A Python guide to handling HTTP request failures

Posted by John Boxall on *October 7 2014*

A lot of things can go wrong when requesting information over HTTP from a remote web server: requests timeout, servers fail, government operatives cut undersea cables. You get the picture.

Identifying and handling failures helps build fault tolerant systems that stay up even when services they rely on are down. A nice side effect is your phone is less likely to beep in the middle of the night with a message from your coworkers talking in all caps.

This guide will introduce you to the common ways HTTP requests fail and how to handle the failures.

The examples use Python's fantastic requests library, but the principles shown work ages. You can follow along on your computer by grabbing `requests` off PyPi.

The `requests.get(url)` method is the cornerstone for all the examples. It makes a synchronous HTTP `GET` request to fetch the content from `url`:

```python
# Importing `requests` is omitted from here on for brevity.
# along with the article, make sure to include before tryin
import requests

response = requests.get(url="https://www.mobify.com/")
```

Where possible, the examples use `httpbin` to illustrate the specific failure scenarios. It's a great service for testing how your code will react in a hostile world!

The guide assumes familiarly with making HTTP requests and uses the following terminology:

- Client: The code making the HTTP requests and the server it lives on.
- Server: The box that delivers the HTTP response we requested.
- Caller: The code which instantiates the client and tells

DNS lookup failures

HTTP requests can fail before the client can even make a connection to the server. If the URL specified by the caller has a domain name, the client must look up its IP address before making the request. If the domain name doesn't resolve it's possible that it isn't configured correctly or doesn't exist.

```
# This domain name doesn't exist!
url = "http://www.definitivelydoesnotexist.com/"

try:
    response = request.get(url)
except requests.exceptions.ConnectionError as e:
    print "These aren't the domains we're looking for."
```

It's important to let the caller know they may have entered the wrong domain!

Errors connecting to the server

Even if the hostname of the URL correctly resolves, we might not always succeed in connecting to the server. If someone tripped on its pc ook it down, it's unlikely it will accept our connection!

Errors of this nature often block the client, tying it up waiting for a server that will never respond. For this reason, it's a good idea to add timeouts to the client. That way, if the server takes too long to respond, the client can move on to do something else rather than waiting indefinitely. `requests` provides both `connect` and `read` timeouts. `connect` is the amount of time the client should wait to establish a connection to the server:

```
# Using a very short `connect_timeout` gives us a feel for
# server is slow to pickup the connection.
connect_timeout = 0.0001

try:
    response = requests.get(url="https://httpbin.org/delay/
                            timeout=(connect_timeout, 10.0)
except requests.exceptions.ConnectTimeout as e:
    print "Too slow Mojo!"
```

`read` is the amount of time it should wait between bytes from the server:

```
# Our resource takes longer than `read_timeout` to send a b
read_timeout = 1.0

try:
    response = requests.get(url="https://httpbin.org/delay/
                                                  read_timeout))
except requests.exceptions.ReadTimeout as e:
    print "Waited too long between bytes."
```

The exact values used for the timeout are usually less important than just setting one. You don't want the client to be blocked forever on a slowpoke server. Start with 10 seconds and watch your logs.

> *Extra Credit: Depending on the profile of the system you're building, you may want to implement dynamic timeouts that use historical data to wait longer for servers that are known to be slow. You may want to ban your client from even trying to connect to servers that always timeout.*

HTTP errors

What if something goes sideways while the server is preparing our response? Maybe its database is unresponsive or it was switched in maintenance mode. Whatever the reason, if the server is able to detect that it isn't functioning correctly, it should respond with a HTTP server error code.

Alternatively, if the client is incorrectly constructing the request, the server may respond with a HTTP client error code.

In most cases we'll want to identify these bad response status codes and let the caller handl this is as easy as calling the `response.raise_for_status()` method on the `response` object:

```
# This URL returns a HTTP 500 Server Error.
url = "https://httpbin.org/status/500"

response = requests.get(url)
try:
    response.raise_for_status()
except requests.exceptions.HTTPError as e:
    print "And you get an HTTPError:", e.message
```

Responses that aren't what we expect

It's possible that the caller could request a resource that our client wasn't designed to handle. For example, what if someone uses our RSS reader to request an MKV file of the last episode of Game of Thrones?

We can assert that `Content-Type` response header matches what we expect. Our RSS reader example might look for the following:

```
class WrongContent(requests.exceptions.RequestException):
    """The response has the wrong content."""
```

```
# This URL sets the `Conte            t/plain`.
url = "https://httpbin.org/response-headers?Content-Type=te

response = requests.get(url)
if response.headers["content-type"]
    raise WrongContent(response=response)
```

Note that even if the `Content-Type` header does match what we are expecting, there is no guarantee that the response's body will. Calling code should account for this. For example, if we're expecting JSON and we don't get back JSON, that's a problem. In `requests`, the `response.json()` method tries to convert the response body into a Python object from JSON:

```
# This URL returns an XML document.
url = "https://httpbin.org/xml"

response = requests.get(url)
try:
    data = response.json()
except ValueError:
    raise WrongContent(response=response)
```

> *Extra Credit: If we're processing text data like HTML, don't forget to detect its charset and correctly decode it. You'll need to check the* `response`*'s* `Content-Type` *header as well as potentially the content itself to avoid decoding errors.*

Responses that are too large

Let's go back to our movie example. Not only is the movie not the content type our RSS reader really big. If we're not careful, these kinds of responses could exhaust our client's resources.

To ensure our client hasn't been asked to download the entire internet, we must track how much content we've received. With `requests`, this takes a little more code:

```
from contextlib import closing

class TooBig(requests.exceptions.RequestException):
    """The response was way too big."""

TOO_BIG = 1024 * 1024 * 10 # 10MB
CHUNK_SIZE = 1024 * 128

url = "https://path-to-a-huge-resource/"

with closing(requests.get(url, stream=True)) as response:
    content_length = 0
    for chunk in response.iter_content(chunk_size=CHUNK_SIZ
        content_length = content_length + CHUNK_SIZE
        if content_length > TOO_BIG:
            raise TooBig(response=response)
```

Requests to unexpected URLs

If the client is located inside your network it may have privileged access to internal servers not addressable from the public internet. For example, what if the caller requests `http://127.0.0.1/admin/`?

If you're letting callers request arbitra check that they are allowed to request what they are asking for.

One strategy is to prevent callers from requesting sensitive hosts using a blacklist. A blacklist checks whether the requested domain is present in a set of restricted domains. If it is, the request is rejected before it's even made. At a minimum, we'll want to blacklist internal IP addresses.

Python 3.3 added the `ipaddress` module to the standard library, and in Python 2 we can install its backport `py2-ipaddress` from PyPi. Here we use it to filter requests for internal IP addresses:

```
import ipaddress
import urlparse

url = "http://127.0.0.1/admin/"
hostname = urlparse.urlparse(url).hostname

# `localhost` isn't an IP address, but we probably don't wa
if hostname == 'localhost':
    raise requests.exceptions.InvalidURL(url)
```

```
# If `hostname` quacks like an IP address, make sure it isn
try:
    ip = ipaddress.ip_address(hostname)
except ValueError:
    pass
else:
    if ip.is_loopback or ip.is_reserved or ip.is_private:
        raise requests.exceptions.InvalidURL(url)
```

We might extend our blacklist to include internal hostnames or other sensitive servers. Maybe we also don't want callers to call the server doing the calling. Otherwise it could be turtles all the way down.

> *Extra Credit: If you want to get serious you'll need to resolve the domain name of the requested resource and check whether it maps to a local IP address.*

Alternatively, if callers should only be able to request from a narrow set of servers it may be easier to use a whitelist to reject requests which aren't directed at a known host:

```
import urlparse

WHITELISTED_HOSTS = {"rainbows.com", "magic.com"}

url = "https://unicorns.com/"

if urlparse.urlparse(url=url).hostname not in WHITELISTED_H
    raise requests.exceptions.InvalidURL(url)
```

> *Extra Credit: Depending on your needs, you might also want to restrict other p P request, including the protocol used, or the ports. Additionally, if you find a caller abusing the syste build a mechanism to ban them!*

Handling errors

So now that we've identified all these errors, what the heck should be do with them?

Logging

What broke? When? Where? Logging failures creates a trail that you can search for patterns. Logs will often give you insight about how you can further tweak your configuration to best suit your system or whether someone is abusing the system.

Retrying

When you're firing bits around the world sometimes you just get unlucky. Depending on what you're doing, it may make sense to just retry the request if you think the error was intermittent. `requests` provides an interface for creating custom `adapters` that can be used to implement retries:

```
# Use a `Session` instance to customize how `requests` hand
session = requests.Session()

# `mount` a custom adapter that retries failed connections
session.mount("http://", requests.adapters.HTTPAdapter(max_
session.mount("https://", requests.adapters.HTTPAdapter(max

# Rejoice with new fault tolerant behaviour!
session.get(url="https://www.service-that-drops-every-odd-r
```

Just make sure you only retry requests that are idempotent!

Notification

Finally, you'll need to raise the error to the caller. You'll want to do it in a way that makes it easy for the caller to handle all possible exceptions, but also in a way that makes it clear why the exception was raised. This is especially important if you will be displaying the error to a non-technical user and you want to provide clear instructions about whether they've mistyped the domain or the server they are trying to connect to is down. In Python, this is a great chance to read up on properly re-raising exceptions!

For Further Consideration

SSL

SSL is pretty cool and we should do more of it. The `requests` library verifies certificates by default. If you're using a different library or language, be sure to check that your client is checking that certificates are valid. You don't want someone MITMing your connection!

```
try:
    response = requests.get(url="https://www.super-sketchy-
                                verify=True)
except requests.exceptions.SSLError as e:
    print "That domain looks super sketchy."
```

Internationalized Domain Names

International Domain Names are a thing. Many libraries will handle these by default now, but you probably want to throw a test case in there that makes sure the snowman works:

```
requests.get(url=u"http://➡.ws/mobify")
```

Performance

Depending on how you've built your client, there are a variety of ways you might be able to improve its performance:

- Consider requesting the compress by setting the header `Accept-Content: gzip`. You'll need to make sure your client can handle decompressing the content.

- Consider having your client connect through an HTTP proxy like Squid or Varnish. If you expect to be requesting the same resources again and again, the proxy's cache may considerably reduce response times for cacheable resources.

- `requests` is blocking. That means that your client will only be able to process one request at a time. If your system needs to support many concurrent requests, you might consider going async using libraries like `request-futures` or `grequests`.

Tooling

There are a number of tools out there that can help simplify putting this all together:

- httpbin.org allows you to quickly test a number of different HTTP response scenarios. It's `delay` and `drip` endpoints are especially useful for testing weird edge conditions.

- HTTPretty is a mocking library that can make cranking out unit tests for all these errors relatively simple.

Wrapping it all up

Wow, there are a lot of ways HTTP requests can fail. TLDR, when making a request:

- Account for DNS lookup failures
- Set a connection and read timeout
- Be sure to handle HTTP errors
- Check that the response has the content type you expect
- Limit the maximum response size
- Ensure that private URLs are not requestable
- Always. Be. SSLing.

Now it's your turn!

Go forth and write fault tolerant services that request data using HTTP!

> *Did we miss anything? Let us kno*
> *below.*

John Boxall

On a day-to-day basis, John oversees the continuous integration and deployment of Mobify's services by our high-performing product team. While John is mainly focused on scaling Mobify's technology services, he is also a regular speaker among the Python and JavaScript communities in both Canada and North America.

View all posts by John Boxall ⟶

Sorin Sbarnea · 6 months ago

I made a feature request for having delayed-retry support directly in python requests, feel free to support it at https://github.com/kennethreit...

Share ›

Mohammad · 6 months ago

Thank you for the article. It is very helpful.
I just didn't figure out the exact exception handling and retry mechanism.
I tried requests.Session to retry request failures but at some points my script is stuck because apparently there is no time-out defined for that. Can you explain how to retry a requests w ithout facing such problem?

Share ›

Alexandru Stanciu · a y ear ago

Thanks for the great w rite-up.
Just one minor fix w hen dealing w ith large responses:

```
with closing(requests.get(url, stream=True)) as response:
    content_length = 0
    for chunk in response.iter_content(chunk_size=CHUNK_SIZE):
        content_length = content_length + len(chunk) # instead of CHUNK_S
        if content_length > TOO_BIG:
            raise TooBig(response=response)
```

1 Share ›

John Boxall Mod Alexandru Stanciu · a y ear ago

Thanks!

`CHUNK_SIZE` w ill be correct for every iteration but the last. I can imagine a service w here this w ould be a problem... for example you advertise you w ill accept up to 1GB video uploads and user carefully compresses a file to have it just squeak in under the limit and bam w e miscount the bytes :) The tiny performance penalty of calling `len(chunk)` on every iteration is probably w orth it.

On a completely unrelated note, I recently had to w rite the same check using Go – here is w hat I came up w ith:

https://play.golang.org/p/aJHd...

Share ›

Recommended reading on Mobify's Developer Blog

A Beginner's Guide to HTTP Cache Headers

Kyle Young, our resident System Engineer, uses this post to explain what HTTP cache headers do to create performant browsing experiences. He also dives in and covers what kind of user-controllable cache optimizations we can make and test.

April 2013

5 Advanced Mobile Web Design Techniques You've Probably Never Seen Before

In this post, **David Fay** takes a detailed look at some of the unique features used on *Style.com*'s mobile website. Check out the double carousel used on the show calendar — bet you've never seen that before.

October 2013

Does Mobile Web Performance Optimization Still Matter?

Peter McLachlan, Mobify's Chief Product Officer, takes an informative glance at the current state of web performance as indicated by latency times, bandwidth and resulting page load times, which leaves us with a few key tactics for optimization.

January 2013

CSS Sprites vs. Data URIs: Which is Faster on Mobile?

In this last featured post, **Peter McLachlan** hits the numbers hard in his research of the performance of Data URIs compared to CSS sprites. The results will surprise you and leave you with a better idea of how to optimize your web pages further.

August 2013

Contact us or 1 (866) 502 5880

Mobify