Group 10

Homework 2

Preface

Installing and running the application

The submitted implementation can be executed by opening a command prompt in the root folder of the project (same root folder as the github repo) and running docker-compose build, followed by docker-compose up, and successively navigating to localhost on port 5000 in any web browser. Celery-Flower is accessible on port 5555.

Running pytest

Pytest will create and delete a new database in the project root folder. Because of this, it should be called from the root folder itself, and when executing a single test it's important to set the execution directory to the root folder. This was done to preserve test consistency with minimal effort. Furthermore, certain tests will synchronously call celery tasks, and for these to complete a running redis server on localhost:6379 is necessary. This address/port combination can be edited within monolith/background.py, at line 19.

Accessing the API documentation

Documentation for all public API routes can be found at the '/doc/public' route on the running app. This documentation was automatically generated from the docstrings of the methods within our views, through the use of the Flask-Autodoc package.

Comment and code style

PEP8 style compliance was enforced with flake8 inspections. Comments for docstrings follow the reStructuredText syntax, while other single-line comments were inserted when necessary.

Implementation

Database

The database is comprised of five main classes.

1. User: this class holds info on all registered users and administrators. These info range from login info such as the salt and hash of their password and their email, to personal info such as their full name and date of birth, to system info such as their login status or security clearance, and number of Lottery points accrued.
2. Message: class holding all messages. There are three kinds of messages identified by their 'status' field, and these are "drafts", "sent", and "delivered" messages. This table will also hold the path on server of any image attachments, and will have utility fields such as "visible_to_sender" and "is_read", used for deletion and notifications.
3. Blacklist: a class holding 1:1 user to blocked user mappings for the block feature.
4. Report: class holding reports made by users against other users, with a description of the report, and the time at which this report was issued.
5. Notification: class of notifications to be displayed for all users, with the associated user email, title, description and timestamp.

Celery Tasks

Under monolith/background.py we can find a set of celery tasks. These are both periodic tasks scheduled with celery-beat and one-off tasks called with a certain delay from within the flask app itself.

1. deliver_message: task which changes the status of a message from '1', sent, to '2', delivered. This task is called both within the monolith.send module and from monolith.background itself, when running the error-prevention task "send_unsent_past_due".
2. create_notification: task to insert a new notification into the database. Currently this is only ever called synchronously because of our implementation, but could be called with a delay as needed.
3. lottery_task: task ran periodically from celery-beat, which simply queries the database for all users and picks one at random, assigning them a fixed number of lottery points and pushing a relevant notification.
4. cleanup_pictures: garbage collection task scheduled every hour with celery-beat. Given the nature of multi-user messages, forwards and deletes, counting the actual occurrences of an image path within the database is the simplest way to make sure no image is deleted while still in use.
5. send_unsent_past_due: in case of catastrophic system failure a deliver_message task might not fire. This method, scheduled with celery-beat, makes sure that no message remains undelivered for more than five minutes.

Tests

Test methods were aimed at both coverage and functionality. Test cases try to execute actions that users would normally call, using, when possible or necessary to prove soundness of methods, the actual API calls rather than directly calling methods.
An exception to this were celery tasks. Since setting up celery within the CI pipeline was seen as superfluous and tests using wait times would be both time-consuming and finnicky, all celery tasks were tested through synchronous calls.

Cooperation

We divided the development into "sprints", each covering one of the three user-story priorities. Usually we met twice per sprint, once at the start and once at the end, chaining two meetings into one to begin a new sprint right away.
During these meetings we would assign tasks to people, and then keep in touch through IM channels in case further coordination was needed. Each member of the team would create a new branch, complete their user story on that branch, and then open a Pull Request which all users could read and comment on.
Issues were used to track who was assigned to which story, and which PR completed them.