Code Challenge: Authorizer

You are tasked with implementing an application that authorizes a transaction for a specific account following a set of predefined rules.

Please make sure you read all the instructions below, and feel free to ask for clarifications if needed.

IMPORTANT: Please anonymize/remove all personal information present on your challenge, special attention to:

- Source files: code, tests, namespaces, packaging, comments and file names;
- Version control author information;
- Automatic comments your development environment may add;
- Documentation provided, such as README.MD files.

Packaging

Your README file should contain a description on relevant code design choices, along with instructions on how to build and run your application.

Building and running the application must be possible under Unix or Mac operating systems. Dockerized builds are welcome.

You may use open source libraries you find suitable, but please refrain as much as possible from adding frameworks and unnecessary boilerplate code.

Sample usage

Your program is going to be provided `json` lines as input in the `stdin`, and should provide a `json` line output for each one — imagine this as a stream of events arriving at the authorizer.

```
$ cat operations
{"account": {"active-card": true, "available-limit": 100}}
{"transaction": {"merchant": "Burger King", "amount": 20, "time":
"2019-02-13T10:00:00.000Z"}}
{"transaction": {"merchant": "Habbib's", "amount": 90, "time":
"2019-02-13T11:00:00.000Z"}}
```

```
$ authorize < operations

{"account": {"active-card": true, "available-limit": 100}, "violations": []}
{"account": {"active-card": true, "available-limit": 80}, "violations": []}
{"account": {"active-card": true, "available-limit": 80}, "violations":
["insufficient-limit"]}
```

State

The program should not rely on any external database. Internal state should be handled by an explicit in-memory structure. State is to be reset at application start.

Operations

The program handles two kinds of operations, deciding on which one according to the line that is being processed:

1. Account creation
2. Transaction authorization

For the sake of simplicity, you can assume:

- All monetary values are positive integers using a currency without cents
- Transactions will arrive in chronological order

---

1. Account creation

Input

Creates the account with `available-limit` and `active-card` set. For simplicity sake, we will assume the application will deal with just one account.

Output

The created account's current state + all business logic violations.

Business rules

- Once created, the account should not be updated or recreated: `account-already-initialized`.

Examples

```
input
    {"account": {"active-card": true, "available-limit": 100}}
    ...
    {"account": {"active-card": true, "available-limit": 350}}


output
    {"account": {"active-card": true, "available-limit": 100}, "violations":
[]}
    ...
    {"account": {"active-card": true, "available-limit": 100}, "violations":
["account-already-initialized" ]}
```

---

2. Transaction authorization

Input

Tries to authorize a transaction for a particular `merchant`, `amount` and `time` given the account's state and last authorized transactions.

Output

The account's current state + any business logic violations.

Business rules

You should implement the following rules, keeping in mind new rules will appear in the future:

- No transaction should be accepted without a properly initialized account: `account-not-initialized`
- No transaction should be accepted when the card is not active: `card-not-active`
- The transaction amount should not exceed available limit: `insufficient-limit`
- There should not be more than 3 transactions on a 2 minute interval: `high-frequency-small-interval`

- There should not be more than 1 similar transactions (same amount and merchant) in a 2 minutes interval: `doubled-transaction`

Examples

Given there is an account with `active-card: true` and `available-limit: 100`:

```
input
    {"transaction": {"merchant": "Burger King", "amount": 20, "time":
"2019-02-13T10:00:00.000Z"}}

output
    {"account": {"active-card": true, "available-limit": 80}, "violations": []}
```

Given there is an account with `active-card: true`, `available-limit: 80` and 3 transaction occurred in the last 2 minutes:

```
input
    {"transaction": {"merchant": "Habbib's", "amount": 90, "time":
"2019-02-13T10:01:00.000Z"}}

output
    {"account": {"active-card": true, "available-limit": 80}, "violations":
["insufficient-limit", "high-frequency-small-interval"]}
```

Error handling

- Please assume input parsing errors will not happen. We will not evaluate your submission against input that breaks the contract.
- Violations of the business rules are not considered to be errors as they are expected to happen and should be listed in the outputs's `violations` field as described on the `output` schema in the examples. That means the program execution should continue normally after any violation.

Our expectations

We at Nubank value simple, elegant, and working code. This exercise should reflect your understanding of it.

Your solution is expected to be production quality, maintainable and extensible. Hence, we will look for:

- Immutability;
- Quality unit and integration tests;
- Documentation where needed;
- Instructions to run the code.

General notes

- This challenge may be extended by you and a Nubank engineer on a different step of the process;
- You should submit your solution source code to us as a compressed file containing the code and possible documentation. Please make sure not to include unnecessary files such as compiled binaries, libraries, etc;
- Do not upload your solution to public repositories in GitHub, BitBucket, etc;
- The project should be implemented as a stream application rather than a Rest API.