ZENSTACK

BE Test

1. Given a list of tuples A = (x, y), where x is an integer from 0 to 13 and y is an integer from 0 to 3, with all possible values of A without repetition and sorted by y then x in ascending order, write a pseudocode to shuffle A.

2. Create a simple application that rewrites the json data structure using Section A to Section B. Please note that Section A nodes are sorted in no particular order. The "subordinate" node must be removed when no child exists (see markcorderoi and richard)

Section A

```
[{
    "manager_name": "nssi",
    "login_name": "nishanthi"
}, {
    "manager_name": "mbarcelona",
    "login_name ": "nssi"
}, {
    "manager_name": "nishanthi",
    "login_name": "markcorderoi"
}, {
    "manager_name": "mbarcelona",
    "login_name ": "richard"
}, {
    "manager_name": "letecia",
    "login_name ": "rudy"
}]
```

Section B

```
[{
    "name": "letecia",
    "subordinate": [{
        "name": "rudy"
    }]
}, {
    "name": "mbarcelona",
    "subordinate": [{
        "name": "nssi",
        "subordinate": [{
            "name": "nishanti",
            "subordinate": [{
                "name": "markcorderoi"
            }]
        }]
    }, {
        "name": "richard"
    }]
}]
```

3. Hypothetically, we are building an app that collects accelerometer data from thin clients (sensors with limited processing capacity)
    a. Assuming that the device is sending continuously at 16000Hz for the X,Y, and Z acceleration measurements, what would your strategy be in handling and processing the data? How would you design the server infrastructure? Please enumerate the steps, software, algorithms, and services that you would use to ensure that the servers can handle the incoming data from our users. Diagrams can be really helpful for this.
    b. If the sensors were upgraded to modern mobile devices, how would you change your architecture from A?

Write an API server that meets the following requirements:

1. Login
    a. POST /api/login
    b. Inputs are username and password in JSON
    c. Case insensitively validates usernames with at least 4 letters and the letters “a”, “b”, and “c” are in the username in that order. Please see the example below
        i. abacca is validated
        ii. Cabbie is not validated since there is no c that comes after ab
        iii. Acaiberrycake is valid
    d. Password will be valid if it is equal to the reverse of the username
2. List letters
    a. GET /api/letters
    b. Seed your database with the following data (you may use an in memory data or use a third party database application. If you do use an external database, make sure that it is included in the submission by submitting a docker image)
        i. {“letter”:”A”, “value”:1, “strokes”:2, “vowel”:true},{“letter”:”B”, “value”:2, “strokes”:1, “vowel”:false}
        ii. You may represent the data in your database however you like as long as the expected API outputs are met

    c. This should return a json of all unique letters in the database sorted by value in ascending order. Given the seed data, this should return {“letters”:[“A”, “B”]}

3. Add letter
    a. POST /api/letter/add
    b. Input is a json object containing the following fields:
        i. Letter: string
        ii. Value: int - any random value
        iii. Strokes: int - any number that is not equal to value
        iv. Vowel: bool - if the letter is treated as a vowel or not
    c. Letters must be unique and not limited to the latin alphabet
        i. Hence a request with {“letter”:”A”, “value”:1, “strokes”:2, “vowel”:true} will fail
    d. Return {‘status”:0} if the letter was added to the database otherwise return {“status”:1}
4. Get letter
    a. GET /api/letter/<letter:str>
    b. Returns the details of the letter in the URL
        i. /api/letter/A will return {“letter”:”A”, “value”:1, “strokes”:2, “vowel”:true}
5. Shuffle letters
    a. GET /api/letter/shuffle
    b. Returns a string containing all the letters in the database without repetition but in a random order
    c. Plus points will be given for implementing your own shuffle function
6. Filter letters
    a. GET /api/letter/filter/<val:int>
    b. Returns a list of letters whose value field is less than or equal to val ordered by when they were added to the database
        i. On the original set, accessing /api/letter/filter/1 this should return {“letters”:[“A”]}