CPSC 449 - Web Back-End Engineering

Project 2, Fall 2020

due October 7 (Section 01) / October 9 (Section 02)

Last updated Wednesday, September 29 12:30 am PDT

In this project you will build two back-end microservices for a microblogging service similar to Twitter.

The project may be completed individually or in a group of no more than three (3) people. All students on the team must be enrolled in the same section of the course.

Services

Your microservices should be written as two separate Flask applications connected to a single SQLite Version 3 database.

Users microservice

Each user has a username, an email address, a hashed password, and a list of users that the user is following.

The following API operations should be exposed:

- `createUser(username, email, password)`

  Registers a new user account.

- `authenticateUser(username, password)`

  Returns `true` if the supplied password matches the hashed password stored for that username in the database.

- `addFollower(username, usernameToFollow)`

  Start following a new user.

- `removeFollower(username, usernameToRemove)`

  Stop following a user.

Timelines microservice

Each post to a timeline should have the author's username, the text of the post, and a timestamp showing when the post was created.

Timelines should be returned in reverse chronological order. Limit the maximum number of posts retrieved for any timeline to 25.

The following API operations should be exposed:

- `getUserTimeline(username)`

  Returns recent tweets from a user.

- `getPublicTimeline()`

  Returns recent tweets from all users.

- `getHomeTimeline(username)`

  Returns recent tweets from all users that this user follows.

- `postTweet(username, text)`

  Post a new tweet.

REST API Definition

Given the service descriptions above, design appropriate JSON data formats, HTTP methods, URL endpoints, and HTTP status codes for each operation. Document each of these for each service and each operation.

Your API should follow the principles of RESTful design as described in class and in the assigned reading.

Note that an API will usually need to document which resources are public and which resources and methods require authentication. In this project, all resources are public, so you may omit that information from your documentation.

API Implementation

Implement your APIs in Python 3 using Flask. You are encouraged, but not required, to use Flask API to obtain additional functionality.

All data, including error messages, should be in JSON format with the `Content-Type` header field set to `application/json`.

Session state

Requests to each microservice must include all information necessary to complete the request; your APIs must not use the Flask `session` object to maintain state between requests.

The resources in each microservice are public and do not require authentication. (Though note that the Users service may be used for authentication by other services.)

Password hashing

The Flask framework is built on top of a WSGI library called Werkzeug, which includes security helper functions for generating and checking hashed passwords. If you have Flask installed, you have access to this library.

Database

Use the Python Standard Library's built-in `sqlite3` module as the database for your Flask application. You are encouraged, but not required, to use PugSQL as an alternative.

Informally, your database schema should be in approximately third normal form. If you are not familiar with database normalization, see Thomas H. Grayson's lecture note Relational Database Design: Rules of Thumb from the MIT OpenCourseWare for MIT Course Number 11.208. Note in particular that data items should be atomic: this means, for example, that the list of users that a user is following should be stored as separate rows in a join table, not as a single column in the user table.

Create a file `schema.sql` to enable foreign key support and run SQL `CREATE TABLE`, `CREATE INDEX`, and `INSERT` statements to create your database and populate it with test data. Enable foreign key support for the database.

Add a custom command to your Flask application so that you can create the initial schema and populate the database by running the following shell command:

```
    $ flask init
```

Managing microservice processes

Write a `Procfile` and use `foreman` to start and manage the Flask processes for each microservice. Configure the `flask run` command to use the `$PORT` environment variable so that Foreman can manage the ports assigned to each process.

Note that while using a `.env` file to set environment variables is a convenient shortcut during development, those values will be shared by any Flask application contained in the same directory. Each microservice in your `Procfile` will need a different setting for FLASK_APP, so you will need to set them individually using the `env(1)` or some other method such as wrapping the invocation of the `flask` command in a shell script.

Implementation tips

- Use the Flask development environment.
- For simplicity, avoid Flask Blueprints and Python packages; implement each microservice in a single `.py` file.
- If you are struggling with the Flask documentation, don't forget that the Syllabus recommends a book on Flask available freely online.
- Get comfortable with the HTTPie and `sqlite3` command-line interfaces.
- On UNIX-based platforms, you can set environment variables temporarily by including them on the same line as the command you want to run. For example, instead of

  ```
  $ export FLASK_APP=timelines

  $ flask init
  ```

  you can type

  ```
  $ FLASK_APP=timelines flask init
  ```

  This feature can also be useful in your `Procfile`.

Test platform

You may use any platform to develop services and tests, but note that per the Syllabus the test environment for projects in this course is a Tuffix VM with Python 3.8.2. It is your responsibility to ensure that your code runs on this platform.

Submission

Submit a compressed tarball (`.tar.gz`, `.tgz`, `.tar.Z`, `.tar.bz2`, or `.tar.xz`) file containing the following through Canvas before class on the due date:

1. The Python source code for each microservice
2. `Procfile` definitions for each service

3. A SQL schema for the database
4. A `README.TXT` file as described in the Syllabus
5. Documentation in PDF format or included in your `README.TXT` showing how to create the database and start the services.
6. Documentation in PDF format defining the REST API for each service
7. Do **not** include compiled `.pyc` files, the contents of `.git/` directories, SQLite database files, or other binary artifacts.
8. If you include other files in your tarball, I will not examine them unless your `README.TXT` states explicitly that they should be included in evaluation of your project.

If the assignment is completed by a team, only one submission is required. Be certain to identify the names of all students on your team in your `README.TXT` and in each PDF document.

Failing to comply with these submission instructions will immediately result in **no credit with no further feedback**. That means that you will need to re-submit, and if there are any problems with the project you will not have a chance to correct them.