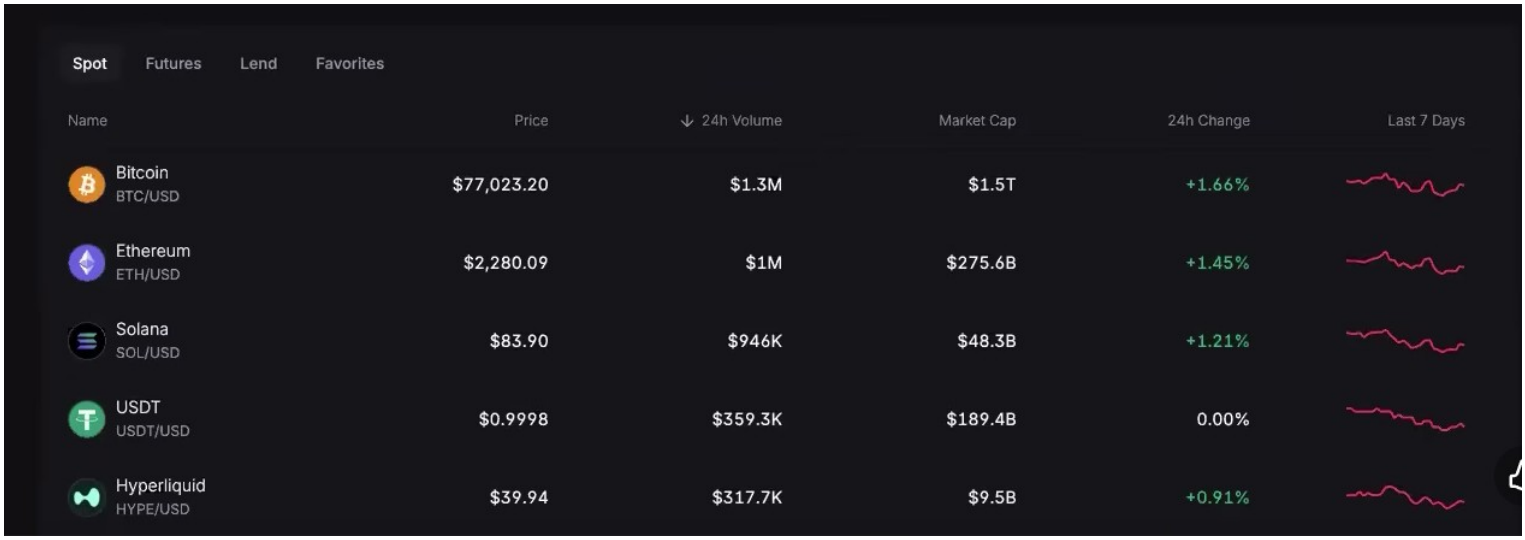
Volume means itna karida becha aaj
using backpack website



see in this the volume means itna aaj trade hua hai and how backpack earns money is by taking some fee on that

today we are making spot and there is something called as futures also

What we're building

A centralized exchange (Binance, NYSE, BSE, Backpack, Coinbase)

Zerodha, grow are all brokers they are just api on top of the BSE(bombay central exchange there used to be for Delhi central exchange also, kolkata central exchange also)

agar zerodha and grow mai kaam karoge toh just api par and user ko joh dikehga uspar kaam karoge but BSE, NYSE mai toh they work on the orderbook

Functional requirements

1. Users should be able to sign up
2. Ideally users should be allowed to KYC (we will skip this)
3. Users should be able to see the performance of various stocks

4. Users should be able to buy/sell stocks
5. Users should be able to Deposit/withdraw INR (razorpay eventually, dummy today)

Other financial instruments/contracts

1. Options/Futures (zerodha's 90% revenue)
2. Perpetual futures (Binance, Backpack have these. Drift was a decentralized perp)
3. CFDs (exness)

jupyter is a decentrailized exchange and today we are making a centralized exchange like binance and backpack
price of stocks such as solana will be different on all the markets such as binance, backpack and jup they all maintain separate market hence has different prices

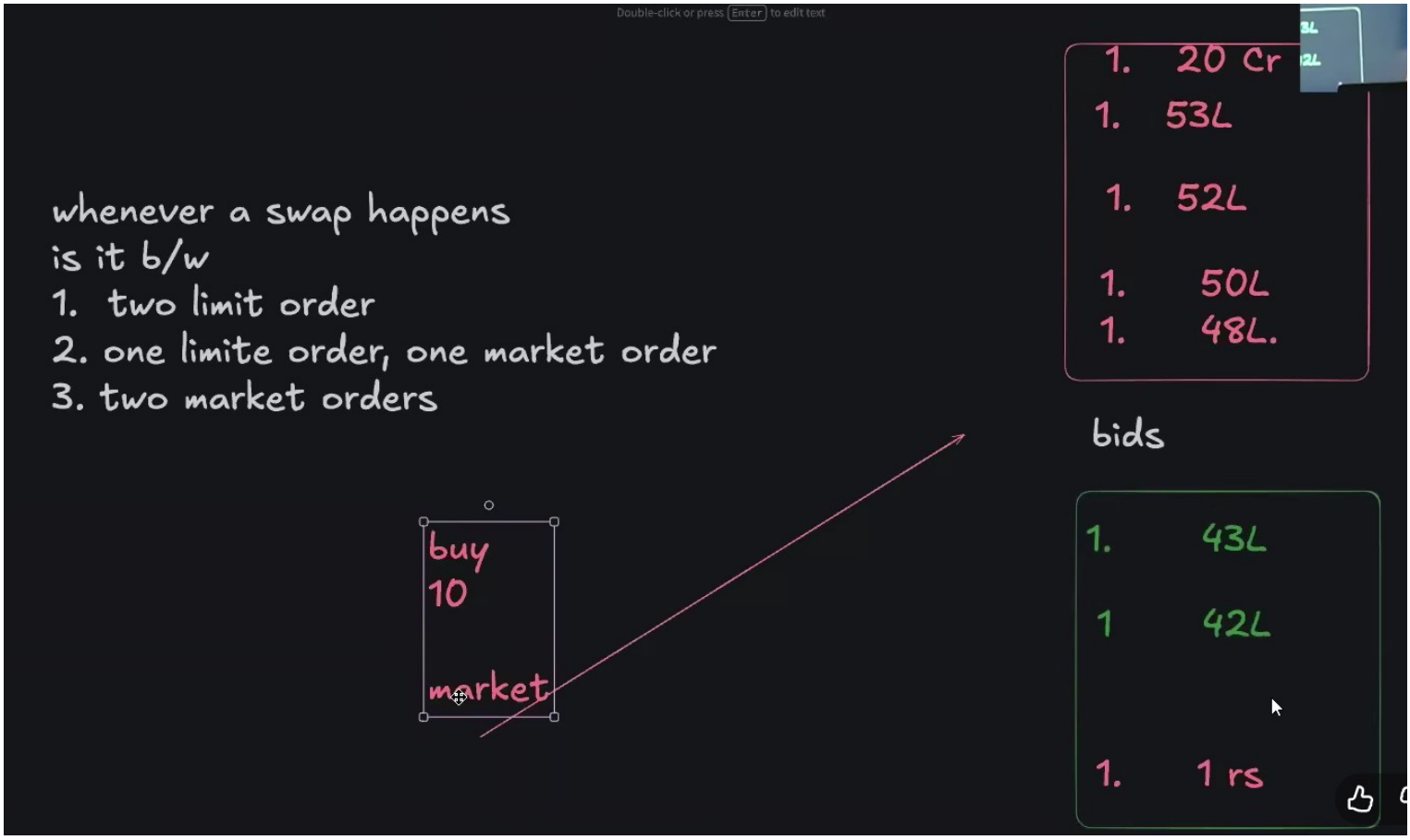
coinmarketcap shows the price of all the coins on different platform on one platform

SUPPLY DEMAND KAE UPAR KISI BHI COIN KA PRICE DECIDE HOTA HAI TOH REMEMBER THAT

DIFF BETWEEN SPOT AND FUTURES
spot is like when you are actually buying the coin and you can keep that in the wallets but in futures you never own a coin you bet on the price of the coins

WHAT ARE LIMIT ORDERS AND MARKET ORDERS?
Limit orders means when you give a price and until that price someone is willing to sell you wait and when someone is willing to sell then the trade happens
market order means you directly see the price and buy it

FAT FINGER ERROR
adding a extra zero in price by mistake



like in this first one is possible because it means like two buyers which as given there bid which is possible

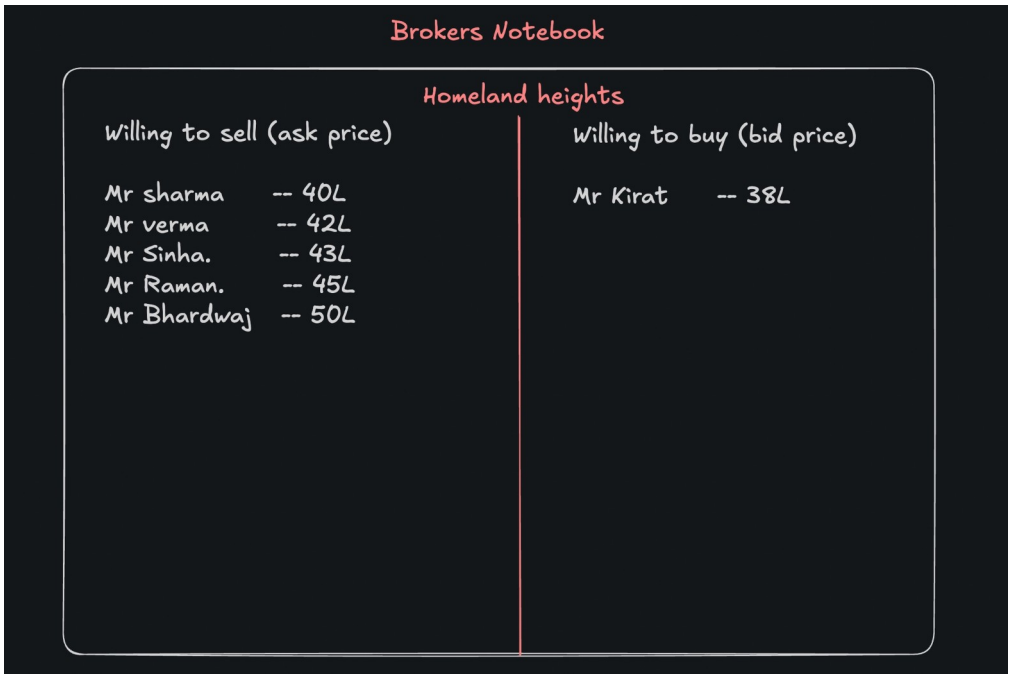
second one is also possible

third one is not possible in this you are saying like one seller comes and one buyer comes and at the same time the trade happens which is not possible

How a Centralized/SPOT exchange works - Orderbooks

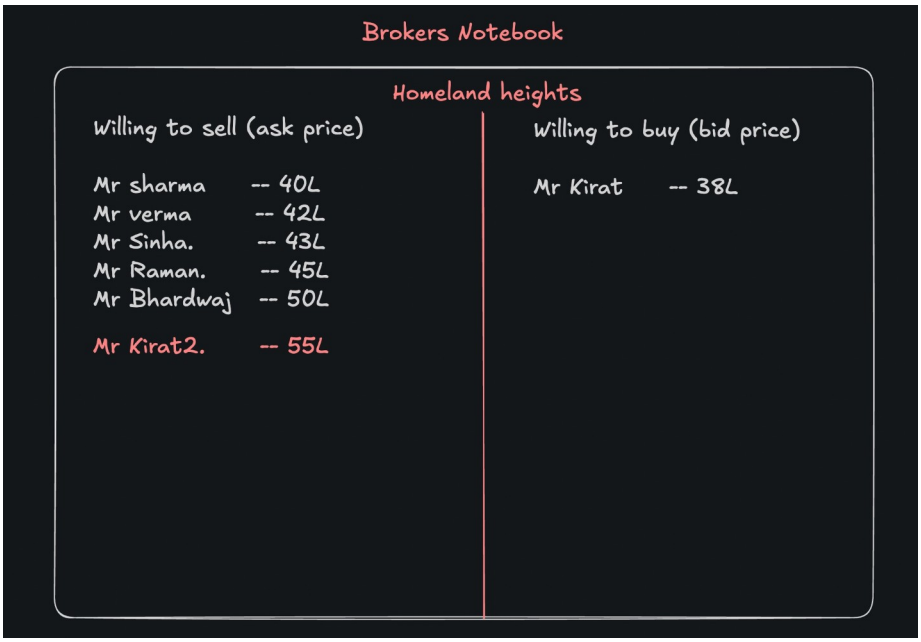
How do you buy land usually?

You go to a broker, who remembers all the plots/flats available in that society and the ask prices . He may optionally ask you to bid a price that you are willing to pay



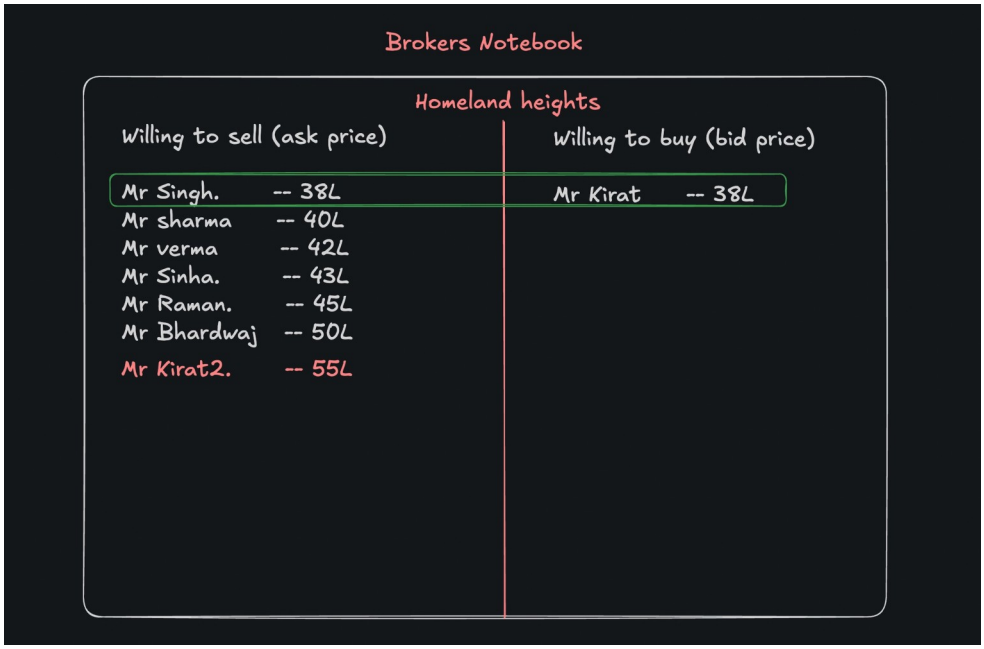
How do you sell land usually?

You go to a broker, who checks all the bids for the houses in the society. You optionally give him an ask for your house.

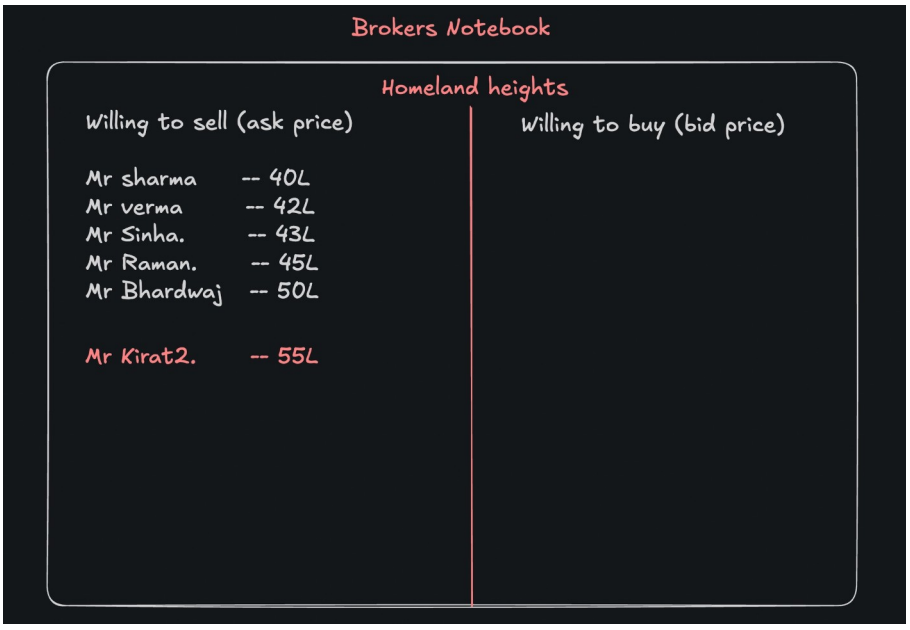


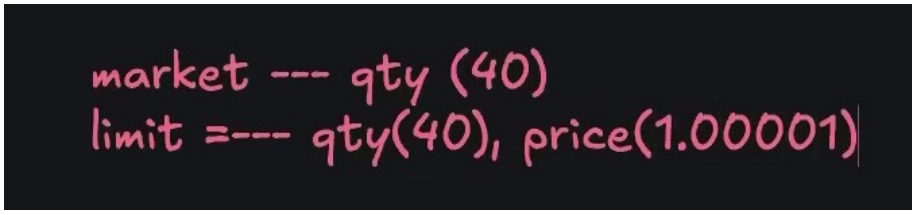
How does the final transaction happen?

If the orderbook ever overlaps, the transaction can be made (unless one party backs out ofcourse)



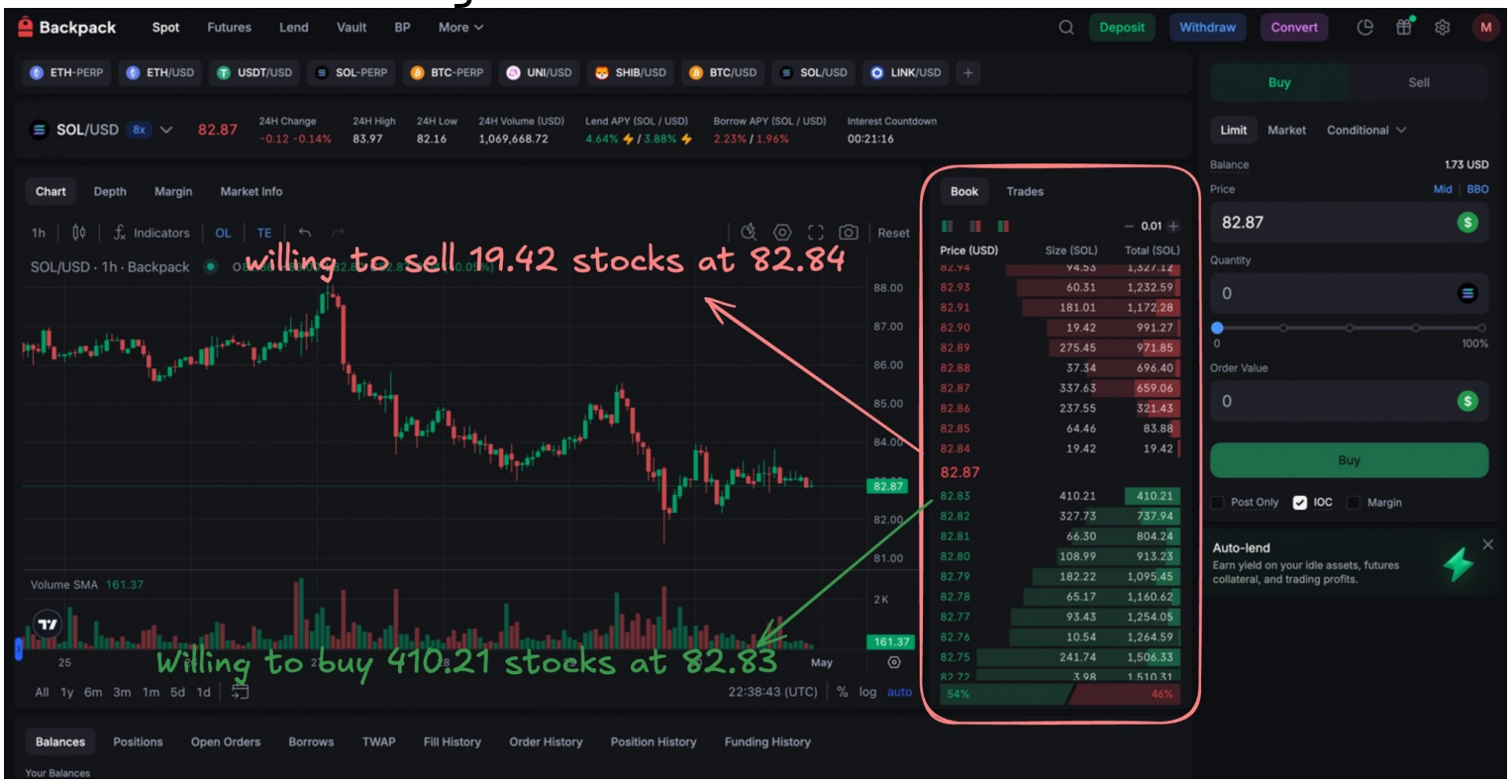
New orderbook



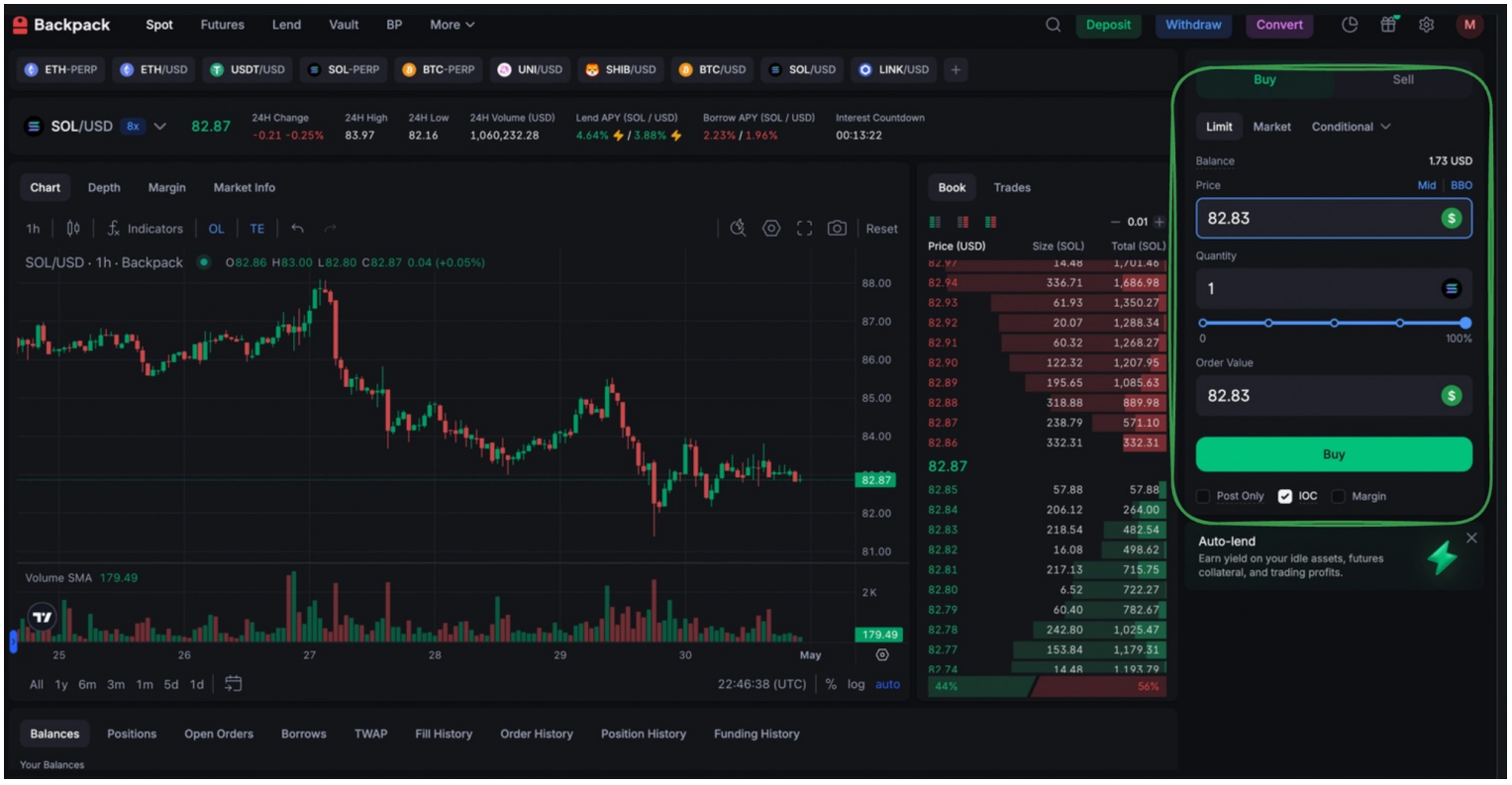


market order and limit order simple difference

Centralized exchanges are the same

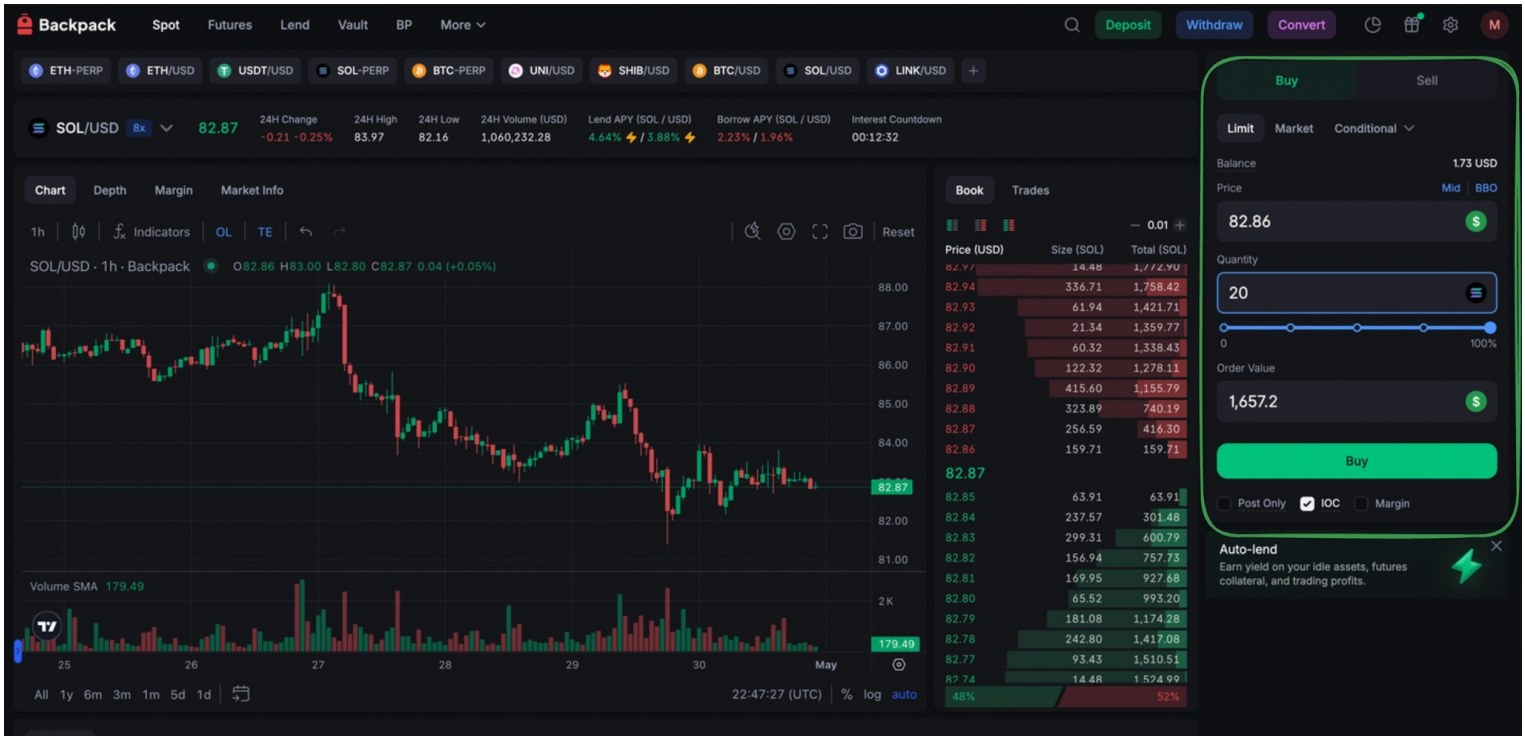


People are bidding what they’re willing to pay to buy a SOL stock
People are asking a certain price to sell their SOL stock.
Question 1



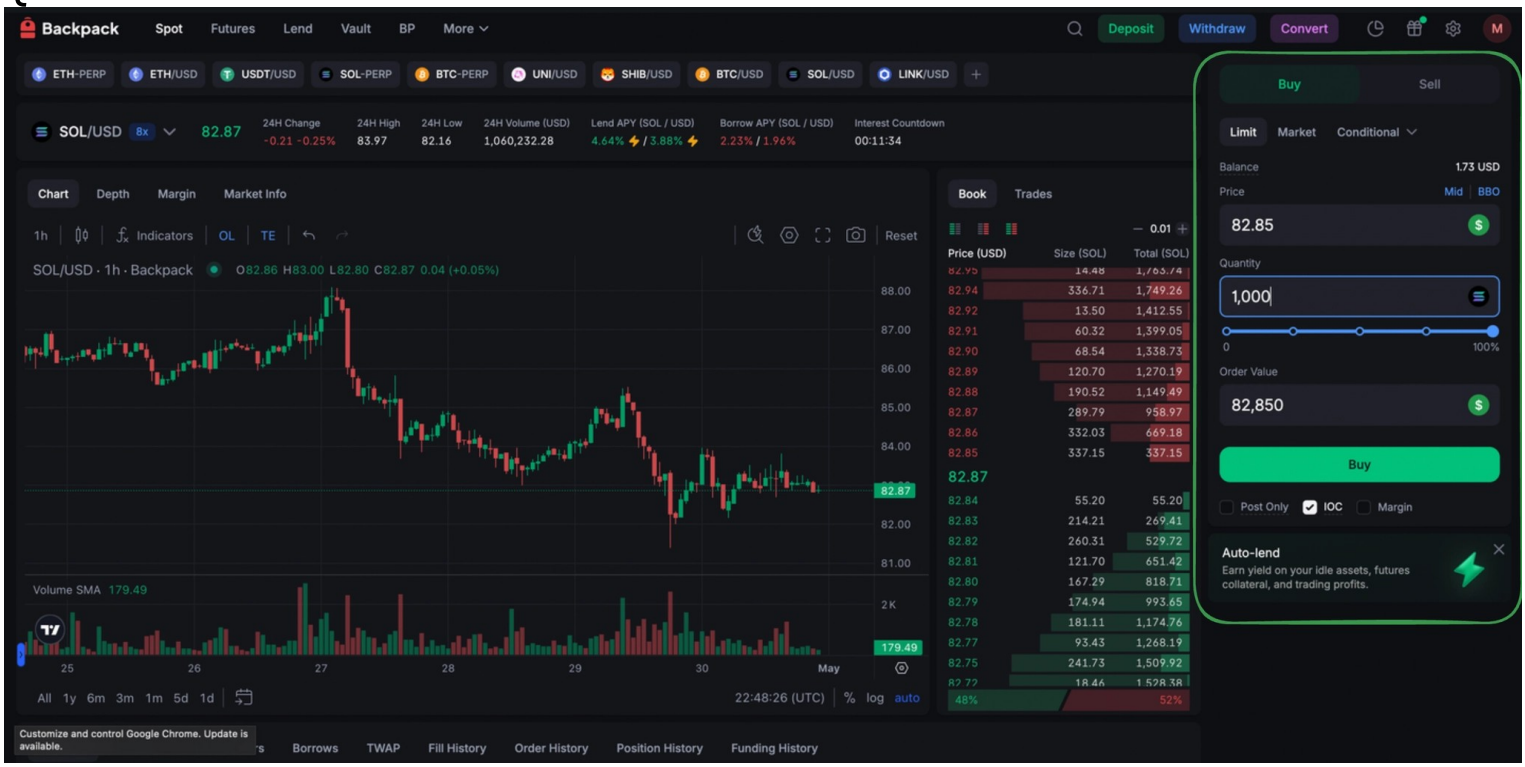
this one will sit at 82.83 and the value will become 219.54

Question 2



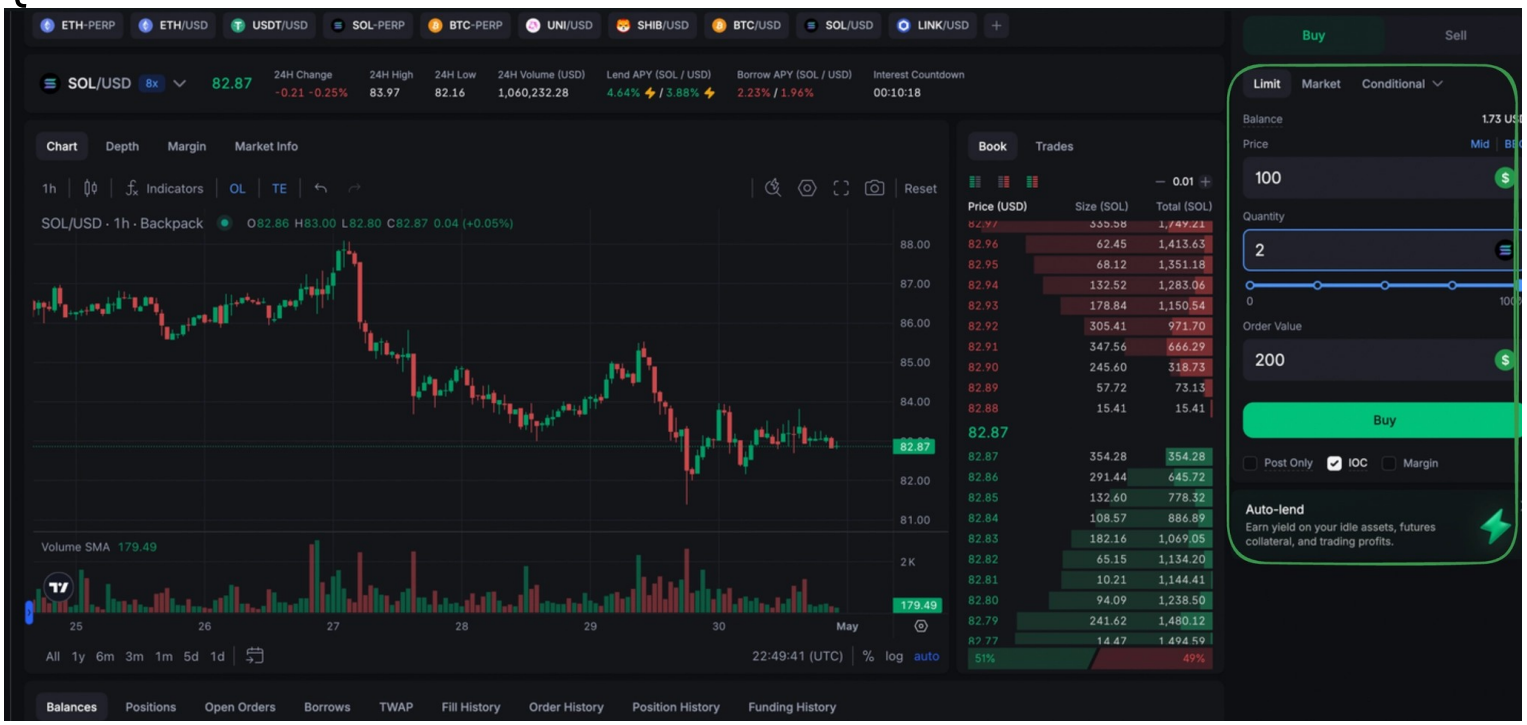
the buyer ask will be swaped but the seller size will reduce from 159.71 to 139.71

Question 3



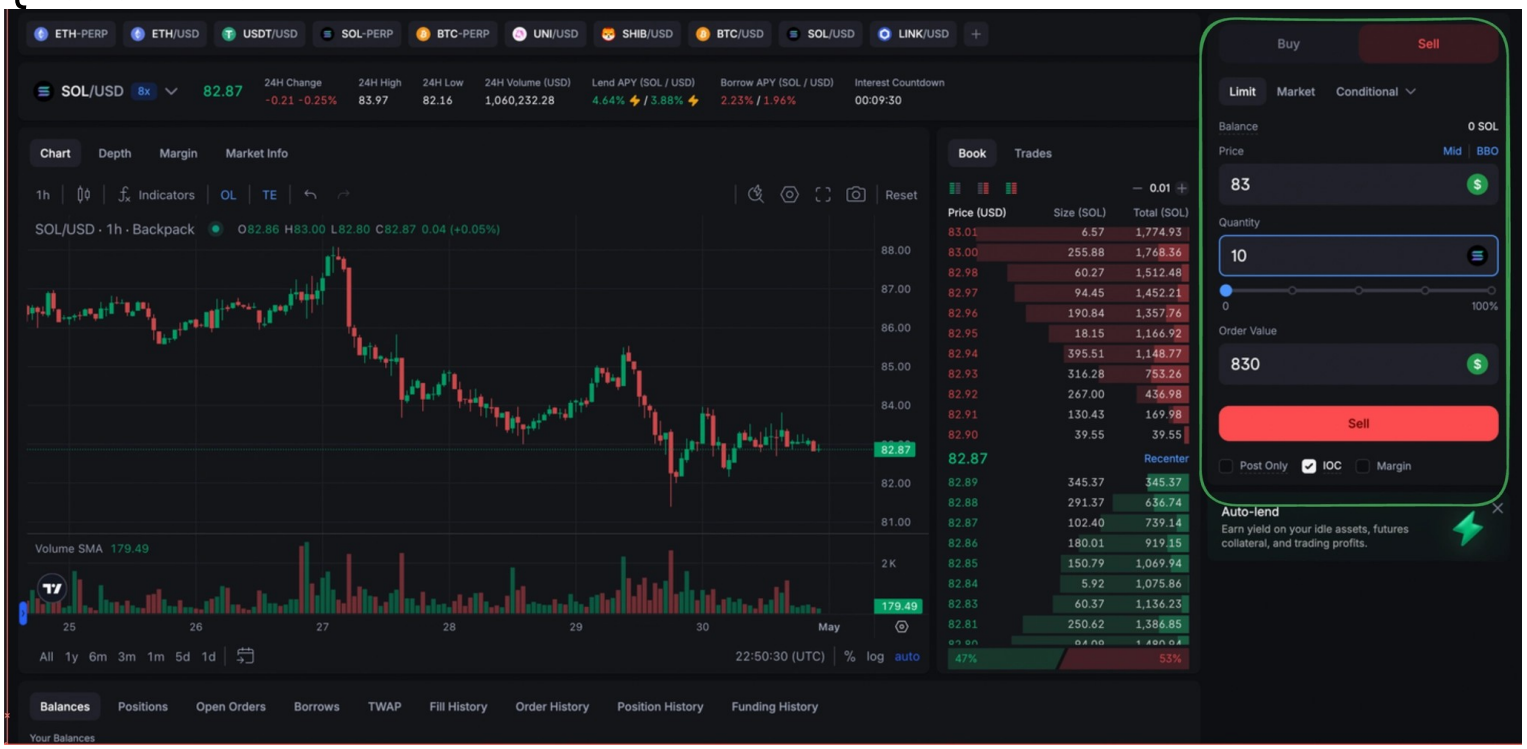
82.85 par jitna avail hai utna swap hojayega and bakki ka niche orderbook mai add hojyagea with 1000-337 = ==

Question 4



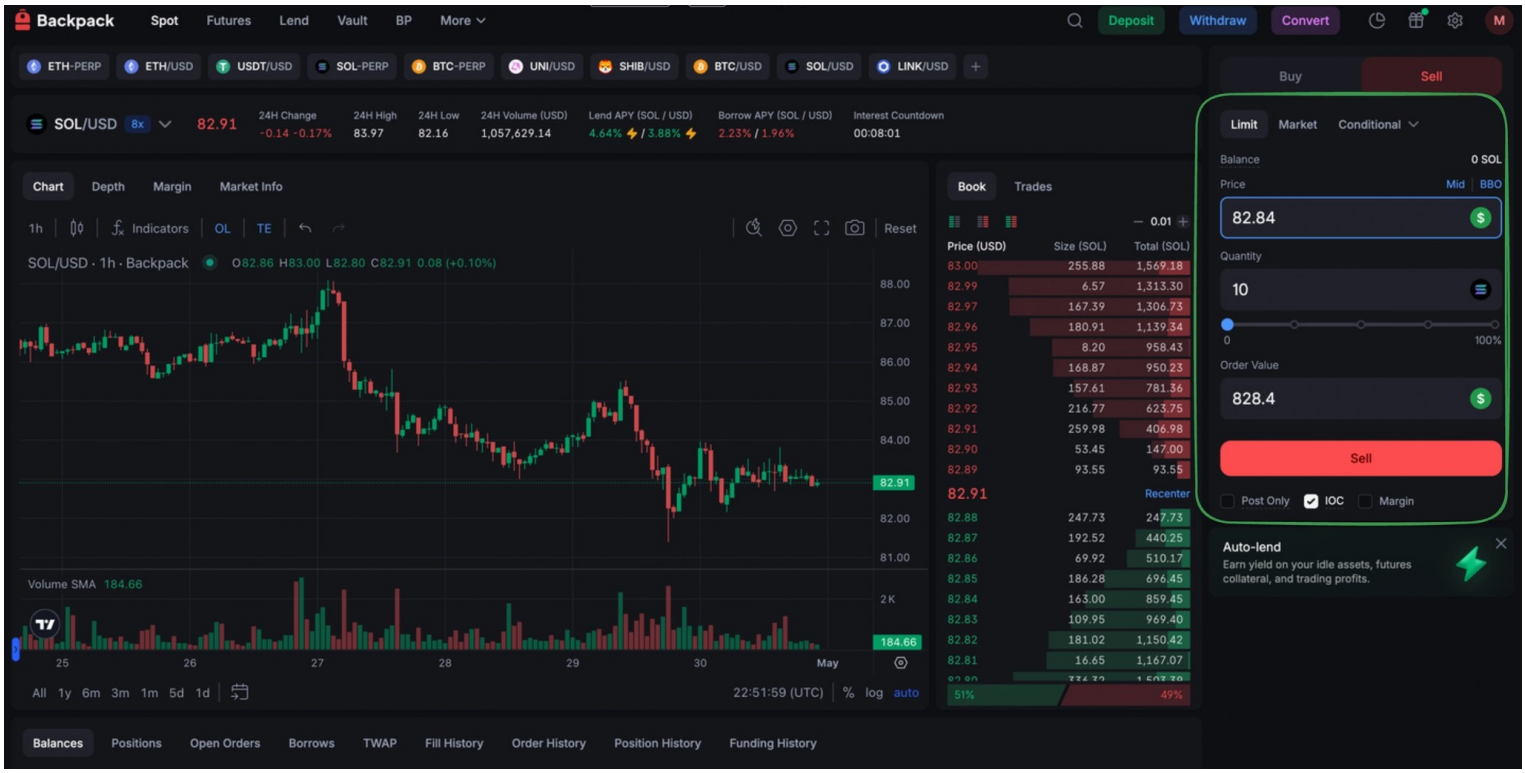
tum jada pay karskte thae but it will choose the best price which is 82.88 and will sell you at that price and count will decrease too 13.41 and other left money will be returned to youu

Question 5



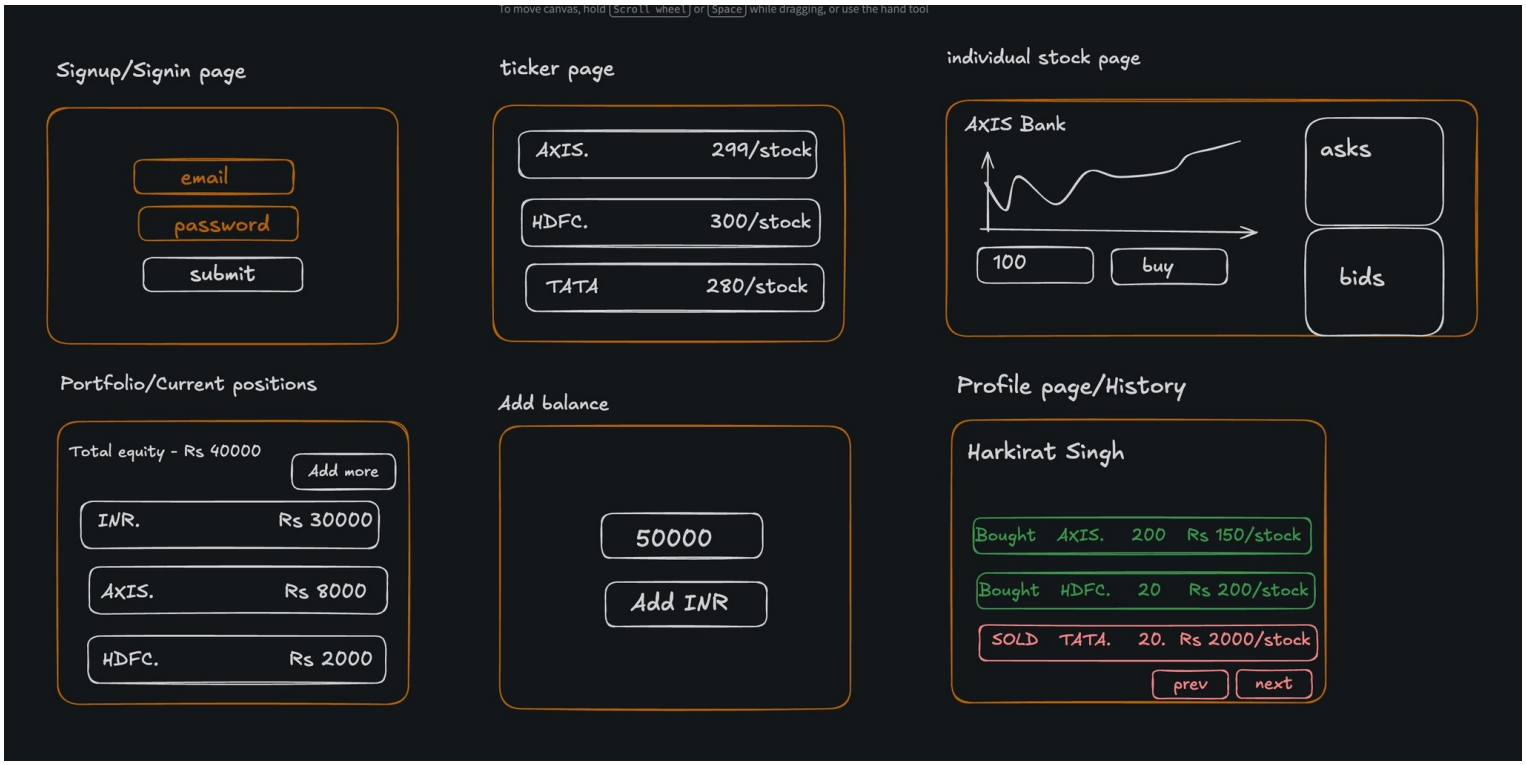
sell kar rhe ho to 83 mai count 255 sae 265 ho jaygea

Question 6



isme direct swap ho jayega 82.84 walae sae and uska count 163 sae 153 hojyega

UI Changes now



Do we remember in memory DBs?

In memory DB basically means storing data in memory to avoid using databases

```ts
import express from "express";

const app = express();

const USERS: { username: string, password: string }[] = [];
const TODOS: { title: string, completed: boolean }[] = [];

app.post("/signup", (req, res) => {
    const { username, password } = req.body;
    USERS.push({ username, password });
    res.status(201).send("User created");
});

app.post("/login", (req, res) => {
    const { username, password } = req.body;
    const user = USERS.find((user) => user.username === username && user.password === password);
    if (!user) {
        res.status(401).send("Invalid username or password");
    }
    res.status(200).send("Login successful");
});

app.listen(3000, () => {
    console.log("Server is running on port 3000");
});
```

Pros

1. Faster access
2. Easy to implement

Cons

1. Data is lost of server crashes
2. Cant horizontally scale

NOW WE STARTED WRITING THE CODE
TWO MAIN THINGS USED IS
Transactions and Locking

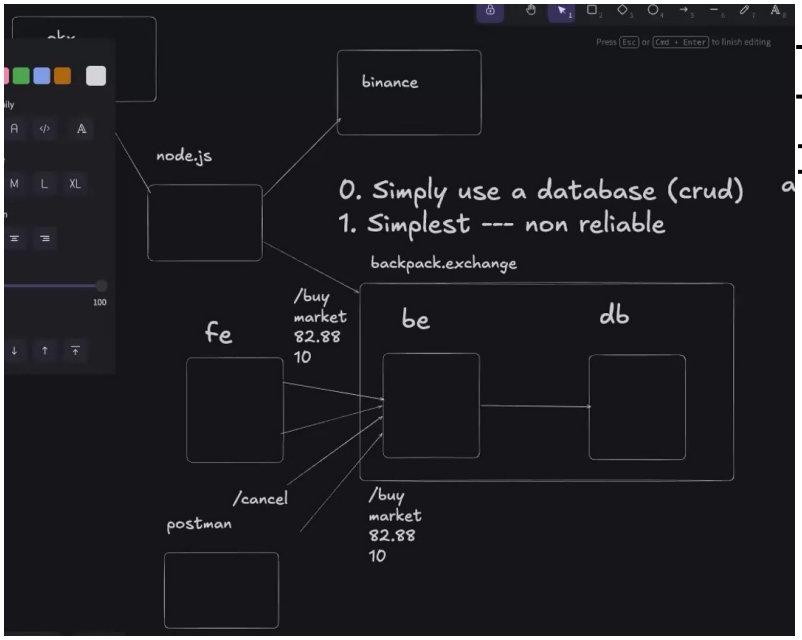
transaction for the whole flow to complete at once
Locking is done to stop a line of code

```
app.post("/order/market", (req, res) => {
    const userId = req.userId;
    const {price, qty, asset, side} = req.body;
    const usdBalance = await db.usdBalance.find({userId})

    if (usdBalance >= price * qty) {
        // matching
        const orderbook = await db.orderbook.find({asset});
        // matching
        filledQty, new_orderbook
        transaction([
            db.usdBalance.decrease();
            db.stocketBalance.update({})
            db.stocketBalance.update({})
            db.orderbook.update();
        )]
    }
```

r2
r1
r2 = 100
100

like in this if r1 comes and search for the usdBalance and it goes to db to search that concurrently during that r2 also comes and for that to it goes for the request to db and when the request for the r1 comes and proceed below assume 100 but for the request r2 also the totalBal is 100 so this is fixed by locking it means until and unless the r1 complete that request it will be locked for r2 and that request will not be given to r2

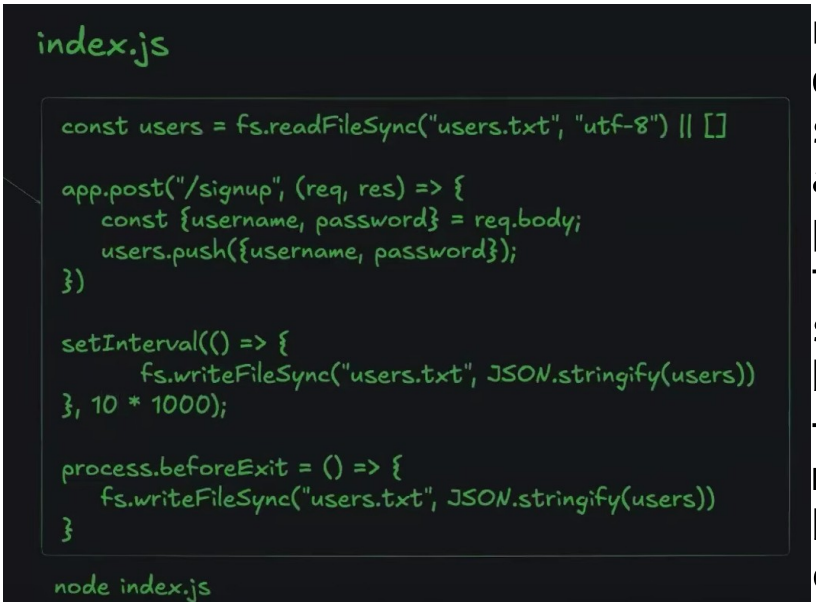
COMMON EXAMPLE WHERE IT IS USED IS
Ecommerce, Book my show
like in that also you use locking to prevent people to buy one seat multiple times
this is also called double booking or double spending



THIS ARCH IS VERY DIFFICULT TO SCALE AND IS LIKE VERY INEFFICIENT

PROS AND CONS OF NON DB CRUD DATABASES
CONS
1. data loss if server crashes
2. cant horizontally scale
PROS
1. fast
2. easy

now we have to somehow fix the cons of this

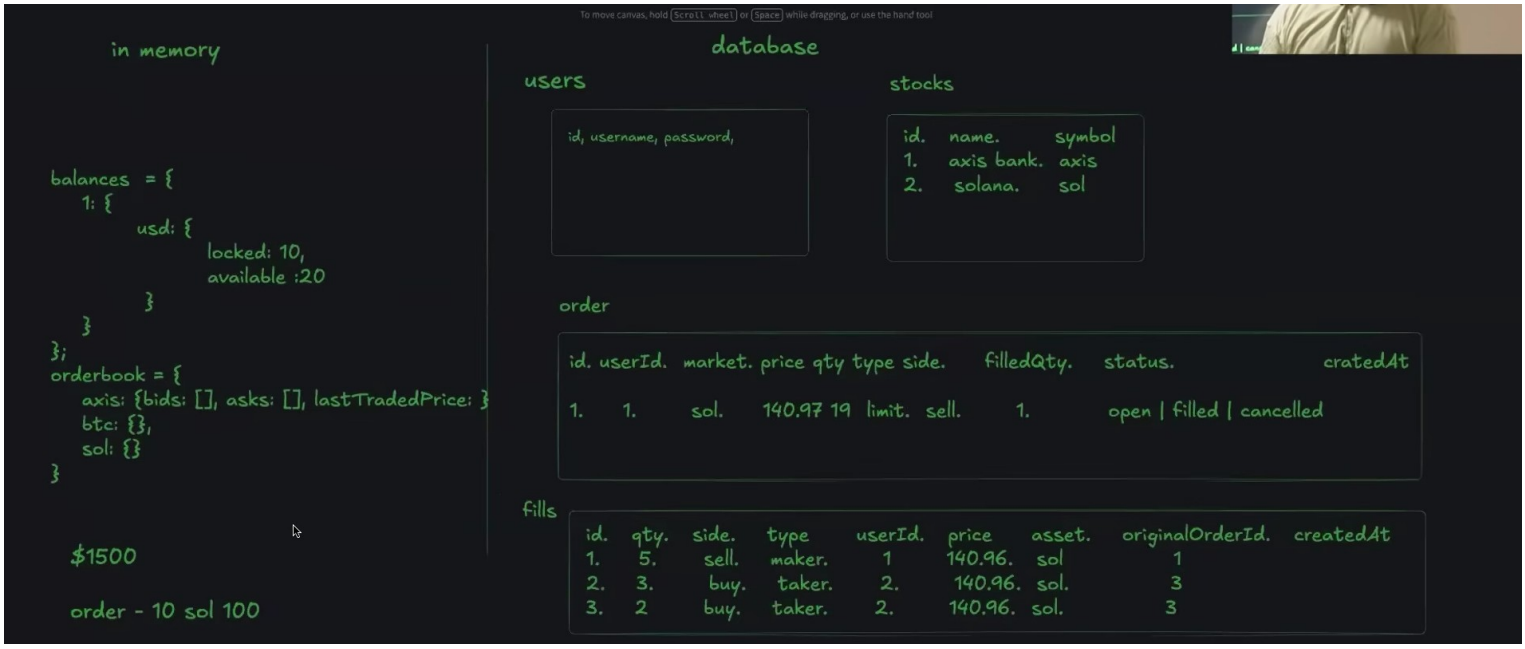


now in this what we are doing is very 10 sec we are storing the data in a file and then whenever the process starts it will read the that but still 10 seconds data can be lost and better arch will store this file in a place where other machine can also access because if this machine crashes the data is lost and we somewhere think the systems are resilience otherwise all the machines can crash at the end and for this we are following this architecture

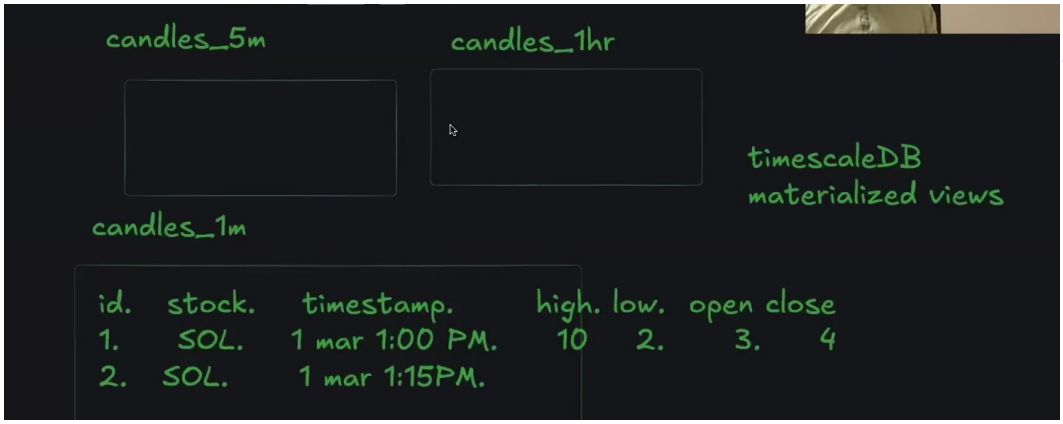
JOH HAAM PRICE DEKHTE HAI KISI BHI STOCK KA YEH WOH PRICE HOTA HAI JISPAR LAST STOCK BIKA HOTA HAI

COINBASE PAR SARAE EXCHANGE PAR JOH WOH STOCK BIKA HOTA HAI USKA AVERAGE PRICE HOTA HAI
BINANCE PAR JOH STOCK KA LAST PRICE THA BIKTE SAMAY WOH HOTA HAI

DB SCHEMA



depth == orderbook



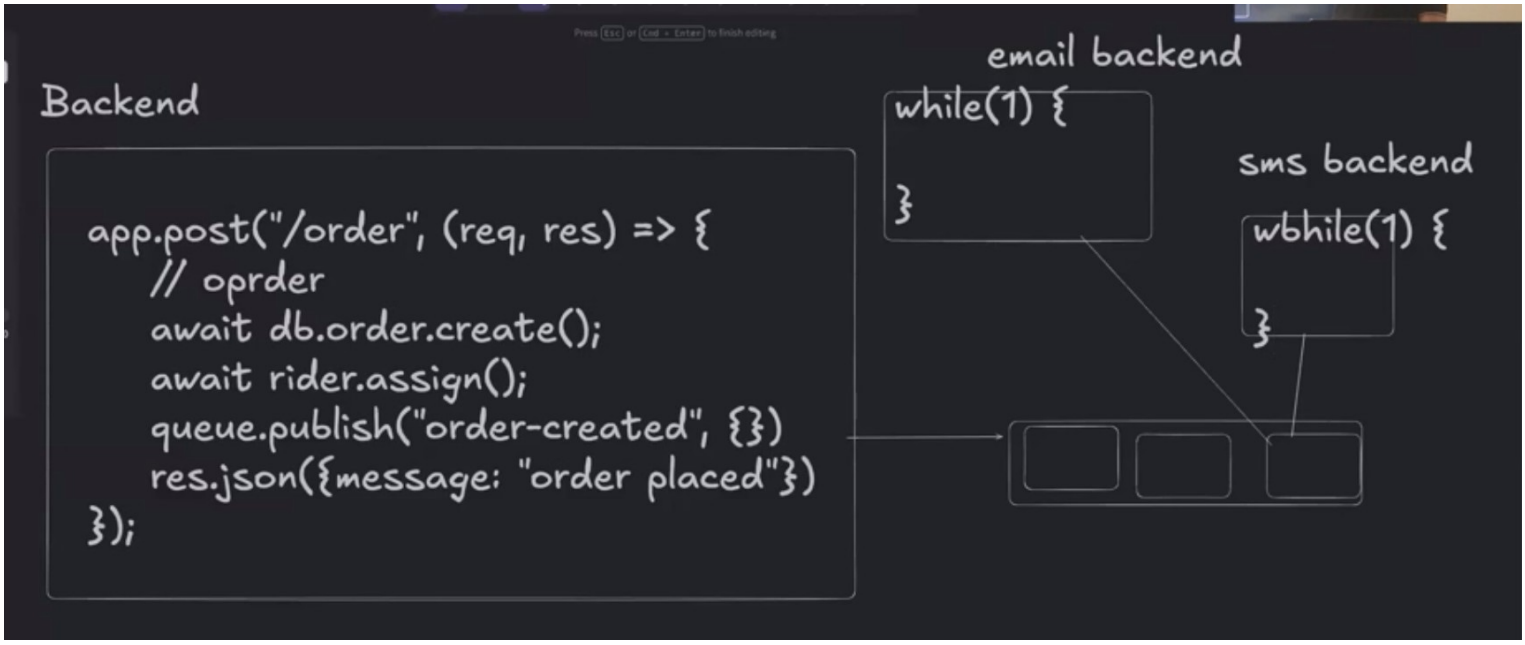
candles schema

SETTING UP THE PROJECT FOR CEX V1
create fe and be and do bun init in the backend folder
then follow the bun with prisma guide
bun add -d prisma @types/pg
bun add @prisma/client @prisma/adapter-pg pg
bunx prisma init

```
const BALANCES = {

}

const ORDERBOOKS = {
    SOL:{},
    BTC:{}
}

app.post("/signup",(req,res) =>{

})

app.post("/signin",(req,res) =>{

    // body={
    //     type:           "market" | "limit",
    //     price:          number | null,
    //     qty:            number,
    //     market_id:      string,
    //     side:           "buy" | "sell"
    // }
    // returns{
    //     orderId:    string,
    //     filledQty:   number,
    //     totalPrice: number
    // }
})

app.post("/order",(req,res) =>{

})
// return the status of an order (partially filled, success, cancle order)
// Also returns the individual files of this order

app.get("/order/:orderId")
app.deLete("/order/:orderId")
app.get("/depth")
app.get("/orders")
```

total price which we are sending is mutliplied with many zeroes to avoid the decimals bcoz it can cause presicision error in js

HOW ONE BACKEND TALK TO MULTIPLE BACKENDS



in this like the order creation and the rider assing thing is more important so we first complete that and the mail and sms is not that important so we put them in a queue ?
But why in a queue?
Bcoz agar 1k req aagyi toh server phat jayega

this is called async backend communication

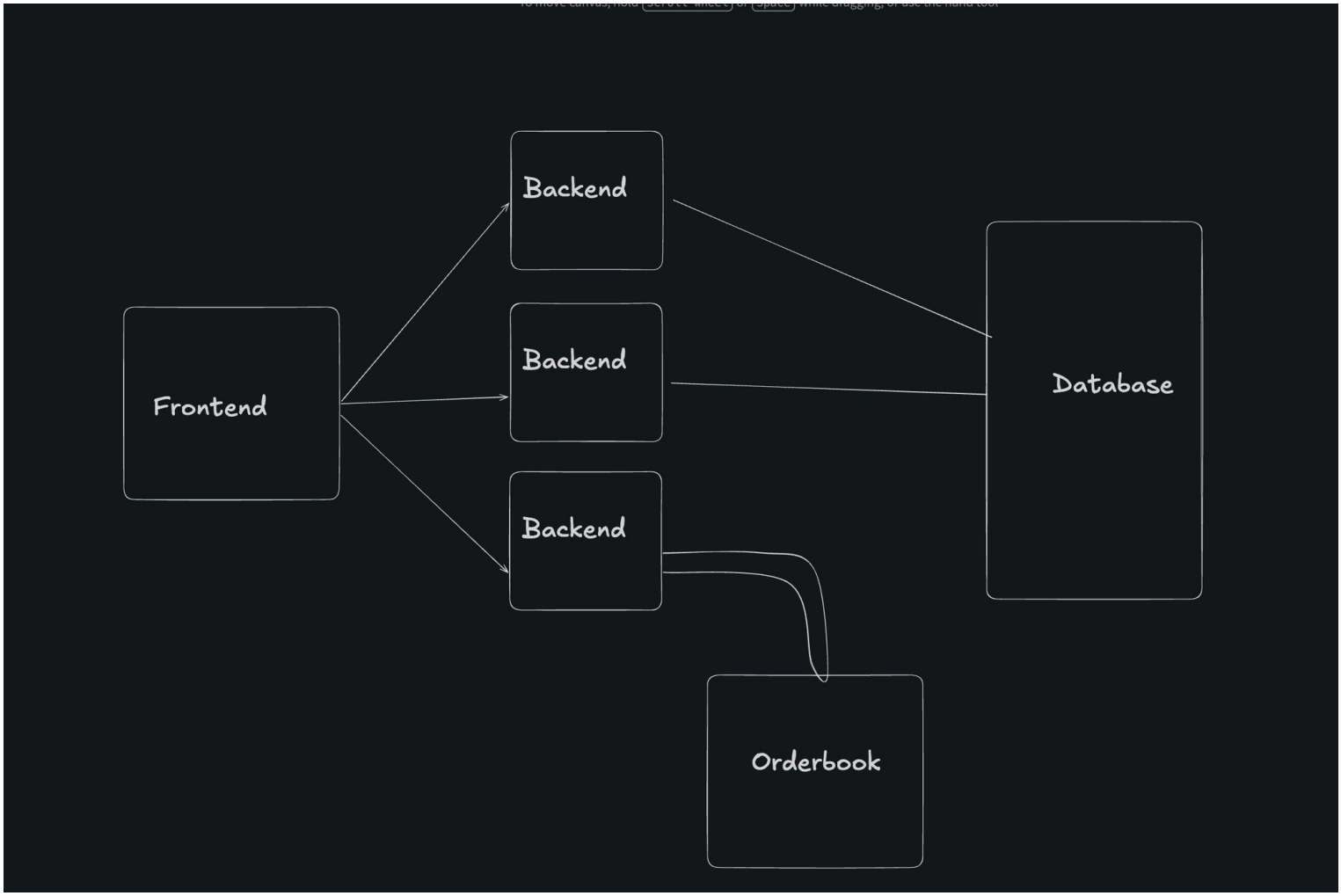
this is also called as workers

sync means we make sure joh request bheji hai dusre backend ko is completed or not

CEX v2 Architecture

Thing to know - queues are faster than db calls.

Benefits of decoupling the orderbook



backend stateless means backend mai koi variable nhi hai jisme haam data store kar rhe ho

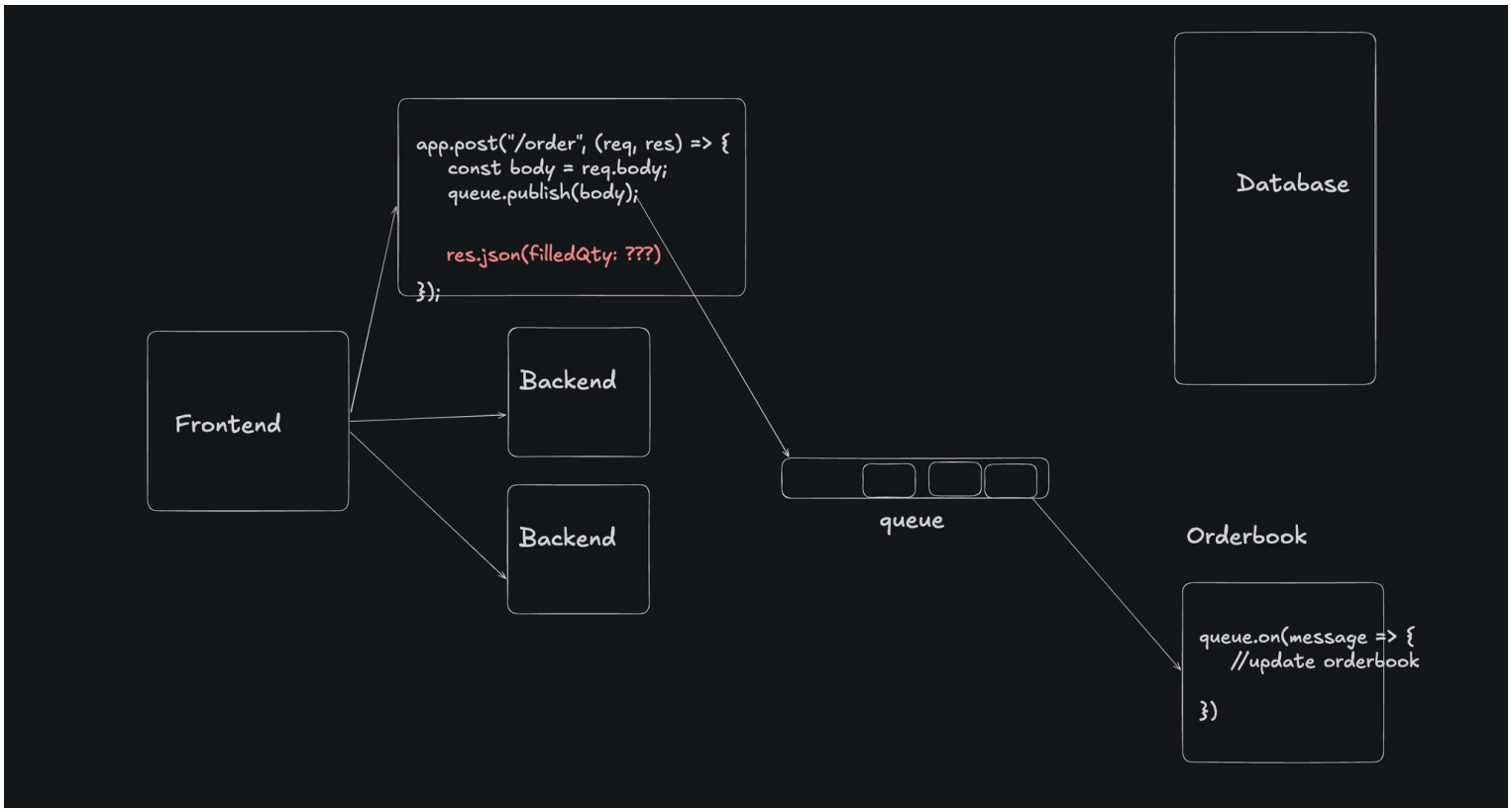
Can horizontally scale the primary backend

1. Primary backend can directly talk to db to take care of simpler calls like signup/signin/get existing orders etc.
2. If orderbook crashes, rest of the backend still remains intact

Problems of decoupling the orderbook

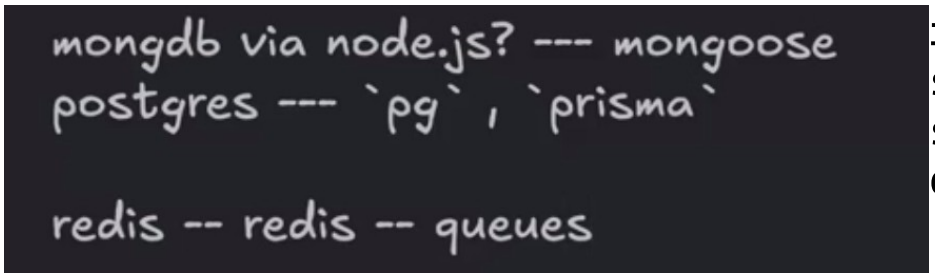
1. Issue still remains the same. The in memory orderbook will get lost if the orderbook ever crashes. We'll come to its fix later

as the data is stored in the memory we can replay the data and then get back the orders and the balances



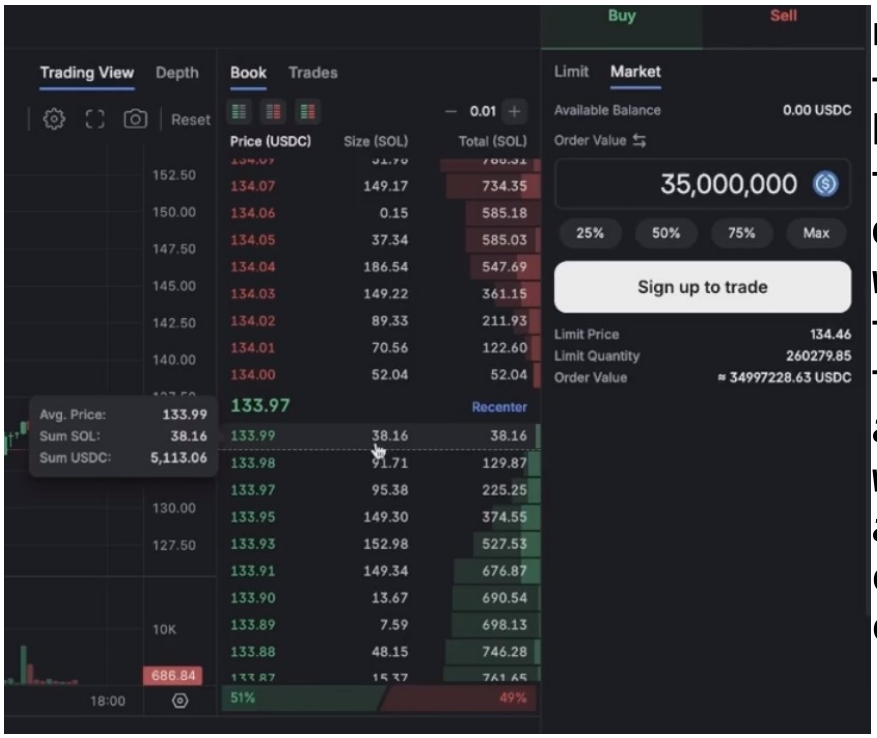
in this we send the request to orderbook using the queue and then the message is sent to the orderbook and then the answer is sent back to the

for queues you can use - reddis(yeh bss aek machine kae aek core par hii chalta hai, but performance high hai), rabbit m,kafka(its better for scaling you can use multiple computers)



redis can be used as something like db, stream and also as a queues

DATABASE MAI HAAM WOH DATA STORE KARTE HAI JOH STRUCTURED HAI AND LIKE IF THE DATA IS NOT STRUCTURED IT CAN BE STORED IN S3 BCOZ USME HAAM KOI QUERIES WAGERA LAGAYENGE HII NHI TOH FHIR KYU DB AND ITS LIKE HAME BSS DATA DUMP KARNA HAI DIRECT



now if I put like a order for like the money we have to divide that with the price and then the quantity which we get is what is the size which we take from that person and that size we eat up
and if the size is more we get is more than available we eat the size of the upper available one also

Jargon

Markets

Whenever you trade, you swap one asset for another. Every pair that you can trade is called a market.
For example -

1. ETH-USDC

ASSET ON ONE SIDE AND THE CURRENCY ON THE OTHER SIDE

2. TATA-INR
3. ETH-BTC (Usualy a combination of two markets, ETH-USDC and BTC-USDC)

Base asset

The asset that is being traded (TATA).

Quote asset

The asset it is being traded with.

Orderbook

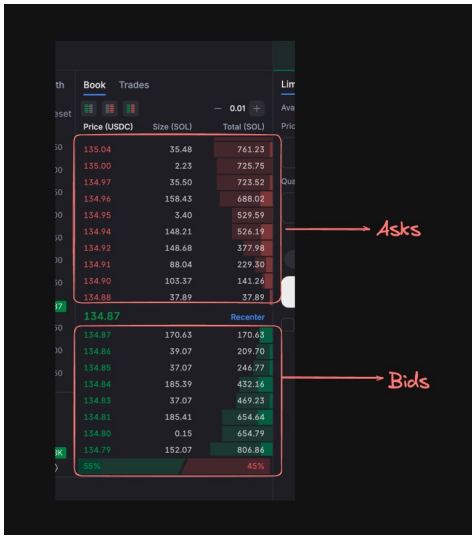
A list of currently available limit orders that people have placed. The more the number of orders, the more liquid the exchange is considered.

Bid

A limit order of the following type -
I will buy 5 stocks of TATA for Rs 200/stock

Ask

A limit order of the following type -
I will sell 5 stocks of TATA for Rs 201/stock



Market makers

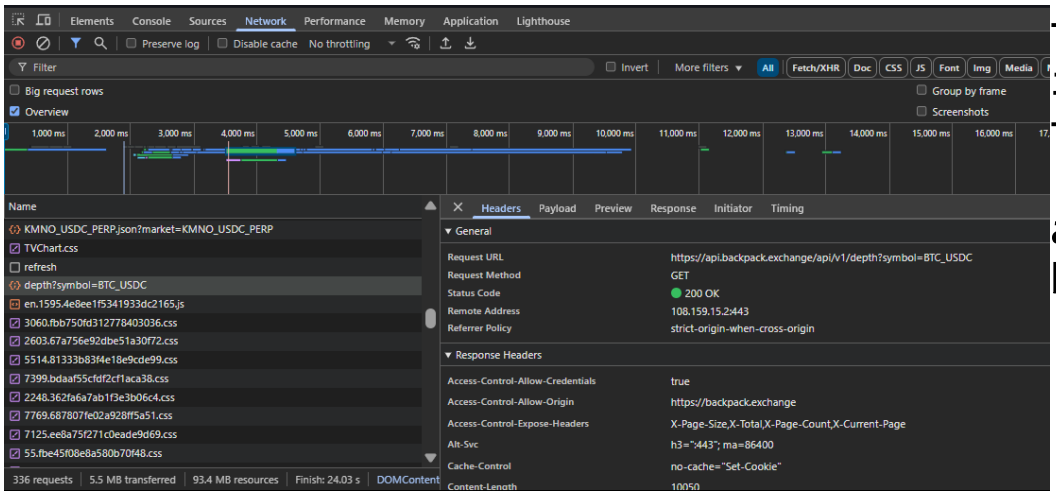
Big companies that keep the orderbook liquid (that have a lot of open orders on the orderbook)

SPREAD MEANS THE DIFF BETWEEN THE BID AND THE ASK

ILLIQUID EXCHANGE MEANS WHEN THE SPREAD IS VERY HIGH WHICH IS IN THE CASE OF THE WAZIRX AS IN THE INDIAN MARKET NO ONE WANTS TO TRADE AS THERE IS SO MUCH TAX INCENTIVE ON THAT

NO MARKET MAKERS WANTS TO COME AND MAKE TRADE

MORE LIQUIDITY MEANS BETTER PRICES AND BETTER THE EXCHANGE IS

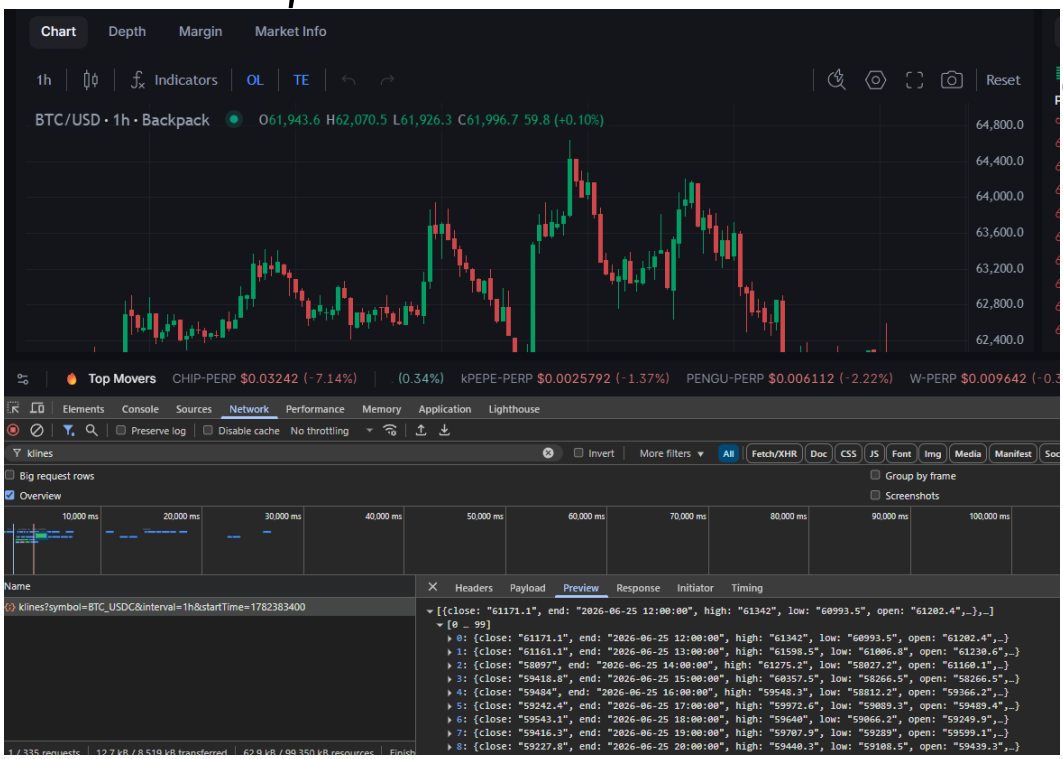


this url sends the initial orderbook to the website

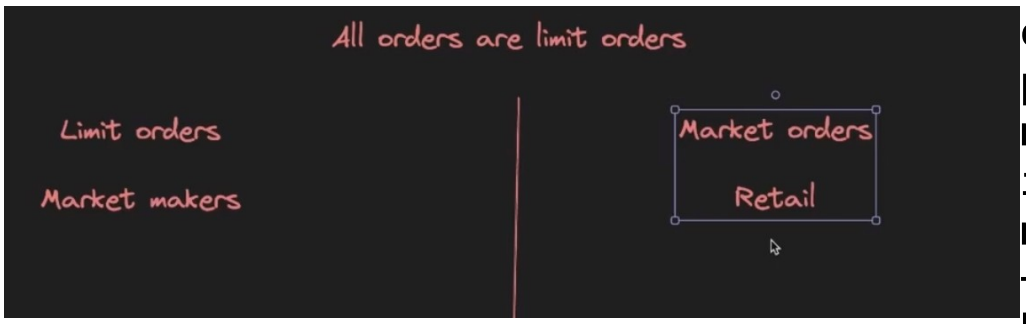
around like 50 bids and 50 asks

binance/docs has this all documented

klines sends all the data for the graph and the candle can be for 1 hr,30 min etc



each candle which is shown is for a one hour interval



even though we are placing like a market order imagine you put a market order like for 8 cr you wanna buy houses and current price for one house is 1.5 cr so you can buy like 5-6 houses so which means you should get that and imagine if those other people

Backend Routes

Users

POST /api/v1/signup - Lets a user signup

POST /api/v1/signin - Lets a user signin, returns a jwt

Orders

POST /api/v1/order - Lets a user place an order. Returns orderId and fill status

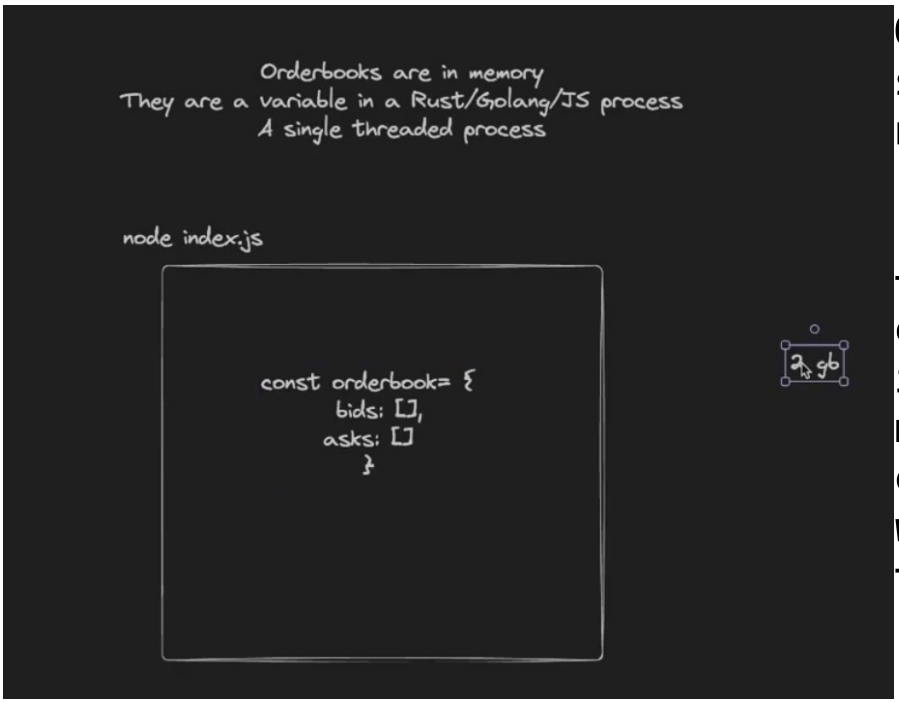
```
type: limit | market
kind: buy | sell
price: number
quantity: number
market: TATA-INR
```

GET /api/v1/order/:orderId - Returns fill status

DELETE /api/v1/order/:orderId - Removes an order off the book (if it isn't filled yet)

POST /api/v1/order/quote

```
kind: buy | sell
quantity: number
market: TATA-INR
```

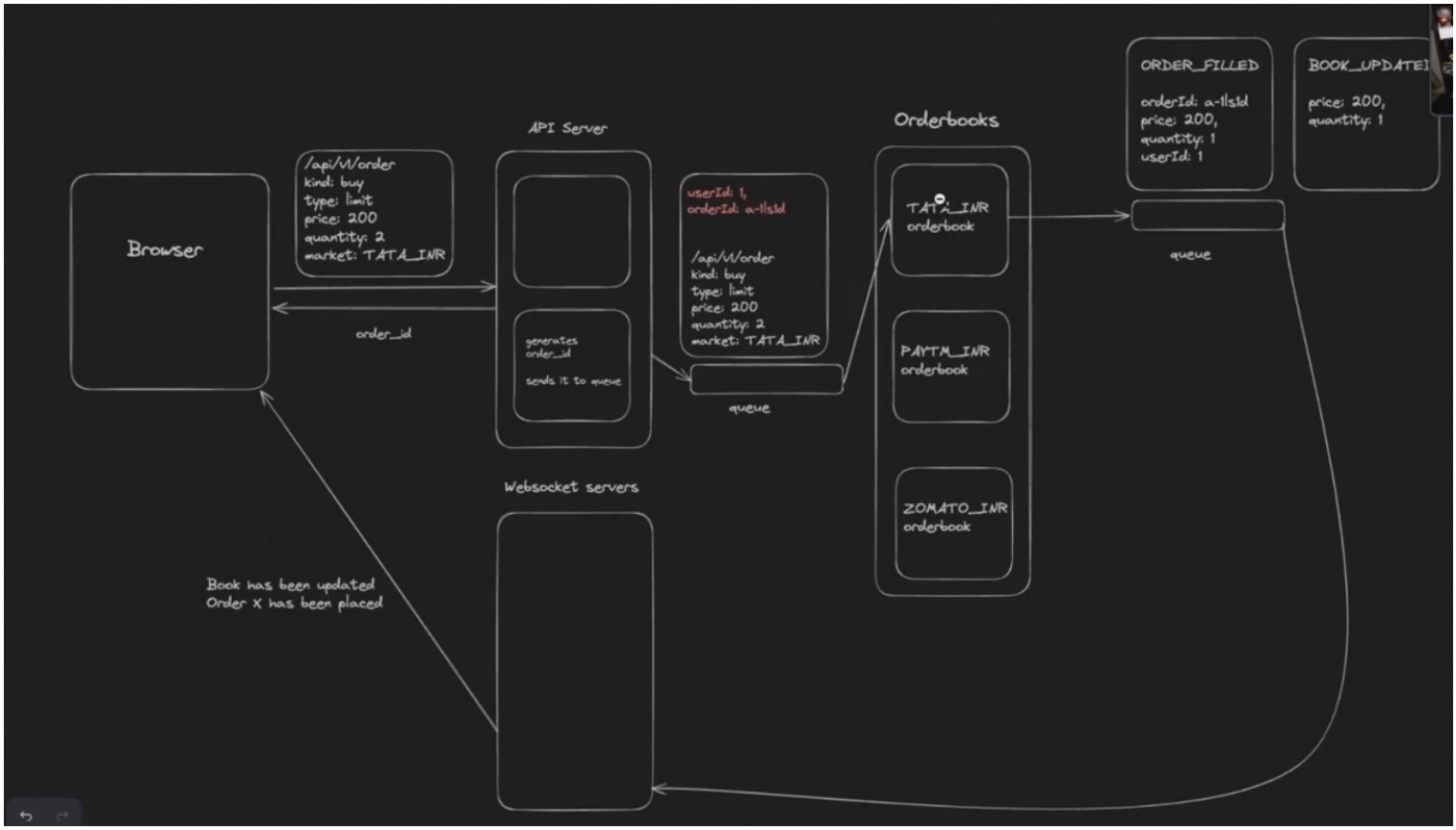


Orderbooks are always stored in memory and never in db

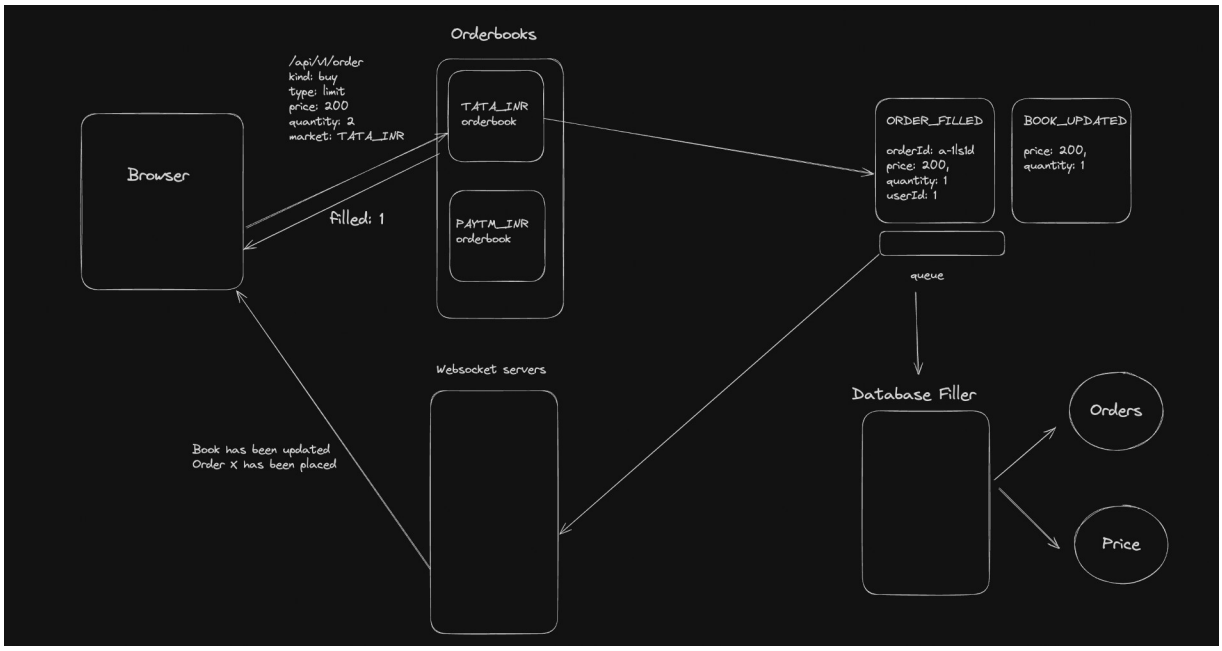
fastest way to make changes in a orderbook is to keep the data in memory rather than calling from db that’s why the exchange needs to be very fast

the issue is if the server dies the variable dies with it

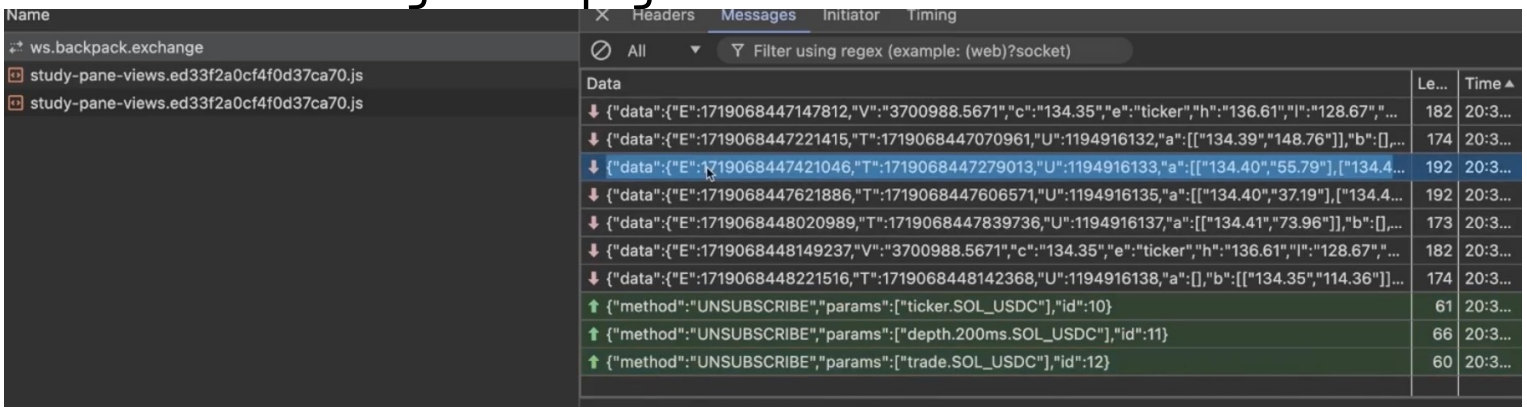
Backend Architecture



Backend Architecture (simpler)



whenever we change the page we unsubscribe the channel



WHY SOMEONE WILL COME TO YOUR TRADE AND WILL TRADE?
Due to low trade price and also at the starting the companies gives like shares of the company or maybe some incentive or something to the heavy investors

if somehow the orderbook goes down, then we can get back the previous state of the orderbook by replaying the event of the queue and use it to make the orderbook again

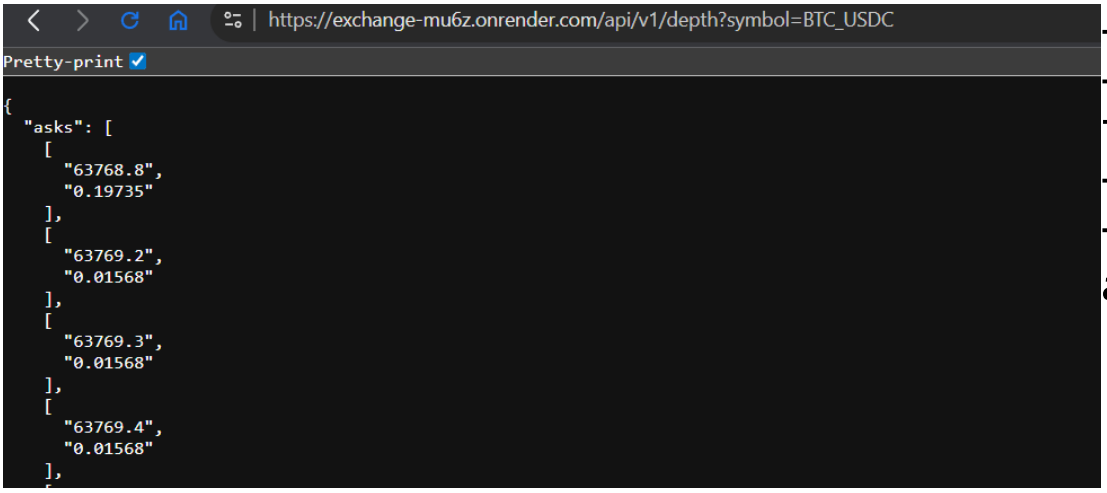
also no more trade will happen

now to recreate the orderbook we just take the snapshot(bids and the asks looks like) of the orderbook like every hour or day and using queue we remake the orderbook

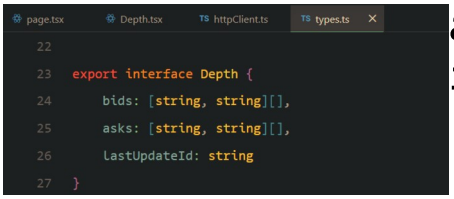
/depth == url gives all the asks and the bids

flex flex-col == means place one below the other

ALSO I HAVE IMPLEMENTED ONE THING LIKE I CANNOT DIRECTLY USE THE BACKPACK EXCHANGE API TO ACCESS THE DATA SO I HAVE CREATED A PROXY WHICH IS DEPLOYED SO THAT I CAN ACCESS THE BACKPACK API AND RENDER THE DATA ON TO THE WEBSITE



this depth url has the data which like is a array of tuples in which the asks and bids are sent

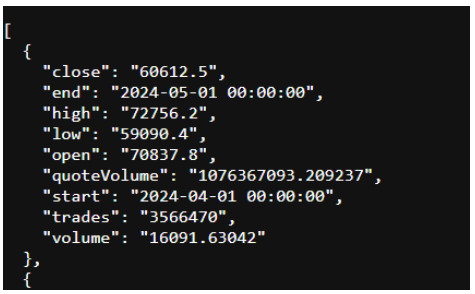


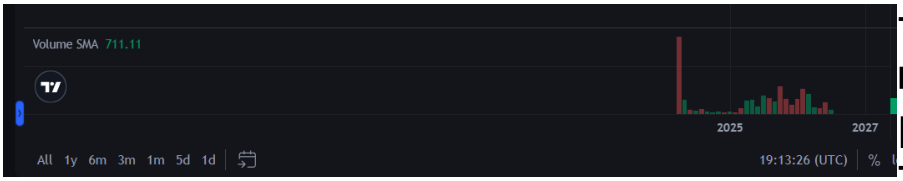
and like the type of the depth is mentioned in the types.ts file

the thin line means the price gone highest there and the top of the candle is where the market closed and the bottom where the market opened

klines sends the candles data
https://exchange-mu6z.onrender.com/api/v1/klines?symbol=BTC_USDC&interval=1month&startTime=917825400

this above url returns the data





the quote volume mentioned shows how many people were trading there

useeffect means whenever the depth function is called the useeffect will run and will workflow

Depth file has all the ui things for the orderbook

Asktable has the logic of maintaining price, size and total

Bidtable has the logic of maintaining price of the below orderbook

logic is simple you can understand the comments explains it

```
export const AskTable = ({ asks }: { asks: [string, string][] }) => {
    let currentTotal = 0;
    const relevantAsks = asks.slice(0, 15);
    /*
     * 129.93 10
     * 129.94 5
     * 132.96 3
     * 132.97 253.03
     */
    relevantAsks.reverse();
    /*
     * 132.97 253.03     270
     * 132.96 3    18
     * 129.94 5    15
     * 129.93 10   10
     */

    let asksWithTotal: [string, string, number][] = [];
    for (let i = relevantAsks.length - 1; i>=0; i--)  {
        const [price, quantity] = relevantAsks[i];
        asksWithTotal.push([price, quantity, currentTotal += Number(quantity)]);
    }
    const maxTotal = relevantAsks.reduce((acc, [_, quantity]) => acc + Number
    (quantity), 0);

     /*
     *    129.93 10   10

     * 129.94 5    15
     * 132.96 3    18
     * 132.97 253.03     270
     */
    asksWithTotal.reverse();
        /*
        * 132.97 253.03     270
        * 132.96 3    18
        * * 129.94 5    15
     *    129.93 10   10

     */

    return <div>
        {asksWithTotal.map(([price, quantity, total]) => <Ask maxTotal=
        {maxTotal} key={price} price={price} quantity={quantity} total={total} /
        >)}
    </div>
}
```

same for bid table read the comment to understand we are like making the orderbook arranging the values from lowest to highest and also maintaing the total values

```jsx
function Ask({price, quantity, total, maxTotal}: {price: string, quantity:
string, total: number, maxTotal: number}) {
    return <div
    style={{
        display: "flex",
        position: "relative",
        width: "100%",
        backgroundColor: "transparent",
        overflow: "hidden",
    }}
>
    <div
        style={{
        position: "absolute",
        top: 0,
        left: 0,
        width: `${(100 * total) / maxTotal}%`,
        height: "100%",
        background: "rgba(228, 75, 68, 0.325)",
        transition: "width 0.3s ease-in-out",
        }}
    ></div>
    <div cLassName="flex justify-between text-xs w-full">
        <div>
            {price}
        </div>
        <div>
            {quantity}
        </div>
        <div>
            {total?.toFixed(2)}
        </div>
    </div>
    </div>
}
```

this renders the graph part on the website

For the visual chart most company uses the paid thing provided by the trading view which gives access to paid library and there free one called lightweight-charts

to use there premium library you have to mail them and then get the access

the **TradeView** page contains the code for the chart

like earlier we used to use the **id** for using the component in a div but better way of using that is by using **useref** hook

like earlier we used to use document.getElementById thing

```tsx
import { useEffect, useRef } from "react";
import { ChartManager } from "../utils/ChartManager";
import { getKlines } from "../utils/httpClient";
import { KLine } from "../utils/types";

export function TradeView({
  market,
}: {
  market: string;
}) {
  const chartRef = useRef<HTMLDivElement>(null);
  const chartManagerRef = useRef<ChartManager>(null);

  const init = async () => {
    let klineData: KLine[] = [];
    try {
      klineData = await getKlines(market, "1h", Math.floor((new Date().getTime() - 1000 * 60 * 60 * 24 * 7) / 1000), Math.floor(new Date().getTime() / 1000));

    } catch (e) { }

    if (chartRef) {
      if (chartManagerRef.current) {
        chartManagerRef.current.destroy();
      }
      const chartManager = new ChartManager(
        chartRef.current,
        [
          ...klineData?.map((x) => ({
            close: parseFloat(x.close),
            high: parseFloat(x.high),
            low: parseFloat(x.low),
            open: parseFloat(x.open),
            timestamp: new Date(x.end),
          })),
        ].sort((x, y) => (x.timestamp < y.timestamp ? -1 : 1)) || [],
        {
          background: "#0e0f14",
          color: "white",
        }
      );
      //@ts-ignore
      chartManagerRef.current = chartManager;
    }
  };

  useEffect(() => {
      init();
  }, [market, chartRef]);

  return (
    <>
      <div ref={chartRef} style={{ height: "520px", width: "100%", marginTop: 4 }}></div>
    </>
  );
}
```

like in this we are referening the chart thing in the div so that it can be used from that

kline we use for the graphs

we are parsing the data into float because the data came was in string only
the volume chart is present in the pro version which we cannot use for now

NOW TO SHOW THE CHANGES ON THE FRONTEND WE CAN USE THE WS SERVER FROM WHICH WE HAVE TO GET THE DATA SUCH AS
SOL_USDC@TRADE == all the realtime trades which happenend
SOL_USDC@DEPTH == all the current realtime orderbook changes
SOL_USDC@TICKER == current price

so whenever we go to a market we subscribe to ws.backpack.exchange and unsubscribe when we leave it

in the code the signaling Manager is the ws connection thing

like right now we have created a ws server which gets all the data which is required and also we can do like create a different ws server for trade, depth, ticker

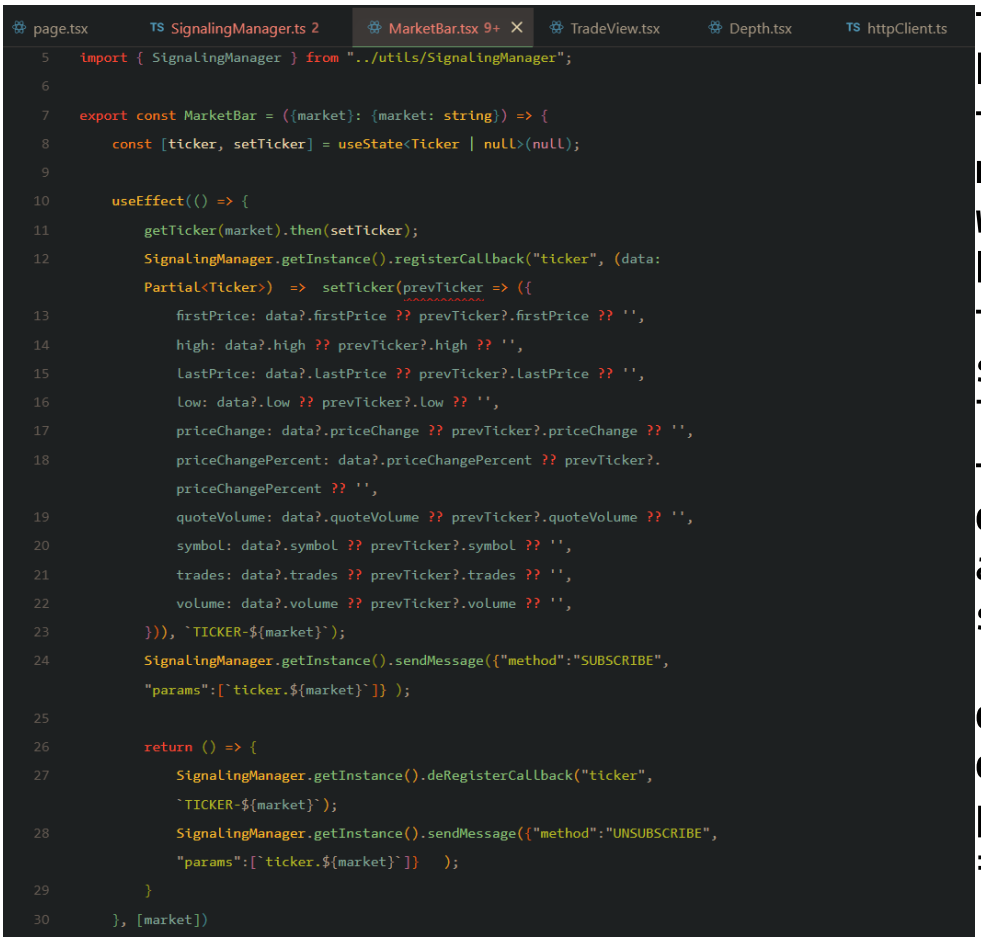
page.tsx | SignalingManager.ts 2 | TradeView.tsx | MarketBar.tsx 1 | Dep

```ts
import { Ticker } from "./types";

export const BASE_URL = "wss://ws.backpack.exchange/"

export class SignalingManager {
    private ws: WebSocket;
    private static instance: SignalingManager;
    private bufferedMessages: any[] = [];
    private callbacks: any = {};
    private id: number;
    private initialized: boolean = false;

    private constructor() {
        this.ws = new WebSocket(BASE_URL);
        this.bufferedMessages = [];
        this.id = 1;
        this.init();
    }

    public static getInstance() {
        if (!this.instance)  {
            this.instance = new SignalingManager();
        }
        return this.instance;
    }
```

for ws we have created a singleton class and all these variables can be accessed using the name of the class and like whenever the getInstance function is called the ws connection is established and only do subscribe and unsubscribe thing

```ts
    async registerCallback(type: string, callback: any, id: string) {
        this.callbacks[type] = this.callbacks[type] || [];
        this.callbacks[type].push({ callback, id });
        // "ticker" => callback
    }

    async deRegisterCallback(type: string, id: string) {
        if (this.callbacks[type]) {
            const index = this.callbacks[type].findIndex(callback => callback.
            id === id);
            if (index !== -1) {
                this.callbacks[type].splice(index, 1);
            }
        }
    }
```

this is the imp thing which tells like whenever the subscribe is done this pushes the new data to the ticker and all and the below function remove that data



this is the important bit as this is triggered whenever the market changes and like whenever it changes it brings the new data and the subscribe msg is sent and when the user leaves the return function is called and deregisters the data and unsubscribe msg is sent

data.firstPrice == new data
prevTicker.firstPrice == old data
‘’ == null

now this can update the current price and to change the orderbook also just add the signaling code to the orderbook code

IN THE COMPANIES THEY USE SOMETHING CALLED AS **CHAIN ANALYSIS** WHICH LIKE ENSURE THAT THE MONEY IS NOT COMING FROM NORTH KOREA OR SOMETHING BEFORE TRANSFERRING THE MONEY AND ALL

the code is similar to the socket io client code

Understood the jargon needed to build an exchange

1. Base asset
2. Quote asset
3. Orderbooks
4. Liquidity
5. Limit orders
6. Market orders
7. KLines

2. Built the frontend for an exchange
    1. Proxied requests via https://exchange-proxy.100xdevs.com/api/v1/depth?symbol=SHFL_USDC
    2. Created a simple frontend which had depth , orderbook, ticker
    3. Assignment - Creating the trades tab on the frontend

What we're doing today

Building the backend ourselves.

HTTP Endpoints

The endpoints we used last week were -

1. Get klines - https://exchange-proxy.100xdevs.com/api/v1/klines?symbol=SOL_USDC&interval=1h&startTime=1718957562&endTime=1719562362
2. Get depth - https://exchange-proxy.100xdevs.com/api/v1/depth?symbol=SHFL_USDC
3. Get Tickers - https://exchange-proxy.100xdevs.com/api/v1/tickers

Websocket streaming

The streams we support include

1. trades@MARKET

2. depth@MARKET
3. ticker@MARKET

Counter intuitive things -

1. Orderbook is in memory
2. User balances are in memory
3. Locking balances in memory

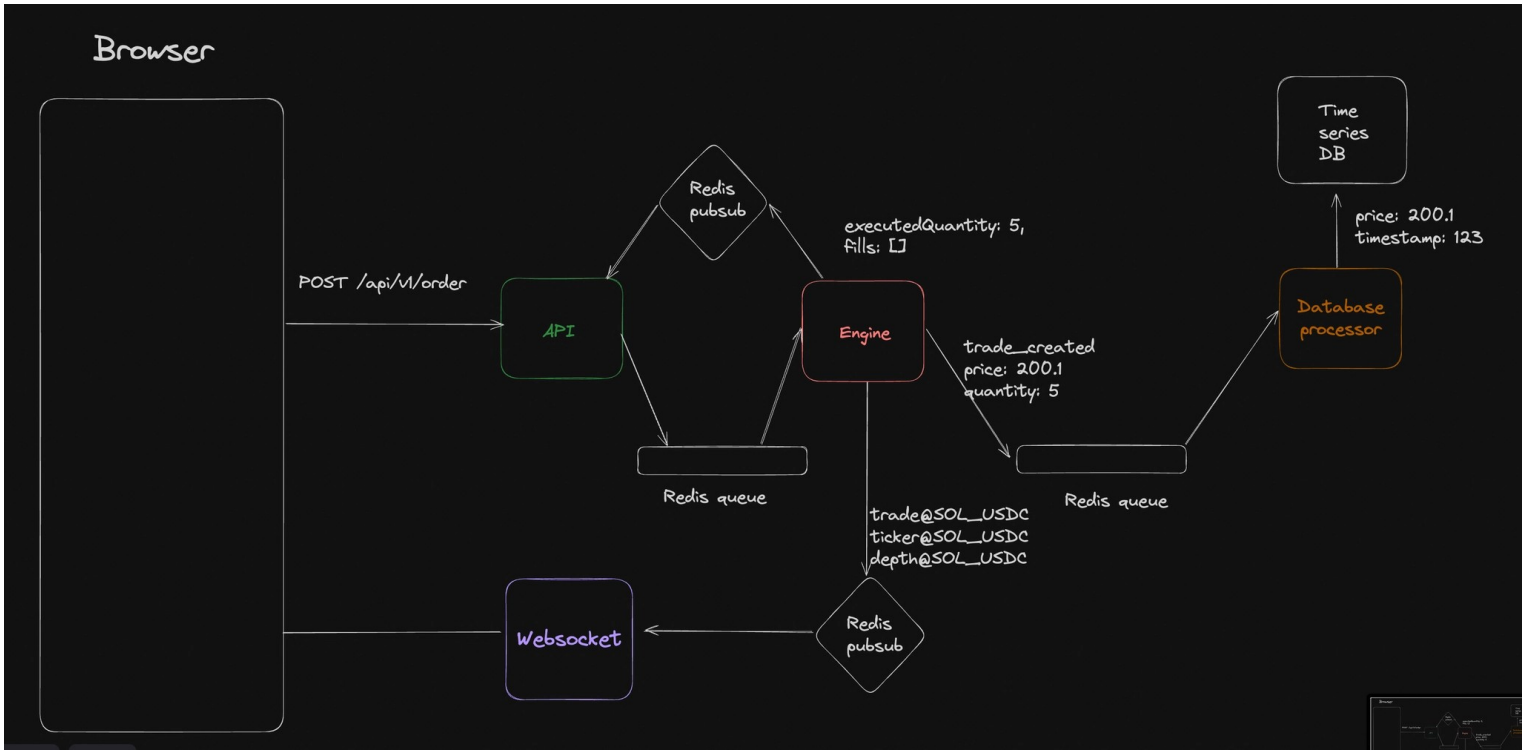
WHAT IS A BAD EXCHANGE?
A bad exchange means which stores its orderbook not in memory but in database

Backend Architecture

We have the following services -

1. **API - An API Server the user sends HTTP requests to**
2. **Engine - Runs various market orderbooks, stores user balances in memory**
3. **Websocket - Websocket server that user subscribes to real time events from**
4. **DB Processor - Processes messages from the Engine and persists them in the DB**
5. Frontend - NextJS app (same as last week, only the URLs would change)
6. Market maker (mm) - Places random orders to keep the book liquid
7. Redis - Queue and pub sub
8. TimescaleDB - Creates buckets of klines based on price feed

There should ideally be a primary database like postgres as well that stores user info/orders/trades etc.



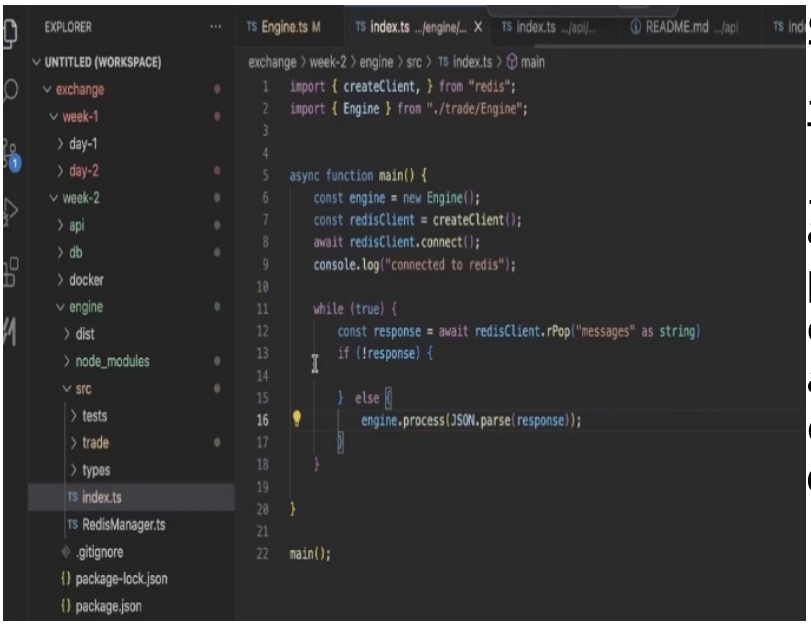
Engine == orderbook manager
the orders are managed sequentially and not like parallely this can lead to concurrency issues

The Engine manages ==
the orderbook, balances of all the users, also flips the balances whenever a match happens between the user,
like maintain the balance of the users

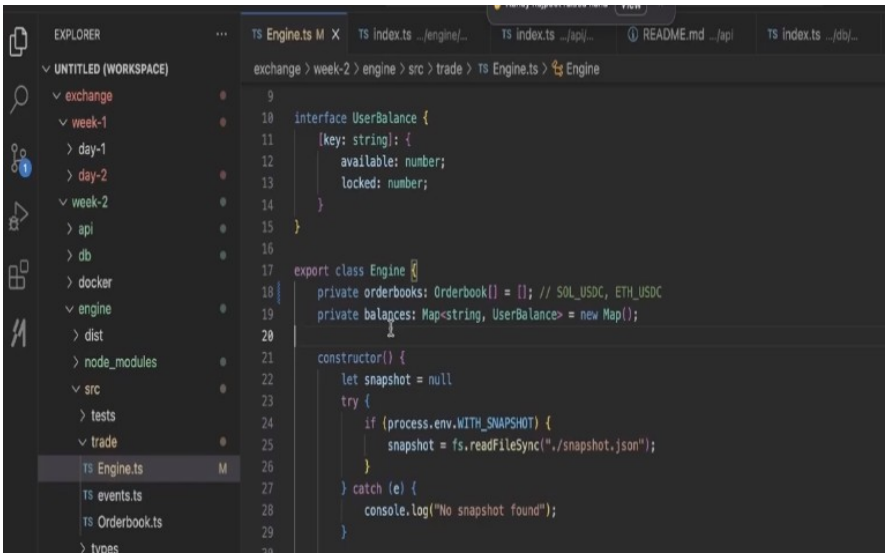
NOW TO CREATE THE STATE OF THE QUEUE IF THE ENGINE GOES DOWN JUST RECREATE THE STATE FROM THE GENESIS USING THE QUEUE AND YOU WILL GET BACK THE ORIGINAL QUEUE

also like to fix this you create a snapshots of the state of orderbook so that you need not to create a whole re run of queue for the state of the orderbook

SNAPSHOTS == VERY BIG JSON BLOB and put it on S3 and do it every like 1 hr(Better way of getting back the state of the orderbook)



See in this like the engine just creates a new engine, then a redie queue, and then just connect to it and then a foreever loops runs which pulls the data and in that engine is never down
and also there is no async or await bcoz we are not doing any db calls or smthg



Now the engine actually manages all the orderbooks and all the balances

Also like we right now using node js processes, if we want to deligate the tasks we can use like the rust processes which will assign the tasks to different cores, it can like spawn threads

multiple threads can make the system faster

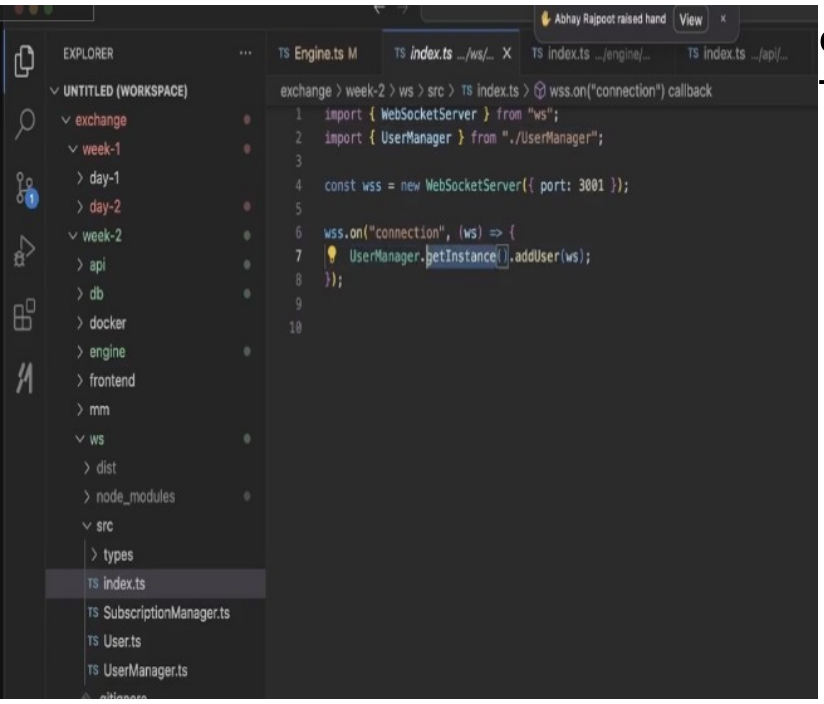
GOOD QUESTION TO ASK
Like if someone has access to DB he can change the balance right?
No because there are multiple datacenters which are running the engine and like to verify the withdrawl every engine must approve the changes and then only the money can be taken out

KAFKA SHINES FOR IT DISTRIBUTED NATURE, WE CAN RUN LIKE MULTIPLE KAFKA SERVERS TO DISTRIBUTE THE LOAD, ALSO BETTER FAULT TOLERANCE, ALSO SLOWER(BECAUSE ITS REPLICATING DATA WHICH MAKES IT SLOWER) AND REDIS IS LIKE SINGLE THREADED IN WHICH A SINGLE PROCESS IS RUNNING AND HANDLING THE REQUEST

NOW TO LIKE INFORM ON THE FRONTEND USING THE API LIKE WHICH ALL TRADES ARE PRESENT AND ALL WE USE PUBSUB OR MAYBE REDIS QUEUE NOT OPTIMAL THO
like to make a Backend talk we usually use a queue or a pubsub its not a good way to communicate using http requests

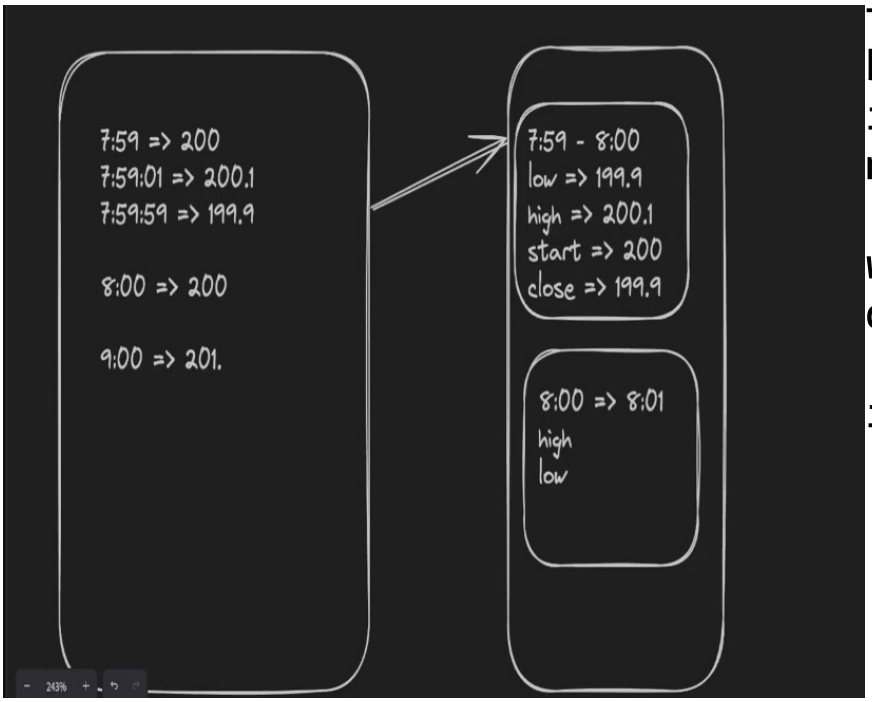
this is done to like see the changes or how many orders are placed and all for that we have to like send the request that these all request are processed and these are not processsed

the websocket server which is present gives the real time updates of the graphs, prices and all using the SUBSCRIBE and UNSUBSCRIBE thing for that we sends a request to the redis pubsub which asks for data from the engine and show to us



creating only one instance for the person required

Another thing is like what all graphs and all data which we store in db and how are they stored



they are stored simply only but converted into the form in which the graph candle making required

which is maintained by the database processsor

in time series db

WHENEVER WE DO A AWAIT ON A FUNCITON IT RETURN TYPE IS ALWAYS A PROMISE == ASYNC FUNCTION IS JS

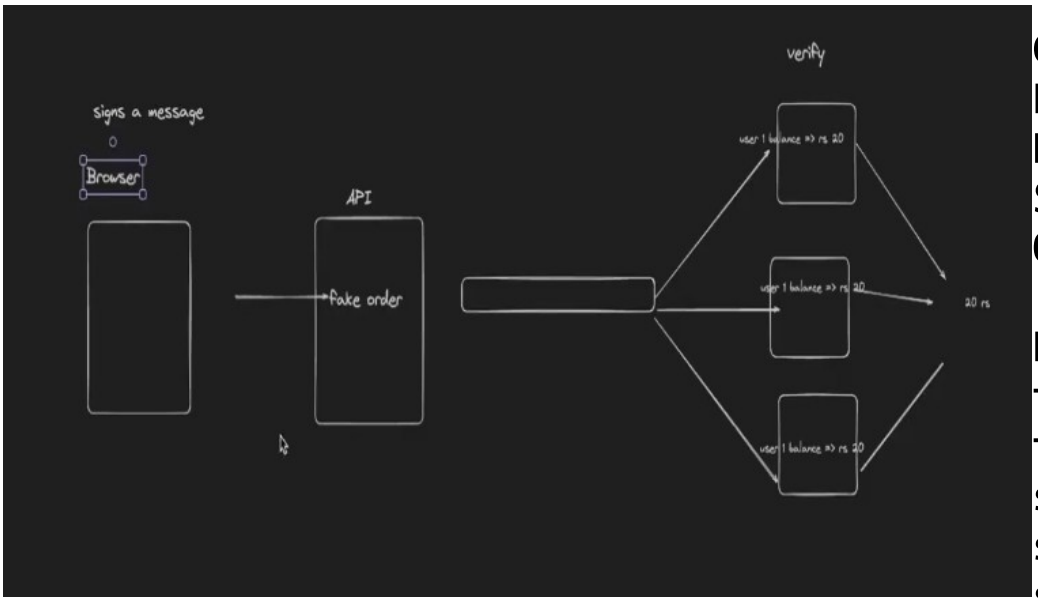
LIKE THE USER WHEN MAKE A ORDER OR SOMETHING IT GIVES A CLIENT ID AND WHEN THE ENGINE PROCESS IT AND SEND BACK THE DATA USING THE PUBSUB IT RETURNS THE CLIENT ID ALSO

HOW TO START THIS PROJECT??
cd docker
docker-compose up

and run all the other processes

YOU SHOULD ALWAYS CREATE TESTS FOR THE ORDERBOOK BCOZ ONE MISTAKE CAN CAUSE A LOT

for like 20 req a second its ok only but the issue is if the number of markets increase and like if users increase we shift to rust process then and make the system faster and better



CAN SOMEONE FAKE THE PRICE ORDER BY HACKING THE API SERVER ITS ONLY ONE ONLY?

No because before the api server when the browser is sending a req it sends it but with signing the message with its private key which protects the thing and limits the dev from doing fake transaction

IN EXCHANGE REDIS IS USED AS THE QUEUE

PUBSUB IS LIKE CONNECTING MULTIPLE WS SERVERS AT ONCE AND DIVIDES THE LOAD AMONG MULTIPLE SERVERS