Contents

1. [**API Performance Test using Python Modules** 2](#_Toc48852701)
2. [Sending registration requests 3](#_Toc48852702)

[Step 3: Start Concurrent Users 4](#_Toc48852703)

[Step 4: Performance Statistics 6](#_Toc48852704)

1. API performance Test

[**https://github.com/peterjpxie**](https://github.com/peterjpxie)

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

  I will continue to explain how you can use existing function tests and scale them up to do performance tests, using Python modules [requests](https://2.python-requests.org/), [threading](https://docs.python.org/3/library/threading.html?highlight=threading#module-threading), and [queue](https://docs.python.org/3/library/queue.html?highlight=queue#module-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.

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": "Hello, World!" }'

# Run in HTTP

app.run(host='127.0.0.1', port='5000')

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!” }.

* Functional Test

# **Functional test** 🡪 perf\_test\_mock\_service.py

import requests

def test\_mock\_service():

url = 'http://127.0.0.1:5000/json'

resp = requests.get(url)

assert resp.status\_code == 200

assert resp.json()["code"] == 1

print(resp.text)

* **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.

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, 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.

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.

# **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()

# 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**.

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)

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.

# perf\_test\_mock\_service\_v2\_concurrent.py

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) )

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.

# [**perf\_test\_mock\_service\_v3\_stats.py**](https://gist.github.com/peterjpxie/eb83bf476fd2b158bb1a72b8b83b7b0e#file-perf_test_mock_service_v3_stats-py)

# module imports - omitted

**# Global variables**

**queue\_results = queue.Queue()**

**#**  **queue\_results**.**put**([**'test\_mock\_service**', **test\_result**, #**elapsed\_time**])

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\_requests))

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) )

[**perf\_test\_mock\_service\_v3\_stats\_part2.py**](https://gist.github.com/peterjpxie/4682fc8e2abe7c959c1b3c472abd139e#file-perf_test_mock_service_v3_stats_part2-py)

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

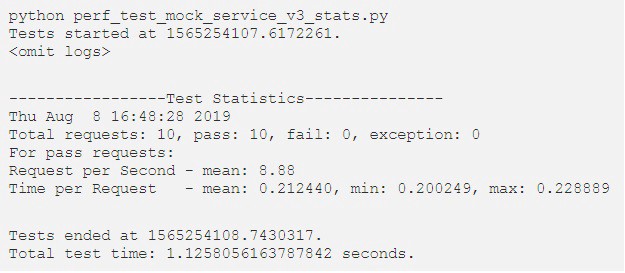
**end\_time** = time.time()

# Performance stats

**stats()**

print('\nTests ended at %s.' % end\_time )

print('Total test time: %s seconds.' % (end\_time - start\_time) )



# 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.

[**perf\_test\_mock\_service\_v4\_test\_timer.py**](https://gist.github.com/peterjpxie/16956219d989894a9062bc5d0f262136#file-perf_test_mock_service_v4_test_timer-py)

# 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=True)

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()

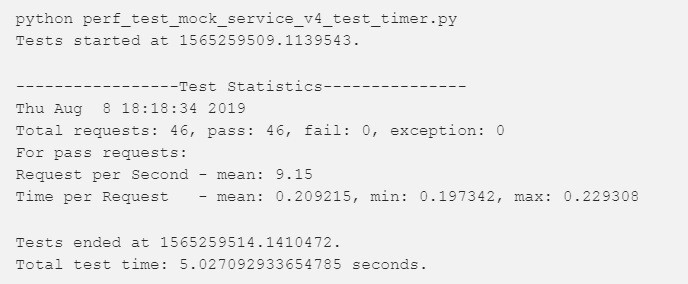
**stats()**

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



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](https://towardsdatascience.com/asyncio-is-not-parallelism-70bfed470489) for examples. And it is always a debatable topic whether asyncio is good or not, like [this post](https://lucumr.pocoo.org/2016/10/30/i-dont-understand-asyncio/).
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.

**Sending registration Requests**

def registration:

URL = "ip"

PARAMS = {**'name'**:'test',**'password'**:'test1','primary\_**email**':'test667@gmail.com','primary\_mobile\_**number'**:'9999999999','country\_abbrev':'US'}

r = **requests.post**(**url** = URL,**params** = PARAMS,**auth**=HTTPDigestAuth('user', 'pass'))

response = r.text

print(response)

* You can contain this in a for loop and append the counter to your name like PARAMS['name']+counter
* Take a look at [Faker Python Package](https://faker.readthedocs.io/en/latest/index.html). This generates fake data for you whether you need to bootstrap your database, create good-looking XML documents, fill-in your persistence to stress test it, or anonymize data taken from a production service, Faker is for you.
* from locust import HttpLocust, TaskSet, task
* class UserBehavior(TaskSet):
* def on\_start(self):
* pass # add code that you want to run during ramp up
* def on\_stop(self):
* pass # add code that you want to run during ramp down
* def registration(self):
* name = fake.first\_name()
* last\_name = fake.last\_name()
* password = ''
* email = name + last\_name + '@gmail.com'
* phone = fake.phone\_number()
* URL = "ip"
* PARAMS = {'name':name,'password': password,'primary\_email': email,'primary\_mobile\_number':phone,'country\_abbrev':'US'}
* self.**client.post**(URL, PARAMS)
* class WebsiteUser(HttpLocust):
* task\_set = UserBehavior
* min\_wait = 5000
* max\_wait = 9000
* To start the load test, run
* **locust -f locust\_files/my\_locust\_file.py --host=http://example.com**
* To start the load test, run
* locust -f locust\_files/my\_locust\_file.py --host=[http://example.com](http://example.com/),

**from locust import HttpLocust, TaskSet**

def **login**(self):

params= {'name':'test','password':'test1','primary\_email':'test667@gmail.com','primary\_mobile\_number':'9999999999','country\_abbrev':'US'}

self.**client.post**(URL, data=params)

#The data parameter or json can both be used here. If it's a dict then data would work but for json replace data with json. For more information you can check out requests package as Locust internally uses requests only.

**class UserBehavior(TaskSet):**

tasks = {index: 2, profile: 1}

def on\_start(self):

login(self)

def on\_stop(self):

pass

@task

def try(self):

pass

class WebsiteUser(HttpLocust):

task\_set = UserBehavior

min\_wait = 5000

max\_wait = 9000

* where host would be your IP. You can then go to **127.0.0.1:8089** to select **the number of virtual users to simulate**. On **windows** there's a limitation of **1024 users only**. But you can use the amazing support of **Master Slave Architecture** provided by Locust.
* **PS:** Anything put in the **on\_start method runs only once for each user**. So since you want to test the limit of the API you should prefer adding that request under the **@task** decorator.

Locust:

1. **The number of users**: the number of users testing your application. Each user opens a TCP connection to your application and tests it.
2. **Hatch rate**: For each second, how many users will be added to the current users until the total amount of users. Each hatch Locust calls the on\_start function if you have.

for example:

* Number of users: 1000
* Hatch rate: 10

Each second 10 users added to current users starting from 0 so in 100 seconds you will have 1000 users. When it reaches to the number of users, the statistic will be reset.

It tries to distribute the load equally for each user. Instead, you can give weight for the tasks, not the users.

If you want to specify the run time for a test, you can do that with --run-time or -t:

**$** locust -f --headless -u 1000 -r 100  1h30m

**Locust Test scenario 1**:

In my scenario, I'm running Locust without the web UI. The command I'm using is

locust -f my\_locust\_file --no\_web -c 20 -r 4 # as a hack I add -t 10s

This corresponds to a 4 users being hatched every second up to a total of 20 users.

My objective is to have each of the 20 locust users execute a task and I'd like the locust run to complete and exit when the last user's (20th user) task completes. The collected statistics should only include the response times associated with each task.

In this scenario, 5 tasks (user scenarios) have been identified which can be randomly associated with a user:

class UserScenarios(TaskSet):

tasks = [Sequence\_One, ServerSequence, Components\_Sequence, Embedded\_Sequence, Connectivity\_Sequence]

class MyLocust(HttpLocust):

def \_\_init\_\_(self):

super().\_\_init\_\_()

MyLocust.counter += 1

print(f"Counter = {MyLocust.counter}")

counter = 0

task\_set = UserScenarios

wait\_time = between(1, 1)

host = 'https://\*\*\*\*\*.com'

Each task (user scenario) corresponds to a different sequence of 3 or 4 pages that should be loaded in order. An example sanitized and simplified sequence consisting of 2 pages is:

class Sequence\_One(TaskSequence):

@seq\_task(1)

def get\_task1(self):

response = self.**client.get**(url='https://\*\*\*\*',

name='https://\*\*\*\*',

timeout=30,

allow\_redirects=False,

headers={...})

@seq\_task(2)

def get\_task2(self):

response = self.client.get(url='https://\*\*\*\*',

name='https://\*\*\*\*',

timeout=30,

allow\_redirects=False,

headers={...})

Is there a way to stop the test after the 20th (nth) user task completes? If every task visits 4 pages for example, I want the test to terminate after the 20\*4 = 80 page request is made. In fact only 80 total page requests should be made as part of this test.

My experience with this test is that page requests continue to be made after the last user task completes until I either manually stop the test or use a time limit which is a bit longer than the tasks actually need to complete.

This is actually possible! While it's not documented, at least anywhere that I could find, stopping a Locust can be achieved by calling raise StopLocust()

In the provided scenario, each of your tasks is a TaskSequence class. In order to quit a Locust after a single task has been performed, you will need to add another step (function) to the overall class which makes a call to StopLocust(). I've provided an example below -

class Scenario(TaskSequence):

@**seq\_task**(1)

def task\_1(self):

self.client.get("/going\_somewhere")

@seq\_task(2)

def task\_2(self):

self.client.get("/going\_somewhere\_else")

@seq\_task(3)

def done(self):

raise StopLocust()

By default, a Locust is set to never end and will continuously pick a task from the available task sets provided to it. Supplying raise StopLocust() will tell the current user (Locust) to stop and no longer pick tasks once it reaches the end of that current task.

**Looks like StopLocust has been removed on locust 1.2, but you can use locust.exception.StopUser**

**Locust Test scenario 2**:

* import random, json
* from locust import HttpUser, task, between, TaskSet, User

class UserBehavior(HttpUser): # class UserBehavior(TaskSet):

* min\_wait = 5000
* max\_wait = 9000
* def \_\_init\_\_(self, parent):
* super(UserBehavior, self).\_\_init\_\_(parent)
* self.token = ""
* self.headers = {}
* def on\_start(self):
* self.token = self.login()
* self.headers = {'Authorization': 'Token ' + self.token}
* def login(self):
* response = self.client.post("/v1/auth/login", data={'phoneNumber': '+666000666', 'password': 'dupadupa'})
* @task
* def index(self):
* self.client.get("/v1/me/profile", headers=self.headers)
* class WebsiteUser(HttpUser):
* task\_set = UserBehavior
* min\_wait = 5000
* max\_wait = 9000