jkbrzt / **httpie**

Watch 796 Star 25,134 Fork 1,572

Code Issues 65 Pull requests 11 Pulse Graphs

CLI HTTP client, user-friendly curl replacement with intuitive UI, JSON support, syntax highlighting, wget-like downloads, extensions, etc. http://httpie.org

853 commits **1** branch **27** releases **54** contributors

Branch: master New pull request Find file Clone or download

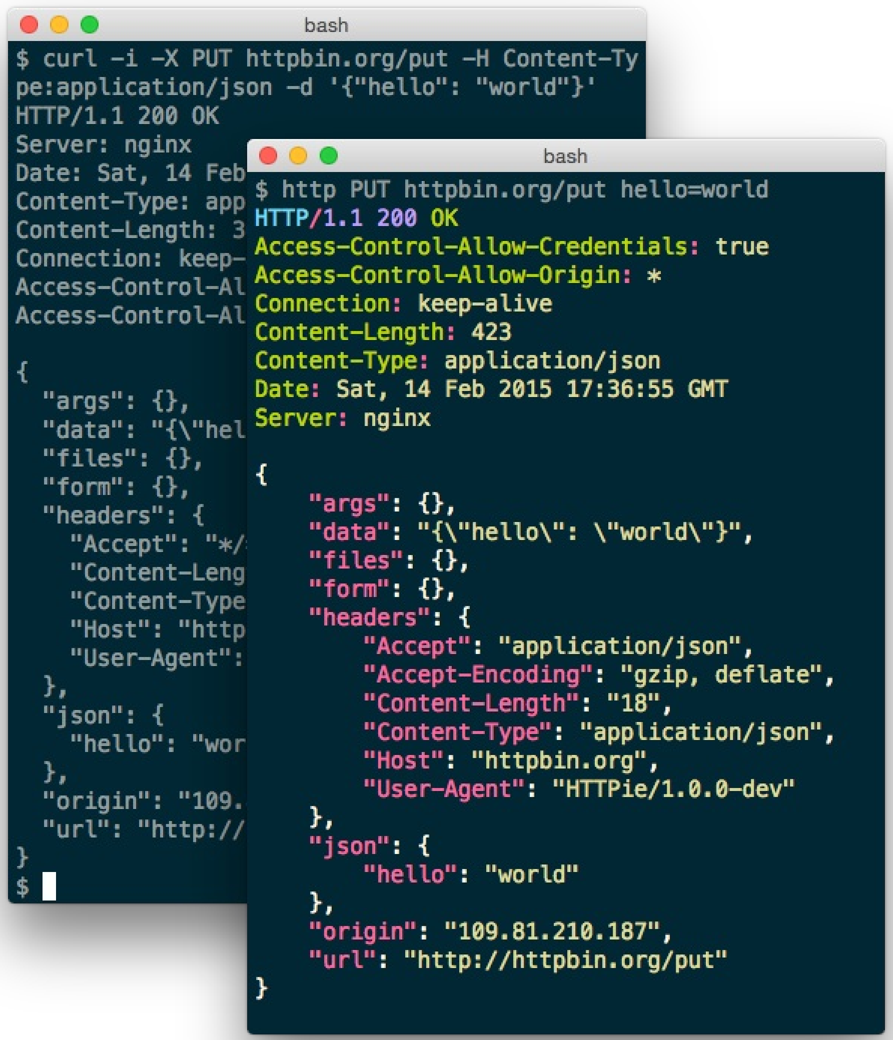
jkbrzt committed on GitHub Update CHANGELOG.rst Latest commit `5efc901` 9 hours ago

extras	Fixed Makefile	21 hours ago
httpie	v0.9.6	21 hours ago
tests	codestyle fixes	26 days ago
.coveragerc	Cleanup	6 months ago
.editorconfig	Added .editorconfig.	a year ago
.gitignore	Added JSON detection when ``--json, -j`` is set	6 months ago
.travis.yml	useful info	19 days ago
AUTHORS.rst	Fixed spelling mistage \`GitHib\` to \`GitHub\`	4 days ago
CHANGELOG.rst	Update CHANGELOG.rst	9 hours ago
CONTRIBUTING.rst	Makefile improvements	8 months ago
LICENSE	Update license year range to 2016	7 months ago
MANIFEST.in	Include AUTHORS.rst in dist; metadata cleanup	2 years ago
Makefile	Fixed Makefile	21 hours ago
README.rst	Updated README.rst to add Arch Linux install docs.	16 hours ago
appveyor.yml	CI	6 months ago
httpie.png	Default --style to "monokai"	2 years ago
pytest.ini	Improved failed test output	6 months ago
requirements-dev.txt	Merge pull request #300 from msabramo/print_info_about_request_on_error	a year ago
setup.cfg	Fixed Makefile, added setup.cfg.	2 years ago
setup.py	Upgrade Pygments version	21 hours ago
tox.ini	useful info	19 days ago

README.rst

HTTPie: a CLI, cURL-like tool for humans

HTTPie (pronounced *aitch-tee-tee-pie*) is a **command line HTTP client**. Its goal is to make CLI interaction with web services as **human-friendly** as possible. It provides a simple `http` command that allows for sending arbitrary HTTP requests using a simple and natural syntax, and displays colorized output. HTTPie can be used for **testing, debugging**, and generally **interacting** with HTTP servers.



```
$ curl -i -X PUT httpbin.org/put -H Content-Ty
pe:application/json -d '{"hello": "world"}'
HTTP/1.1 200 OK
Server: nginx
Date: Sat, 14 Feb
Content-Type: app
Content-Length: 3
Connection: keep-
Access-Control-Al
Access-Control-Al

{
  "args": {},
  "data": "{\"hel
  "files": {},
  "form": {},
  "headers": {
    "Accept": "*/
    "Content-Leng
    "Content-Type
    "Host": "http
    "User-Agent":
  },
  "json": {
    "hello": "wor
  },
  "origin": "109.
  "url": "http://
}
$
```

```
$ http PUT httpbin.org/put hello=world
HTTP/1.1 200 OK
Access-Control-Allow-Credentials: true
Access-Control-Allow-Origin: *
Connection: keep-alive
Content-Length: 423
Content-Type: application/json
Date: Sat, 14 Feb 2015 17:36:55 GMT
Server: nginx

{
    "args": {},
    "data": "{\"hello\": \"world\"}",
    "files": {},
    "form": {},
    "headers": {
        "Accept": "application/json",
        "Accept-Encoding": "gzip, deflate",
        "Content-Length": "18",
        "Content-Type": "application/json",
        "Host": "httpbin.org",
        "User-Agent": "HTTPie/1.0.0-dev"
    },
    "json": {
        "hello": "world"
    },
    "origin": "109.81.210.187",
    "url": "http://httpbin.org/put"
}
```

HTTPie is written in Python, and under the hood it uses the excellent Requests and Pygments libraries.

---

latest stable version v0.9.9 unix build unknown windows build passing coverage 94% chat on gitter

---

- Main features
- Installation
- Usage
- HTTP method
- Request URL
- Request items
- JSON
- Forms
- HTTP headers
- Authentication
- HTTP redirects
- Proxies
- HTTPS
- Output options
- Redirected Input
- Terminal output
- Redirected output
- Download mode

- Streamed responses
- Sessions
- Config
- Scripting
- Interface design
- Support
- Authors
- Logo
- Contribute
- Change log
- Licence
- Related projects

Main features

- Expressive and intuitive syntax
- Formatted and colorized terminal output
- Built-in JSON support
- Forms and file uploads
- HTTPS, proxies, and authentication
- Arbitrary request data
- Custom headers
- Persistent sessions
- Wget-like downloads
- Python 2.6, 2.7 and 3.x support
- Linux, Mac OS X and Windows support
- Plugins
- Documentation
- Test coverage

Installation

On **Mac OS X**, HTTPie can be installed via Homebrew (recommended):

```
$ brew install httpie
```

A MacPorts *port* is also available:

```
$ port install httpie
```

Most **Linux** distributions provide a package that can be installed using the system package manager, e.g.:

```
# Debian-based distributions such as Ubuntu:
$ apt-get install httpie

# RPM-based distributions:
$ yum install httpie

# Arch Linux
$ pacman -S httpie
```

A **universal installation method** (that works on **Windows**, Mac OS X, Linux, …, and provides the latest version) is to use pip:

```
# Make sure we have an up-to-date version of pip and setuptools:
$ pip install --upgrade pip setuptools
$ pip install --upgrade httpie
```

(If `pip` installation fails for some reason, you can try `easy_install httpie` as a fallback.)

Development version

The **latest development version** can be installed directly from GitHub:

```
# Mac OS X via Homebrew
$ brew install httpie --HEAD

# Universal
$ pip install --upgrade https://github.com/jkbrzt/httpie/archive/master.tar.gz
```

Python version

Although Python 2.6 and 2.7 are supported as well, it is recommended to install HTTPie against the latest Python 3.x whenever possible. That will ensure that some of the newer HTTP features, such as SNI (Server Name Indication), work out of the box. Python 3 is the default for Homebrew installations starting with version 0.9.4. To see which version HTTPie uses, run `http --debug`.

Usage

Hello World:

```
$ http httpie.org
```

Synopsis:

```
$ http [flags] [METHOD] URL [ITEM [ITEM]]
```

See also `http --help`.

Examples

Custom HTTP method, HTTP headers and JSON data:

```
$ http PUT example.org X-API-Token:123 name=John
```

Submitting forms:

```
$ http -f POST example.org hello=World
```

See the request that is being sent using one of the output options:

```
$ http -v example.org
```

Use Github API to post a comment on an issue with authentication:

```
$ http -a USERNAME POST https://api.github.com/repos/jkbrzt/httpie/issues/83/comments body='HTTPie is aw
```

Upload a file using redirected input:

```
$ http example.org < file.json
```

Download a file and save it via redirected output:

```
$ http example.org/file > file
```

Download a file `wget` style:

```
$ http --download example.org/file
```

Use named sessions to make certain aspects or the communication persistent between requests to the same host:

```
$ http --session=logged-in -a username:password httpbin.org/get API-Key:123

$ http --session=logged-in httpbin.org/headers
```

Set a custom `Host` header to work around missing DNS records:

```
$ http localhost:8000 Host:example.com
```

What follows is a detailed documentation. It covers the command syntax, advanced usage, and also features additional examples.

HTTP method

The name of the HTTP method comes right before the URL argument:

```
$ http DELETE example.org/todos/7
```

Which looks similar to the actual `Request-Line` that is sent:

```
DELETE /todos/7 HTTP/1.1
```

When the `METHOD` argument is **omitted** from the command, HTTPie defaults to either `GET` (with no request data) or `POST` (with request data).

Request URL

The only information HTTPie needs to perform a request is a URL. The default scheme is, somewhat unsurprisingly, `http://`, and can be omitted from the argument – `http example.org` works just fine.

Additionally, curl-like shorthand for localhost is supported. This means that, for example `:3000` would expand to `http://localhost:3000` If the port is omitted, then port 80 is assumed.

```
$ http :/foo
```

```
GET /foo HTTP/1.1
Host: localhost
```

```
$ http :3000/bar
```

```
GET /bar HTTP/1.1
Host: localhost:3000
```

```
$ http :
```

```
GET / HTTP/1.1
Host: localhost
```

If you find yourself manually constructing URLs with **querystring parameters** on the terminal, you may appreciate the `param==value` syntax for appending URL parameters. With that, you don't have to worry about escaping the `&` separators for you shell. Also, special characters in parameter values, will also automatically escaped (HTTPie otherwise expects the URL to be already escaped). To search for `HTTPie logo` on Google Images you could use this command:

```
$ http www.google.com search=='HTTPie logo' tbm==isch
```

```
GET /?search=HTTPie+logo&tbm=isch HTTP/1.1
```

You can use the `--default-scheme <URL_SCHEME>` option to create shortcuts for other protocols than HTTP:

```
$ alias https='http --default-scheme=https'
```

Request items

There are a few different *request item* types that provide a convenient mechanism for specifying HTTP headers, simple JSON and form data, files, and URL parameters.

They are key/value pairs specified after the URL. All have in common that they become part of the actual request that is sent and that their type is distinguished only by the separator used: `:`, `=`, `:=`, `==`, `@`, `=@`, and `:=@`. The ones with an `@` expect a file path as value.

Item Type	Description
HTTP Headers `Name:Value`	Arbitrary HTTP header, e.g. `X-API-Token:123`.
URL parameters `name==value`	Appends the given name/value pair as a query string parameter to the URL. The `==` separator is used.
Data Fields `field=value`, `field=@file.txt`	Request data fields to be serialized as a JSON object (default), or to be form-encoded (`--form, -f`).
Raw JSON fields `field:=json`, `field:=@file.json`	Useful when sending JSON and one or more fields need to be a `Boolean`, `Number`, nested `Object`, or an `Array`, e.g., `meals:='["ham","spam"]'` or `pies:=[1,2,3]` (note the quotes).
Form File Fields `field@/dir/file`	Only available with `--form, -f`. For example `screenshot@~/Pictures/img.png`. The presence of a file field results in a `multipart/form-data` request.

You can use `\` to escape characters that shouldn't be used as separators (or parts thereof). For instance, `foo\==bar` will become a data key/value pair (`foo=` and `bar`) instead of a URL parameter.

Often it is necessary to quote the values, e.g. `foo='bar baz'`.

If any of the field names or headers starts with a minus (e.g., `-fieldname`), you need to place all such items after the special token `--` to prevent confusion with `--arguments`:

```
$ http httpbin.org/post  --  -name-starting-with-dash=foo --Weird-Header:bar
```

```
POST /post HTTP/1.1
--Weird-Header: bar

{
    "-name-starting-with-dash": "value"
}
```

Note that data fields aren't the only way to specify request data: Redirected input allows for passing arbitrary data to be sent with the request.

JSON

JSON is the *lingua franca* of modern web services and it is also the **implicit content type** HTTPie by default uses:

If your command includes some data items, they are serialized as a JSON object by default. HTTPie also automatically sets the following headers, both of which can be overwritten:

`Content-Type`	`application/json`
`Accept`	`application/json, */*`

You can use `--json, -j` to explicitly set `Accept` to `application/json` regardless of whether you are sending data (it's a shortcut for setting the header via the usual header notation – `http url Accept:application/json, */*`). Additionally, HTTPie will try to detect JSON responses even when the `Content-Type` is incorrectly `text/plain` or unknown.

Simple example:

```
$ http PUT example.org name=John email=john@example.org
```

```
PUT / HTTP/1.1
Accept: application/json, */*
Accept-Encoding: gzip, deflate
Content-Type: application/json
Host: example.org

{
    "name": "John",
    "email": "john@example.org"
}
```

Non-string fields use the `:=` separator, which allows you to embed raw JSON into the resulting object. Text and raw JSON files can also be embedded into fields using `=@` and `:=@` :

```
$ http PUT api.example.com/person/1 \
    name=John \
    age:=29 married:=false hobbies:='["http", "pies"]' \  # Raw JSON
    description=@about-john.txt \   # Embed text file
    bookmarks:=@bookmarks.json      # Embed JSON file
```

```
PUT /person/1 HTTP/1.1
Accept: application/json, */*
Content-Type: application/json
Host: api.example.com

{
    "age": 29,
    "hobbies": [
        "http",
        "pies"
    ],
    "description": "John is a nice guy who likes pies.",
    "married": false,
    "name": "John",
    "bookmarks": {
        "HTTPie": "http://httpie.org",
    }
}
```

Send JSON data stored in a file (see redirected input for more examples):

```
$ http POST api.example.com/person/1 < person.json
```

Forms

Submitting forms is very similar to sending JSON requests. Often the only difference is in adding the `--form, -f` option, which ensures that data fields are serialized as, and `Content-Type` is set to, `application/x-www-form-urlencoded; charset=utf-8`.

It is possible to make form data the implicit content type instead of JSON via the config file.

Regular forms

```
$ http --form POST api.example.org/person/1 name='John Smith' \
    email=john@example.org cv=@~/Documents/cv.txt
```

```
POST /person/1 HTTP/1.1
Content-Type: application/x-www-form-urlencoded; charset=utf-8

name=John+Smith&email=john%40example.org&cv=John's+CV+...
```

File upload forms

If one or more file fields is present, the serialization and content type is `multipart/form-data`:

```
$ http -f POST example.com/jobs name='John Smith' cv@~/Documents/cv.pdf
```

The request above is the same as if the following HTML form were submitted:

```
<form enctype="multipart/form-data" method="post" action="http://example.com/jobs">
    <input type="text" name="name" />
    <input type="file" name="cv" />
</form>
```

Note that `@` is used to simulate a file upload form field, whereas `=@` just embeds the file content as a regular text field value.

HTTP headers

To set custom headers you can use the `Header:Value` notation:

```
$ http example.org  User-Agent:Bacon/1.0  'Cookie:valued-visitor=yes;foo=bar'  \
    X-Foo:Bar  Referer:http://httpie.org/
```

```
GET / HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Cookie: valued-visitor=yes;foo=bar
Host: example.org
Referer: http://httpie.org/
User-Agent: Bacon/1.0
X-Foo: Bar
```

There are a couple of default headers that HTTPie sets:

```
GET / HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
User-Agent: HTTPie/<version>
```

```
Host: <taken-from-URL>
```

Any of the default headers can be overwritten and some of them unset.

To unset a header that has already been specified (such a one of the default headers), use `Header:` :

```
$ http httpbin.org/headers Accept: User-Agent:
```

To send a header with an empty value, use `Header;` :

```
$ http httpbin.org/headers 'Header;'
```

Authentication

The currently supported authentication schemes are Basic and Digest (see auth plugins for more). There are two flags that control authentication:

`--auth, -a`	Pass a `username:password` pair as the argument. Or, if you only specify a username (`-a username`), you'll be prompted for the password before the request is sent. To send an empty password, pass `username:` . The `username:password@hostname` URL syntax is supported as well (but credentials passed via `-a` have higher priority).
`--auth-type, -A`	Specify the auth mechanism. Possible values are `basic` and `digest` . The default value is `basic` so it can often be omitted.

Basic auth:

```
$ http -a username:password example.org
```

Digest auth:

```
$ http -A digest -a username:password example.org
```

With password prompt:

```
$ http -a username example.org
```

Authorization information from your `~/.netrc` file is honored as well:

```
$ cat ~/.netrc
machine httpbin.org
login httpie
password test

$ http httpbin.org/basic-auth/httpie/test
HTTP/1.1 200 OK
[...]
```

Auth plugins

- httpie-oauth: OAuth
- httpie-hmac-auth: HMAC
- httpie-ntlm: NTLM (NT LAN Manager)
- httpie-negotiate: SPNEGO (GSS Negotiate)
- requests-hawk: Hawk
- httpie-api-auth: ApiAuth

- httpie-edgegrid: EdgeGrid
- httpie-jwt-auth: JWTAuth (JSON Web Tokens)

HTTP redirects

By default, HTTP redirects are not followed and only the first response is shown. To instruct HTTPie to follow the `Location` header of `30x` responses and show the final response instead, use the `--follow, -F` option.

If you additionally wish to see the intermediary requests/responses, then use the `--all` option as well.

To change the default limit of maximum 30 redirects, use the `--max-redirects=<limit>` option.

```
$ http --follow --all --max-redirects=5 httpbin.org/redirect/3
```

Proxies

You can specify proxies to be used through the `--proxy` argument for each protocol (which is included in the value in case of redirects across protocols):

```
$ http --proxy=http:http://10.10.1.10:3128 --proxy=https:https://10.10.1.10:1080 example.org
```

With Basic authentication:

```
$ http --proxy=http:http://user:pass@10.10.1.10:3128 example.org
```

You can also configure proxies by environment variables `HTTP_PROXY` and `HTTPS_PROXY`, and the underlying Requests library will pick them up as well. If you want to disable proxies configured through the environment variables for certain hosts, you can specify them in `NO_PROXY`.

In your `~/.bash_profile`:

```
export HTTP_PROXY=http://10.10.1.10:3128
export HTTPS_PROXY=https://10.10.1.10:1080
export NO_PROXY=localhost,example.com
```

SOCKS

To enable SOCKS proxy support please install `requests[socks]` using `pip`:

```
$ pip install -U requests[socks]
```

Usage is the same as for other types of proxies:

```
$ http --proxy=http:socks5://user:pass@host:port --proxy=https:socks5://user:pass@host:port example.org
```

HTTPS

Server SSL certificate verification

To skip the **host's SSL certificate verification,** you can pass `--verify=no` (default is `yes`):

```
$ http --verify=no https://example.org
```

You can also use `--verify=<CA_BUNDLE_PATH>` to set a **custom CA bundle** path:

```
$ http --verify=/ssl/custom_ca_bundle https://example.org
```

The path can also be configured via the environment variable `REQUESTS_CA_BUNDLE` (picked up by the underlying python-requests library):

```
$ REQUESTS_CA_BUNDLE=/ssl/custom_ca_bundle http https://example.org
```

Client side SSL certificate

To use a **client side certificate** for the SSL communication, you can pass the path of the cert file with `--cert`:

```
$ http --cert=client.pem https://example.org
```

If the **private key** is not contained in the cert file you may pass the path of the key file with `--cert-key`:

```
$ http --cert=client.crt --cert-key=client.key https://example.org
```

SSL version

Use the `--ssl=<PROTOCOL>` to specify the desired protocol version to use. This will default to SSL v2.3 which will negotiate the highest protocol that both the server and your installation of OpenSSL support. The available protocols are `ssl2.3`, `ssl3`, `tls1`, `tls1.1`, `tls1.2`. (The actually available set of protocols may vary depending on your OpenSSL installation.)

```
# Specify the vulnerable SSL v3 protocol to talk to an outdated server:
$ http --ssl=ssl3 https://vulnerable.example.org
```

SNI (Server Name Indication)

If you use HTTPie with Python version lower than 2.7.9 (can be verified with `http --debug`) and need to talk to servers that use **SNI (Server Name Indication)** you need to install some additional dependencies:

```
$ pip install --upgrade pyopenssl pyasn1 ndg-httpsclient
```

You can use the following command to test SNI support:

```
$ http https://sni.velox.ch
```

Output options

By default, HTTPie only outputs the final response and the whole response message is printed (headers as well as the body).

You can control what should be printed via several options:

`--headers, -h`	Only the response headers are printed.
`--body, -b`	Only the response body is printed.
`--verbose, -v`	Print the whole HTTP exchange (request and response). This option also enables `--all` (see bellow).
`--print, -p`	Selects parts of the HTTP exchange.

`--verbose` can often be useful for debugging the request and generating documentation examples:

```
$ http --verbose PUT httpbin.org/put hello=world
PUT /put HTTP/1.1
Accept: application/json, */*
Accept-Encoding: gzip, deflate
Content-Type: application/json
```

```
Host: httpbin.org
User-Agent: HTTPie/0.2.7dev

{
    "hello": "world"
}


HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 477
Content-Type: application/json
Date: Sun, 05 Aug 2012 00:25:23 GMT
Server: gunicorn/0.13.4

{
    […]
}
```

All the other options are just a shortcut for `--print, -p`. It accepts a string of characters each of which represents a specific part of the HTTP exchange:

Character	Stands for
`H`	request headers
`B`	request body
`h`	response headers
`b`	response body

Print request and response headers:

```
$ http --print=Hh PUT httpbin.org/put hello=world
```

Viewing intermediary requests/responses

To see *all* the HTTP communication, i.e. the final request/response as well as any possible intermediary requests/responses, use the `--all` option. The intermediary HTTP communication include followed redirects (with `--follow`), the first unauthorized request when HTTP digest authentication is used (`--auth=digest`), etc.

```
# Include all responses that lead to the final one:
$ http --all --follow httpbin.org/redirect/3
```

The intermediary requests/response are by default formatted according to `--print, -p` (and its shortcuts described above). If you'd like to change that, use the `--history-print, -P` option. It takes the same arguments as `--print, -p` but applies to the intermediary requests only.

```
# Print the intermediary requests/responses differently than the final one:
$ http -A digest -a foo:bar --all -p Hh -P H httpbin.org/digest-auth/auth/foo/bar
```

Conditional body download

As an optimization, the response body is downloaded from the server only if it's part of the output. This is similar to performing a `HEAD` request, except that it applies to any HTTP method you use.

Let's say that there is an API that returns the whole resource when it is updated, but you are only interested in the response headers to see the status code after an update:

```
$ http --headers PATCH example.org/Really-Huge-Resource name='New Name'
```

Since we are only printing the HTTP headers here, the connection to the server is closed as soon as all the response headers have been received. Therefore, bandwidth and time isn't wasted downloading the body which you don't care about.

The response headers are downloaded always, even if they are not part of the output

Redirected Input

A universal method for passing request data is through redirected `stdin` (standard input). Such data is buffered and then with no further processing used as the request body. There are multiple useful ways to use piping:

Redirect from a file:

```
$ http PUT example.com/person/1 X-API-Token:123 < person.json
```

Or the output of another program:

```
$ grep '401 Unauthorized' /var/log/httpd/error_log | http POST example.org/intruders
```

You can use `echo` for simple data:

```
$ echo '{"name": "John"}' | http PATCH example.com/person/1 X-API-Token:123
```

You can even pipe web services together using HTTPie:

```
$ http GET https://api.github.com/repos/jkbrzt/httpie | http POST httpbin.org/post
```

You can use `cat` to enter multiline data on the terminal:

```
$ cat | http POST example.com
<paste>
^D
```

```
$ cat | http POST example.com/todos Content-Type:text/plain
- buy milk
- call parents
^D
```

On OS X, you can send the contents of the clipboard with `pbpaste`:

```
$ pbpaste | http PUT example.com
```

Passing data through `stdin` cannot be combined with data fields specified on the command line:

```
$ echo 'data' | http POST example.org more=data   # This is invalid
```

To prevent HTTPie from reading `stdin` data you can use the `--ignore-stdin` option.

Request data from a filename

An alternative to redirected `stdin` is specifying a filename (as `@/path/to/file`) whose content is used as if it came from `stdin`.

It has the advantage that **the** `Content-Type` **header is automatically set** to the appropriate value based on the filename extension. For example, the following request sends the verbatim contents of that XML file with `Content-Type: application/xml`:

```
$ http PUT httpbin.org/put @/data/file.xml
```

Terminal output

HTTPie does several things by default in order to make its terminal output easy to read.

Colors and formatting

Syntax highlighting is applied to HTTP headers and bodies (where it makes sense). You can choose your preferred color scheme via the `--style` option if you don't like the default one (see `$ http --help` for the possible values).

Also, the following formatting is applied:

- HTTP headers are sorted by name.
- JSON data is indented, sorted by keys, and unicode escapes are converted to the characters they represent.

One of these options can be used to control output processing:

`--pretty=all`	Apply both colors and formatting. Default for terminal output.
`--pretty=colors`	Apply colors.
`--pretty=format`	Apply formatting.
`--pretty=none`	Disables output processing. Default for redirected output.

Binary data

Binary data is suppressed for terminal output, which makes it safe to perform requests to URLs that send back binary data. Binary data is suppressed also in redirected, but prettified output. The connection is closed as soon as we know that the response body is binary,

```
$ http example.org/Movie.mov
```

You will nearly instantly see something like this:

```
HTTP/1.1 200 OK
Accept-Ranges: bytes
Content-Encoding: gzip
Content-Type: video/quicktime
Transfer-Encoding: chunked

+-----------------------------------------+
| NOTE: binary data not shown in terminal |
+-----------------------------------------+
```

Redirected output

HTTPie uses **different defaults** for redirected output than for terminal output:

- Formatting and colors aren't applied (unless `--pretty` is specified).
- Only the response body is printed (unless one of the output options is set).
- Also, binary data isn't suppressed.

The reason is to make piping HTTPie's output to another programs and downloading files work with no extra flags. Most of the time, only the raw response body is of an interest when the output is redirected.

Download a file:

```
$ http example.org/Movie.mov > Movie.mov
```

Download an image of Octocat, resize it using ImageMagick, upload it elsewhere:

```
$ http octodex.github.com/images/original.jpg | convert - -resize 25% -  | http example.org/Octocats
```

Force colorizing and formatting, and show both the request and the response in `less` pager:

```
$ http --pretty=all --verbose example.org | less -R
```

The `-R` flag tells `less` to interpret color escape sequences included HTTPie`s output.

You can create a shortcut for invoking HTTPie with colorized and paged output by adding the following to your `~/.bash_profile`:

```
function httpless {
    # `httpless example.org'
    http --pretty=all --print=hb "$@" | less -R;
}
```

Download mode

HTTPie features a download mode in which it acts similarly to `wget`.

When enabled using the `--download, -d` flag, response headers are printed to the terminal (`stderr`), and a progress bar is shown while the response body is being saved to a file.

```
$ http --download https://github.com/jkbrzt/httpie/archive/master.tar.gz
```

```
HTTP/1.1 200 OK
Content-Disposition: attachment; filename=httpie-master.tar.gz
Content-Length: 257336
Content-Type: application/x-gzip

Downloading 251.30 kB to "httpie-master.tar.gz"
Done. 251.30 kB in 2.73862s (91.76 kB/s)
```

If not provided via `--output, -o`, the output filename will be determined from `Content-Disposition` (if available), or from the URL and `Content-Type`. If the guessed filename already exists, HTTPie adds a unique suffix to it.

You can also redirect the response body to another program while the response headers and progress are still shown in the terminal:

```
$ http -d https://github.com/jkbrzt/httpie/archive/master.tar.gz |  tar zxf -
```

If `--output, -o` is specified, you can resume a partial download using the `--continue, -c` option. This only works with servers that support `Range` requests and `206 Partial Content` responses. If the server doesn't support that, the whole file will simply be downloaded:

```
$ http -dco file.zip example.org/file
```

Other notes:

- The `--download` option only changes how the response body is treated.
- You can still set custom headers, use sessions, `--verbose, -v`, etc.
- `--download` always implies `--follow` (redirects are followed).
- HTTPie exits with status code `1` (error) if the body hasn't been fully downloaded.
- `Accept-Encoding` cannot be set with `--download`.

Streamed responses

Responses are downloaded and printed in chunks, which allows for streaming and large file downloads without using too much RAM. However, when colors and formatting is applied, the whole response is buffered and only then processed at once.

You can use the `--stream, -S` flag to make two things happen:

1. The output is flushed in **much smaller chunks** without any buffering, which makes HTTPie behave kind of like `tail -f` for URLs.
2. Streaming becomes enabled even when the output is prettified: It will be applied to **each line** of the response and flushed immediately. This makes it possible to have a nice output for long-lived requests, such as one to the Twitter streaming API.

Prettified streamed response:

```
$ http --stream -f -a YOUR-TWITTER-NAME https://stream.twitter.com/1/statuses/filter.json track='Justin
```

Streamed output by small chunks alá `tail -f`:

```
# Send each new tweet (JSON object) mentioning "Apple" to another
# server as soon as it arrives from the Twitter streaming API:
$ http --stream -f -a YOUR-TWITTER-NAME https://stream.twitter.com/1/statuses/filter.json track=Apple \
| while read tweet; do echo "$tweet" | http POST example.org/tweets ; done
```

Sessions

By default, every request is completely independent of any previous ones. HTTPie also supports persistent sessions, where custom headers (except for the ones starting with `Content-` or `If-`), authorization, and cookies (manually specified or sent by the server) persist between requests to the same host.

Named sessions

Create a new session named `user1` for `example.org`:

```
$ http --session=user1 -a user1:password example.org X-Foo:Bar
```

Now you can refer to the session by its name, and the previously used authorization and HTTP headers will automatically be set:

```
$ http --session=user1 example.org
```

To create or reuse a different session, simple specify a different name:

```
$ http --session=user2 -a user2:password example.org X-Bar:Foo
```

To use a session without updating it from the request/response exchange once it is created, specify the session name via `--session-read-only=SESSION_NAME` instead.

Named sessions' data is stored in JSON files in the directory `~/.httpie/sessions/<host>/<name>.json` (`%APPDATA%\httpie\sessions\<host>\<name>.json` on Windows).

Anonymous sessions

Instead of a name, you can also directly specify a path to a session file. This allows for sessions to be re-used across multiple hosts:

```
$ http --session=/tmp/session.json example.org
$ http --session=/tmp/session.json admin.example.org
$ http --session=~/.httpie/sessions/another.example.org/test.json example.org
$ http --session-read-only=/tmp/session.json example.org
```

Warning: All session data, including credentials, cookie data, and custom headers are stored in plain text.

Note that session files can also be created and edited manually in a text editor; they are plain JSON.

See also Config.

Config

HTTPie uses a simple configuration file that contains a JSON object with the following keys:

`__meta__`

HTTPie automatically stores some of its metadata here. Do not change.

`default_options`

An `Array` (by default empty) of default options that should be applied to every invocation of HTTPie.

For instance, you can use this option to change the default style and output options: `"default_options": ["--style=fruity", "--body"]` Another useful default option could be `"--session=default"` to make HTTPie always use sessions (one named `default` will automatically be used). Or you could change the implicit request content type from JSON to form by adding `--form` to the list.

Default options from config file can be unset for a particular invocation via `--no-OPTION` arguments passed on the command line (e.g., `--no-style` or `--no-session`). The default location of the configuration file is `~/.httpie/config.json` (or `%APPDATA%\httpie\config.json` on Windows). The config directory location can be changed by setting the `HTTPIE_CONFIG_DIR` environment variable.

Scripting

When using HTTPie from **shell scripts**, it can be handy to set the `--check-status` flag. It instructs HTTPie to exit with an error if the HTTP status is one of `3xx`, `4xx`, or `5xx`. The exit status will be `3` (unless `--follow` is set), `4`, or `5`, respectively.

The `--ignore-stdin` option prevents HTTPie from reading data from `stdin`, which is usually not desirable during non-interactive invocations.

Also, the `--timeout` option allows to overwrite the default 30s timeout:

```
#!/bin/bash

if http --check-status --ignore-stdin --timeout=2.5 HEAD example.org/health &> /dev/null; then
    echo 'OK!'
else
    case $? in
        2) echo 'Request timed out!' ;;
        3) echo 'Unexpected HTTP 3xx Redirection!' ;;
        4) echo 'HTTP 4xx Client Error!' ;;
        5) echo 'HTTP 5xx Server Error!' ;;
        6) echo 'Exceeded --max-redirects=<n> redirects!' ;;
        *) echo 'Other Error!' ;;
    esac
fi
```

Interface design

The syntax of the command arguments closely corresponds to the actual HTTP requests sent over the wire. It has the advantage that it's easy to remember and read. It is often possible to translate an HTTP request to an HTTPie argument list just by inlining the request elements. For example, compare this HTTP request:

```
POST /collection HTTP/1.1
X-API-Key: 123
User-Agent: Bacon/1.0
Content-Type: application/x-www-form-urlencoded

name=value&name2=value2
```

with the HTTPie command that sends it:

```
$ http -f POST example.org/collection \
  X-API-Key:123 \
  User-Agent:Bacon/1.0 \
  name=value \
  name2=value2
```

Notice that both the order of elements and the syntax is very similar, and that only a small portion of the command is used to control HTTPie and doesn't directly correspond to any part of the request (here it's only `-f` asking HTTPie to send a form request).

The two modes, `--pretty=all` (default for terminal) and `--pretty=none` (default for redirected output), allow for both user-friendly interactive use and usage from scripts, where HTTPie serves as a generic HTTP client.

As HTTPie is still under heavy development, the existing command line syntax and some of the `--OPTIONS` may change slightly before HTTPie reaches its final version `1.0`. All changes are recorded in the change log.

Support

Please use the following support channels:

- GitHub issues for bug reports and feature requests.
- Our Gitter chat room to ask questions, discuss features, and for general discussion.
- StackOverflow to ask questions (please make sure to use the httpie tag).
- You can also tweet directly to @jkbrzt.

Authors

Jakub Roztocil (@jkbrzt) created HTTPie and these fine people have contributed.

Logo

See claudiatd/httpie-artwork

Contribute

See CONTRIBUTING.

Change log

See CHANGELOG.

Licence

See LICENSE.

Related projects

- jq — a command-line JSON processor that works great in conjunction with HTTPie
- http-prompt — an interactive shell for HTTPie featuring autocomplete and command syntax highlighting