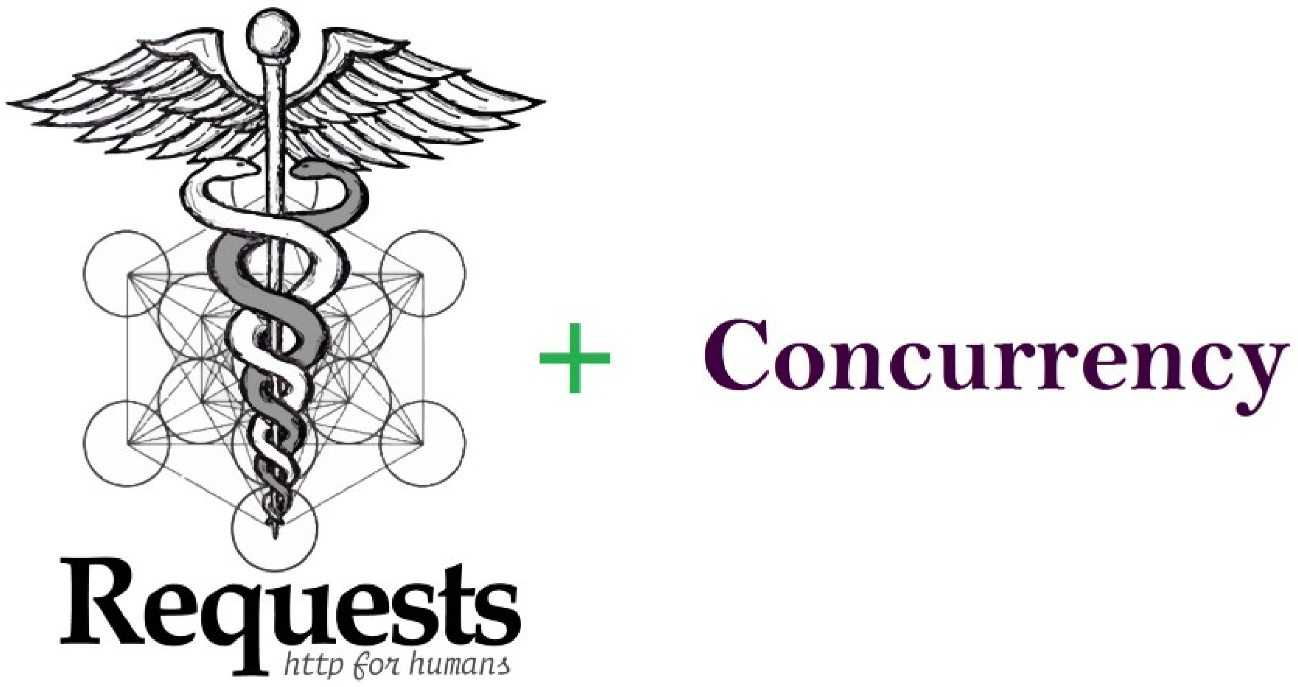
Scale-up REST API Functional Tests to Performance Tests in Python

Peter Xie Follow
Aug 8, 2019 · 5 min read

Imagine you have written REST API function tests in Python, how to scale them up to do performance tests?



I have previously written an article about how to create REST API function tests using Python. In this article, I will continue to explain how you can use existing function tests and scale them up to do performance tests, using Python modules requests, threading, and queue.

· · ·

Let's use the same flask mock service endpoint I used for the functional tests, but just add a `time.sleep(0.2)` to simulate network delay 0.2 seconds.

```python
from flask import Flask
import time

app = Flask(__name__)

@app.route('/json', methods=['POST', 'GET'])
def test_json():
    time.sleep(0.2) # simulate delay
    return '{"code": 1, "message": '[illegible]lo, World!" }'

# Run in HTTP
app.run(host='127.0.0.1', port='5000')
```

flask_mock_simple_service_w_delay.py hosted with ♡ by GitHub view raw

Save the code as a file, e.g. flask_mock_simple_service.py, and run it by `python flask_mock_simple_service.py`.

Now you can access this service by typing http://127.0.0.1:5000/json in a browser or by running the functional test code as below, `pytest -sv test_mock_service.py`. And you will get response content `{"code": 1, "message": "Hello, World!" }`.

```python
import requests

def test_mock_service():
    url = 'http://127.0.0.1:5000/json'
    resp = requests.get([illegible]
    assert resp.status_code == 200
    assert resp.json()["code"] == 1
    print(resp.text)
```

test_mock_service.py hosted with ♡ by GitHub view raw

Functional test

. . .

Let's see how to convert function tests to performance tests.

Step 1: Modify existing functional tests

To create performance tests using existing function tests, first we need to modify the function test functions a bit to suit performance tests. Let's copy & paste the existing test_mock_service.py as a new file perf_test_mock_service.py and modify it for performance tests.

We use `assert` to verify something in the response, e.g. status_code and "code" value in the body content, in pytest functional tests. It needs to be converted to checking (some people use term validate), so one single request failure won't stop the whole performance tests. We still mark the test as fail if validation fails, in the return values.

```python
# Updated test function in perf_test_mock_service.py
def test_mock_service():
    url = 'http://127.0.0.1:5000/json'
    resp = requests.get(url)
    if resp.status_code != 200:
        print('Test failed with response status code %s.' % resp.status_code )
        return 'fail', resp.elapsed.total_seconds()
    elif resp.json()["code"] != 1:
        print('Test failed with code %s != 1.' %  resp.json()["code"] )
        return 'fail', resp.elapsed.total_seconds()
    else:
        print('Test passed.')
        return 'pass', resp.elapsed.total_seconds()
```

test_mock_service_mod_for_perf.py hosted with ♡ by GitHub view raw

The second change is we need to return the response time, i.e. the amount of time elapsed between sending the request and the arrival of the response. This is so easy to just return `resp.elapsed.total_seconds()`, e.g., 0.210752.

Step 2: Loop test function

We need to create a loop test function so it will continuously send requests. As you can from the code below, it just loops one or more API functional tests with a wait-time, and stops once loop times(default infinite) is reached. A queue variable is used to store the results so we can calculate performance stats later. Note Queue is multiple thread safe.

```python
import requests
import queue
import sys
import time

queue_results = queue.Queue()
# def test_mock_service():  - omitted

def loop_test(loop_wait=0, loop_times=sys.maxsize):
    looped_times = 0
    while looped_times < loop_times:
        # run an API test
        test_result, elapsed_time = test_mock_service()
        # put results into a queue for statistics
        queue_results.put(['test_mock_service', test_result, elapsed_time])

        # You can add more API tests in a loop here.
        looped_times += 1
        time.sleep(loop_wait)

if __name__ == '__main__':
    loop_test(loop_times=3)
```

perf_test_mock_service_v1_loop.py hosted with ♡ by GitHub view raw

Run the test code and you will see the result as below:

```
python perf_test_mock_service_v1_loop.py
Test passed.
Test passed.
Test passed.
```

Step 3: Start Concurrent Users

This is the most important step, to create and start concurrent threads to simulate concurrent users. To add one thread, just create a Thread object and provide a function (loop_test here) to run the thread, and function arguments (loop_times here) if any. Then start the thread by the `start()` method, and call the `join()` method to wait for the thread to finish before proceeding in the main thread.

Note: Thread parameter `daemon=True` tells spawned threads to exit if main thread exits.

```python
import requests
import threading
import queue
import sys
import time

# Global variables
queue_results = queue.Queue()
start_time = 0

# def test_mock_service(): - omitted
# def loop_test(loop_wait=0, loop_times=sys.maxsize):  - omitted

if __name__ == '__main__':
    ### Test Settings ###
    concurrent_users = 2
    loop_times = 3

    workers = []
    start_time = time.time()
    print('Tests started at %s.' % start_time )

    # start concurrent user threads
    for i in range(concurrent_users):
        thread = threading.Thread(target=loop_test, kwargs={'loop_times': loop_times}, daemon=True)
        thread.start()
        workers.append(thread)

    # Block until all threads finish.
    for w in workers:
        w.join()

    end_time = time.time()
    print('\nTests ended at %s.' % end_time )
    print('Total test time: %s seconds.' %  (end_time - start_time) )
```

perf_test_mock_service_v2_concurrent.py hosted with ♡ by GitHub view raw

For instance, we start 2 threads and each thread loops 3 times. Run the test code and you will see the result as below:

```
python perf_test_mock_service_v2_concurrent.py
Tests started at 1565252504.3480494.
```

```
Test passed.
Test passed.
Test passed.
Test passed.
Test passed.
Test passed.

Tests ended at 1565252505.0206523.
Total test time: 0.6726028919219971 seconds.
```

Step 4: Performance Statistics

Now that we have been able to run concurrent performance tests, we can add code to calculate performance metrics.

- **Time per Request(TPR):** measure min, max and mean(avg) value using `resp.elapsed.total_seconds()` for all pass requests.
- **Requests per Second(RPS)**: measure mean value by dividing total pass requests by total test time.

Function `stats()` is added for this purpose, and we just call this function at the end of the main thread. As you see from the code below, we get the test results from the queue until it is empty or current queue size is reached, and measure TPR, RPS as well as total fail, exception and pass requests.

```python
# module imports - omitted
# Global variables
queue_results = queue.Queue()
start_time = 0

# def test_mock_service(): - omitted
# def loop_test(loop_wait=0, loop_times=sys.maxsize):  - omitted
def stats():
    # request per second
    rps_mean = 0
    total_tested_requests = 0
    total_pass_requests = 0
    # time per request
    tpr_min = 999
    tpr_mean = 0
    tpr_max = 0
    sum_response_time = 0
    # failures
    total_fail_requests = 0
    total_exception_requests = 0

    global start_time
    end_time = time.time()
    # get the approximate queue size
    qsize = queue_results.qsize()
    loop = 0
    for i in range(qsize):
        try:
            result=queue_results.get_nowait()
            loop +=1
        except Empty:
            break
        # calc stats
        if result[1] == 'exception':
            total_exception_requests += 1
        elif result[1] == 'fail':
            total_fail_requests += 1
        elif result[1] == 'pass':
            total_pass_requests += 1
            sum_response_time += result[2]
            # update min and max time per request
            if result[2] < tpr_min:
                tpr_min = result[2]
            if result[2] > tpr_max:
                tpr_max = result[2]

    total_tested_requests += loop
    # time per requests - mean (avg)
    if total_pass_requests != 0:
        tpr_mean = sum_response_time / total_pass_requests

    # requests per second - mean
    if start_time == 0:
        print('stats: start_time is not set, skipping rps stats.')
    else:
        tested_time = end_time - start_time
        rps_mean = total_pass_requests / tested_time

    # print stats
    print('\n-----------------Test Statistics---------------')
    print(time.asctime())
    print('Total requests: %s, pass: %s, fail: %s, exception: %s'
        % (total_tested_requests, total_pass_requests, total_fail_requests, total_exception_requ
    if total_pass_requests > 0:
        print('For pass requests:')
        print('Request per Second - mean: %.2f' % rps_mean)
        print('Time per Request   - mean: %.6f, min: %.6f, max: %.6f'
            % (tpr_mean, tpr_min, tpr_max) )
```

Then add stats() function into main as below.

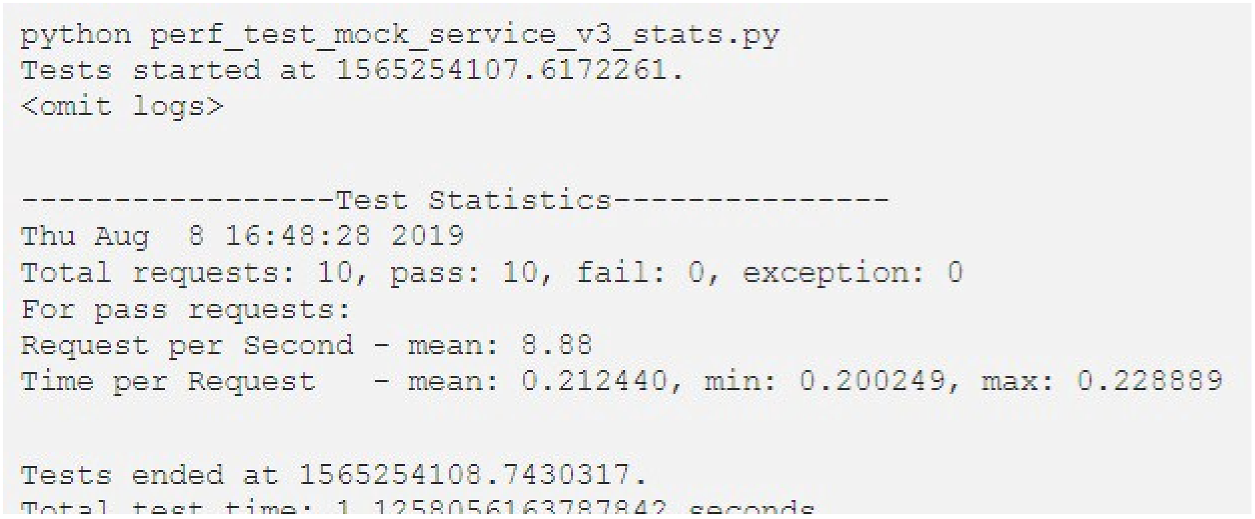
```
if __name__ == '__main__':
    ### Test Settings ###
    concurrent_users = 2
    loop_times = 5

    workers = []
    start_time = time.time()
    print('Tests started at %s.' % start_time )
    # start concurrent user threads - omitted
    end_time = time.time()

    # Performance stats
    stats()
    print('\nTests ended at %s.' % end_time )
    print('Total test time: %s seconds.' %  (end_time - start_time) )
```

perf_test_mock_service_v3_stats_part2.py hosted with ♡ by GitHub view raw

Example output of 2 threads and 5 loop times are as follows:



```
python perf_test_mock_service_v3_stats.py
Tests started at 1565254107.6172261.
<omit logs>


-----------------Test Statistics---------------
Thu Aug  8 16:48:28 2019
Total requests: 10, pass: 10, fail: 0, exception: 0
For pass requests:
Request per Second - mean: 8.88
Time per Request   - mean: 0.212440, min: 0.200249, max: 0.228889


Tests ended at 1565254108.7430317.
Total test time: 1.1258056163787842 seconds
Total test time: [illegible] seconds.
```

It looks good, isn’t it?

Step 5: Test timer

Normally we want to control the duration of performance tests by time as well, and we stop the test either loop times is reached or time is up.

First, we need to create a global Event (`event_time_up = threading.Event()`) to notify loop_test when time is up.

Second, we create a function (`set_event_time_up`) to set the event.

Finally, we create a Timer (`timer = threading.Timer(test_time, set_event_time_up)`) and start it after performance tests are started. The timer will wait for `test_time` and call function `set_event_time_up` . Note we also need to cancel the timer if loop_times is reached earlier than this timer.

```python
# module imports - omitted
# Global variables - omitted
# event flag to set and check test time is up.
event_time_up = threading.Event()

# def test_mock_service(): - omitted
def loop_test(loop_wait=0, loop_times=sys.maxsize):
    looped_times = 0
    while (looped_times < loop_times
        and not event_time_up.is_set()):
        test_result, elapsed_time = test_mock_service()
        queue_results.put(['test_mock_service', test_result, elapsed_time])
        looped_times += 1
        time.sleep(loop_wait)

#def stats(): - omitted
def set_event_time_up():
    if not event_time_up.is_set():
        event_time_up.set()

if __name__ == '__main__':
    ### Test Settings ###
    concurrent_users = 2
    loop_times = 100
    test_time = 5 # time in seconds, e.g. 36000

    workers = []
    start_time = time.time()
    print('Tests started at %s.' % start_time )

    # start concurrent user threads
    for i in range(concurrent_users):
        thread = threading.Thread(target=loop_test, kwargs={'loop_times': loop_times}, daemon=Tr
        thread.start()
        workers.append(thread)

    # set a timer to stop testing
    timer = threading.Timer(test_time, set_event_time_up)
    timer.start()

    # Block until all threads finish.
    for w in workers:
        w.join()

    # stop timer if loop_times is reached first.
    if not event_time_up.is_set():
        timer.cancel()

    end_time = time.time()
    stats()
    print('\nTests ended at %s.' % end_time )
    print('Total test time: %s seconds.' %  (end_time - start_time) )
```

Example output of below settings when test time is reached first.

concurrent_users = 2
loop_times = 100
test_time = 5 # time in seconds

```
python perf_test_mock_service_v4_test_timer.py
Tests started at 1565259509.1139543.

----------------Test Statistics---------------
```

```
Thu Aug  8 18:18:34 2019
Total requests: 46, pass: 46, fail: 0, exception: 0
For pass requests:
Request per Second - mean: 9.15
Time per Request   - mean: 0.209215, min: 0.197342, max: 0.229308

Tests ended at 1565259514.1410472.
Total test time: 5.027092933654785 seconds.
```

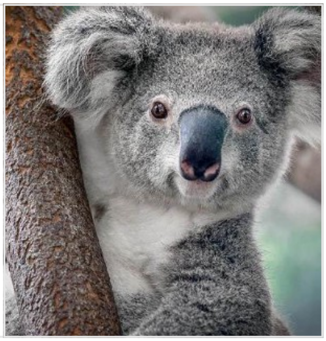
Put It All Together

I've added this performance test script into my Python REST API test framework below, which now cover both functional and performance tests. The script includes everything mentioned above and more, such as print stats continuously in an interval, e.g., 5 minutes.

peterjpxie/REST_API_Test_Framework_Python

REST API Test Framework example using Python requests and flask for both functional and performance tests. ...

github.com



. . .

Why Not Coroutine

We have implemented the performance tests using threading. There are two reasons why I don't recommend using coroutine like asyncio in this case.

1. Coroutine is complicated and tricky to use. You may measure the response time wrong if you don't **really** understand coroutine. See my post for examples. And it is always a debatable topic whether asyncio is good or not, like this post.

2. The beautiful reqests package we used in funtional tests is not coroutine based, so you cannot reuse the same functions if you want to use coroutine for performance tests.

However, if you really want to use coroutine instead of threading for performance tests, I would recommend using a professionally written package like locust.

Python Rest Api Restful Performance Testing