Take Home Test

At Coins we make extensive use of Django REST framework (DRF) when developing in Python and domain-driven design (DDD) with Go kit when developing in Go. For the task we ask to implement generic Wallet service with a RESTful API. For extra context and perspective assume it will be used as a core service in your fintech startup.

User Stories

Here are user stories that should be covered:

- I want to be able to send a payment from one account to another (same currency)
- I want to be able to see all payments
- I want to be able to see available accounts

Assumptions and requirements

- Only payments within the same currency are supported (no exchanges)
- There are no users in the system (no auth)
- Balance can't go below zero
- More than one instance of the application can be launched

It's okay to make other assumptions as long as you are explicit about them. Feel free to ask further questions/clarifications once you have those.

Example entries

Account entity:

```
id: bob123
balance: 100.00
currency: USD

id: alice456
balance: 0.01
currency: USD
```

Payment entity:

```
account: bob123
amount: 100.00
to_account: alice456
direction: outgoing

account: alice456
amount: 100.00
from_account: bob123
direction: incoming
```

Evaluation

Our goal with this test project is to see your **best/most idiomatic code** in Go or Python. Attention will be put on but not limited by the listed aspects:

- Documentation:
  - Code documentation
  - API docs (ideally in markdown format, e.g., docs/api.md)
  - Human oriented README explaining your project's purpose, how to set it up, run tests, code linting, start contributing
  - Doc strings which should explain "real world" problem you're solving, attributes, params documenting
- Architecture and Design
  - Simplicity
  - Expected design patterns, decoupling, complexity isolation
  - DDD (Domain Driven Design) if you choose Go
  - proper usage of Django and DRF components if you choose Django
- Code:
  - Hosted on any public/private git storage (github/gitlab/bitbucket)
  - Human-oriented with recomended conventions (Django, Effective Go, naming in Go, Go for Industrial Programming, Practical Go), descriptive names of variables, classes, functions, apps, packages, repository itself
  - no "dead code" inside a repository (e.g., empty modules, unused settings, excessive blank lines, etc)
  - Idiomatic (data model and structures, best practices)
  - Proven to work (covered by tests)
  - Implementation using **go-kit**(if you write in Go) or **Django REST framework**(if you write in Python)
- Infrastructure:
  - Deployability
  - External dependencies (if any) choice justification
  - PostgreSQL as storage engine (be aware of DB transactions, locks, race conditions)
- Bonus points:
  - Deployment with Docker/Docker-compose

Timeframe

There is no strict deadline but we would like to get ETA from you to finish the project upon reading the document and understanding it.