.7.4 (2011-10-26)

- Session Hooks fix.

0.7.3 (2011-10-23)

- Digest Auth fix.

0.7.2 (2011-10-23)

- PATCH Fix.

0.7.1 (2011-10-23)

- Move away from urllib2 authentication handling.
- Fully Remove AuthManager, AuthObject, &c.
- New tuple-based auth system with handler callbacks.

0.7.0 (2011-10-22)

- Sessions are now the primary interface.
- Deprecated InvalidMethodException.
- PATCH fix.
- New config system (no more global settings).

0.6.6 (2011-10-19)

- Session parameter bugfix (params merging).

0.6.5 (2011-10-18)

- Offline (fast) test suite.
- Session dictionary argument merging.

0.6.4 (2011-10-13)

- Automatic decoding of unicode, based on HTTP Headers.
- New `decode_unicode` setting.
- Removal of `r.read/close` methods.
- New `r.faw` interface for advanced response usage.*
- Automatic expansion of parameterized headers.

0.6.3 (2011-10-13)

- Beautiful `requests.async` module, for making async requests w/ gevent.

0.6.2 (2011-10-09)

- GET/HEAD obeys allow_redirects=False.

0.6.1 (2011-08-20)

- Enhanced status codes experience `\o/`
- Set a maximum number of redirects (`settings.max_redirects`)
- Full Unicode URL support
- Support for protocol-less redirects.
- Allow for arbitrary request types.
- Bugfixes

0.6.0 (2011-08-17)

- New callback hook system
- New persistient sessions object and context manager
- Transparent Dict-cookie handling
- Status code reference object
- Removed Response.cached
- Added Response.request
- All args are kwargs
- Relative redirect support
- HTTPError handling improvements
- Improved https testing
- Bugfixes

0.5.1 (2011-07-23)

- International Domain Name Support!
- Access headers without fetching entire body (`read()`)
- Use lists as dicts for parameters
- Add Forced Basic Authentication
- Forced Basic is default authentication type
- `python-requests.org` default User-Agent header
- CaseInsensitiveDict lower-case caching
- Response.history bugfix

0.5.0 (2011-06-21)

- PATCH Support
- Support for Proxies
- HTTPBin Test Suite
- Redirect Fixes
- settings.verbose stream writing
- Querystrings for all methods
- URLErrors (Connection Refused, Timeout, Invalid URLs) are treated as explicity raised `r.requests.get('hwe://blah'); r.raise_for_status()`

0.4.1 (2011-05-22)

- Improved Redirection Handling
- New 'allow_redirects' param for following non-GET/HEAD Redirects
- Settings module refactoring

0.4.0 (2011-05-15)

- Response.history: list of redirected responses
- Case-Insensitive Header Dictionaries!
- Unicode URLs

0.3.4 (2011-05-14)

- Urllib2 HTTPAuthentication Recursion fix (Basic/Digest)
- Internal Refactor
- Bytes data upload Bugfix

0.3.3 (2011-05-12)

- Request timeouts
- Unicode url-encoded data
- Settings context manager and module

0.3.2 (2011-04-15)

- Automatic Decompression of GZip Encoded Content
- AutoAuth Support for Tupled HTTP Auth

0.3.1 (2011-04-01)

- Cookie Changes
- Response.read()
- Poster fix

0.3.0 (2011-02-25)

- Automatic Authentication API Change
- Smarter Query URL Parameterization
- Allow file uploads and POST data together
- **New Authentication Manager System**
    - Simpler Basic HTTP System
    - Supports all build-in urllib2 Auths
    - Allows for custom Auth Handlers

0.2.4 (2011-02-19)

- Python 2.5 Support
- PyPy-c v1.4 Support
- Auto-Authentication tests
- Improved Request object constructor

0.2.3 (2011-02-15)

- **New HTTPHandling Methods**
    - Response.__nonzero__ (false if bad HTTP Status)
    - Response.ok (True if expected HTTP Status)
    - Response.error (Logged HTTPError if bad HTTP Status)
    - Response.raise_for_status() (Raises stored HTTPError)

0.2.2 (2011-02-14)

- Still handles request in the event of an HTTPError. (Issue #2)
- Eventlet and Gevent Monkeypatch support.
- Cookie Support (Issue #1)

0.2.1 (2011-02-14)

- Added file attribute to POST and PUT requests for multipart-encode file uploads.
- Added Request.url attribute for context and redirects

0.2.0 (2011-02-14)

- Birth!

0.0.1 (2011-02-13)

- Frustration
- Conception

CAPÍTULO 5

Documentación del API

Si buscas información acerca de una función, clase o método en específico está parte de la documentación es para ti.

5.1 Interfaz para Desarrolladores

Esta parte de la documentación cubre todas las interfaces de Requests. En las partes donde Requests depende de librerías externas, documentamos las más importantes aquí mismo, y proveemos enlaces a la documentación canónica.

5.1.1 Interfaz Principal

Se puede acceder a todas las funcionalidades de Requests a través de estos 7 métodos. Todos ellos devuelven una instancia de :class: *Response <Response>*.

`requests.`**`request`**(*method*, *url*, ***kwargs*)

Constructs and sends a *Request*. Returns *Response* object.

Parámetros

- **method** – method for the new *Request* object.
- **url** – URL for the new *Request* object.
- **params** – (optional) Dictionary or bytes to be sent in the query string for the *Request*.
- **data** – (optional) Dictionary, bytes, or file-like object to send in the body of the *Request*.
- **headers** – (optional) Dictionary of HTTP Headers to send with the *Request*.
- **cookies** – (optional) Dict or CookieJar object to send with the *Request*.
- **files** – (optional) Dictionary of 'name': file-like-objects (or {'name': ('filename', fileobj)}) for multipart encoding upload.
- **auth** – (optional) Auth tuple to enable Basic/Digest/Custom HTTP Auth.
- **timeout** – (optional) Float describing the timeout of the request.
- **allow_redirects** – (optional) Boolean. Set to True if POST/PUT/DELETE redirect following is allowed.
- **proxies** – (optional) Dictionary mapping protocol to the URL of the proxy.
- **verify** – (optional) if `True`, the SSL cert will be verified. A CA_BUNDLE path can also be provided.

- **stream** – (optional) if `False`, the response content will be immediately downloaded.
- **cert** – (optional) if String, path to ssl client cert file (.pem). If Tuple, ('cert', 'key') pair.

Usage:

```
>>> import requests
>>> req = requests.request('GET', 'http://httpbin.org/get')
<Response [200]>
```

`requests.`**`head`**(*url*, ***kwargs*)
Sends a HEAD request. Returns *Response* object.

Parámetros

- **url** – URL for the new *Request* object.
- ****kwargs** – Optional arguments that `request` takes.

`requests.`**`get`**(*url*, ***kwargs*)
Sends a GET request. Returns *Response* object.

Parámetros

- **url** – URL for the new *Request* object.
- ****kwargs** – Optional arguments that `request` takes.

`requests.`**`post`**(*url*, *data=None*, ***kwargs*)
Sends a POST request. Returns *Response* object.

Parámetros

- **url** – URL for the new *Request* object.
- **data** – (optional) Dictionary, bytes, or file-like object to send in the body of the *Request*.
- ****kwargs** – Optional arguments that `request` takes.

`requests.`**`put`**(*url*, *data=None*, ***kwargs*)
Sends a PUT request. Returns *Response* object.

Parámetros

- **url** – URL for the new *Request* object.
- **data** – (optional) Dictionary, bytes, or file-like object to send in the body of the *Request*.
- ****kwargs** – Optional arguments that `request` takes.

`requests.`**`patch`**(*url*, *data=None*, ***kwargs*)
Sends a PATCH request. Returns *Response* object.

Parámetros

- **url** – URL for the new *Request* object.
- **data** – (optional) Dictionary, bytes, or file-like object to send in the body of the *Request*.
- ****kwargs** – Optional arguments that `request` takes.

`requests.`**`delete`**(*url*, ***kwargs*)
Sends a DELETE request. Returns *Response* object.

Parámetros

- **url** – URL for the new *Request* object.
- ****kwargs** – Optional arguments that `request` takes.

Clases de bajo nivel

class requests.**Request**(*method=None*, *url=None*, *headers=None*, *files=None*, *data={}*, *params={}*, *auth=None*, *cookies=None*, *hooks=None*)

A user-created *Request* object.

Used to prepare a *PreparedRequest*, which is sent to the server.

Parámetros

- **method** – HTTP method to use.
- **url** – URL to send.
- **headers** – dictionary of headers to send.
- **files** – dictionary of {filename: fileobject} files to multipart upload.
- **data** – the body to attach the request. If a dictionary is provided, form-encoding will take place.
- **params** – dictionary of URL parameters to append to the URL.
- **auth** – Auth handler or (user, pass) tuple.
- **cookies** – dictionary or CookieJar of cookies to attach to this request.
- **hooks** – dictionary of callback hooks, for internal usage.

Usage:

```
>>> import requests
>>> req = requests.Request('GET', 'http://httpbin.org/get')
>>> req.prepare()
<PreparedRequest [GET]>
```

deregister_hook(*event*, *hook*)

Deregister a previously registered hook. Returns True if the hook existed, False if not.

prepare()

Constructs a *PreparedRequest* for transmission and returns it.

register_hook(*event*, *hook*)

Properly register a hook.

class requests.**Response**

The *Response* object, which contains a server's response to an HTTP request.

apparent_encoding

The apparent encoding, provided by the lovely Charade library (Thanks, Ian!).

content

Content of the response, in bytes.

cookies = None

A CookieJar of Cookies the server sent back.

elapsed = None

The amount of time elapsed between sending the request and the arrival of the response (as a timedelta)

encoding = None

Encoding to decode with when accessing r.text.

headers = None
Case-insensitive Dictionary of Response Headers. For example, `headers['content-encoding']` will return the value of a `'Content-Encoding'` response header.

history = None
A list of *Response* objects from the history of the Request. Any redirect responses will end up here. The list is sorted from the oldest to the most recent request.

iter_content (*chunk_size=1*, *decode_unicode=False*)
Iterates over the response data. This avoids reading the content at once into memory for large responses. The chunk size is the number of bytes it should read into memory. This is not necessarily the length of each item returned as decoding can take place.

iter_lines (*chunk_size=512*, *decode_unicode=None*)
Iterates over the response data, one line at a time. This avoids reading the content at once into memory for large responses.

json (***kwargs*)
Returns the json-encoded content of a response, if any.

Parámetros ****kwargs** – Optional arguments that `json.loads` takes.

links
Returns the parsed header links of the response, if any.

raise_for_status ()
Raises stored *HTTPError*, if one occurred.

raw = None
File-like object representation of response (for advanced usage). Requires that ‘‘stream=True‘ on the request.

status_code = None
Integer Code of responded HTTP Status.

text
Content of the response, in unicode.

if Response.encoding is None and chardet module is available, encoding will be guessed.

url = None
Final URL location of Response.

5.1.2 Request Sessions

class `requests.Session`
A Requests session.

Provides cookie persistience, connection-pooling, and configuration.

Basic Usage:

```
>>> import requests
>>> s = requests.Session()
>>> s.get('http://httpbin.org/get')
200
```

auth = None
Default Authentication tuple or object to attach to *Request*.

cert = None
SSL certificate default.

close()
Closes all adapters and as such the session

delete(*url*, ***kwargs*)
Sends a DELETE request. Returns *Response* object.

Parámetros

- **url** – URL for the new *Request* object.
- ****kwargs** – Optional arguments that `request` takes.

get(*url*, ***kwargs*)
Sends a GET request. Returns *Response* object.

Parámetros

- **url** – URL for the new *Request* object.
- ****kwargs** – Optional arguments that `request` takes.

get_adapter(*url*)
Returns the appropriate connnection adapter for the given URL.

head(*url*, ***kwargs*)
Sends a HEAD request. Returns *Response* object.

Parámetros

- **url** – URL for the new *Request* object.
- ****kwargs** – Optional arguments that `request` takes.

headers = None
A case-insensitive dictionary of headers to be sent on each *Request* sent from this *Session*.

hooks = None
Event-handling hooks.

max_redirects = None
Maximum number of redirects to follow.

mount(*prefix*, *adapter*)
Registers a connection adapter to a prefix.

options(*url*, ***kwargs*)
Sends a OPTIONS request. Returns *Response* object.

Parámetros

- **url** – URL for the new *Request* object.
- ****kwargs** – Optional arguments that `request` takes.

params = None
Dictionary of querystring data to attach to each *Request*. The dictionary values may be lists for representing multivalued query parameters.

patch(*url*, *data=None*, ***kwargs*)
Sends a PATCH request. Returns *Response* object.

Parámetros

- **url** – URL for the new *Request* object.
- **data** – (optional) Dictionary, bytes, or file-like object to send in the body of the *Request*.

- ****kwargs** – Optional arguments that `request` takes.

post(*url*, *data=None*, ***kwargs*)
Sends a POST request. Returns *Response* object.

Parámetros

- **url** – URL for the new *Request* object.
- **data** – (optional) Dictionary, bytes, or file-like object to send in the body of the *Request*.
- ****kwargs** – Optional arguments that `request` takes.

proxies = None
Dictionary mapping protocol to the URL of the proxy (e.g. {'http': 'foo.bar:3128'}) to be used on each *Request*.

put(*url*, *data=None*, ***kwargs*)
Sends a PUT request. Returns *Response* object.

Parámetros

- **url** – URL for the new *Request* object.
- **data** – (optional) Dictionary, bytes, or file-like object to send in the body of the *Request*.
- ****kwargs** – Optional arguments that `request` takes.

resolve_redirects(*resp*, *req*, *stream=False*, *timeout=None*, *verify=True*, *cert=None*, *proxies=None*)
Receives a Response. Returns a generator of Responses.

send(*request*, ***kwargs*)
Send a given PreparedRequest.

stream = None
Stream response content default.

trust_env = None
Should we trust the environment?

verify = None
SSL Verification default.

Excepciones

exception `requests.`**RequestException**
There was an ambiguous exception that occurred while handling your request.

exception `requests.`**ConnectionError**
A Connection error occurred.

exception `requests.`**HTTPError**(**args*, ***kwargs*)
An HTTP error occurred.

exception `requests.`**URLRequired**
A valid URL is required to make a request.

exception `requests.`**TooManyRedirects**
Too many redirects.

Búsqueda de Códigos de Estado

requests.**codes**()
Dictionary lookup object.

```
>>> requests.codes['temporary_redirect']
307

>>> requests.codes.teapot
418

>>> requests.codes['\o/']
200
```

Cookies

Codificaciones

Clases

class requests.**Response**
The *Response* object, which contains a server's response to an HTTP request.

apparent_encoding
The apparent encoding, provided by the lovely Charade library (Thanks, Ian!).

content
Content of the response, in bytes.

cookies = None
A CookieJar of Cookies the server sent back.

elapsed = None
The amount of time elapsed between sending the request and the arrival of the response (as a timedelta)

encoding = None
Encoding to decode with when accessing r.text.

headers = None
Case-insensitive Dictionary of Response Headers. For example, `headers['content-encoding']` will return the value of a `'Content-Encoding'` response header.

history = None
A list of *Response* objects from the history of the Request. Any redirect responses will end up here. The list is sorted from the oldest to the most recent request.

iter_content(*chunk_size=1*, *decode_unicode=False*)
Iterates over the response data. This avoids reading the content at once into memory for large responses. The chunk size is the number of bytes it should read into memory. This is not necessarily the length of each item returned as decoding can take place.

iter_lines(*chunk_size=512*, *decode_unicode=None*)
Iterates over the response data, one line at a time. This avoids reading the content at once into memory for large responses.

json(***kwargs*)
Returns the json-encoded content of a response, if any.

Parámetros ****kwargs** – Optional arguments that `json.loads` takes.

links
Returns the parsed header links of the response, if any.

raise_for_status()
Raises stored *HTTPError*, if one occurred.

raw = None
File-like object representation of response (for advanced usage). Requires that ‘‘stream=True‘ on the request.

status_code = None
Integer Code of responded HTTP Status.

text
Content of the response, in unicode.

if Response.encoding is None and chardet module is available, encoding will be guessed.

url = None
Final URL location of Response.

class requests.**Request**(*method=None*, *url=None*, *headers=None*, *files=None*, *data={}*, *params={}*, *auth=None*, *cookies=None*, *hooks=None*)
A user-created *Request* object.

Used to prepare a *PreparedRequest*, which is sent to the server.

Parámetros

- **method** – HTTP method to use.
- **url** – URL to send.
- **headers** – dictionary of headers to send.
- **files** – dictionary of {filename: fileobject} files to multipart upload.
- **data** – the body to attach the request. If a dictionary is provided, form-encoding will take place.
- **params** – dictionary of URL parameters to append to the URL.
- **auth** – Auth handler or (user, pass) tuple.
- **cookies** – dictionary or CookieJar of cookies to attach to this request.
- **hooks** – dictionary of callback hooks, for internal usage.

Usage:

```
>>> import requests
>>> req = requests.Request('GET', 'http://httpbin.org/get')
>>> req.prepare()
<PreparedRequest [GET]>
```

deregister_hook(*event*, *hook*)
Deregister a previously registered hook. Returns True if the hook existed, False if not.

prepare()
Constructs a *PreparedRequest* for transmission and returns it.

register_hook(*event*, *hook*)
Properly register a hook.

class requests.**PreparedRequest**
The fully mutable *PreparedRequest* object, containing the exact bytes that will be sent to the server.

Generated from either a *Request* object or manually.

Usage:

```
>>> import requests
>>> req = requests.Request('GET', 'http://httpbin.org/get')
>>> r = req.prepare()
<PreparedRequest [GET]>

>>> s = requests.Session()
>>> s.send(r)
<Response [200]>
```

body = None
request body to send to the server.

deregister_hook(*event*, *hook*)
Deregister a previously registered hook. Returns True if the hook existed, False if not.

headers = None
dictionary of HTTP headers.

hooks = None
dictionary of callback hooks, for internal usage.

method = None
HTTP verb to send to the server.

path_url
Build the path URL to use.

prepare_auth(*auth*)
Prepares the given HTTP auth data.

prepare_body(*data*, *files*)
Prepares the given HTTP body data.

prepare_cookies(*cookies*)
Prepares the given HTTP cookie data.

prepare_headers(*headers*)
Prepares the given HTTP headers.

prepare_hooks(*hooks*)
Prepares the given hooks.

prepare_method(*method*)
Prepares the given HTTP method.

prepare_url(*url*, *params*)
Prepares the given HTTP URL.

register_hook(*event*, *hook*)
Properly register a hook.

url = None
HTTP URL to send the request to.

class requests.**Session**
A Requests session.

Provides cookie persistience, connection-pooling, and configuration.

Basic Usage:

```
>>> import requests
>>> s = requests.Session()
>>> s.get('http://httpbin.org/get')
200
```

auth = None
Default Authentication tuple or object to attach to *Request*.

cert = None
SSL certificate default.

close()
Closes all adapters and as such the session

delete(*url*, ***kwargs*)
Sends a DELETE request. Returns *Response* object.

Parámetros

- **url** – URL for the new *Request* object.
- ****kwargs** – Optional arguments that `request` takes.

get(*url*, ***kwargs*)
Sends a GET request. Returns *Response* object.

Parámetros

- **url** – URL for the new *Request* object.
- ****kwargs** – Optional arguments that `request` takes.

get_adapter(*url*)
Returns the appropriate connnection adapter for the given URL.

head(*url*, ***kwargs*)
Sends a HEAD request. Returns *Response* object.

Parámetros

- **url** – URL for the new *Request* object.
- ****kwargs** – Optional arguments that `request` takes.

headers = None
A case-insensitive dictionary of headers to be sent on each *Request* sent from this *Session*.

hooks = None
Event-handling hooks.

max_redirects = None
Maximum number of redirects to follow.

mount(*prefix*, *adapter*)
Registers a connection adapter to a prefix.

options(*url*, ***kwargs*)
Sends a OPTIONS request. Returns *Response* object.

Parámetros

- **url** – URL for the new *Request* object.

- ****kwargs** – Optional arguments that `request` takes.

params = None
Dictionary of querystring data to attach to each *Request*. The dictionary values may be lists for representing multivalued query parameters.

patch(*url*, *data=None*, ***kwargs*)
Sends a PATCH request. Returns *Response* object.

Parámetros

- **url** – URL for the new *Request* object.
- **data** – (optional) Dictionary, bytes, or file-like object to send in the body of the *Request*.
- ****kwargs** – Optional arguments that `request` takes.

post(*url*, *data=None*, ***kwargs*)
Sends a POST request. Returns *Response* object.

Parámetros

- **url** – URL for the new *Request* object.
- **data** – (optional) Dictionary, bytes, or file-like object to send in the body of the *Request*.
- ****kwargs** – Optional arguments that `request` takes.

proxies = None
Dictionary mapping protocol to the URL of the proxy (e.g. {'http': 'foo.bar:3128'}) to be used on each *Request*.

put(*url*, *data=None*, ***kwargs*)
Sends a PUT request. Returns *Response* object.

Parámetros

- **url** – URL for the new *Request* object.
- **data** – (optional) Dictionary, bytes, or file-like object to send in the body of the *Request*.
- ****kwargs** – Optional arguments that `request` takes.

resolve_redirects(*resp*, *req*, *stream=False*, *timeout=None*, *verify=True*, *cert=None*, *proxies=None*)
Receives a Response. Returns a generator of Responses.

send(*request*, ***kwargs*)
Send a given PreparedRequest.

stream = None
Stream response content default.

trust_env = None
Should we trust the environment?

verify = None
SSL Verification default.

5.1.3 Migrando a 1.x

Esta sección expone las principales diferencias entre las versiones 0.x y 1.x, y pretende facilitar las molestias a la hora de actualizar.

Cambios en el API

- `Response.json` ahora es un invocable y no una propiedad de una respuesta.

```
import requests
r = requests.get('https://github.com/timeline.json')
r.json()    # This *call* raises an exception if JSON decoding fails
```

- El API de `Session` ha cambiado. Los objetos `Session` ya no reciben

parámetros. `Session` se escribe ahora con mayúscula, pero aún puede ser instanciado como `session` en minúscula, por razones de compatibilidad hacia atrás.

```
s = requests.Session()    # anteriormente session tomaba parámetros
s.auth = auth
s.headers.update(headers)
r = s.get('http://httpbin.org/headers')
```

- Todos los *hooks* de petición han sido eliminados, con excepción de

'response'.

- Los ayudantes de autenticación han sido separados en módulos independientes.

Ver requests-oauthlib y requests-kerberos.

- El parámetro para realizar peticiones de *streaming* ha cambiado de

`prefetch` a `stream` y la lógica ha sido invertida. Adicionalmente, `stream` ahora es obligatorio para leer respuestas en crudo.

```
# en 0.x, pasar prefetch=False haría lo mismo que
r = requests.get('https://github.com/timeline.json', stream=True)
r.raw.read(10)
```

- El parámetro `config` en los métodos de peticiones ha sido eliminado.

Algunas de estas opciones ahora se configuran en una sesión `Session`, tales como *keep-alive* y el número máximo de redirecciones. La opción de verbosidad debe ser manipulada configurando las opciones de registro (*logging*).

```
import requests
import logging

# estas dos líneas habilitan el debugg a nivel httplib (requests->urllib3->httplib)
# verás el REQUEST, incluído HEADERS y DATA, así como RESPONSE sin HEADERS pero sin DATA.
# la única cosa que falta será el response.body, que no es registrado.
import httplib
httplib.HTTPConnection.debuglevel = 1

logging.basicConfig() # necesitas inicializar logging, de otra manera no verás nada desde reques
logging.getLogger().setLevel(logging.DEBUG)
requests_log = logging.getLogger("requests.packages.urllib3")
requests_log.setLevel(logging.DEBUG)
requests_log.propagate = True

requests.get('http://httpbin.org/headers')
```

Licenciamiento

Una diferencia clave que no tiene nada que ver con el API es el cambio en la licencia, de ISC a Apache 2.0. La licencia Apache 2.0 asegura que las contribuciones a Requests también estén cubiertas por la licencia Apache 2.

5.1.4 Migrando a 2.x

Comparado con el release 1.0, hay relativamente pocos cambios compatibles con versiones anteriores, pero aún hay algunos pocos temas de los cuales estar pendientes en esta liberación.

Para más detalles en los cambios de este release, incluidos nuevas API, enlaces relevantes a los issues en GitHub y algunas de las correcciones de bug, lee el blog de Cory al respecto

Cambios en la API

- Hay un par de cambios en cómo Requests maneja las excepciones.

`RequestException` es ahora una subclase de `IOError` en vez de `RuntimeError` debido a que categoriza con mayor precisión este tipo de error. Además, una secuencia de escape URL inválida ahora arroja una subclase de `RequestException` en vez de `ValueError`.

```
requests.get('http://%zz/')    # arroja un requests.exceptions.InvalidURL
```

Por último, las excepciones `httplib.IncompleteRead` causadas por una incorrecta codificación por trozos (*chunked*) ahora arrojará un `ChunkedEncodingError`.

- La API de proxy ha cambiado ligeramente: ahora se requiere el esquema

para la URL del proxy.

```
proxies = {
  "http": "10.10.1.10:3128",    # ahora usa http://10.10.1.10:3128
}

# En Requests 1.x, esto era legal, en Requests 2.x,
#  esto arroja requests.exceptions.MissingSchema
requests.get("http://example.org", proxies=proxies)
```

Cambios en el Comportamiento

- Las llaves en el diccionario `headers` ahora son cadenas nativas strings

para todas las versiones de Python , por ejemplo, `bytestring` en Python 2 y unicode on Python 3. Si las llaves no son cadenas nativas (unicode en Python2 o bytestring en Python 3) serán convertidas al tipo de cadena nativa asumiendo codificación UTF-8.

- Los Timeouts se comportan ligeramente diferente. En peticiones de streaming, sólo aplicann al intento de conexión. En peticiones regulares, el timeout es aplicado al proceso de conexión y a la descarga completa.

```
tarball_url = 'https://github.com/kennethreitz/requests/tarball/master'

# Un segundo de timeout para el intento de conexión
# Tiempo ilimitado para descargar el tarball
r = requests.get(tarball_url, stream=True, timeout=1)

# Un segungo de para el intento de conexión
```

```
    # Otro segundo completo para descargar el tarball
    r = requests.get(tarball_url, timeout=1)
```

CAPÍTULO 6

Guía del contribuidor

Si quieres contribuir con el proyecto, esta parte de la documentación es para ti.

6.1 Filosofía del desarrollo

Requests es una librería abierta pero es dogmática, creada por un desarrollador abierto pero con opiniones propias.

6.1.1 Dictador Benevolente (*Benevolent Dictador*)

Kenneth Reitz es el Dictador Benevolente de por vida, BDFL. Él tiene la última palabra en cualquier decisión relacionada con Requests.

6.1.2 Valores

- Simplicidad siempre es mejor que funcionalidad.
- Escucha a todos, luego has caso omiso de ello.
- El API es todo lo que importa. Todo lo demás es secundario.
- Ajústate al 90 % de los casos de uso. Ignora a los pocos que llevan la

contraria.

6.1.3 Versionado semántico

Por muchos años, la comunidad open source ha estado plagada con algún tipo de distonía en lo que se refiere a la numeración de versiones, las cuales no tienen ningún sentido práctico.

Requests usa Versionado Semántico (*Semantic Versioning*). Está especificación tiene como objetivo poner fín a este sinsentido por medio de un conjunto pequeño de reglas prácticas que tu y tu colegas pueden usar en sus proyectos.

6.1.4 ¿Biblioteca Estándar?

Requests no tiene ningún plan *activo* para ser incluído en la librería estándar. Esta decisión ha sido ampliamente discutida con Guido, así como un buen número de desarrolladores del núcleo de Python.

Esencialmente, porque la biblioteca estándar es donde van a morir las bibliotecas. Para un modulo es apropiado ser incluido cuando el desarrollo activo ya no es necesario.

Requests acaba de llegar a su versión 1.0.0. Esto es un gran avance, y marca un paso en la dirección correcta.

6.1.5 Paquetes en Distros de Linux

Se han hecho algunas distribuciones para algunos repositorios de Linux incluyendo: Ubuntu, Debian, RHEL, y Arch.

Estas distribuciones por lo general corresponden a *forks* que divergen del principal, o en otros casos no siempre están actualizados con el último código y con las correcciones de errores. PyPI (y sus espejos) y GitHub son la fuente oficial de distribución; otras alternativas no están soportadas en este proyecto.

6.2 Cómo ayudar

Requests está bajo desarrollo activo y tus contribuciones son más que bienvenidas!

1. Revisa los reportes (*issues*) abiertos, o abre uno nuevo para empezar una discusión acerca de un error o bug. Existe una etiqueta llamada *Contributor Friendly* (amigable al contribuyente), para reportes que son ideales para personas que no están familiarizadas con el código fuente aún.
2. Haz un *fork* al repositorio en GitHub y hazle cambios a tu propio *branch*.
3. Escribe una prueba que demuestre que se ha solucionado el problema.
4. Envía un *pull request* y molesta al mantenedor hasta que este se consolide y sea publicado. :) Asegúrate de agregarte a lista de autores: AUTHORS.

6.2.1 Feature Freeze

A partir de la versión 1.0.0, Requests ha entrado en un estado de *feature freeze* (congelamiento de funcionalidades). Las peticiones para nuevas funcionalidades y los *pull requests* implementándolas no serán aceptadas.

6.2.2 Dependencias en el desarrollo

Necesitarás instalar py.test con el fín ejecutar el conjunto de pruebas de Requests:

```
$ pip install -r requirements.txt
$ invoke test
py.test
platform darwin -- Python 2.7.3 -- pytest-2.3.4
collected 25 items

test_requests.py .........................
25 passed in 3.50 seconds
```

6.2.3 Entornos de ejecución

Actualmente, Requests soporta las siguientes versión de Python:

- Python 2.6
- Python 2.7

- Python 3.1
- Python 3.2
- Python 3.3
- PyPy 1.9

El soporte para Python 3.1 y 3.2 quitarse en cualquier momento.

Nunca se soportará oficialmente Google App Engine. Los *Pull requests* para mejorar la compatibilidad con este serán aceptados, siempre y cuando no introduzcan complicaciones en el código.

6.2.4 ¿Estás loco?

- Soporte para SPDY sería genial. Pero que no usen extensiones en C.

6.2.5 Reempaquetado

Si estás reempaquetando Requests, nota por favor que también debes redistribuir el archivo `cacerts.pem` con el fin de obtener la funcionalidad SSL correcta.

6.3 Authors

Requests es escrito y mantenido por Kenneth Reitz y varios contribuidores:

6.3.1 Líder de Desarrollo

- Kenneth Reitz <me@kennethreitz.com>

6.3.2 Urllib3

- Andrey Petrov <andrey.petrov@shazow.net>

6.3.3 Parches y Sugerencias

- Various Pocoo Members
- Chris Adams
- Flavio Percoco Premoli
- Dj Gilcrease
- Justin Murphy
- Rob Madole
- Aram Dulyan
- Johannes Gorset
- (Megane Murayama)
- James Rowe

- Daniel Schauenberg
- Zbigniew Siciarz
- Daniele Tricoli 'Eriol'
- Richard Boulton
- Miguel Olivares <miguel@moliware.com>
- Alberto Paro
- Jérémy Bethmont
- (Xu Pan)
- Tamás Gulácsi
- Rubén Abad
- Peter Manser
- Jeremy Selier
- Jens Diemer
- Alex <@alopatin>
- Tom Hogans <tomhsx@gmail.com>
- Armin Ronacher
- Shrikant Sharat Kandula
- Mikko Ohtamaa
- Den Shabalin
- Daniel Miller <danielm@vs-networks.com>
- Alejandro Giacometti
- Rick Mak
- Johan Bergström
- Josselin Jacquard
- Travis N. Vaught
- Fredrik Möllerstrand
- Daniel Hengeveld
- Dan Head
- Bruno Renié
- David Fischer
- Joseph McCullough
- Juergen Brendel
- Juan Riaza
- Ryan Kelly
- Rolando Espinoza La fuente
- Robert Gieseke

- Idan Gazit
- Ed Summers
- Chris Van Horne
- Christopher Davis
- Ori Livneh
- Jason Emerick
- Bryan Helmig
- Jonas Obrist
- Lucian Ursu
- Tom Moertel
- Frank Kumro Jr
- Chase Sterling
- Marty Alchin
- takluyver
- Ben Toews (mastahyeti)
- David Kemp
- Brendon Crawford
- Denis (Telofy)
- Cory Benfield (Lukasa)
- Matt Giuca
- Adam Tauber
- Honza Javorek
- Brendan Maguire <maguire.brendan@gmail.com>
- Chris Dary
- Danver Braganza <danverbraganza@gmail.com>
- Max Countryman
- Nick Chadwick
- Jonathan Drosdeck
- Jiri Machalek
- Steve Pulec
- Michael Kelly
- Michael Newman <newmaniese@gmail.com>
- Jonty Wareing <jonty@jonty.co.uk>
- Shivaram Lingamneni
- Miguel Turner
- Rohan Jain (crodjer)

- Justin Barber <barber.justin@gmail.com>
- Roman Haritonov <@reclosedev>
- Josh Imhoff <joshimhoff13@gmail.com>
- Arup Malakar <amalakar@gmail.com>
- Danilo Bargen (dbrgn)
- Torsten Landschoff
- Michael Holler (apotheos)
- Timnit Gebru
- Sarah Gonzalez
- Victoria Mo
- Leila Muhtasib
- Matthias Rahlf <matthias@webding.de>
- Jakub Roztocil <jakub@roztocil.name>
- Ian Cordasco <graffatcolmingov@gmail.com> @sigmavirus24
- Rhys Elsmore
- André Graf (dergraf)
- Stephen Zhuang (everbird)
- Martijn Pieters
- Jonatan Heyman
- David Bonner <dbonner@gmail.com> @rascalking
- Vinod Chandru
- Johnny Goodnow <j.goodnow29@gmail.com>
- Denis Ryzhkov <denisr@denisr.com>
- Wilfred Hughes <me@wilfred.me.uk> @dontYetKnow
- Dmitry Medvinsky <me@dmedvinsky.name>
- Bryce Boe <bbzbryce@gmail.com> @bboe
- Colin Dunklau <colin.dunklau@gmail.com> @cdunklau
- Bob Carroll <bob.carroll@alum.rit.edu> @rcarz
- Hugo Osvaldo Barrera <hugo@osvaldobarrera.com.ar> @hobarrera
- Łukasz Langa <lukasz@langa.pl> @llanga
- Dave Shawley <daveshawley@gmail.com>
- James Clarke (jam)
- Kevin Burke <kev@inburke.com>
- Flavio Curella
- David Pursehouse <david.pursehouse@gmail.com> @dpursehouse
- Jon Parise

- Alexander Karpinsky @homm86
- Marc Schlaich @schlamar
- Park Ilsu <daftonshady@gmail.com> @daftshady
- Matt Spitz @mattspitz
- Vikram Oberoi @voberoi
- Can Ibanoglu <can.ibanoglu@gmail.com> @canibanoglu
- Thomas Weißschuh <thomas@t-8ch.de> @t-8ch
- Jayson Vantuyl <jayson@aggressive.ly> @kagato
- Pengfei.X <pengphy@gmail.com>
- Kamil Madac <kamil.madac@gmail.com>
- Michael Becker <mike@beckerfuffle.com> @beckerfuffle
- Erik Wickstrom <erik@erikwickstrom.com> @erikwickstrom
- @podshumok

r

R

S

T

U

V