

# 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

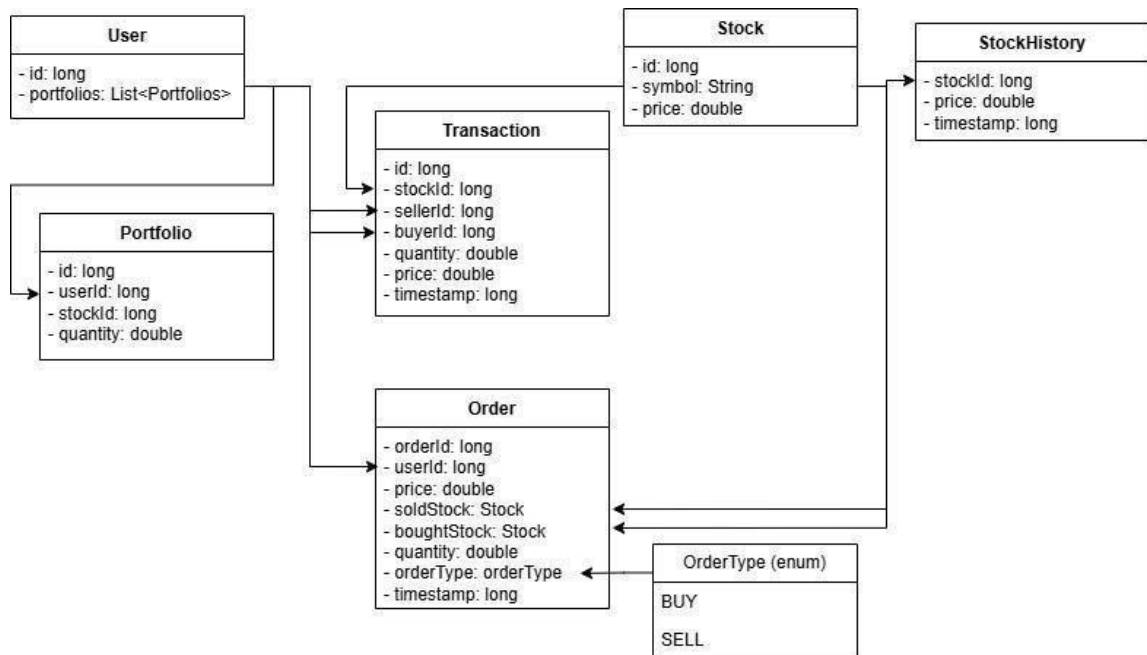
Acest proiect reprezintă simularea unei burse de acțiuni. Structura este de tip server-clienți, având următoarele roluri:

- Serverul are ca sarcini:
  - o procesarea comenzilor (de cumpărare și vânzare): este posibilă crearea, modificarea și anularea unui ordin de tranzacționare pe piață;
  - o utilizatorul poate plasa ordine limită care vor fi executate la prețul stabilit de utilizator în ordinul de tranzacționare sau unul mai avantajos, în funcție de stadiul pieței
  - o gestionarea acțiunilor de tip CRUD (Create Read Update Delete) asupra anumitor obiecte ce vor fi stocate într-o baza de date;
  - o verificarea condițiilor necesare funcționării corecte din punct de vedere concurrent (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 concurrent 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 requesturi (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 dependențele 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 sa poată fi blocat printr-un lock de tip ReentrantLock. Astfel atunci când un user plasează o comanda, aceasta trebuie procesata pana la capăt pentru ca același user sa poată plasa o comanda noua.

```
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 trimiterii 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;

Problemele b) si c) sunt rezolvate prin blocarea fiecărei perechi de comenzi care alcătuiesc o tranzacție. Metoda „matchOrder” caută o comanda de cumpărare/vânzare care sa fie potrivita pentru comanda de tip vânzare/cumpărare pe care o primește ca parametru. Primul lucru pe care îl face metoda este sa pună un lock peste comanda data ca parametru, astfel încât sa nu se modifice in timpul in care se caută o comanda corespondenta.

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

        orderRepository.lockOrder(order.getOrderId());

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

Lock-ul este realizat prin metoda „lockOrder”, care este folosita si pe fiecare comanda înregistrată până în prezent care ar putea fi compatibila cu obiectul de tip „Order” din parametru. Aceasta se datoreaza nevoii de a evita schimbarea sau stergerea unei comenzi din lista chiar in timpul procesului de selectie. Selectia in sine are loc inauntrul 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.

```

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;
        }
    }
}

```

```

} 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 {
    private final Map<Long, Order> orderStore = new ConcurrentHashMap<>();
    private final AtomicLong idCounter = new AtomicLong(1);
    private final ConcurrentHashMap<Long, ReentrantLock> orderLocks = new ConcurrentHashMap<>();

    @Override
    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.

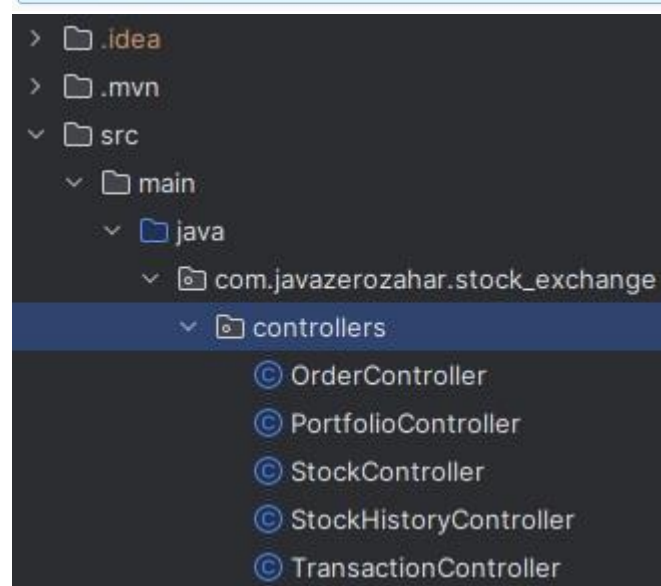
## 4. Specificația OpenAPI

Url pentru specificație: <http://localhost:8080/swagger-ui/index.html>

Servers

<http://localhost:8080> - Generated server url

stock-controller		^
GET	/stocks	▼
POST	/stocks	▼
PATCH	/stocks	▼
GET	/stocks/{stockId}	▼
order-controller		^
GET	/orders	▼
POST	/orders	▼
DELETE	/orders	▼
PATCH	/orders	▼
GET	/orders/{orderId}	▼
GET	/orders/user/{userId}	▼
transaction-controller		^
GET	/transactions	▼
GET	/transactions/{transactionId}	▼
stock-history-controller		^
GET	/stock-history/{stockId}	▼
portfolio-controller		^
GET	/portfolios	▼



Interfata de programare de tip OpenAPI/Swagger usureaza utilizarea endpointurilor API-ului definit de catre noi prin controllerele implementate cu Java Spring.

Pentru a adauga o comanda (de vanzare/cumparare) trebuie sa folosim un request de tip POST cu un RequestBody care sa contina datele noii comenzi intr-un format json, la fel ca in imaginea urmatoare:

The screenshot shows a REST client interface with the following details:

- Method:** POST
- URL:** /orders
- Parameters:** No parameters
- Request body:** required, application/json
- JSON Body:**

```
{  "orderId": 1,  "userId": 2,  "price": 5,  "soldStockId": 1,  "boughtStockId": 2,  "quantity": 20,  "orderType": "BUY",  "timestamp": 25}
```
- Buttons:** Execute, Clear

In urma apasarii butonului „Execute”, server-ul primeste request-ul introdus, pe baza caruia va trimite un response:

The screenshot shows the server response details in the REST client interface:

- Request URL:** http://localhost:8080/orders
- Server response:**
  - Code:** 200
  - Details:** Response headers
    - connection: keep-alive
    - content-length: 0
    - date: Tue, 26 Nov 2024 16:46:30 GMT
    - keep-alive: timeout=60
    - vary: Origin, Access-Control-Request-Method, Access-Control-Request-Headers
- Responses:**
  - Code:** 200
  - Description:** OK

Prin interfata ultimelor 2 imagini am apelat metoda urmatoare din clasa OrderController:

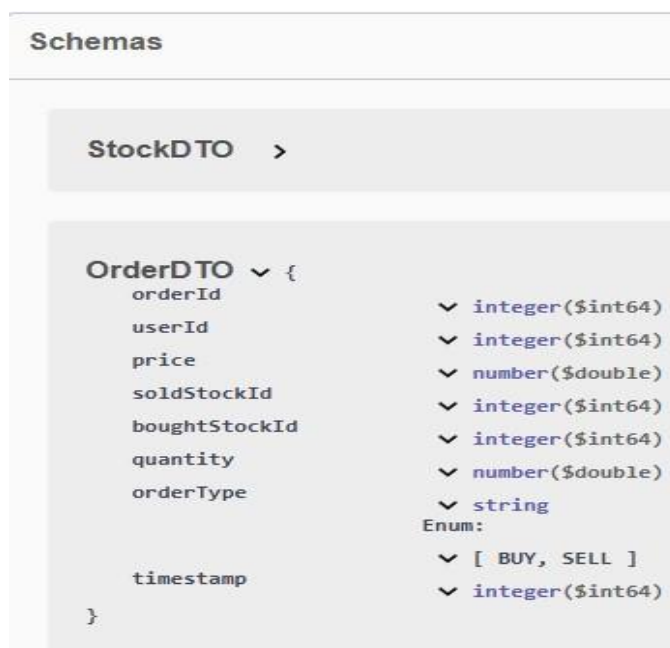
```
@PostMapping  Dani Munteanu
public void createOrder(@RequestBody OrderDTO orderDTO) { orderService.placeOrder(orderDTO, orderStrategy: "create"); }
```



## 5. Integrarea serverului concurrent

Funcționalitatea descrisă anterior – care presupune plasarea, potrivirea ordinelor și realizarea tranzacțiilor, respectiv operațiunile conexe acestora – a fost integrată într-o aplicație Spring Boot, păstrând arhitectura existentă și extinzând serviciile deja expuse.

Astfel, acestea sunt accesibile printr-un API dedicat, gestionat de frameworkul Spring prin controlere REST dedicate, iar interacțiunea se realizează prin obiecte de transfer de date (DTO), după cum se observă și în documentația Swagger menționată anterior. Utilizarea DTO-urilor are ca scop decuplarea nivelului de prezentare al aplicației de modalitatea de stocare a informațiilor în baza de date.



De asemenea, se renunță la modalitatea personalizată de a gestiona instanțele obiectelor de către clasa SingletonFactory, apelând în acest scop la mecanismul Spring de gestiune a beans-urilor și injectarea automată a dependențelor.

În ceea ce privește asigurarea funcționării corecte a aplicației în context concurrent, se apelează și în acest caz la mecanismele puse la dispoziție de Spring. Pentru a garanta consistența și coerența bazei de date, am asigurat atomicitatea operațiilor prin utilizarea adnotărilor specifice (@Transactional, @Lock etc.) în segmente critice de cod. Un exemplu, în acest sens, este blocarea accesului la rândurile asociate ordinelor și portofoliilor utilizatorilor implicați într-un proces de potrivire a unui ordin, și anume plasatorul și potențialele potriviri, conform strategiei de matching implementate.

În plus, pentru a asigura tratarea cererilor de plasare a cererilor de tranzacționare în ordinea primirii acestora, am integrat un sistem de comunicații prin mesaje (RabbitMQ) care gestionează aceste evenimente printr-un sistem de cozi. Astfel, am redus nevoia de



a implementa la nivel de cod mecanisme care prevină erori de concurență prin introducerea unui set de producători și consumatori RabbitMQ între componentele OrderPlacer și OrderMatcher, respectiv OrderMatcher și TransactionService. În scopul testării, am implementat și o modalitate de a urmări numărul mesajelor aflate în curs de procesare (MessageTracker) pentru a asigura finalizarea tranzacțiilor înaintea verificării valorilor așteptate, procesarea RabbitMQ realizându-se asincron.