SoftRobot

Backend developer test

1 Purpose

The goal of this test is to provide a relevant, fun, and inspirational foundation for a technical conversation for an upcoming interview. The submitted solution together with an interview will provide the interviewer with an understanding of the applicant's coding style and skills. The solution itself should not take more than half a day to write, though it will be difficult to write a fully finished and excellent solution in such a short time span. The goal is by no means is to get a solution covering all special cases in a 100% robust way. Requiring this would be naive. At least the functions described shall be error free when used correctly, how everything else is handled is subject to the creativity and ambition of the applicant.
We will pay attention to code structure, threading, and understanding of the problem, as well as the discussion on the choices of design and data structures.

2 Description

Write a HTTP-based mini game backend server application that allows users to login, register game scores, and to retrieve a list of high scores for the levels.

Deliver a zip file containing:

- A file named "app.py" in the root folder that can be executed with Python 3.
- Additional code in a folder named "src"
- An optional readme.txt or pdf file with thoughts and considerations about the program

Note: This type of exhaustive specification is extremely rare and is only needed based on the circumstances of this test.

3 Nonfunctional requirements

- The server will be handling a lot of simultaneous requests, so make good use of the available memory and CPU power while not compromising data integrity.
- Do not use any external frameworks, except if testing is performed. For HTTP, prefer using the http.server Python.
- The server should run on the port 8080
- There is no need for persistence storage of data to disk.

- The application should be able to continuously run indefinitely without experiencing a crash or similar.

4 Functional requirements

The endpoints are described in detail below and the notation "`<value>`" indicates a call parameter value or a return value.

- If the call is successful, then the HTTP status code 200 should be returned, else the call is unsuccessful and any status code but 200 should be returned. If an appropriate status code is applicable for an unsuccessful call, then return that status code.
- Numbers, parameters, and return values in the response body are to be in decimal ASCII representation as expected, i.e. not in a binary format.
- Response data is to be returned as valid JSON objects.
- Users and levels are created "adhoc", the first time they are referenced.
- Note the case capitalisation of the request string fields as well as the JSON object property names.

4.1 Login

Given a user identifier this endpoint should return a personal session key in the form of an alphanumeric string, consisting only of digits and characters from the English alphabet, which shall be valid for use as authentication for all other functions for the next 10 minutes. The session keys should be "reasonably unique".

```
Request: GET /<userid>/login
Response: JSON object containing the sessionkey of the user
Details:
<userid>     : 31 bit unsigned integer number representing the
               user identifier
<sessionkey> : A string representing the session that should be
               valid for the next 10 minutes.

Example:
  Request:
    http://localhost:8080/4711/login
  Response body:
    {"user": "UICSNDK"}
```

4.2 Post a user's score to a level

Given a level identifier, a user's session key, and that user's score, this endpoint should add or replace the current score for the user for the given level if all constraints are met for the new score. The constraints of the score is that it must be better than the user's previous score if one exists and that it must be among the top 15 greatest scores of the given level. If no previous score exists for the user, then the new score must only be among the top 15 greatest scores for the given level. Note that this function has no limits in the number of times it can be called by the same user and level combination. The request should be processed if, and only if, the supplied session key is valid. If the score

```
Request: POST /<levelid>/score?sessionkey=<sessionkey>
Request body:
{
     "score": <score>
}
Response: (nothing)
Details:
<levelid>    : 31 bit unsigned integer number representing the
               level identifier
<sessionkey> : A session key string retrieved from the login
               function.
<score>      : 31 bit unsigned integer number representing the
               user's score

Example:
  Request:
    http://localhost:8080/2/score?sessionkey=UICSNDK
    given the post body: {"score": 1500}
```

4.3 Get a high score list for a level

Given a level identifier, this endpoint retrieves the high score list for that level. The entries of the high score list should be in descending order by the scores of the users. If two or more users have the same score, then those entries are to be ordered by the user identifiers in ascending order lexicographically. No more than 15 scores are to be saved for any one level since data is not stored persistently. A user may only appear once per high score list for any given level, and that appearance should be with that user's greatest high score. If a user hasn't submitted a score for a level, then no score is present in the high score list for that user and level combination. If no high scores are recorded for any user for a given level, then an empty list should be returned for that level.

```
Request: GET /<levelid>/highscorelist
Response: JSON object containing the scores of the users
Details:
<levelid> : 31 bit unsigned integer number representing the
level
            identifier

Example:
  Request:
    http://localhost:8080/2/highscorelist
  Response body:
    [{"user": 4711, "score": 1500},{"user": 131, "score":
1220}]
```