Kenytt Avery, Computer Science Department, California State University, Fullerton

CPSC 449 - Web Back-End Engineering

Project 2, Fall 2021

due Friday, October 22 at 9:45 pm PDT

Last updated Sunday October 10, 6:55 pm PDT

In this project you will build two RESTful back-end services and prepare them for production deployment.

The project may be completed individually, or in a team of up to three students as long as all students are enrolled in the same section of the course.

Platforms

You may use any platform to develop the code for this project, but per the Syllabus the test environment for projects in this course is Tuffix 2020 with Python 3.8.10. It is your responsibility to ensure that your code runs on this platform.

Note: this is a back-end project only. While your browser may be able to display JSON objects returned in response to an HTTP GET request, you do not need to implement a front-end or other user interface for testing API methods. Use an HTTP client program such as HTTPie or curl.

Libraries and Code

This project must be implemented using the Hug, sqlite_utils, and Requests libraries.

Code from the Python documentation, the library documentation, examples provided by the instructor, and *A Whirlwind Tour of Python*. All other code must be your own original work or the original work of other members of your team.

Problem domain

We plan to build a microblogging service similar to Twitter

Each user has a *username*, a *bio*, an *email address*, and a *password*. Users can *follow* each other, and they can post *messages*.

Each post includes the author's *username*, the *text* of the post, and a *timestamp* showing when the post was created. If the post is a repost, it should also include the *URL* of the original post.

Posts are aggregated into *timelines*. There are three different timelines:

1. Each user has a *user timeline* consisting of the posts that they have made.
2. Each user has a *home timeline* consisting of recent posts by all users that this user follows.
3. There is a *public timeline* consisting of all posts from all users.

Timelines are organized in reverse chronological order, with the newest posts listed first.

Services

Create two RESTful microservices, one for users and one for timelines.

API Implementation

Use Hug routing to define RESTful resources for users and posts. Your resources should make appropriate use of HTTP methods, status codes, and headers.

Your API should follow the principles of RESTful design as described in class and in the assigned reading, with the exception that all input and output representations should be in JSON format with the `Content-Type` header field set to `application/json`.

Authentication

Calls to the users service do not need to be authenticated. (This is not particularly realistic -- obviously operations such as changing passwords or updating user details should require both authentication and authorization, but for the purposes of the project you may assume that this service will only be accessed by authorized users from behind a firewall.)

Most resources exposed by the posts service should be public, with two exceptions:

1. Only authenticated users should be able to create new posts.
2. A user's home timeline should only be accessible to themselves.

These resources should require HTTP Basic access authentication using Hug's authentication support. The wrapped function that you pass as the `requires` argument to these routes should use the Requests library to authenticate the user against the corresponding resource in the users service.

Databases

The Python Standard Library includes the `sqlite3` embedded database. Use this database via the sqlite_utils Python library for your services.

Each service should have its own independent database. Services should be accessed only through their HTTP interfaces, and neither service should use the other service's database.

Informally, your database schemas 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 a user table.

Getting ready for production

The HTTP server started by the `hug` command-line application is suitable for development, but is not intended for production use. (It is based on the same `http.server` module that you used in Project 1.)

Adding a WSGI server

In production, Python applications are run via the Web Service Gateway Interface. Gunicorn is a WSGI server that can be used to run Hug applications. Start your services with Gunicorn and test that HTTP requests are handled correctly.

Managing processes

Write a `Procfile` for your services and use `foreman` to start both services using Gunicorn. Bind each service using Foreman's `$PORT` environment variable so that they will not conflict.

In order to avoid missing output from Foreman, create a `.env` file and set `PYTHONUNBUFFERED=True`.

Use the foreman `--formation` switch to start one instance of the users service and three instances of the timeline service. Test that each service instance can be accessed separately

Load balancing

Now that multiple instances of the timeline service are running, you will need to be able to load-balance requests across those instances. Configure HAProxy to act as an HTTP load balancer for the timelines service and proxy for the users service.

Testing

Routing works correctly when all API calls can be accessed successfully through HAProxy.

You can check load-balancing by examining the logs shown by foreman. Make several requests to HAProxy for the timelines service, and verify that they are routed to different service instances (e.g. `timelines.1`, `timelines.2`, and `timelines.3`).

HTTP Basic Authentication works correctly when HTTPie calls to protected endpoints without the `--auth` switch fail with `HTTP 401 Unauthorized`, or when visiting an endpoint in your browser pops up an “Authentication Required” dialog box.

Submission

Your project should include summary information in a README file as described in the Syllabus. Only one submission is required. Be certain to identify the names of all team members at the top of the README.

Submit a tarball (`.tar.gz`, `.tgz`, `.tar.Z`, `.tar.bz2`, or `.tar.xz`) file containing the following through Canvas before 9:45 pm PDT on the due date:

1. A README file as described in the Syllabus
2. The Python source code for each microservice
3. `Procfile` definitions for each service
4. An initialization script and CSV or SQL schema files for each database
5. Any other configuration files
6. Documentation in PDF format or included in your `README.TXT` including:
    a. How to create the databases and start the services
    b. REST API documentation for each service

Do **not** include compiled `.pyc` files, the contents of `__pycache__` directories, or other binary files. If you use git, this includes the contents of the `.git/` directory. See Git Archive: How to export a git project for details.

If you include other files in your tarball, I will not examine them unless your README states explicitly that they should be included in the evaluation of your project.

The Canvas submission deadline includes a grace period of an hour. Canvas will mark submissions after the first submission deadline as late, but your grade will not be penalized. If you miss the second deadline, you will not be able to submit and will not receive credit for the project.

Note: do not attempt to submit projects via email. Projects must be submitted via Canvas, and instructors cannot submit projects on students’ behalf.

See the following sections of the Canvas documentation for instructions on group submission:

- How do I join a group as a student?
- How do I submit an assignment on behalf of a group?

Grading

The grade for the project will be assigned on the following five-point scale:

Exemplary (5 points)

Results are correct; code and documentation are clearly written; organization makes it easy to review.

Basically Correct (4 points)

Results are (mostly) correct, but the code or documentation is not easy to follow or is not clear.

Right Idea (3 points)

The approach is appropriate, but the work has mistakes in code or documentation that undermine the correctness of the results.

Solid Start (2 points)

The work makes a good start, but does not include documentation or has fundamental conceptual problems in code that do not produce legitimate results.

Did Something (1 point)

The solution began an attempt, but is either insufficient to assess correctness or is on entirely the wrong track.

Did Nothing (0 points)

Project was not submitted, submitted code belonging to someone other than the members of the team, or submission was of such low quality that there is nothing to assess.

Acknowledgements: this grading scale is drawn from the general rubric used by Professor Michael Ekstrand at Boise State University.