Digital Wallet:

We are going to build Golang backend, that maintains user wallet and performs money transfer between users inside the platform. You will be creating two microservices user and transactions service.,

User-service:

1. CreateUser API / HTTP POST:
This request should create an entry in users table (columns: user_id, email, created_at) inside user-service and then push an event ("user-created") into kafka with payload = { user_id: <id>, email: '<user email>', created_at : <user_created_at> }, This payload should be consumed by the transaction-service and it should also have an user table, that store these columns (user_id, balance, created_at) and makes an entry from the payload received.

2. Balance API / HTTP GET:
When this request is called, you should fetch the user's latest balance. Request Body: { email: " "} | Response Body: { email: "" , balance: "" }

You should not store user balance data in user microservice and it should be fetched only from transaction-service by make a service-to-service call using nats sync subcriber (https://github.com/nats-io/nats.go#basic-usage).

Request flow => HTTP Request -> User service -> [nats sync req] -> transaction service -> [nats sync res] -> User service -> Http response

Transactions-service:

1. Add money API / HTTP POST:
This API credits money to a user account and you should update user's current balance in the user table. And every balance change should be recorded in another table called transactions. And API should respond with updated user balance.

Request params: { user_id, amount }
Response params: { updated_balance: }

2. Transfer money:
This API transfers funds from one user to another, we should also make 2 entires in transactions table, 1) for debiting funds from user A and 2) for crediting funds into user B

Req params: { from_user_id: , to_user_id, amount_to_transfer

Evaluation criteria:
- We need a complete working version as described and postgres db should be used.
- We pay attention to robustness and the careful handling of money transactions without any bugs.
- we pay a close attention at how you handle concurrency risks, when you update user balance to ensure idempotency when you perform internal transfer, as part of testing we will send thousands of transfer request calls, to make sure user wallet is not credited/debited twice.

Tips to save time:
- No need to implement any login/signup features
- You can refer any materials in internet, any libraries.
- Code doesn't have to be pretty, we don't check code refractor, design patterns and no test cases is required.

Deliverables:
You should provide final code in zip format, and the readme file should clearly explain, setup procedure in our local environment. Please use docker wherever possible.