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

CPSC 449 - Web Back-End Engineering

Project 3, Spring 2022 - due April 29

Last updated Tuesday April 13, 2:15 pm PDT

In this project you will build an additional RESTful back-end microservice and run multiple instances of the new service with load-balancing and sharding.

The following are the learning goals for this project:

1. Gaining further experience with implementing back-end APIs in Python and FastAPI.
2. Running and debugging multiple instances of a web service process.
3. Implementing application-level sharding for a relational database.
4. Configuring and testing an HTTP reverse proxy and load balancer.

Project Teams

This project must be completed in a team of three or four students. All students on the team must be enrolled in the same section of the course. You are free to choose the initial members of your team, but the instructor may adjust membership as necessary to accommodate everyone.

Teams should be formed by the beginning of class on Monday, April 11.

In order to submit a project as a team, you must join a group for that project in Canvas:

- Several groups have been pre-created by the instructor, and your team must join one of these for your submission. Projects cannot be submitted using groups that you create yourself.
- The first team member to join will automatically be assigned as the group leader.
- Teams are specific to the project, so you must join a new group for each project, even if you worked with the same team on a previous project.
- If there are no empty groups available, email the instructor immediately to request a new group to be created.

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?

Platforms

Per the Syllabus, this project is written for Tuffix 2020. Instructions are provided only for that platform. While it may be possible for you to run portions of this project on other platforms, debugging or troubleshooting any issues you may encounter while doing so are entirely your responsibility.

Libraries, Tools, and Code

The requirements for this project are the same as for Project 2, with the addition of the Traefik application proxy.

Other than the sources specified in the previous project and the StackOverflow article referenced below, all other code must be your own original work or the original work of other members of your team.

Adding a New Service

Create a RESTful microservice for tracking user wins and losses. Your service should expose the following operations:

- Posting a win or loss for a particular game, along with a timestamp and number of guesses.

  Note: this endpoint only needs to record the final result when a user has finished a game; you do not need to keep track of a user's score in the middle of a game. This functionality will be added in Project 4.

- Retrieving the statistics for a user. If you visit Wordle, open your browser's developer tools, and examine local storage for the entry `nyt-wordle-statistics`, you'll see that it uses the following format, which your service should also return:

```
{
   "currentStreak": 4,
   "maxStreak": 5,
   "guesses":{
      "1": 0,
      "2": 0,
      "3": 2,
      "4": 3,
      "5": 4,
      "6": 2,
      "fail": 1
   },
   "winPercentage": 92,
   "gamesPlayed": 12,
   "gamesWon": 11,
   "averageGuesses": 5
}
```

Note: the names `"1"` through `"6"` are not valid identifiers for class members in Python, so if you are returning a Pydantic model as a response you will need to use Field customization to set an alias for each of these.

- Retrieving the top 10 users by number of wins
- Retrieving the top 10 users by longest streak

As with Project 2, your service should be implemented using RESTful principles.

Sharding a Database

The statistics service should have its own database. The schema is provided at `https://github.com/ProfAvery/cpsc449/tree/master/stats` in `share/stats.sql`, and complete, populated version of the database is available in `share/sqlite3-populated.sql`. (This version was created with the `sqlite3 .dump` command after running `bin/stats.py`.)

Alternatively, you can install the Faker library with

```
$ sudo apt install --yes faker
```

and run `bin/stats.py` on your own to create a populated database.

The database includes one hundred thousand users and one million game win/loss records. There are two views, `wins` and `streaks`, which can be queried as tables.

Adding Databases

Once the API for your service is working correctly, replace the single SQLite database configuration with four SQLite databases: three databases holding shards of the `games` table, and one database containing the `users` tables (and, in the next project, materialized views).

Modify the `Settings` object and database dependencies to include connections to all databases. Split the game win/loss records into shards based on the user ID so that records for each game will be spread across all three databases. Map the shard key modulo 3 to the database partition.

User IDs as Shard Keys

Since the SQLite databases are independent from each other, using `INTEGER PRIMARY KEY` or similar techniques for primary keys in sharded tables can lead to duplicate values (i.e., a particular value of `user_id` may be present in more than one database). One way to solve this problem is to use universally unique identifiers as keys.

While SQLite does not directly support UUIDs, see the accepted StackOverflow answer to Proper way to store GUID in sqlite for an example of configuring the Python `sqlite3` module to use `uuid.UUID` objects as primary keys.

Note: the call to `buffer()` is no longer required in Python 3.

Configuring Load Balancing

Use the foreman `--formation` switch to start one instance of each of the services from the previous project, one instance of Traefik, and three instances of the new service. Test that each service instance can be accessed separately.

Note: the original article Introducing Foreman shows more than one process being run using the command-line switch `-c` (for "concurrency"), but current versions use `-m` or `--formation` instead.

Configuring Traefik

Now that multiple instances of the win/loss statistics service are running, you will need to be able to load-balance requests across those instances.

Configure your services to sit behind a proxy with Traefik acting as a reverse proxy for the services from Project 2 as a load balancer for the three instances of the new service.

Note: for Tuffix 2020, you should download the `linux_amd64` version build of Traefik.

Testing

Reverse proxying is correctly configured when API calls for all three services can be accessed successfully through Traefik (e.g, on port 9999). Load balancing is correctly configured consecutive calls to the statistics service through Traefik are routed to separate instances.

You can check that load-balancing is working correctly by examining the foreman logs. Make several requests to Traefik for the statistics service, and verify that they are routed to different service instances (e.g. `stats.1`, `stats.2`, and `stats.3`).

Submission

Your submission should consist of a tarball (`.tar.gz`, `.tgz`, `.tar.Z`, `.tar.bz2`, or `.tar.xz`) file containing the following items:

1. A README file identifying the members of the team and describing how to initialize the databases and start the services.

2. The Python source code for each microservice.
3. `Procfile` definitions for the services.
4. Your `traefik.toml` configuration file
5. Initialization and population scripts for each database, and the script that you used to shard the existing entries in the stats database.
6. Any other necessary configuration files.

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

You do not need to write separate documentation for the APIs, but be sure to name functions and parameters appropriately and test your services using the automatic documentation.

Submit your tarball through Canvas before 9:45 pm PDT on the due date. Only one submission is required for a team.

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.

Reminder: do not attempt to submit projects via email. Projects must be submitted via Canvas, and instructors cannot submit projects on students' behalf. If you miss a submission deadline, contact the instructor as soon as possible. If the late work is accepted, the assignment will be re-opened for submission via Canvas.

Grading

The project itself will be evaluated on the following five-point scale, inspired by the general rubric used by Professor Michael Ekstrand at Boise State University:

Exemplary (5 points)

Code is correct and internal documentation is clearly written; organization makes it easy to review.

Basically Correct (4 points)

Results are reasonable, but the code is not easy to follow, does not contain internal documentation, or has minor mistakes.

Solid Start (3 points)

The approach is appropriate, but the work has mistakes in code or design that undermine the functionality of the result.

Serious Issues (2 points)

The work contains fundamental conceptual problems in procedure, design, or code such that it will not lead to a working system.

Did Something (1 point)

The solution began an attempt, but is either insufficiently complete to assess its quality or is on entirely the wrong track.

Did Nothing (0 points)

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

The individual contributions of team members will be confidentially evaluated by the rest of the team along the following axes:

- API implementation
- Database design and implementation
- Reverse proxy and load balancing configuration
- Discussion, research, and problem solving
- Status updates and regular contact with the rest of the team
- Willingness to help other members of team, or take on unpopular jobs

Each team member will assign scores on a scale ranging from +2 (contributed the most) to -2 (contributed the least), subject to the constraint that the sum of contributions to each factor across all team members must be 0. (For example, if you were to evaluate one team member's contribution as +2, you would also need to evaluate another team member's contribution as -2, or evaluate two other team members as -1.)