Play Network Take Home Problem Analysis

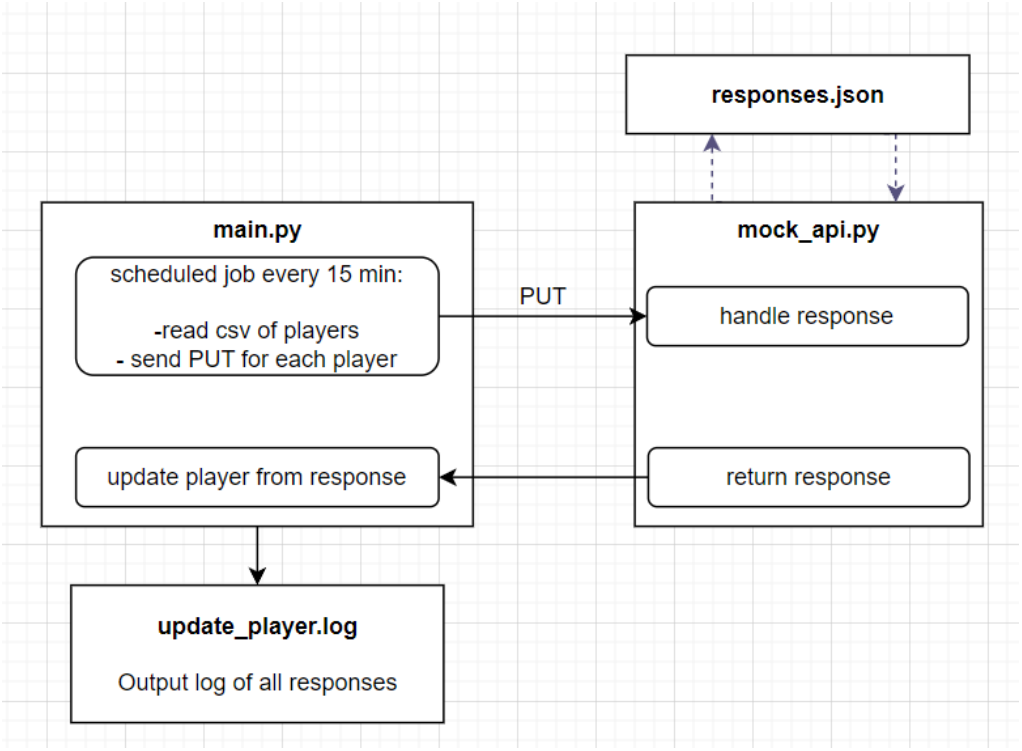
Rayan Isran

October 9, 2024

1 The problem

We have thousands of music players in the wild. Each player queries an API on a server every 15 minutes to check whether a new version is out. If so, the music player updates itself. Provided alongside this document is a tool that automates the update of a thousand such players listed in a .csv file. In this document, I explain 3 things: the structure of the code, assumptions behind technical decisions, and limitations of the tool.

The below diagram summarizes the flow of the code. It runs a scheduled task that reads the *profile* of each music player and sends a PUT request to a mock API, which is a locally running Flask server that simulates responses. Each response is read from *responses.json*. Once a player is updated, the next PUT request is sent until all players are updated.



The project directory contains the following files:

- **Documentation:**
  - `USER.md`: Documentation for the user.
  - `README.md`: Documentation for developers wishing to add to or edit the tool.
- **Python Scripts:**
  - `mock_api.py`: A local Flask server that simulates API responses.

  - `main.py`: The main script to run.
  - `tests\tests.py`: Unit tests that simulate all possible server responses.

- **Other Files:**
  - `test_players.csv`: An input file containing 1,000 generated MAC addresses. This file is regenerated when running the main script.
  - `requirements.txt`: A list of Python packages to install.
  - `keys.conf`: A file containing the client ID and authentication token sent in the PUT request header.
  - `responses.json`: A list of possible responses and status codes for requests.
  - `ci.yml`: A workflow that automatically runs the unit tests once changes are pushed to the main branch.
  - `main.bat`: An optional script that runs the server and main script, saving the end user a few keystrokes and clicks.

2 My approach & code style

Python is generally my go-to language for prototyping; it is cross-platform compatible, and to test a simple client-server system Flask is easy enough to setup.

My preferences were to keep the number of code files small, the number of lines in each file relatively small, and thirdly, for a user to understand the code easily after reading the diagram above. Some included files are optional, but were added to enable better readability and scalability. For example, a separate file for responses it should be easy to add a response to *responses.json*, so it can be referenced in more than just the *mock_api.py* file if necessary. Although this would add a layer of complexity, a schema could have been used to validate the request and response objects. Instead, simple conditions with clear messages have been used as validation checks (side note: I despite seeing "something went wrong" messages!)

The authentication token and Client ID would be better stored in environment variables on the system. However, the current approach was used for the purpose of demonstration. It is important to note that the The *keys.conf* should, at the very least, belong in a .gitignore file.

3 Limitations & Possible Improvements

To keep the code relatively simple, I used synchronous requests. In reality, it might be more effective to use asynchronous calls, especially as the number of music players will likely increase.

Currently, there is no quick and easy method to check which player updates failed, since the generated .log file is far from pretty. To enhance support and improve the developer experience, it would be beneficial and cool to create a geographical map displaying the locations of all Juke-Boxes with their status reports.

Furthermore, we could implement schema validators and more advanced error handling, such as rate-limiting the API or retrying updates for players that failed on the first attempt.

Finally, hosting the documentation on a separate site could enhance readability and accessibility for users and developers.