Code Challenge: Authorizer

You are tasked with implementing an application which 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.

:warning: **IMPORTANT:** Please remove all 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 containing:

- Discussing regarding the technical and archtectural decisions;
- Reasoning about the frameworks used (if any framework/library was used);
- Instructions on how to compile and run the project;
- Additional notes you consider important to be evaluated.

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

Sample usage of the Authorizer

How the program should work

Your program is going to receive `json` lines as input through `stdin` and should provide a `json` line output for each of the inputs through `stdout`. You can imagine this as a stream of events being processed by the authorizer.

How the program should be executed

Given a file called `operations` that contains several 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 according 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 to execute based on the line that is being processed:

1. Account creation
2. Transaction authorization for 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 no violations happen during operation processing, 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 account creation operation, 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 no violations happen during operation processing, 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 be no more than 3 transactions within a 2 minutes interval: `high-frequency-small-interval`
- There should be no more than 1 similar transaction (same `amount` and `merchant`) within a 2 minutes interval: `doubled-transaction`

Examples

Processing a transaction successfully

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

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

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

Processing a transaction which violates the `account-not-initialized` logic

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 which violates `card-not-active` logic

Given an account with an inactive card (`active-card: false`), any transaction submit to the Authorizer should be rejected and return the `card-not-active` violation:

```
# Input
    {"account": {"active-card": false, "available-limit": 100}}
    {"transaction": {"merchant": "Burger King", "amount": 20, "time": "2019-02-
13T11:00:00.000Z"}}
    {"transaction": {"merchant": "Habbib's", "amount": 15, "time": "2019-02-
13T11:15:00.000Z"}}

# Output
    {"account": {"active-card": false, "available-limit": 100}, "violations": []}
    {"account": {"active-card": false, "available-limit": 100}, "violations": ["card-
not-active"]}
    {"account": {"active-card": false, "available-limit": 100}, "violations": ["card-
not-active"]}
```

Processing a transaction which violates `insufficient-limit` logic:

Given an account with an active card (`active-card: true`), the available limit of 1000 (`available-limit: 1000`), any transaction above the limit of `1000` should be rejected and return the `insufficient-limit` violation:

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

# Output
    {"account": {"active-card": true,"available-limit": 1000}, "violations": []}
    {"account": {"active-card": true,"available-limit": 1000}, "violations":
["insufficient-limit"]}
    {"account": {"active-card": true,"available-limit": 1000}, "violations":
["insufficient-limit"]}
    {"account": {"active-card": true,"available-limit": 200}, "violations": []}
```

Processing a transaction which violates the `high-frequency-small-interval` logic

Given 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 new operation and return the `high-frequency-small-interval` violation:

```
# Input
    {"account": {"active-card": true, "available-limit": 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"}}
    {"transaction": {"merchant": "Burger King", "amount": 10, "time": "2019-02-13T12:00:00.000Z"}}

# Output
    {"account": {"active-card": true, "available-limit": 100}, "violations": []}
    {"account": {"active-card": true, "available-limit": 80}, "violations": []}
    {"account": {"active-card": true, "available-limit": 60}, "violations": []}
    {"account": {"active-card": true, "available-limit": 40}, "violations": []}
    {"account": {"active-card": true, "available-limit": 40}, "violations": ["high-frequency-small-interval"]}
    {"account": {"active-card": true, "available-limit": 30}, "violations": []}
```

Processing a transaction which violates the `doubled-transaction` logic

Given an account with an active card (`active-card: true`), the available limit of 100 (`available-limit: 100`) and some transactions that occurred successfully in the last 2 minutes. The authorizer should reject the new transaction if it shares the same amount and merchant as any of previously accepted transactions and return the `doubled-transaction` violation:

```
# Input
    {"account": {"active-card": true, "available-limit": 100}}
    {"transaction": {"merchant": "Burger King", "amount": 20, "time": "2019-02-13T11:00:00.000Z"}}
    {"transaction": {"merchant": "McDonald's", "amount": 10, "time": "2019-02-13T11:00:01.000Z"}}
    {"transaction": {"merchant": "Burger King", "amount": 20, "time": "2019-02-13T11:00:02.000Z"}}
    {"transaction": {"merchant": "Burger King", "amount": 15, "time": "2019-02-13T11:00:03.000Z"}}

# Output
    {"account": {"active-card": true, "available-limit": 100}, "violations": []}
    {"account": {"active-card": true, "available-limit": 80}, "violations": []}
    {"account": {"active-card": true, "available-limit": 70}, "violations": []}
    {"account": {"active-card": true, "available-limit": 70}, "violations": ["doubled-transaction"]}
    {"account": {"active-card": true, "available-limit": 55}, "violations": []}
```

Processing transactions that violate multiple logics

Given a transaction that violates more than one logic, the authorizer must return all violations that occurred:

```
# Input
    {"account": {"active-card": true, "available-limit": 100}}
    {"transaction": {"merchant": "McDonald's", "amount": 10, "time": "2019-02-13T11:00:01.000Z"}}
    {"transaction": {"merchant": "Burger King", "amount": 20, "time": "2019-02-
13T11:00:02.000Z"}}
    {"transaction": {"merchant": "Burger King", "amount": 5, "time": "2019-02-
13T11:00:07.000Z"}}
    {"transaction": {"merchant": "Burger King", "amount": 5, "time": "2019-02-
13T11:00:08.000Z"}}
    {"transaction": {"merchant": "Burger King", "amount": 150, "time": "2019-02-
13T11:00:18.000Z"}}
    {"transaction": {"merchant": "Burger King", "amount": 190, "time": "2019-02-
13T11:00:22.000Z"}}
    {"transaction": {"merchant": "Burger King", "amount": 15, "time": "2019-02-
13T12:00:27.000Z"}}

# Output
    {"account":{"active-card":true,"available-limit":100},"violations":[]}
    {"account":{"active-card":true,"available-limit":90},"violations":[]}
    {"account":{"active-card":true,"available-limit":70},"violations":[]}
    {"account":{"active-card":true,"available-limit":65},"violations":[]}
    {"account":{"active-card":true,"available-limit":65},"violations":["high-
frequency-small-interval","double-transaction"]}
    {"account":{"active-card":true,"available-limit":65},"violations":["insufficient-
limit","high-frequency-small-interval"]}
    {"account":{"active-card":true,"available-limit":65},"violations":["insufficient-
limit","high-frequency-small-interval"]}
    {"account":{"active-card":true,"available-limit":50},"violations":[]}
```

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 start of the application.

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": {"active-card": true, "available-limit": 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": {"active-card": true,"available-limit": 1000}, "violations": []}
    {"account": {"active-card": true,"available-limit": 1000}, "violations":
["insufficient-limit"]}
    {"account": {"active-card": true,"available-limit": 1000}, "violations":
```

```
["insufficient-limit"]}
    {"account": {"active-card": true,"available-limit": 200}, "violations": []}
    {"account": {"active-card": true,"available-limit": 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 formatted incorrectly, or which breaks the contract.

- Violations of the business rules **are not considered errors** as they are expected to happen and should be listed in the `violations` field of the output as described on the `output` schema in the examples. This means that the program execution should continue normally after any kind of violation.

Our expectations

We at Nubank value the following criteria:

- **Simplicity**: the solution is expected to be a small project and easy to understand;
- **Elegance**: the solution is expected to be easy to maintain, have a clear seperation of concerns and well structured code organization;
- **Operational**: the solution is expected to solve the problem, cover possible corner cases and be extensible for future design decisions.

We will look for:

- The use of referential transparency when applicable;
- Quality unit and integration tests;
- Documentation where it is needed;
- Instructions on how to run the code.

Lastly, but not least expected:

- You may use open source libraries you find suitable to support you in solving the challenge, e.g. json parsers; Please refrain as much as possible from adding frameworks and unnecessary boilerplate code.
- The challenge expects a **standalone** command line application; please refrain from adding unecessary infrastructure and/or dependencies. You are expected to be able to identify which tools are required to solve the problem without adding extra layers of complexity.

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 documentation. Please make sure not to include unnecessary files such as compiled binaries, libraries, etc;
- Please do not upload your solution to public repositories such as GitHub, BitBucket, etc;
- The Authorizer application should receive the operations data through `stdin` and return the processing result through `stdout`, rather than through for example a REST API.