The Future of Python HTTP

2012

3 min read • 766 words

I like to think Requests is mostly analogous to Werkzeug in terms of purpose, functionality, and goals. One is for servers while the other is for clients.Werkzeug and Flask were huge inspirations for Requests' design.

> This acknowledgment of Werkzeug and Flask's influence reveals Kenneth's design philosophy: learning from existing excellent libraries and adapting their principles to new domains. Armin Ronacher's elegant API design patterns clearly influenced the human-friendly approach of Requests.

As a matter of fact, Requests contains a decent bit of Werkzeug's internal data structures.

So, why are they separate projects?

Brainstorming

At PyCon 2012 a few weeks ago, Andrey Petrov, Armin Ronacher, Paul McMillan, and myself got into a room for a brainstorming session around the possibility of formally combining our efforts.

My expectations going in weren't that high, but that quickly changed once we were all in the same room. We discussed the general state of Python HTTP, security concerns,distributed services, and web application testing.

Today, making real HTTP Requests to an in-process WSGI app with a real HTTP client is not simple. Can you imagine writing real OAuth tests for your application with the same HTTP consumer your clients will use?

The root of the problem is that WSGI doesn't map `1:1` to HTTP.

> This insight about WSGI's impedance mismatch with HTTP was prescient. WSGI was designed as a Python-specific abstraction that doesn't fully capture HTTP's semantics, creating friction when trying to write realistic tests or implement certain HTTP features.

So, instead of taking the WebOb approach of using WSGI as the common protocol between services, why not use HTTP itself? The rest of the world uses HTTP as the most-common denominator after all.

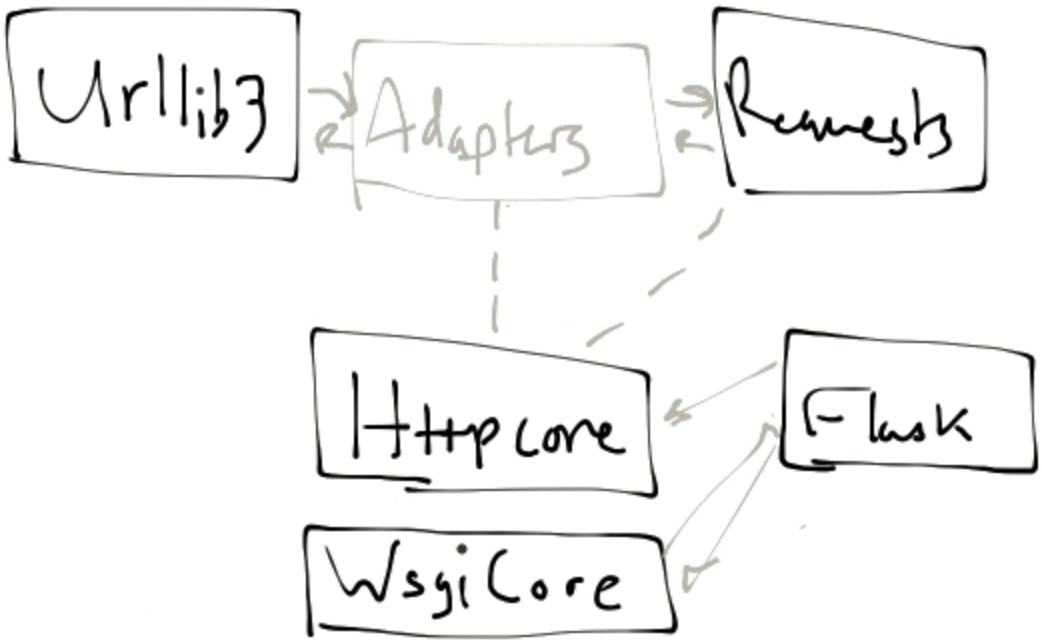
After a few hours, we drafted up a solid plan:

- Consolidate shared code between Requests and Werkzeug into a new `httpcore` module.
- Move WSGI-specific Werkzeug code into a new `wsgicore` module.
- Make HTTP (vs WSGI) the common protocol between services.
- Provide a transport adapter mechanism for mocking and emulating HTTP services.

The Architecture

Requests, Flask, and Werkzeug will remain the same to the end user.

Behind the scenes, the same functions used to generate a request will be used to consume it. For example, stream handling, header parsing, and form-encoding will all be synchronous functions from `httpcore`.



Adapters

Transport Adapters will provide a mechanism to define interaction methods for an "HTTP" service. They will allow you to fully mock a web service to fit your needs.

Gloriously simplified example (implementation subject to change):

```
class DistributedAdapter(BaseAdapter):
    def __init__(self):
        self.connect_pool = …

    def send(self, request):
        """Takes a Request object, returns a Response object."""
        # Whatever needs to happen here.
        …
```

HTTPCore

HTTPCore will be comprised of the code currently shared by Requests and Werkzeug, general HTTP utilities, and base objects / data structures.

Specifically, it will provide:

- Request and Response objects
- General HTTP Utilities
- Common Data Structures (`MultiDict`, `CaseInsensitiveDict`)
- Common Data Structure Utilities (`merge_kwargs`)
- Stream Handling (`make_line_iter`, `make_chunk_iter`)
- HTTP Parsing (`http-parser`)
- URI/IRI Parsing and Handling Functions (`uricore`)
- SSL Utilities
- Cookie Handling
- Base `TransportAdapter`

WSGICore

WSGICore will extend the framework that HTTPCore provides to be used by WSGI applications. It will replace the WSGI-specific parts of Werkzeug:

- Request and Response objects
- WSGI Transport Adapter (`HTTP <-> WSGI`)
- WSGI Utility Functions

Distributed Services

In addition to testing web applications, this new Adapter system will provide a fantastic mechanism for distributed services.

In Requests, you'll be able to mount external services to the routing mechanism bymocking HTTP. To Requests, it'll be an HTTP Service, but in reality the service could be anything: a random number generator, ZeroMQ socket, proxy, WSGI application,&c.

Here's some theoretical example code:

```
import requests
from webscale import DevNullAdapter
from wsgicore.adapters import WsgiAdapter
from haystackapp.core import app as haystack

s = requests.session()
s.mount('null:', DevNullAdapter())
s.mount('http://haystack', WsgiAdapter(app=haystack))

# Make a request via DevNullAdapter
r = s.get('null://someurl')

# Make a request via Haystack WSGI App
r = s.get('http://haystack/index')

# Make a request via standard HTTPS
r = s.get('https://github.com/')
```

Long-term Advantages

There's a number of advantages to this design and approach in the future:

- Requests will be able to use the same cache backends for HTTP Requests that Flask/Werkzeug does for views. They will be moved to `cachecore`.
- Security enhancements (e.g. DNSSEC) can live in `httpcore` rather than waiting for a PEP or standard library implementation.
- Django could potentially utilize the security features provided by `httpcore`.
- Django/Flask could potentially use Requests as their respective official test clients.

Development

If you have thoughts to share, feel free to discuss this with us on Freenode at `#cores`.

There's little code to show at the moment, but you can track the development over on GitHub:

https://github.com/core
