

JavaZeroZahar: StockExchange

Membrii echipei:

Murariu Marius

Munteanu Mircea-Georgian

Munteanu Daniel

Vasile Ioana

Costean Ionuț

Link repository: <https://github.com/MirceaTT8/StockExchange>

1. Descrierea proiectului

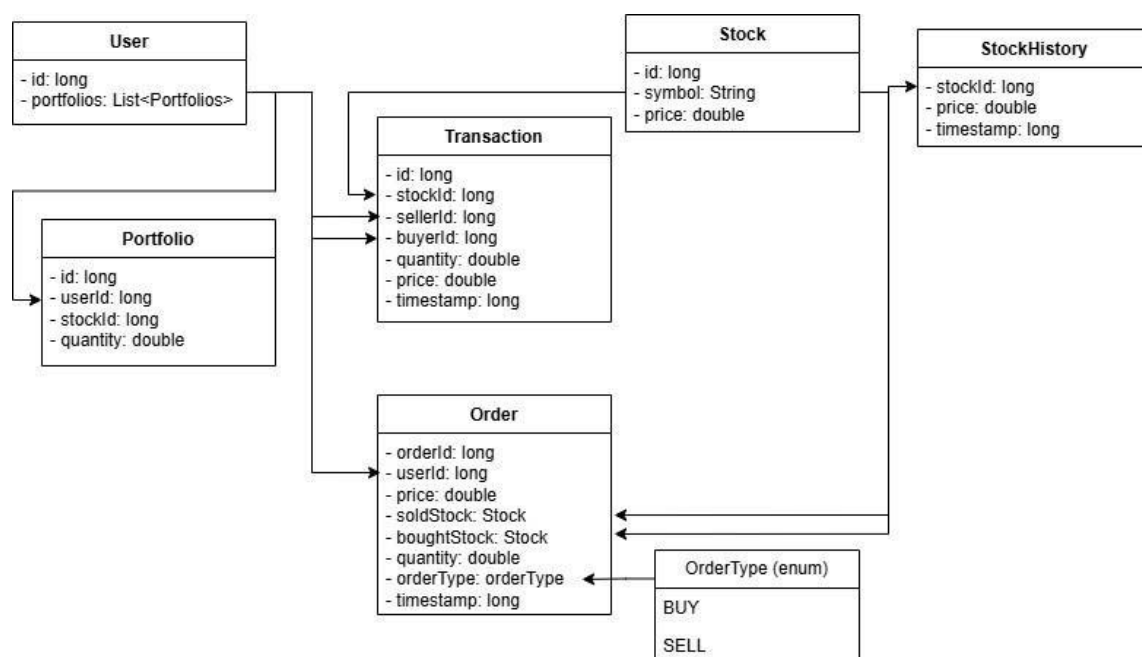
Acest proiect reprezintă simularea unei burse de acțiuni. Structura este de tip server-clienți, având următoarele roluri:

- Serverul are ca sarcini:
 - procesarea comenzilor (de cumpărare și vânzare): este posibilă crearea, modificarea și anularea unui ordin de tranzacționare pe piață;
 - utilizatorul poate plasa ordine limită care vor fi executate la prețul stabilit de utilizator în ordinea de tranzacționare sau unul mai avantajos, în funcție de stadiul pieței
 - gestionarea acțiunilor de tip CRUD (Create Read Update Delete) asupra anumitor obiecte ce vor fi stocate într-o bază de date;
 - verificarea condițiilor necesare funcționării corecte din punct de vedere concurent (potențiale probleme explicate la punctul 4), dar și din alte puncte de vedere (de exemplu, verificarea ca un cumpărător să aibă suficient de multe fonduri pentru a lansa o cerere de cumpărare cu valoarea respectivă).

Testarea serverului concurent s-a realizat folosind frameworkul JUnit5 prin plasarea paralelă a mai multor ordine de tranzacționare care sunt preluate de server, folosind serviciul dedicat.

- Clientul reprezintă platforma prin care utilizatorii pot interacționa cu Server-ul prin trimiterea de request-uri (folosind un API HTTP pentru Java), cu ajutorul unui framework pentru frontend (de exemplu Vue.js) - implementat în stadiile ulterioare ale proiectului

2. Entități principale



În această diagramă sunt reprezentate clasele de obiecte care vor fi introduse într-o baza de date și dependentele dintre ele (de exemplu, un portofoliu trebuie să conțină ca atribut valoarea id-ului user-ului care îl detine).

3. Posibile probleme de concurență

Printre cazurile care ar putea duce la conflicte de concurență am identificat următoarele:

- Probleme de tipul producer-consumer la adăugarea simultană a mai multor ordine de tranzacționare pentru o acțiune;

```
public class OrderPlacer { 3 usages  ± MurariuMarius

    private final StockService stockService; 3 usages

    private final Map<String, OrderPlacementStrategy> orderPlacementStrategies; 5 usages

    private final ConcurrentHashMap<Long, Lock> userLocks = new ConcurrentHashMap<>(); 1 usage

    public OrderPlacer() { no usages  ± MurariuMarius
        this.stockService = SingletonFactory.getInstance(StockService.class);

        this.orderPlacementStrategies = new HashMap<>();
        orderPlacementStrategies.put("create", SingletonFactory.getInstance(CreateOrderPlacementStrategy.class));
        orderPlacementStrategies.put("update", SingletonFactory.getInstance(UpdateOrderPlacementStrategy.class));
        orderPlacementStrategies.put("delete", SingletonFactory.getInstance(DeleteOrderPlacementStrategy.class));
    }
}
```

În clasa dedicată plasării de comenzi folosim ca atribut o variabilă de tip `ConcurrentHashMap` pentru a reține id-urile utilizatorilor și starea de blocare a fiecăruia. Prin stare de blocare ne referim la posibilitatea ca un user să poată fi blocat printr-un lock de tip `ReentrantLock`. Astfel atunci când un user plasează o comandă, aceasta trebuie procesată până la capăt pentru ca același user să poată plasa o comandă nouă.

```

public Order placeOrder(Order order, String orderPlacementStrategy) { 1 usage  ⬆ MurariuMarius

    Lock lock = userLocks.computeIfAbsent(order.getUserId(), _ -> new ReentrantLock(fair: true));

    lock.lock();
    try {
        return orderPlacementStrategies.get(orderPlacementStrategy).placeOrder(order);
    } finally {
        lock.unlock();
    }
}

```

- b) Verificarea disponibilității fondurilor pentru vânzare/cumpărare de stocuri în cazul trimerii simultane de către același utilizator a mai multor comenzi de modificare a unui ordin;
- c) Asigurarea actualizării concurente a fondurilor disponibile în fiecare portofoliu la momentul plasării unui ordin, astfel încât satisfacerea ordinelor viitoare plasate din același portofoliu să fie evaluată în funcție de valoarea reală a fondului;

```

public class OrderMatcher {
    public Order matchOrder(Order order) { 1 usage  ⬆ Mura

        orderRepository.lockOrder(order.getOrderid());

        List<Long> matchedOrderIds = new ArrayList<>();

```

```

        PriorityQueue<Order> matchingQueue = matchingOrders.getValue();

        matchedOrderIds = matchingOrders.getValue().stream().map(Order::getOrderId).toList();
        matchedOrderIds.forEach(orderRepository::lockOrder);

        while (!matchingQueue.isEmpty()) {
            Order matchingOrder = matchingQueue.poll();

            if (order.getOrderType().equals(OrderType.BUY) && matchingOrder.getPrice() <= order.getPrice() ||
                order.getOrderType().equals(OrderType.SELL) && matchingOrder.getPrice() >= order.getPrice()) {

                double matchedQuantity = Math.min(order.getQuantity(), order.getOrderType().equals(OrderType.SELL) ?
                    matchingOrder.getQuantity() / matchingOrder.getSoldStock().getPrice() :
                    matchingOrder.getQuantity() / matchingOrder.getBoughtStock().getPrice());

                order.setQuantity(order.getQuantity() - matchedQuantity);
                matchingOrder.setQuantity(
                    matchingOrder.getQuantity() - (currencyConverter.convert(order.getPrice(), matchingOrder.getPrice(), matchedQuantity)));

                if (matchingOrder.getQuantity() == 0) {
                    orderRepository.remove(matchingOrder);
                }

                transactionService.createTransaction(order, matchingOrder, matchedQuantity);

                if (order.getQuantity() == 0) {
                    orderRepository.remove(order);
                    break;
                }
            }
        }
    }
}

```

Problemele b) și c) sunt rezolvate prin blocarea fiecărei perechi de comenzi care alcatuiesc o tranzacție. Metoda „matchOrder” caută o comandă de cumpărare/vânzare care să fie potrivită pentru comanda de tip vânzare/cumpărare pe care o primește ca parametru. Primul lucru pe care îl face metoda este să pună un lock peste comanda dată ca parametru, astfel încât să nu se modifice în timpul în care se caută o comandă corespondentă. Lock-ul este realizat prin metoda „lockOrder”, care este folosită și pe fiecare comandă înregistrată până în prezent care ar putea fi compatibilă cu obiectul de tip „Order” din parametru. Aceasta se datorează nevoii de a evita schimbarea

sau stergerea unei comenzi din lista chiar in timpul procesului de selectie. Selectia in sine are loc in cadrul unui „try” statement, definit dupa blocarea comenzii parametru si se termina cu un „finally” statement, unde atat comanda parametru, cat si potentialele comenzi pentru potrivire sunt deblocate pentru a putea fi folosite in alte metode/clase.

```
} finally {  
    orderRepository.unlockOrder(order.getOrderId());  
    matchedOrderIds.forEach(orderRepository::unlockOrder);  
}
```

- d) Menținerea coerenței bazei de date în cazul încercărilor simultane de a citi / scrie informații. Astfel, este garantat faptul că firele de execuție operează cu aceeași stare a bazei de date.

```
public class OrderRepositoryImpl implements OrderRepository {  ⚡ MurariuMarius +1  
  
    private final Map<Long, Order> orderStore = new ConcurrentHashMap<>(); 7 usages  
    private final AtomicLong idCounter = new AtomicLong( initialValue: 1); 2 usages  
  
    private final ConcurrentHashMap<Long, ReentrantLock> orderLocks = new ConcurrentHashMap<>();  
  
    @Override  ⚡ MirceaTT +1  
    public Order save(Order order) {  
        if (order.getOrderId() == null) {  
            order.setOrderId(idCounter.incrementAndGet());  
        }  
        orderStore.put(order.getOrderId(), order);  
        return order;  
    }  
}
```

```
@Override 4 usages  ⚡ MurariuMarius  
public void reset() {  
    orderStore.clear();  
    this.idCounter.set(1);  
    orderLocks.clear();  
}
```

Aici nu am folosit doar un ConcurrentHashMap pentru sincronizarea comenzilor, ci si un counter de tip AtomicLong pe care il folosim pentru a determina id-ul pe care il va avea o comanda atunci cand e creata. Nu am folosit o variabila doar de tip Long, deoarece 2 sau mai multe comenzi plasate simultan ar fi riscat sa aibe acelasi id, ceea ce nu ne dorim din moment ce sunt comenzi diferite care ar trebui sa aibe un id unic.

Acest counter este util si daca dorim sa resetam baza de date a comenzilor, avand posibilitatea de a se intoarce la valoarea initiala intr-un mod thread-safe.