JavaZeroZahar: StockExchange

Membrii echipei:

Murariu Marius

Munteanu Mircea-Georgian

Munteanu Daniel

Vasile Ioana

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

1. Descrierea proiectului

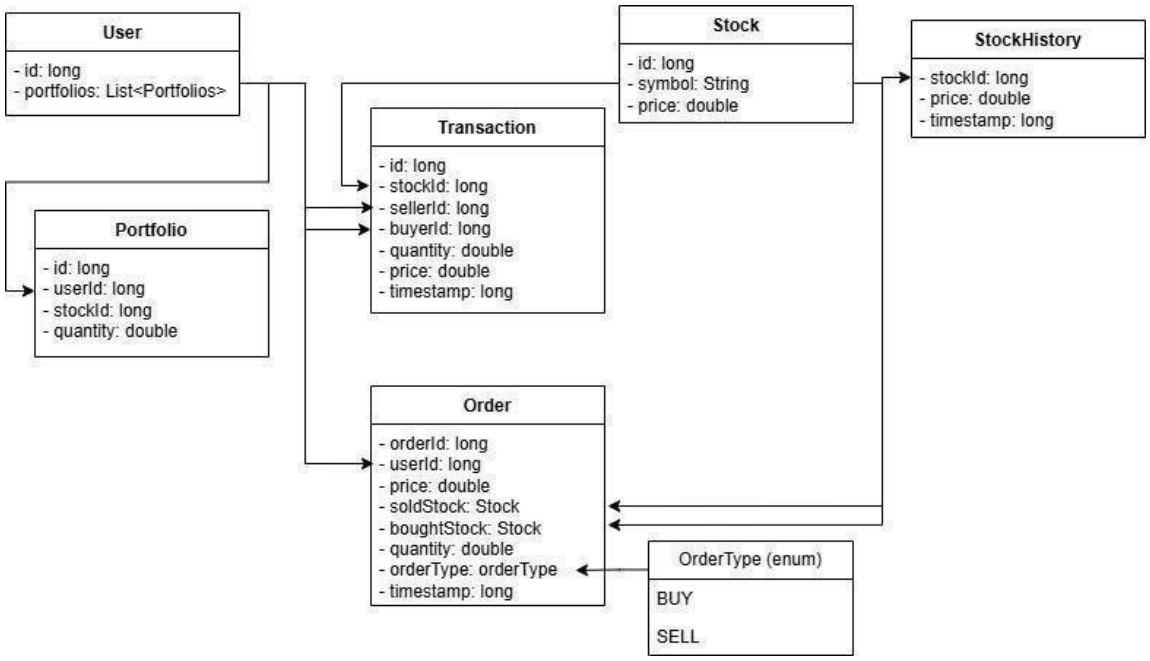
Acest proiect reprezintă simularea unei burse de acțiuni. Structura este de tip server-clienți, având următoarele roluri:

- 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 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 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));
    }
```

În clasa dedicate plasării de comenzi folosim ca atribut o variabila de tip ConcurrentHashMap pentru a retine id-urile utilizatorilor si 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<>();
```

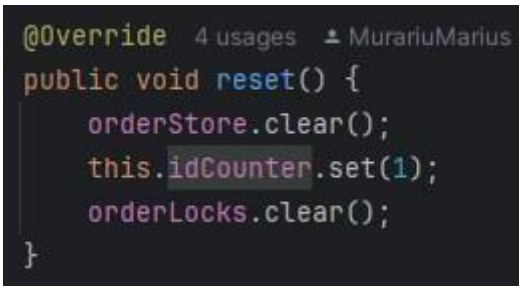
Lock-ul este realizat prin metoda „lockOrder", care este folosita si pe fiecare comanda înregistrată până în prezent care ar putea fi compatibila cu obiectul de tip „Order" din parametru. Aceasta se datoreaza nevoii de a evita schimbarea sau stergerea unei comenzi din lista chiar in timpul procesului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



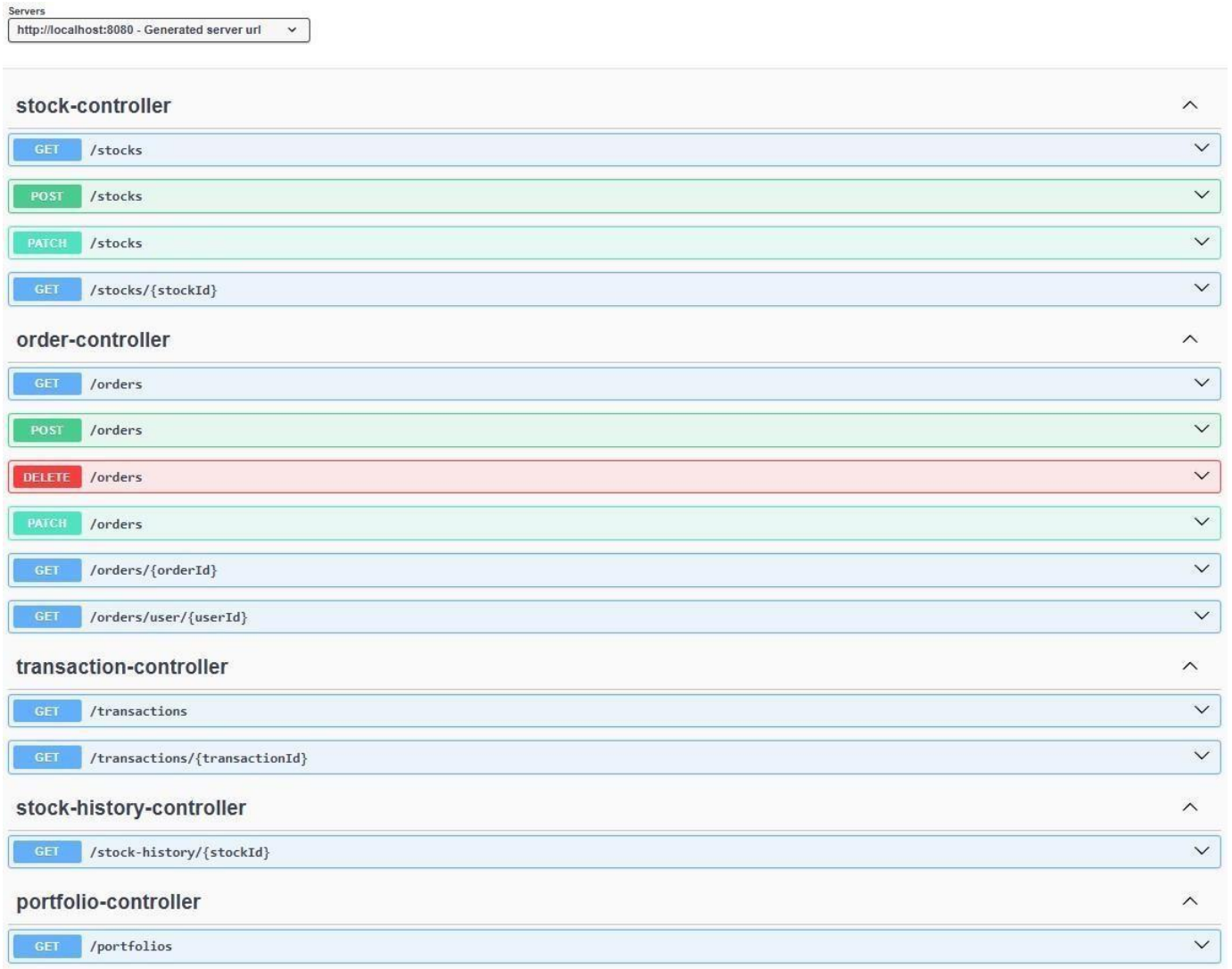
Aici nu am folosit doar un ConcurrentHashMap pentru sincronizarea comenzilor, ci si un counter de tip AtomicLong pe care il folosim pentru a determina id -ul pe care il va avea o comanda atunci cand e creata. Nu am folosit o variabila doar de tip Long, deoarece 2 sau mai multe comenzi plasate simultan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

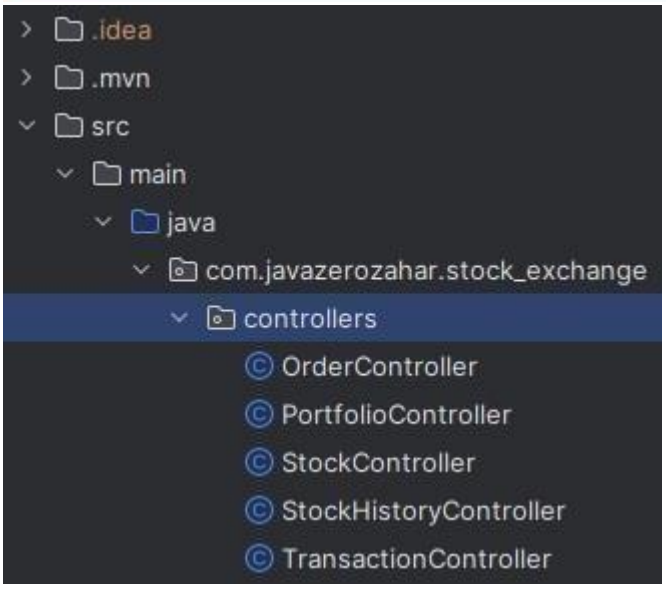
Acest counter este util si daca dorim sa resetam baza de date a comenzilor, avand posibilitatea de a se intoarce la valoarea initiala intr-un mod thread-safe.

4. Specificația OpenAPI

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

/v3/api-docs: http://localhost:8080/v3/api-docs

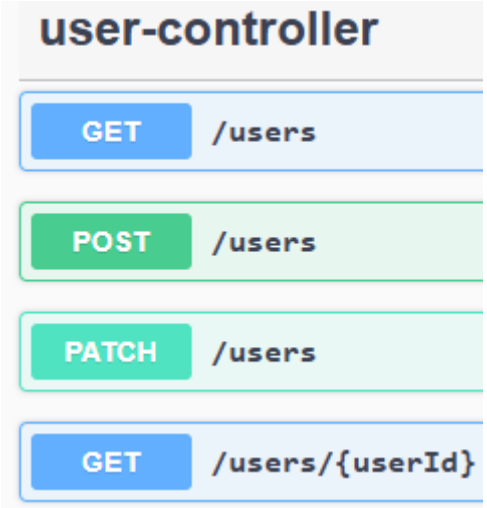




Interfata de programare de tip OpenAPI/Swagger usureaza utilizarea endpointurilor API-ului definit de catre noi prin controllerele implementate cu Java Spring.

5. Rolurile endpoint-urilor

a)

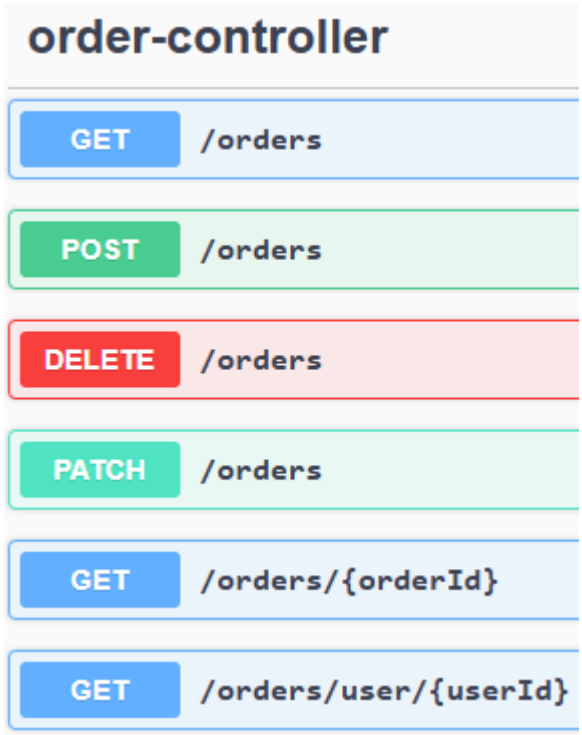


b)

Scopul endpoint-urilor de la a) si b) in ordine de sus in jos:

- Obtinerea tuturor user-ilor/stock-urilor
- Crearea unui nou user/stock. Valorile atributelor noului obiect vor fi definite in corpul request-ului
- Actualizarea unui atribut(sau mai multe) ale unui user/stock (user-ul/stock-ul actualizat va fi stabilit in functie de id-ul scris in RequestBody)
- Obtinerea unui anumit user/stock in functie de id-ul scris ca parametru in corpul request-ului (de exemplu: /users/2 asteapta ca raspuns datele celui de-al doilea user; la fel si pentru stock-uri)

c)



Functiile endpoint-urilor de la c) :

- Obtinerea tuturor comenzilor
- Crearea unei noi comenzi
- Stergerea unei comenzi definite in format json in corpul request-ului
- Actualizarea unei comenzi (asemanator cu patch-ul din a) si b))
- Obtinerea comenzii care are id-ul egal cu cel dat ca parametru {orderId)}
- Obtinerea tuturor comenzilor facute de un anumit user (cel care are id-ul egal cu valoarea data ca parametru {userId})

transaction-controller

GET `/transactions`

Endpoint pentru obtinerea tuturor tranzactiilor (pot fi folositi parametri optionali pentru filtrare in functie de user sau stock)

Parameters

Name	Description
userId integer($int64) *(query)*	userId
stockId integer($int64) *(query)*	stockId

GET `/transactions/{transactionId}`

Endpoint pentru obtinerea tranzactiei cu id-ul cerut de parametrul mandatoriu

Parameters

Name	Description
transactionId * required integer($int64) *(path)*	transactionId

stock-history-controller

GET `/stock-history/{stockId}`

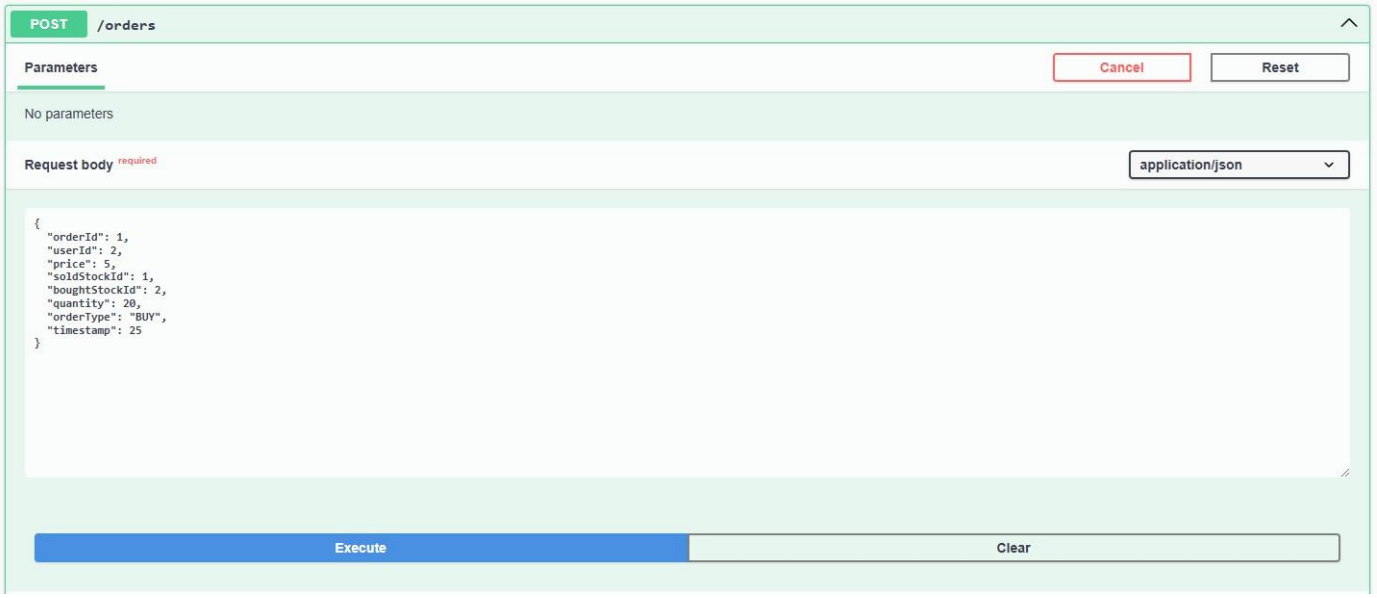
Endpoint pentru obtinerea istoricului de schimbari la care a fost supus stock-ul cu id-ul cerut

portfolio-controller

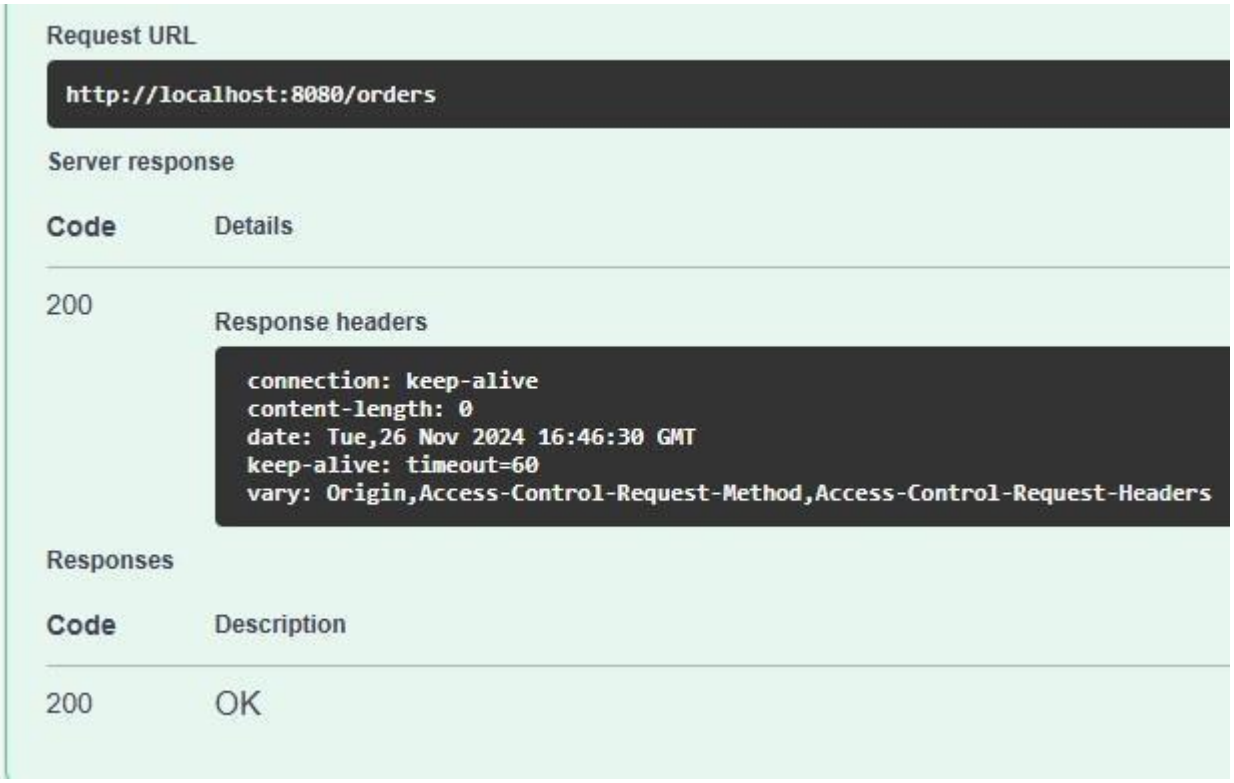
GET `/portfolios`

Endpoint pentru obtinerea tuturor portofoliilor (pot fi folositi parametri optionali pentru filtrare in functie de user sau stock)

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:



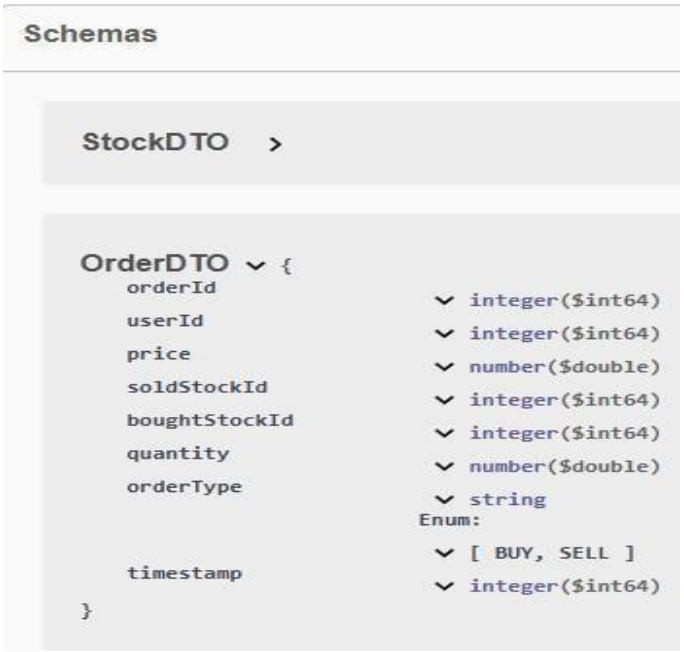
Prin interfata ultimelor 2 imagini am apelat metoda urmatoare din clasa OrderController:

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

6. Integrarea serverului concurent

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 concurent, 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 previn 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.

Traseul de la endpoint-uri la serverul concurent se desfășoară în felul următor. Atunci când creem o nouă comandă prin intermediul interfeței Swagger, controller-ul API-ului aplică o metodă din clasa OrderService (care pune la dispoziție funcții de adăugare/actualizare/ștergere/obținere a comenzilor) pentru a o plasa. Acea metodă, la rândul ei, aplică metoda „placeOrder" a unui obiect de tip OrderPlacer. Metodă „placeOrder" aplică metoda „sendOrder" a clasei OrderPlacerProducer. Aceasta clasa are rolul de a trimite comanda sub formă de mesaj către clasa OrderPlacerConsumer. Obiectul de tip OrderPlacerConsumer primește comanda de la producer și o procesează. În metoda de procesare a clasei este folosit un obiect de tip OrderMatcher pentru a se încerca găsirea unei comenzi create anterior care să se potrivească cu cea tocmai primită de Consumer în scopul creării unei tranzacții automate. Metodă pentru matching a clasei OrderMatcher folosește un obiect de tip OrderRepository și altul de tip PortfolioService pentru a pune un „lock" asupra comenzii curente. Blocarea propriu-zisă are loc în câteva din metodele celor 2 clase. De exemplu, OrderRepository:

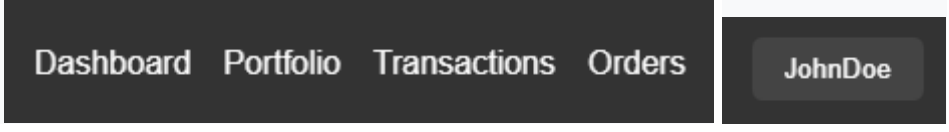
```
public interface OrderRepository extends JpaRepository<Order, Long> {

    @Lock(LockModeType.PESSIMISTIC_WRITE)  1 usage  MurariuMarius
    @Query("SELECT o FROM Order o WHERE o.boughtStock.id = :stockId")
    List<Order> findByBoughtStockIdWithLock(Long stockId);

    @Lock(LockModeType.PESSIMISTIC_WRITE)  1 usage  MurariuMarius
    @Query("SELECT o FROM Order o WHERE o.soldStock.id = :stockId")
    List<Order> findBySoldStockIdWithLock(Long stockId);

    @Lock(LockModeType.PESSIMISTIC_WRITE)  1 usage  MurariuMarius
    @Query("SELECT o FROM Order o WHERE o.orderId = :id")
    Optional<Order> findByIdWithLock(Long id);
```

7. Descrierea componentelor dezvoltate

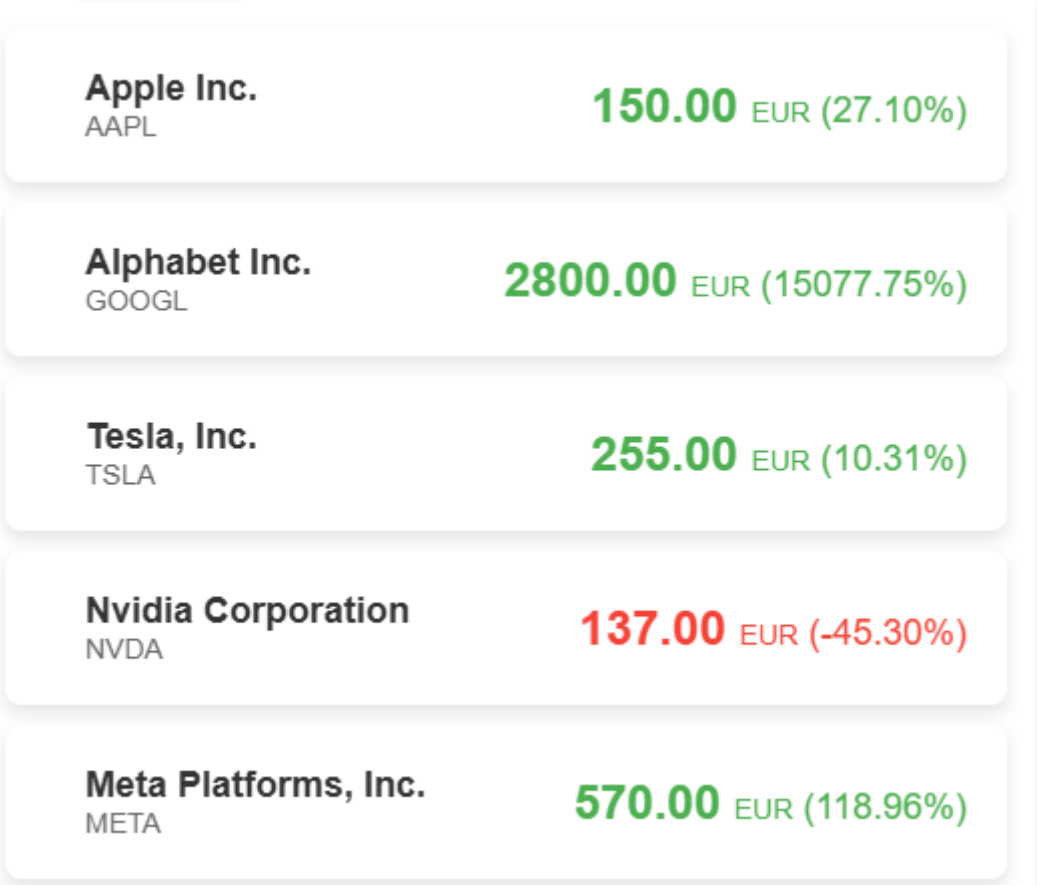
Componentele realizate sunt serverul central pentru backend(pe care l-am descris in inceputul documentatiei) si client-ul, pe care urmeaza sa il descriem mai detaliat.

Client-ul este o aplicatie dezvoltata pe un server local, folosind framework-ul Vue.js din JavaScript. Astfel, sunt folosite template-uri html, stilizate cu css, care utilizeaza script-uri de js pentru a transmite sau primi date/functii. Scopul aplicatiei este de a oferi utilizatorilor posibilitatea de a interactiona cu serviciile oferite de serverul backend.

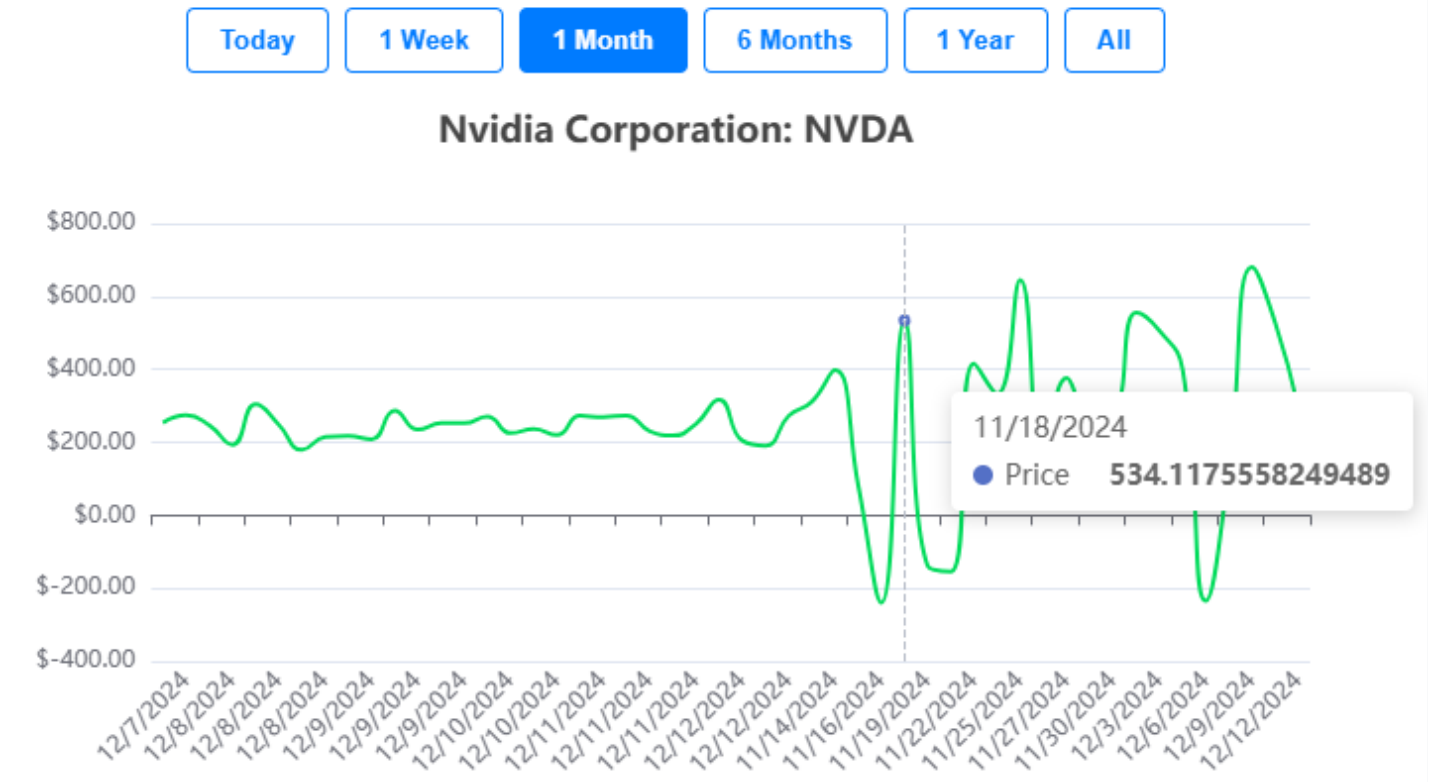
De exemplu, navigation bar-ul din partea de sus a paginii are rolul de a redirectiona prin link-uri utilizatorul catre pagina corespunzatoare tipului de serviciu dorit.



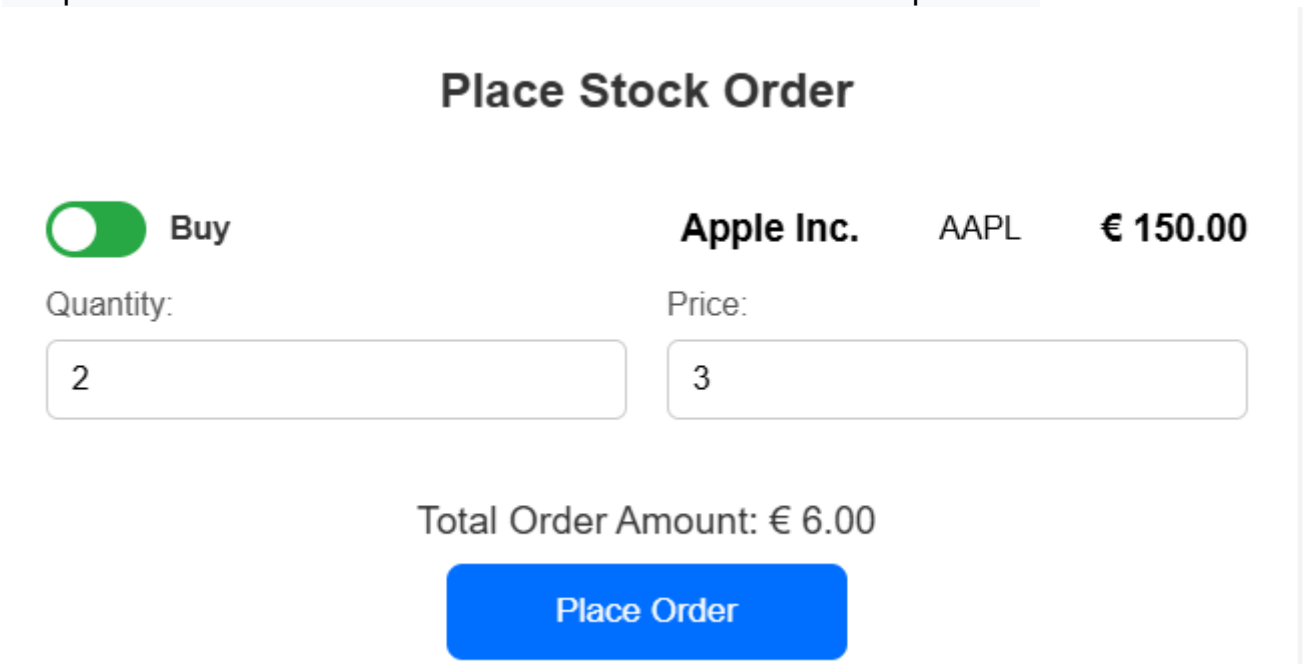
a) Pagina de Dashboard are ca rol afisarea tuturor stock-urilor disponibile pentru tranzactii, alaturi de valorile lor actuale in EURO si procentul cu care acestea au crescut fata de ziua anterioara.



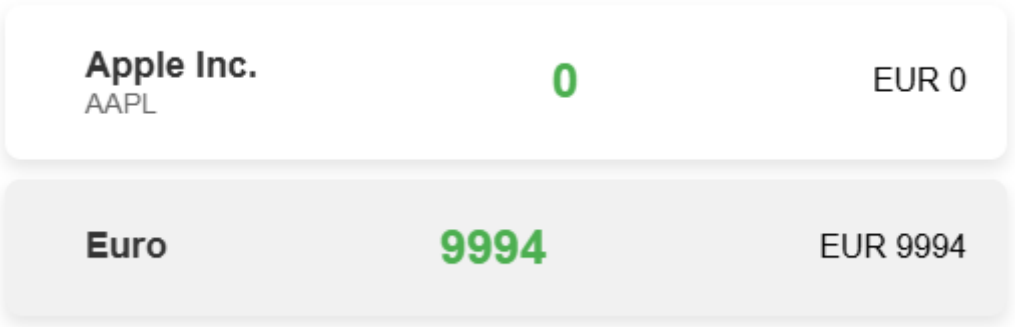
Dashboard-ul permite user-ului si sa observe progresul valorii unui stock la alegere in ultimul perioada. Fiecare timestamp reprezinta un obiect de tipul StockHistory, care retine valoarea (generata random, dar influentata de pretul de baza si fluctuatia pietei) unui stock la un moment dat.



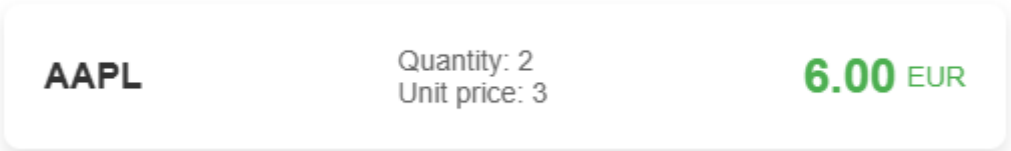
O alta functionalitate importanta a paginii este aceea de a plasa o comanda de cumparare/vanzare folosind stock-ul selectat din lista celor disponibile:



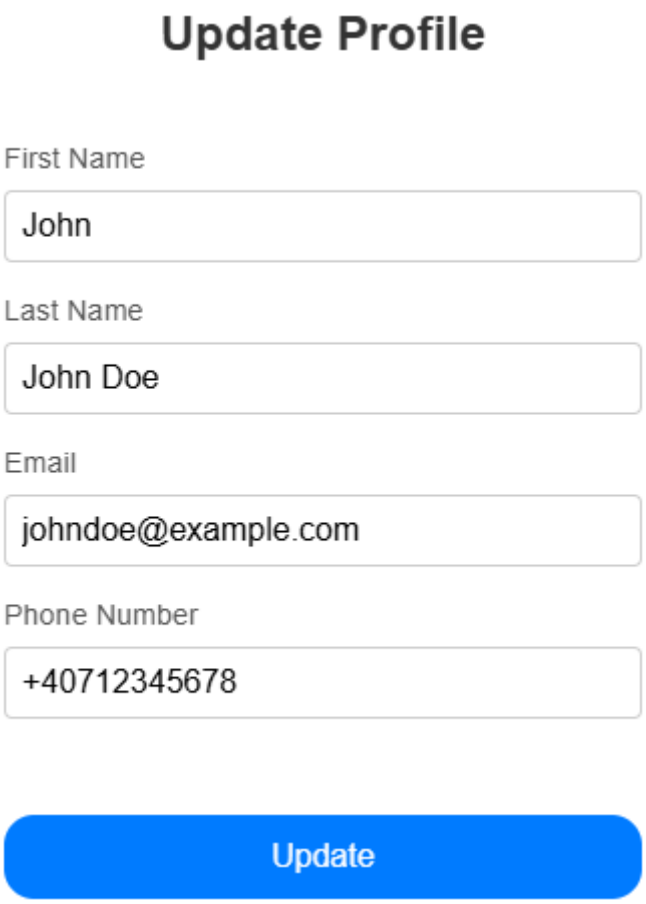
b) Pagina Portfolio este pagina in care utilizatorul curent poate urmari cantitatea de stock-uri pe care o detine si valoarea lor in EURO. In urma plasarii comenzii anterioare (initial user-ul avea 10000 EURO), suma de 6 euro a fost retrasa din portofoliul de EUR, iar apoi a fost creat portofoliul de AAPL care inca nu are niciun stock, din moment ce nu a fost inca gasit un cumparator.



c) In pagina Orders, utilizatorul poate observa comenzile plasate de acesta si care inca nu au primit match. De aceea, in cazul nostru, este prezenta comanda plasata la punctul a)



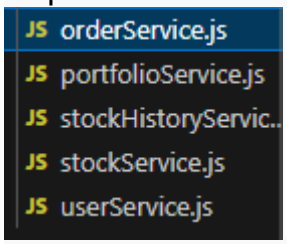
d) In partea dreapta a Navbar-ului se afla un link catre pagina de profil a utilizatorului. De aici este posibila modificarea datelor personale ale user-ului.



O alta componenta este client-ul pentru administrator, care are posibilitatea de a adauga un User/Portofolio/Order/Stock prin completarea unui formular asemanator vizual cu cel din poza anterioara.

8. Tehnologiile folosite pentru comunicarea backend-frontend

Pentru schimbul de date dintre serverul central si aplicatia Vue, sunt folosite fisiere JavaScript continand functii care trimit request-uri API-ului, pentru ca apoi sa returneze datele utile din response sau sa arunce exceptii in cazul unui request necorespunzator.



De exemplu, pagina PortfolioView contine o componenta <PortofolioList />.

```
<h1>This is PortfolioView</h1>
<div class="container">
  <div class="portfolios">
    <PortfolioList v-if="loadedPortfolios"
                   :portfolios="portfolios"
                   @portfolio-selected="handlePortfolioSelected"
    />
  </div>
```

```
onMounted(async () => {


  const fetchedPortfolios = await fetchPortfolios(getCurrentUser().id);
```

Toate portofoliile utilizatorului curent sunt citite din baza de date a serverului central prin functia JS „fetchPortfolios”.
<PortofolioList />, la randul ei este o lista compusa din mai multe elemente de tip <Portfolio /> si retine valorile portofoliilor citite anterior.

Fiecare componenta <Portfolio /> foloseste datele unui obiect JavaScript numit „portfolio”, care este transmis ca parametru de catre parintele <PortofolioList />.

9. Stadiul general al dezvoltării sistemului

Pe parcursul milestone-urilor se poate observa ca proiectul a trecut prin schimbari.

Pentru primul milestone a fost realizata componenta de server concurent, pentru care a fost folosit doar Java pur (si JUnit pentru testare), prevenind potentialele probleme de concurenta prin elemente apartinand librariei java.util.concurrent. Obiectele de tip User/Portofolio/Order/Stock/StockHistory au fost stocate in liste, iar pentru operatii de tip CRUD asupra acestor obiecte foloseam clasele din folderul repository.

In cazul celui de-al doilea milestone au fost folosite si alte tehnologii:

- PostgreSQL pentru stocarea obiectelor intr-o baza de date
- RabbitMQ pentru integrarea unui sistem de comunicații prin mesaje care este alcatuit din producatori si consumatori, astfel incat sa previna erorile de concurenta care ar putea aparea la plasarea unei comenzi
- Docker pentru a se asigura ca fiecare membru al echipei foloseste aceleasi configuratii de PostgreSQL si RabbitMQ
- Spring pentru adnotarile folosite pentru pornirea serverului/testelor, Lock-uri, queries in baza de date, sau implementarea endpoint-urilor pentru API-ul server-ului
- Postman pentru testarea API-ului
- Swagger/OpenAPI pentru generarea interfetei API-ului

Acestea au facilitat comunicarea dintre serverul concurent si endpoint-uri, realizand astfel serverul central care a urmat sa fie folosit ca si componenta de backend pentru proiect.

Dupa cel de-al doilea milestone au fost implementate 2 tipuri de componente client (unul pentru utilizatorul obisnuit, iar celalalt pentru administrator). Acestea au fost implementate in aceeasi maniera, ambele utilizand npm pentru pornirea aplicatiei client si Vue.js pentru folosirea de template-uri html, care pot transmite intre ele date, functii, sau declansa evenimente definite cu JavaScript. Acelasi limbaj de programare este responsabil si pentru implementarea functiilor care trimit request-uri si primesc raspunsuri la endpoint-urile definite in milestone-ul trecut, astfel facand legatura dintre componenta frontend si backend.

Urmatorul pas al acestui proiect il constituie testarea aplicatiei in posibilele scenarii in care 2 sau mai multi utilizatori sunt conectati simultan la server pentru a studia siguranta acestuia in contextul concurentei.