Test task (Elixir)

Test task for Elixir developers. Candidate should write a simple banking OTP application in Elixir language.

General acceptance criteria

- All code is in git repo (candidate can use his/her own github account).
- OTP application is a standard mix project.
- Application name is :ex_banking (main Elixir module is ExBanking).
- Application interface is just set of public functions of ExBanking module (no API endpoint, no REST / SOAP API, no TCP / UDP sockets, no any external network interface).
- Application should not use any database / disc storage. All needed data should be stored only in application memory.
- Candidate can use any Elixir or Erlang library he/she wants to (but app can be written in pure Elixir / Erlang / OTP).
- Solution will be tested using our auto-tests for this task. So, please follow specifications accurately.
- Public functions of ExBanking module described in this document is the only one thing tested by our auto-tests. If anything else needs to be called for normal application functioning then probably tests will fail.
- Code accuracy also matters. Readable, safe, refactorable code is a plus.

Money amounts

- Money amount of any currency should not be negative.
- Application should provide 2 decimal precision of money amount for any currency.
- Amount of money incoming to the system should be equal to amount of money inside the system + amount of withdraws (money should not appear or disappear accidentally).
- User and currency type is any string. Case sensitive. New currencies / users can be added dynamically in runtime. In the application, there should be a special public function (described below) for creating users. Currencies should be created automatically (if needed).

API reference

Requirements for public functions provided by ExBanking module. Any function should return success result or error result. Success result is different for each function, error result is generic
@spec create_user(user :: String.t) :: :ok | banking_error

- Function creates new user in the system
- New user has zero balance of any currency

@spec deposit(user :: String.t, amount :: number, currency :: String.t) :: {:ok, new_balance :: number} | banking_error

- Increases user's balance in given currency by amount value
- Returns new_balance of the user in given format

@spec withdraw(user :: String.t, amount :: number, currency :: String.t) :: {:ok, new_balance :: number} | banking_error

- Decreases user's balance in given currency by amount value
- Returns new_balance of the user in given format

@spec get_balance(user :: String.t, currency :: String.t) :: {:ok, balance :: number} | banking_error

- Returns balance of the user in given format

@spec send(from_user :: String.t, to_user :: String.t, amount :: number, currency :: String.t) :: {:ok, from_user_balance :: number, to_user_balance :: number} | banking_error

- Decreases from_user's balance in given currency by amount value
- Increases to_user's balance in given currency by amount value
- Returns balance of from_user and to_user in given format

Performance

- In every single moment of time the system should handle 10 or less operations for every individual user (user is a string passed as the first argument to API functions). If there is any new operation for this user and he/she still has 10 operations in pending state - new operation for this user should immediately return too_many_requests_to_user error until number of requests for this user decreases < 10
- The system should be able to handle requests for different users in the same moment of time
- Requests for user A should not affect to performance of requests to user B (maybe except send function when both A and B users are involved in the request)

Notes

- If you have any questions about the test task, send an email, we will answer
- Completed Elixir test tasks can be sent to **hr@pure-agency.co**