Code Challenge: Authorizer

You are tasked with implementing an application that authorizes transactions 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 remove all the personal information from the files of the challenge before submitting the solution. Pay special attention to the following:

- Source files like code, tests, namespaces, packaging, comments, and file names;
- Automatic comments your development environment may add to solution files;
- Code documentation such as annotations, metadata, and README.MD files;
- Version control configuration and author information.

If you plan to use git as the version control system, execute the following in the repository root to export the solution anonymized:

```
git archive --format=zip --output=./authorizer.zip HEAD
```

Packaging

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

It must be possible to build and run the application must be possible under Unix or Mac operating systems. Dockerized builds are welcome.

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

Sample usage of the Authorizer

How the program should work?

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

How the program should be executed?

Given a file called `operations` that contains many lines describing operations in `json format`:

```
$ 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"}}
{"transaction": {"merchant": "McDonald's", "amount": 30, "time": "2019-02-
13T12:00:00.000Z"}}
```

The application should be able to receive the file content through `stdin`, and for each processed operation return an output equivalent to the business rules:

```
$ 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"]}
{"account": {"active-card": true, "available-limit": 50}, "violations": []}
```

Authorizer Operations

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

1. Account creation
2. Transaction authorization in the account

For the sake of simplicity, you can assume the following:

- All monetary values are positive integers using a currency without cents;
- The transactions will arrive at the Authorizer in chronological order.

---

1. Account creation

Input

Creates the account with the attributes `available-limit` and `active-card`. For simplicity's sake, we will assume that the Authorizer will deal with just one account.

Output

The created account's current state with all business logic violations. If in the operation processing does not happen any violation, the field `violations` should return an empty vector `[]`.

Business rules

- Once created, the account should not be updated or recreated. If the application receives another operation of account creation, it should return the following violation: `account-already-initialized`.

Examples

Creating an account successfully

Creating an account with an inactive card (`active-card: false`) and an available limit of 750 (`available-limit: 750`):

```
# Input
    {"account": {"active-card": false, "available-limit": 750}}

# Output
    {"account": {"active-card": false, "available-limit": 750}, "violations":
[]}
```

Creating an account that violates the Authorizer logic

Given there is an account with an active card (`active-card: true`) and the available limit of 175 (`available-limit: 175`), tries to create another account:

```
# Input
    {"account": {"active-card": true, "available-limit": 175}}
    {"account": {"active-card": true, "available-limit": 350}}

# Output
    {"account": {"active-card": true, "available-limit": 175}, "violations":
[]}
    {"account": {"active-card": true, "available-limit": 175}, "violations":
["account-already-initialized"]}
```

---

2. Transaction authorization

Input

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

Output

The account's current state with any business logic violations. If in the operation processing does not happen any violation, the field `violations` should return an empty vector `[]`.

Business rules

You should implement the following rules, keeping in mind that **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 the 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

Processing a transaction successfully

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

```
# Input
    {"account": {"activeCard": true, "availableLimit": 100}}
    {"transaction": {"merchant": "Burger King", "amount": 20, "time": "2019-02-13T11:00:00.000Z"}}

# Output
    {"account": {"activeCard": true, "availableLimit": 100}, "violations": []}
    {"account": {"activeCard": true, "availableLimit": 80}, "violations": []}
```

Processing a transaction for a not initialized account

When a transaction operation is processed but there is not a previously created account, the Authorizer should return the `account-not-initialized` violation:

```
# Input
    {"transaction": {"merchant": "Uber Eats", "amount": 25, "time": "2020-12-01T11:07:00.000Z"}}
    {"account": {"active-card": true, "available-limit": 225}}
    {"transaction": {"merchant": "Uber Eats", "amount": 25, "time": "2020-12-01T11:07:00.000Z"}}
# Output
    {"account": {}, "violations": ["account-not-initialized"]}
    {"account": {"active-card": true, "available-limit": 225}, "violations": []}
    {"account": {"active-card": true, "available-limit": 200}, "violations": []}
```

Processing a transaction that violates the Authorizer logic

Given there is an account with an active card (`active-card: true`), the available limit of 100 (`available-limit: 100`), and 3 transactions that occurred successfully in the last 2 minutes, the Authorizer should reject the operation and return the `high-frequency-small-interval` violation:

```
# Input
    {"account": {"activeCard": true, "availableLimit": 100}}
    {"transaction": {"merchant": "Burger King", "amount": 20, "time": "2019-02-13T11:00:00.000Z"}}
    {"transaction": {"merchant": "Habbib's", "amount": 20, "time": "2019-02-13T11:00:01.000Z"}}
    {"transaction": {"merchant": "McDonald's", "amount": 20, "time": "2019-02-13T11:01:01.000Z"}}
    {"transaction": {"merchant": "Subway", "amount": 20, "time": "2019-02-13T11:01:31.000Z"}}

# Output
    {"account": {"activeCard": true, "availableLimit": 100}, "violations": []}
    {"account": {"activeCard": true, "availableLimit": 80}, "violations": []}
    {"account": {"activeCard": true, "availableLimit": 60}, "violations": []}
    {"account": {"activeCard": true, "availableLimit": 40}, "violations": []}
    {"account": {"activeCard": true, "availableLimit": 40}, "violations": ["high-frequency-small-interval"]}
```

State

The program **should not** rely on any external database, and the application's internal state should be handled by an explicit in-memory structure. The application state needs to be reset at the application start.

Authorizer operations that had violations should not be saved in the application's internal state. For example, the following should not trigger the `high-frequency-small-interval` violation:

```
# Input
    {"account": {"activeCard": true, "availableLimit": 1000}}
    {"transaction": {"merchant": "Vivara", "amount": 1250, "time": "2019-02-
13T11:00:00.000Z"}}
    {"transaction": {"merchant": "Samsung", "amount": 2500, "time": "2019-02-
13T11:00:01.000Z"}}
    {"transaction": {"merchant": "Nike", "amount": 800, "time": "2019-02-
13T11:01:01.000Z"}}
    {"transaction": {"merchant": "Uber", "amount": 80, "time": "2019-02-
13T11:01:31.000Z"}}

# Output
    {"account": {"activeCard": true,"availableLimit": 1000}, "violations": []}
    {"account": {"activeCard": true,"availableLimit": 1000}, "violations":
["insufficient-limit"]}
    {"account": {"activeCard": true,"availableLimit": 1000}, "violations":
["insufficient-limit"]}
    {"account": {"activeCard": true,"availableLimit": 200}, "violations": []}
    {"account": {"activeCard": true,"availableLimit": 120}, "violations": []}
```

Error handling

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

Our expectations

We at Nubank value **simple, elegant, and working code**. This exercise should reflect your understanding of it. It is expected that:

- Your solution to be **production quality**;
- The application to be **maintainable**, being easy to maintain, and well structured;
- The application to be **extensible**, being easy to implement new business rules.

Hence, we will look for:

- Thoughtful use of immutability in the solution;
- Quality unit and integration tests;
- Documentation where it is needed;
- Instructions about how 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 (zip) containing the code and all 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 Authorizer application should receive the operations data through `stdin` and return the processing result through `stdout`, rather than through a REST API communication.