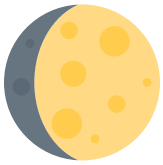

Buycoins Engineering Challenge

You're the newest member of the Buycoins Engineering team and on your second day at work, you're faced with the following challenge:

- Buycoins is building a new application in which users add their bank accounts by providing their **bank account number**, selecting a **bank name**, and writing their **name** as registered with their banks. Eg a corporate user such as STER, might input "0157148304" as their account number, select GTB and type "*Stand to End Rape Initiative*". In this scenario, the application should verify the user (ie the application already has a registered ***user*** object and sets the ***is_verified*** attribute to true). The frontend and backend work together by doing the following dance:
- The backend provides the frontend with a GraphQL mutation that accepts three arguments: ***user_account_number***, ***user_bank_code***, ***user_account_name***. The backend will then do some validation against an external API service to validate that indeed this **user_account_name** belongs to the given **account_number**. (Ignore case validation when validating or storing this)
- How does the backend do this validation? Well, the backend service will make a call to the Paystack API. Paystack has an account number resolution API that takes in an **account_number** and a **bank_code** and returns the **account_name**. If the names match, the user is marked as verified in the DB and this result is returned to the front end.

- However, we understand that this is Nigeria and data inputs are odd. A user might accidentally type their name with one letter off or the bank might have stored their name with one letter off eg "*Stand to End Rape initaitive*". Or "Paystack" might be written as "Paystac". In scenarios like this, we still want to verify this user. As such, you've decided to compute the Levenshtein Distance between the user inputed **account_name** and the **account_name** provided by the API. You've decided to still verify users who have an input that is within a Levenshtein Distance of 2 from what is provided by Paystack. (You don't need to implement your own LD).

- Now, Buycoins has decided to further create a GraphQL query that takes in a **bank_code** and **account_number** and returns an **account_name**. We assume users are more reliable than banks' stored data. As such, our API returns the **account_name** inputed by the user if available, otherwise it returns the name that Paystack would otherwise have provided (all in sentence case). Eg if the user inputed "Timi Ajiboye" but the Paystack API returned "Tim Ajiboye", we should verify the user and our new GraphQL query should return the user's name as "Timi Ajiboye".
- Please **build the backend component** of this task. Your backend project should provide the GraphQL mutation and query referenced above. It should also solve the various components expected as described by the problem space stated above.
- Please write some tests.
- In 100 words or less, provide an answer to this in your readme: What's a good reason why the pure Levenshtein Distance algorithm might be a more effective solution than the broader Damerau–Levenshtein Distance algorithm in this specific scenario.
- If you make any major assumptions about any part of your solution, please state them in the readme

Tools you might need:

- Free unverified Paystack account (to get test API keys to use with the Paystack documentation)
- Paystack documentation
- Postman

Estimated time to complete this challenge:

120 minutes

Link to Submission

https://forms.gle/6R8CTFMCbaNnY8Qt8

Deadline

Monday, September 27, 2021 10:00 pm WAT