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

CPSC 449 - Web Back-End Engineering

Project 5, Spring 2022 - due May 20

Last updated Thursday April 28, 7:00 pm PDT

To this point, the services you've built have been entirely independent of each other, even going so far as to use separate databases. Even when you place the services behind a reverse proxy like Traefik so that all APIs can all be accessed through a common hostname and port (sometimes called the Gateway Routing pattern), the responsibility falls on the client to manage gameplay.

While this may be an entirely reasonable thing to do for a rich client such as a desktop or mobile app, a simpler, unified interface may be preferable for a web version of the game. In this final project you'll implement the Backends for Frontends pattern to unify the services behind a single, much simpler API, orchestrating calls behind the scenes to coordinate functionality across services.

The following are the learning goals for this project:

1. Understanding several different cloud design patterns for coordinating microservices.
2. Coordinating stateful and stateless services to accomplish a high-level task.
3. Gaining experience integrating services from several different teams.
4. Using an HTTP client library to make HTTP calls and parse JSON results.
5. Making and coordinating asynchronous calls to API calls to proceed in parallel.

Project Teams

This project must be completed in a team of three or four students. The instructor will assign teams for this project in Canvas.

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

- 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 4, with the addition of the HTTPX library for making HTTP calls.

In order to complete this project, you will need to have access to working versions of the services from each of the previous projects. Members of the team are welcome to re-use the services they developed with other teams in previous projects. If no member of your team was on a team that produced a complete, working version of a service from a previous project, you are encouraged to collaborate and share code across teams.

Note, however, that **all code for the new service** specified below must be your own original work or the original work of other members of your team, with the usual exception of sources specified in previous projects such as documentation for the various libraries.

Putting it all together

You have been building multiple separate microservices, and at this point it may seem as if your back-end is far more complicated than Wordle itself. But from the perspective of the front-end, it is possible to unify most of the functionality behind two endpoints:

`POST /game/new`	Start a new game
`POST /game/{game_id}`	Guess a word

Implement a new "Backend for Frontend" service to expose this API.

Making HTTP calls

The API endpoints in your new services should operate by making a series of HTTP calls to the services you created in the previous projects (or to services you have borrowed from other teams — see the section on *Libraries, Tools, and Code* above.)

To install HTTPX on Tuffix, use the following command:

```
$ python3 -m pip install httpx
```

To get started, see the QuickStart page.

Accessing services

Your new service should interact with the other services entirely via HTTP — it should not have access to any SQLite or Redis database.

Making changes to existing services

If you find that functionality required to implement the interaction shown below is missing from the existing services, add it and document your changes.

Sharing changes

Since, as noted above, you may be working with services borrowed or inherited from other teams, it is only polite to offer to share those changes with the original authors. Since they may require the same changes, you may coordinate with other teams to implement them together. You may **not**, however, share the code for the new service you are creating.

Game flow

POSTing a username to the `/game/new` endpoint should:

1. Find the user ID for the specified username
2. Choose a new game ID for the user to play

POSTing a new guess to the `/game/{game_id}` endpoint should:

1. Verify that the guess is a word allowed by the dictionary
2. Check that the user has guesses remaining

If both of these are true,

3. Record the guess and update the number of guesses remaining
4. Check to see if the guess is correct

If the guess is correct…

5. Record the win
6. Return the user's score

If the guess is incorrect and no guesses remain…

5. Record the loss
6. Return the user's score

If the guess is incorrect and additional guesses remain…

5. Return which letters are included in the word and which are correctly placed

Note that this flow includes points where calls to different services do not depend on each other's results. Use the [async support](#) in HTTPX and the [`asyncio.gather()`](#) method to run these tasks concurrently.

Example flow

You may find the following example of gameplay helpful in understanding the flow shown above. Requests are shown in bold, and the corresponding replies are shown indented underneath.

`POST /game/new`
`{ "username": "ProfAvery" }`

```
      200 OK
      {
        "status": "new",
        "user_id": "866e0602-23b1-4c8d-a0e9-205b0884247f",
        "game_id": 20220424
      }
```

`POST /game/20220424`
`{ "user_id": "866e0602-23b1-4c8d-a0e9-205b0884247f", "guess": "AROSE" }`

```
      200 OK
      {
        "status": "incorrect",
        "remaining": 5,
        "letters": {
          "correct": [],
          "present": ["R", "E"]
        }
      }
```

`POST /game/20220424`
`{ "user_id": "866e0602-23b1-4c8d-a0e9-205b0884247f", "guess": "XYZZY" }`

```
      400 Bad Request
      {
        "status": "invalid",
        "remaining": 5
      }
```

`POST /game/20220424`
`{ "user_id": "866e0602-23b1-4c8d-a0e9-205b0884247f", "guess": "RIVEN" }`

```
      200 OK
      {
        "status": "incorrect",
        "remaining": 4,
        "letters": {
          "correct": [],
          "present": ["R", "I", "E", "N"]
        }
      }
```

```
POST /game/new
{ "username": "ProfAvery" }

      200 OK
      {
        "status": "in-progress",
        "user_id": "866e0602-23b1-4c8d-a0e9-205b0884247f",
        "game_id": 20220424,
        "remaining": 4,
        "guesses": ["AROSE", "RIVEN"],
        "letters": {
          "correct": [],
          "present": ["R", "I", "E", "N"]
        }
      }

POST /game/20220424
{ "user_id": "866e0602-23b1-4c8d-a0e9-205b0884247f", "guess": "INTER" }

      200 OK
      {
        "remaining": 3,
        "status": "incorrect",
        "letters": {
          "correct": ["I", "N"],
          "present": ["T", "E", "R"]
        }
      }

POST /game/20220424
{ "user_id": "866e0602-23b1-4c8d-a0e9-205b0884247f", "guess": "INERT" }

      200 OK
      {
        "status": "win",
        "remaining": 2,
        "currentStreak": 1,
        "maxStreak": 3,
        "guesses":{
           "1": 0,
           "2": 0,
           "3": 1,
           "4": 2,
           "5": 3,
           "6": 1,
           "fail": 1
        },
        "winPercentage": 85,
        "gamesPlayed": 7,
        "gamesWon": 6,
        "averageGuesses": 5
      }
```

```
POST /game/20220424
{ "user_id": "866e0602-23b1-4c8d-a0e9-205b0884247f", "guess": "EXTRA" }

      204 No Content
```

Note that this example does not account for all possible error conditions, and that your service output may depend on the underlying back-end services it calls.

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 the new BFF service.
3. The modified Python source code for any of the underlying services, if they needed to be changed.
4. `Procfile` definitions for the services.
5. Any other necessary configuration files or scripts, including from previous projects (e.g. Traefik configuration, `crontab`).

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
- Data model design
- View materialization and scheduling
- 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.)