Assignment 2 – First Price Twitter

This assignment will build upon the web system we made in assignment 1. Note: Not all choices made for our system are the best choices; some are added because it'll make for a better assignment.

Table of Contents

Features

The website should mainly consume an API and should support the following basic features. Features new for assignment 2 being underlined.

- Post tweets
- Edit tweets
- Delete tweets
- List/show tweets
- Search for tweets
- Search for hashtags
- Like tweets

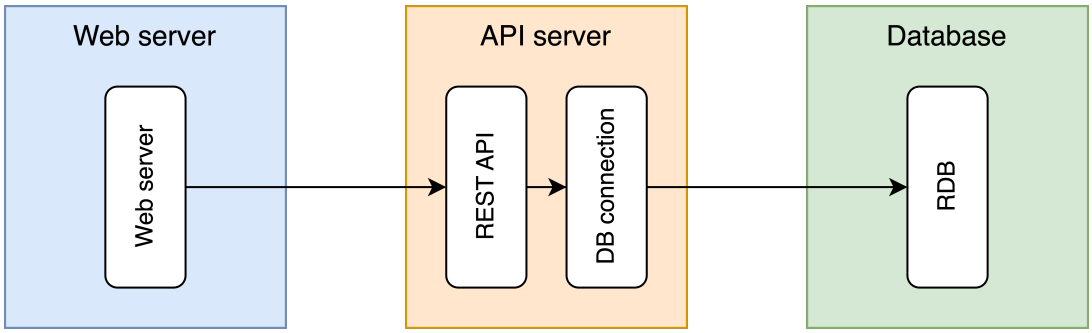
- Make account
- List accounts
- Search for account

And other features you deem necessary for the site.

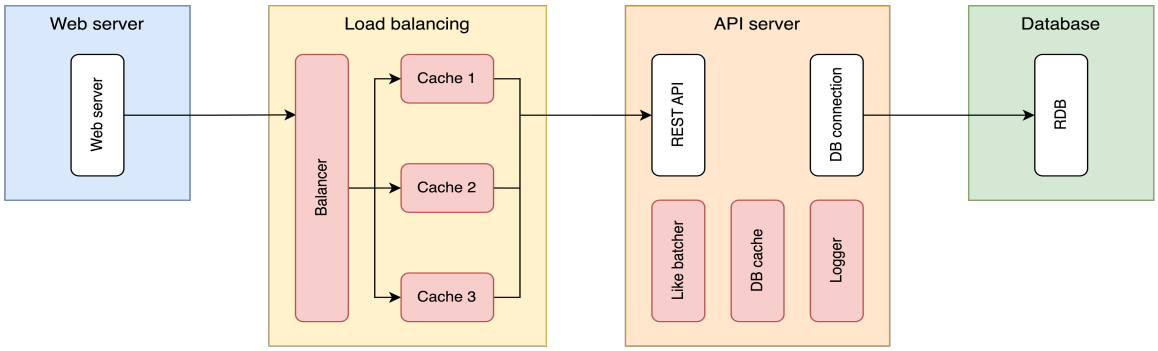
Note: The like button does not have to track which users has liked. Idempotence be damned! It can simply be a “make the database number one larger each time I click it”-button. Terrible design, but simpler.

Architecture

The first assignment had the following architecture.



This assignments architecture should be the same basic web service, but with some added complexity – for scaling purposes.



White blocks are the same as from assignment 1, while red ones are new.

Load balancer + caches

We want to limit unnecessary read requests to the API and therefore use caches between the web server and API. These should only cache reads and not writes or deletes. They should be written in Python, and can use a like structure the dictionary. The cached requests should automatically be removed when more than 1 minute old. Normally, we may want a different thread to deal with this. To reduce complexity, we can simply trigger the invalidation check whenever we use the cache. So a cache access could be something like the following.

1. Run invalidation check on cache.
2. Check if GET request is already cached.

3. If yes: Return. If no: Send request further (API) and update cache.

The caches should run as Docker containers.

The load balancer should split the API requests between the caches, using whichever mechanism you want. Here, you can use pre-built systems or containers, like an nginx Docker container.

This whole subsystem should be the kind you could chose to add or remove later, and the rest of the system should be relatively unaffected (except some IP address modifications).

Like batcher

Imagine our Twitter clone becomes enormously popular, and 10s of thousands of likes gets sent every second. That kind of traffic would be a problem for our server. If every like is sent from the API to the database, it may overload the database. A better solution may be the following.

1. When the API server receives a “like” post request, do not send it on immediately. Rather, store it locally. For example, in a dictionary or a small database like SQLite.
2. At some point, send many likes at once to the database. Instead of a “+1” 10000 times, we may get a “+5000” twice.

That is, instead of sending likes one at a time, we want to gather them in a batch in the API and send them when we have received multiple. Whenever a like counter gets one of the following criteria, they should be sent to the server.

- A post (tweet) has more than 10 likes.
- A post (tweet) has not passed on its likes in more than 1 minute.

This means that each post ID receiving a like will have to be stored along with the current time. It should be written in Python. As a bad pseudo-code example:

```
{
    "post-id-1234": {"likes": 1442, "time": 123456789},
    ...
}
```

Though make your structured code as you think makes sense.

DB cache

These caches should be more or less the same as the ones described above. Updated are just passed along. 1 minute outdate timer. Only target reads.

Logger

Add another endpoint to the API at /logs, which should give a list of API calls ran. It should be in order. It should include all calls since start-up of the server. It should include which method at which endpoint but does not need to include the body. E.g., {..., ["GET", "/like/123"], ...}. The logger should also log number database access. This can later be used to compare savings from use of the caches. (Something we won’t be doing here, but hypothetically.)

Hosting

You have two options in this assignment, on how you can host these services. You can:

1. Host them in Render.com or a similar service (e.g., AWS, Railway, etc.). This is what you most likely did for assignment 1.
2. Host them locally, using Docker compose, in which case, you must include the Docker compose file.

Check list

As a brief check list of what is to be done:

Goals

- Implement "likes" as a part of your website, if not already added.
- Limit unnecessary traffic to the API and database using caching. Both between web server and API, and API and database.
- Limit unnecessary traffic to the database by sending likes in batches/in bulk.
- Improve the speed of the architecture by adding load balancing.
- Implement logging of requests for the API. Logs should be accessible from the /logs endpoint.

Technical requirements

- Implement the caches, the like batcher, the logger and the API using Python. (API should be implemented already, from assignment 1.)
- Use Docker to implement the caches and the API/logger/like batcher. It is here up to your whether you place the API. Logger and like batcher in multiple files in a single docker image, or if you make individual docker images for them.

Bonus points

How to get a good grade: Just do what the assignment asks. But, in the case where you are missing some points from some of the other requirements, the bonus points can fill in. Thus, bonus points are optional, but recommended.

Unit testing

From the API, the DB cache and the like batcher, make at least 2 unit tests for at least 2 different Python files. I.e., at least 4 total unit tests. They should all pass.

Style guides and type hinting

For the API server elements, follow style guides (PEP 8) and use at least some type hinting. Make sure the tools flake8 and mypy runs successfully on your coding files.

API load testing

Make a separate Python script which does tests our API. We should do two kinds of testing:

- **Basic functionality testing**: Perform some basic tests on the API by POST-ing and GET-ing from the API, and make sure the results are as expected. E.g., POST a tweet and check that the GET can find it.
- **Load testing**: Perform some heavy load tests, by sending many requests in quick succession and seeing how well the API can deal with it.

Report

The report should be similar to the one for assignment 1 but mainly focus on the assignment 2 additions. That is, you do not have to re-describe things from the assignment 1 report.

Comment on the design and architecture choices you made. Comment on how they relate to the work we have done.

Additionally, discuss the design and architecture choices made in this assignment were. For example, the use of a like batcher: Does it make sense? Does it depend on the traffic we get?

Group

The groups should be the same as for assignment 1. If not, you must specify which members worked on which assignment.

Deliverables

Deliver the following to Inspera:

- Report x2 (PDF-files)
- Repos x2 (compressed files – preferably zip)