Purple

Purple é um projeto de autorização de cadastro de conta e transações

Installation

Seguir o tutorial de instalação do docker

Seguir o tutorial de instalação do docker-compose

Baixar o source e extrair

```
unzip purple.zip
cd purple
```

```
ls

Dockerfile  entrypoint.sh  HELP.md  mvnw  mvnw.cmd  operations  pom.xml
purple.iml  README.md  src
```

OBS : Dockerfile com multistage, utilizar versões recentes do Docker

```
cat Dockerfile



FROM maven:3.6.2-jdk-11 as maven

COPY . purple/

WORKDIR purple

#RUN mvn dependency:go-offline package -B

RUN mvn clean install

FROM openjdk:11-jre-slim

COPY --from=maven /purple/target/purple-1.0.jar
/usr/local/purple/purple.jar

COPY entrypoint.sh /usr/local/purple/entrypoint.sh

WORKDIR /usr/local/purple

ENTRYPOINT ["bash","entrypoint.sh"]
```

Após extrair,vamos buildar a imagem

```
docker build --network=host -t purple .
```

Usage

```
cd purple
cat operations

{ "account": { "activeCard": true, "availableLimit": 1000 } }
{ "account": { "activeCard": true, "availableLimit": 200 } }
{ "transaction": { "merchant": "Habbib's", "amount": 30, "time": "2019-02-
13T11:01:00.000Z" } }
{ "transaction": { "merchant": "Avon", "amount": 60, "time": "2019-02-
13T11:01:00.000Z" } }
{ "transaction": { "merchant": "Ragazzo", "amount": 10, "time": "2019-02-
13T11:01:00.000Z" } }
{ "transaction": { "merchant": "Carrefour", "amount": 11, "time": "2019-02-
13T11:04:00.000Z" } }
{ "transaction": { "merchant": "Havan", "amount": 11, "time": "2019-02-
13T11:04:00.000Z" } }
{ "transaction": { "merchant": "Riachuelo", "amount": 11, "time": "2019-02-
13T11:04:00.000Z" } }
{ "transaction": { "merchant": "KFC", "amount": 11, "time": "2019-02-
13T11:04:00.000Z" } }
{ "transaction": { "merchant": "Brooxklin", "amount": 11, "time": "2019-02-
13T11:04:00.000Z" } }
{ "transaction": { "merchant": "Outlet", "amount": 11, "time": "2019-02-
13T11:01:00.000Z" } }
{ "transaction": { "merchant": "Hashicorp", "amount": 11, "time": "2019-02-
13T11:01:00.000Z" } }
{ "transaction": { "merchant": "Forever 21", "amount": 11, "time": "2019-
02-13T11:01:00.000Z" } }
{ "transaction": { "merchant": "Tayan", "amount": 11, "time": "2019-02-
13T11:08:00.000Z" } }
{ "transaction": { "merchant": "Baked Potato", "amount": 11, "time": "2019-
02-13T11:09:00.000Z" } }
```

Para modificar as transações realizadas, basta sobrescrever o arquivo operations

```
cd purple
echo { "account": { "activeCard": true, "availableLimit": 1000 } } >
operations
cat operations

{ "account": { "activeCard": true, "availableLimit": 1000 } }
```

Para executar a aplicação, basta chamar o docker run

```
cd purple
cat operations | docker run -i purple -c 'cat'
```

Decisões Arquiteturais

- Spring Boot: Utilizado para Injeção de dependência e EventListener;
- Java: Afinidade técnica;
- Padrão de projeto Strategy: Encapsular regras de validação, sobre o mesmo contrato, mantendo a extensibilidade e a manutenabilidade;
- Padrão de projeto Cadeia de Responsabilidade: Estrutura condicional hierarquica, no melhor caso e no caso médio, evita algumas validações e processamentos desnecessários;
- Eventos: Com o EventListener, consigo processar mais rapidamenta o meu input, pois as chamadas de publicações de eventos, são assíncronas;
- Docker: Facilitar a construção e a execução do projeto;
- Implementação própria do Consumidor de Runnables: A aplicação lê linha a linha o input em seguida cria um runnable e adiciona na fila, toda esta operação ocorre de maneira assíncronas, no entanto, preciso garantir o FIFO, então instanciei uma BlockingQueue e controlo o consumo desta fila;
- Testes unitários: O foco dos testes ficou no Core da aplicação, evitando testes de get e set em modelos, só para aumentar a cobertura de código;
- Criei classes de serviço e de persistência, todas baseadas em interfaces, para garantir a manutenabilidade, para trocar a implementação, basta criar uma nova classe que implementa a interface e especificar na injeção de dependência;

Implementações futuras

- Métodos de busca em coleções, a atual implementação no pior caso, pode percorrer totalmente as coleções, ou seja, a complexidade deste algoritmo é da ordem de $O(n^2)$
  As alternativas para resolver esse problema, é trocar o algoritmo de busca, utilizando uma árvore binária ou algo semelhante ao QuickSort.
- O input de dados na fila, é feito de maneira assíncrona, pelos meus producers de eventos, o consumo da fila está assíncrono, mas não paralelizado, tomei essa decisão para manter a ordem de execução
  A alternativa para melhorar essa implementação, seria criar um modelo de divisão de producers e consumers. Exemplo: Tenho 10 entradas, e 5 threads no meu pool, cada thread, seria responsável por duas entradas e cada consumer, seria responsável por ler essas entradas. Eles iriam compartilhar a mesma fila, fazendo com que eu mantesse a ordem de execução. **Obs: Manter a ordem de execução dessa forma, não garante 100%, pois dependendo das validações dessa transação, ela pode demorar mais do que outras, mas é um trade-off para melhorar o desempenho.**

Bibliotecas e Frameworks

- Jackson: Fazer parse do input para o meu modelo;
- JUnit: Para fazer os testes unitários;
- Mockito: Para mockar as classes dependentes ao teste;

- Maven: Instruções de construção e dependências do projeto

Ambiente de Desenvolvimento

- Kubuntu (18.04)
- Intellij IDE