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

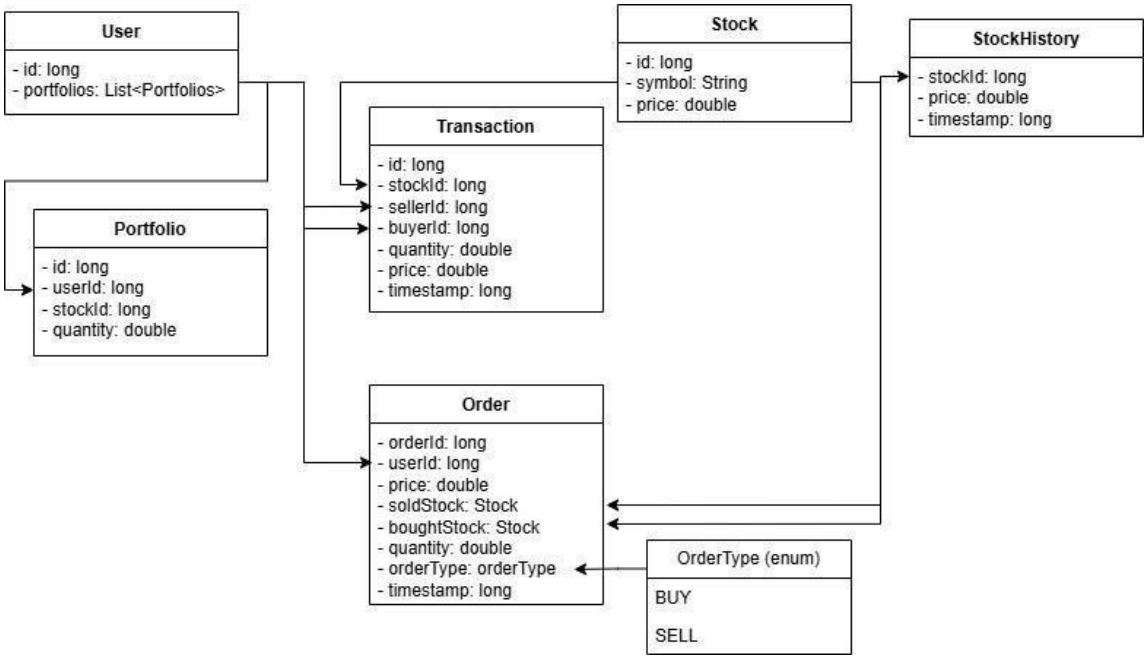
Acest proiect reprezintă simularea unei burse de acțiuni. Structura este de tip server-clienți, având următoarele roluri:

- Serverul are ca sarcini:
  - procesarea comenzilor (de cumpărare si 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 ordinul 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 baza de date;
  - verificarea condițiilor necesare funcționarii 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 si dependentele dintre ele (de exemplu, un portofoliu trebuie sa conțină ca atribut valoarea id-ului user-ului care il detine).

3. Posibile probleme de concurență

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

a) 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));
    }
```

In clasa dedicata plasarii de comenzi folosim ca atribut o variabila de tip ConcurrentHashMap pentru a retine id-urile utilizatorilor si starea de blocare a fiecaruia. Prin stare de blocare ne referim la posibilitatea ca un user sa poata fi blocat printr-un lock de tip ReentrantLock. Astfel atunci cand un user plaseaza o comanda, aceasta trebuie procesata pana la capat pentru ca acelasi user sa poata plasa o comanda noua.

```java
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;

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

        orderRepository.lockOrder(order.getOrderId());

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

```java
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) si c) sunt rezolvate prin blocarea fiecarei perechi de comenzi care alcatuiesc o tranzactie. Metoda „matchOrder” cauta o comanda de cumparare/vanzare care sa fie potrivita pentru comanda de tip vanzare/cumparare pe care o primeste ca parametru. Primul lucru pe care il face metoda este sa puna un lock peste comanda data ca parametru, astfel incat sa nu se modifice in timpul in care se cauta o comanda corespondenta.
Lock-ul este realizat prin metoda „lockOrder”, care este folosita si pe fiecare comanda inregistrata pana in 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.

```
} 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;
    }
```

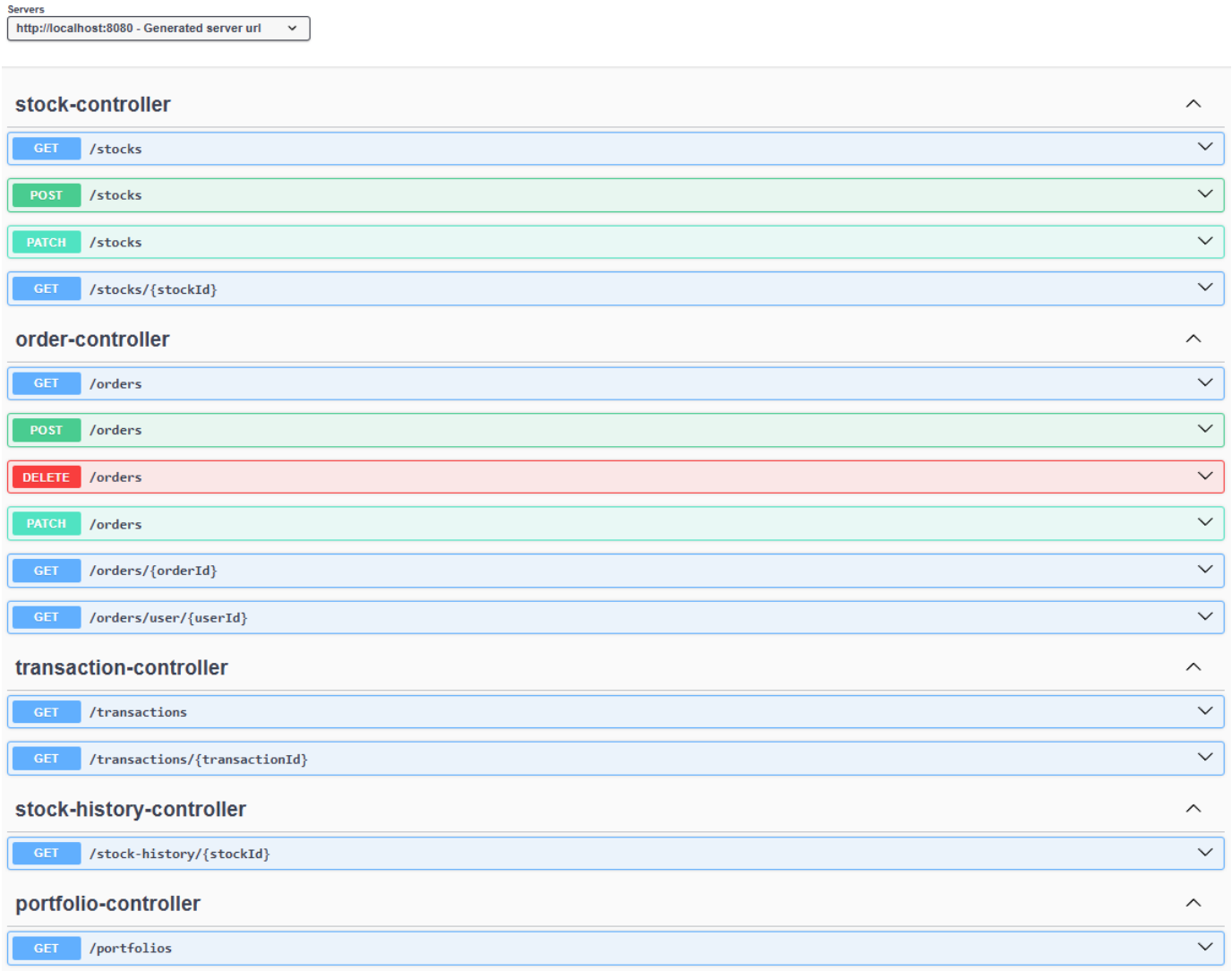
```
@Override  4 usages  MurariuMarius
public void reset() {
    orderStore.clear();
    this.idCounter.set(1);
    orderLocks.clear();
}
```

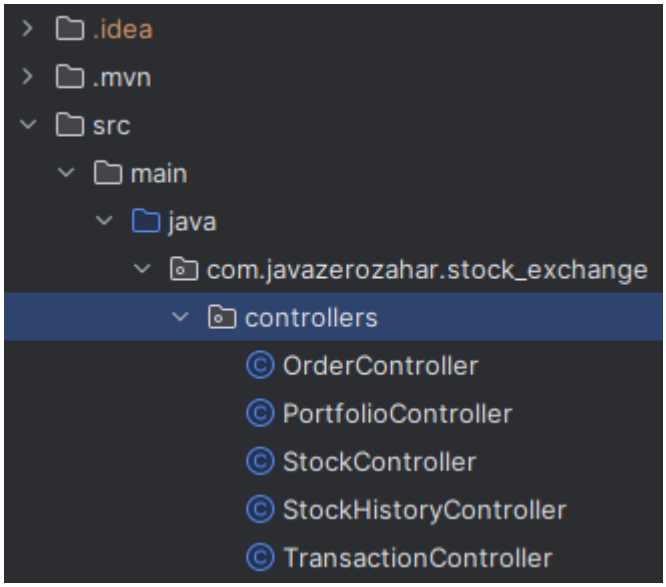
Aici nu am folosit doar un ConcurrentHashMap pentru sincronizarea comenzilor, ci si un counter de tip AtomicLong pe care il folosim pentru a determina id-ul pe care il va avea o comanda atunci cand e creata. Nu am folosit o variabila doar de tip Long, deoarece 2 sau mai multe comenzi plasate simultan 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.

4. Specificația OpenAPI

Url pentru specificatie: http://localhost:8080/swagger-ui/index.html

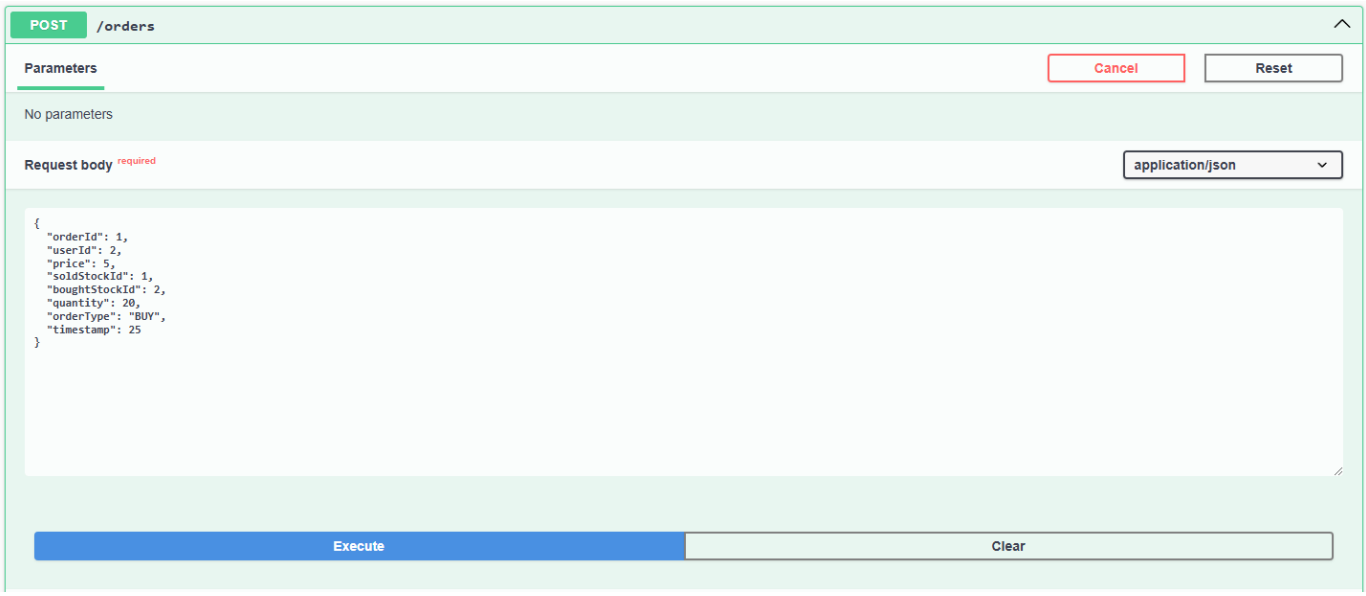




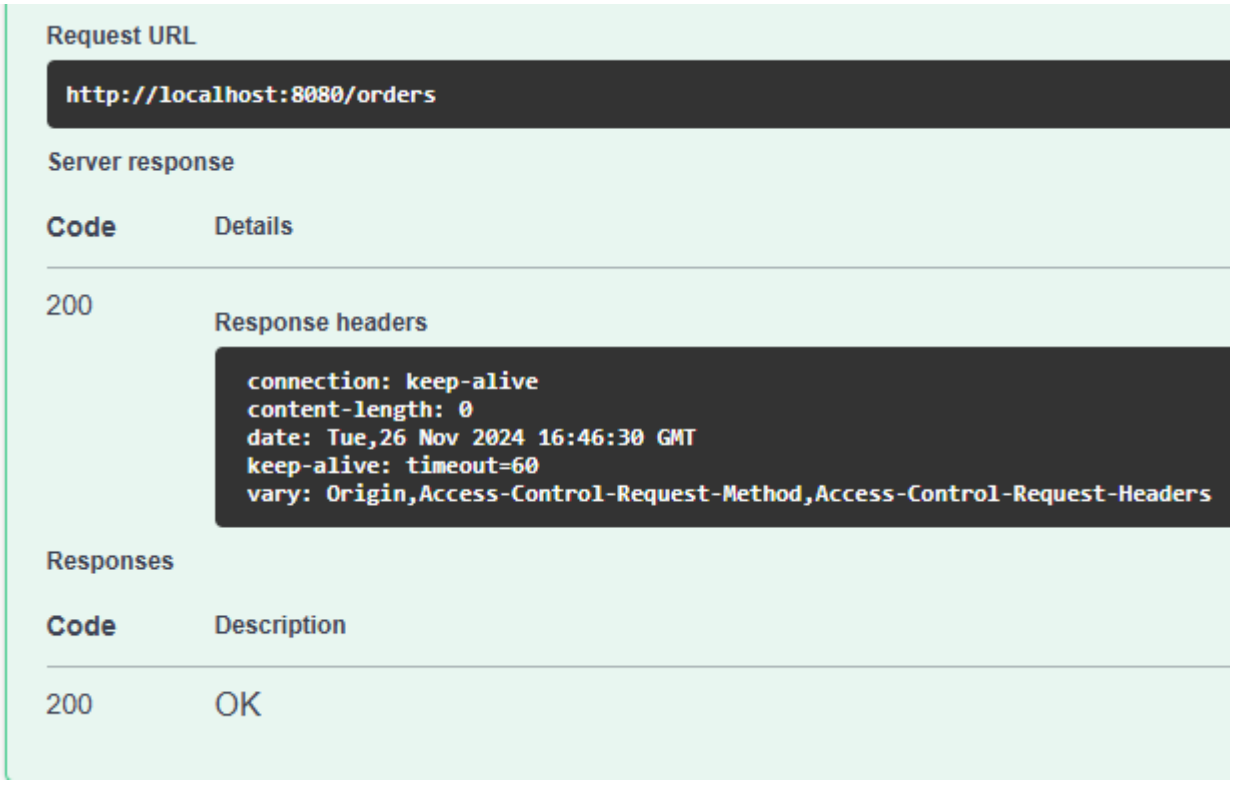
Interfata de programare de tip OpenAPI/Swagger usureaza utilizarea metodelor API-ului definit de catre noi prin controller-ele implementate cu Java Spring.

Un exemplu potrivit il reprezinta OrderController:

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:



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



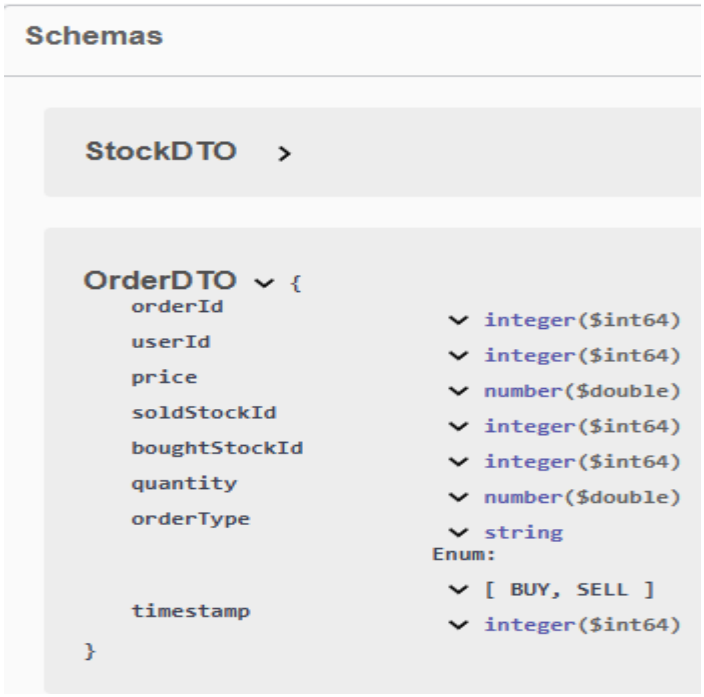
Prin interfata ultimelor 2 imagini am apelat metoda urmatoare din clasa OrderController:

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

5. Legatura cu componenta de server concurent

Controller-ele folosesc metodele unei clase de servicii corespunzatoare cu tipul de entitate. In exemplul clasei OrderService, utilizam un obiect de tip OrderRepository (pentru a putea accesa baza de date a comenzilor deja existente), dar si un obiect de tip OrderPlacer (care prin metoda „placeOrder” efectueaza o actiune asupra repository-ului, in functie de strategia Create/Update/Delete aleasa).

Este folosit totodata si un OrderConverter pentru a obtine un obiect de tip DTO (Data Transfer Object) dintr-un obiect de tip Order. Tipul de clasa DTO este de regula folosit pentru serializarea obiectelor transmise ca parametru in request-uri sau cele primite ca raspunsuri. Obiectele de tip DTO contin doar valori de atribute, asa cum se poate observa si din reprezentarea structurii lor in interfata Swagger mentionata anterior.




```
@Service  3 usages  MurariuMarius +1
@AllArgsConstructor
public class OrderService {

    private final OrderPlacer orderPlacer;
    private final OrderRepository orderRepository;
    private final OrderConverter orderConverter;
```

```
@Service  2 usages  MurariuMarius +1 *
@AllArgsConstructor
@Log4j2
@Transactional
public class OrderPlacer {

    private final OrderPlacerProducer orderPlacerProducer;
    private final OrderPlacementStrategyFactory orderPlacementStrategyFactory;
```

Clasa OrderPlacer foloseste, la randul ei, un obiect de clasa OrderPlacerProducer:

```java
@Service  2 usages  MurariuMarius +1
@RequiredArgsConstructor
@Log4j2
@Transactional
public class OrderPlacerProducer {

    @Value("order-queue")
    private String queueName;

    private final OrderConverter orderConverter;
    private final RabbitTemplate rabbitTemplate;
    private final MessageTracker messageTracker;
```

Aceasta clasa este responsabila pentru adaugarea comenzilor intr-o coada de asteptare, astfel incat sa evitam potentialele probleme de concurenta mentionate in sectiunea 3. Folosim RabbitMQ pentru un RabbitTemplate, cu scopul de a simplifica trimiterea mesajelor. Pe langa asta, utilizam si un MessageTracker, obiect care asigura thread-safety-ul server-ului prin utilizarea de ConcurrentHashMap si CompletableFuture.

```java
@Component  10 usages  MurariuMarius
public class MessageTracker {
    private final Map<String, Integer> pendingCounts = new ConcurrentHashMap<>();  3 usages
    private final Map<String, CompletableFuture<Void>> completionFutures = new ConcurrentHashMap<>();

    public void increment(String queueName) {  2 usages  MurariuMarius
        pendingCounts.merge(queueName,  value: 1, Integer::sum);

        completionFutures.put(queueName, new CompletableFuture<>());
    }

    public void decrement(String queueName) {  2 usages  MurariuMarius
        pendingCounts.computeIfPresent(queueName, (_, v) -> v > 1 ? v - 1 : null);

        if (pendingCounts.get(queueName) == null) {
            completionFutures.remove(queueName).complete( value: null);
        }
    }

    public void waitUntilQueueEmpty(String queueName) {  20 usages  MurariuMarius
        CompletableFuture<Void> future = completionFutures.get(queueName);
        if (future != null) {
            future.join();
        }
    }
}
```