3 July 2017

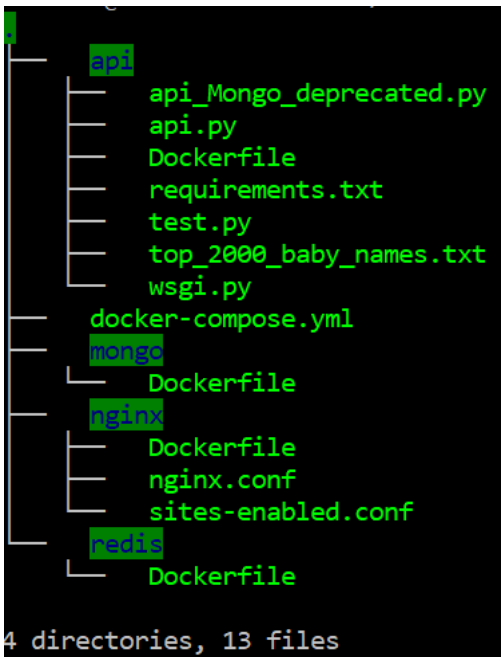
TEXTMAN

An ephemeral text messaging service

Sean A. Leeka | *sleeka@swbell.net*

About this Repo

This is the repo of the Docker unofficial package for TEXTMAN, maintained by Sean Leeka. Textman handles 180 texts /sec (360 requests /sec) with several thousand users.



I'm tweaking nginx.conf to prevent the request timeouts I'm receiving. CPU use is 30% on the API servers, 10% on nginx, and negligible on Redis – with RAM use in a few hundred MB.

Redis is a powerful upgrade. Key 'global_message_id' increments and keeps ids unique. Each user is a key and their value is a list of ids. Each id is a key, valued with message and an expiration through Redis based on the timeout.

The Docker containers include:

'**api**_{1,2,3,4}' – four application servers using Python's Flask framework

'**nginx**' – a web server and reverse-proxy load-balancer

'**redis**' – NoSQL cached database with the option for persistence

Installation

In the root directory of Textman, execute:

1) `docker-compose build`

This will download and install the minimum number of dependencies. Flask is run through Python3.6-slim, a lightweight distribution (as seen in flask.dockerfile). Mongo is run through Mongo 3.4 as seen in mongo.dockerfile.

2) `docker-compose up` will run Textman `-d` for daemon mode.

Access

POST: `curl http://127.0.0.1:5000/chat -d username=Sleeka -d message=Interesting`
GET: `curl -G http://127.0.0.1:5000/chat?username=Sleeka`

http://127.0.0.1:5000/chat is the Flask API address. It accepts and validates the following ***POST*** requests:

Name	Type	Description	Required	Default Value
username	String	The recipient of the message	Y	n/a
text	String	The content of the message	Y	n/a
timeout	Integer	The number of seconds the message should live before expiring	N	60

A successful POST response has a status code of **201 Created** and a JSON object containing the id of the newly-created message, for example:

{ "id": 1337 }

The API accepts and validates the following ***GET*** requests:

Name	Type	Description	Required
username	String	The recipient of the message	Y

A success response will include: a JSON array of messages, each of which contains the message's ID and text, for example:
[

```
        {
                "id": 345,
                "text": "This is a message"
        },
        {
                "id": 95958,
                "text": "This is also a message"
        }
]
```

---

1) What could be the different scenarios/assumptions you might have to make in order to scale your service? Are there questions you would ask to product owners to help you understanding how you need to scale your service?

→ Millions of daily users will be handled by several web servers. Distribution through horizontal sharding on a database handles this load too – these are assumptions. Disk write *has* been a bottleneck though – transactions per second are a challenge, and CPU use is high under load tests up to 100,000 users.

What percentage of users are repeat, and can I divide them by physical location? What is our maximum and average timeout? Is persistence of data necessary? How frequently do I need to back up data?

What if 5% of recipient phones are turned off for several hours, perhaps at night? When is there is a surge in TPS?

2) What technology choices would you pick and why? Are you keeping your current technology stack or will you make changes.

→ I will use Go! Go and Erlang (RabbitMQ) are the fastest technologies for requests /sec. They out-perform Python by 10x

Redis has an efficient data structure – automatically expiring keys based on timeout. Note that it's not optimized through Docker because of enabled Transparent Huge Pages (> 4KB memory blocks).

I'm curious about RabbitMQ . “Get messages” will consume texts that have been received and scheduled workers with Celery expire/delete those that time out – this would be rebuilding much of what Redis does though.

I group users by location, unique timeouts, and by user activity.

With SQL (and NoSQL) I have often seen DELETE / CREATE take twice as long as UPDATE. If long-term persistence is required until guaranteed delivery, I'll update an 'expired' flag and DELETE rows when they emerge from a scheduled queue later.

Redis is fast and works for short timeouts, despite being single-threaded. It has doubled performance. With MongoDB, memory allocation initially slows down 3-5x.

3) How would you monitor/measure a service like this?

→ Linode Longview. What seems crucial here is preemptive testing, which I'm eager to continue with this project. Disk write is a concern. These data models could grow exponentially, depending on the division of database tables/queues and the logic to clean up expired texts – this growth-rate and distribution of unique users is key. I'm working with several low-level C daemons which integrate with Python for data science – netdata, psutil to automatically log/graph CPU/RAM/Disk use.

4) What are the things that could go wrong with each approach?

→ With non-persistent Redis, any server outage would drop all data. Python may struggle with a high volume of TPS. Disk failure, or overuse, would be fatal and disk is heavily used. Extreme timeouts or a rapid spike in users could overload disks.