A technical introduction to the Openbook DEX

28 December 2022

This document offers a no-frills, introductory technical explanation of the core components of the Openbook DEX.

We won't get too deep into the details – this will cover what's necessary to make a single trade on a DEX and not much more! And we'll link to the relevant source code and resources throughout.

Note that the Openbook community has ambitious plans for a large ecosystem. We recommend giving the white paper a read before continuing with this document – it'll give you a sense for what that larger ecosystem looks like and where the Openbook DEX fits into it.

Solana, in a nutshell

SPL tokens, in a nutshell

The DEX, at some length
- The general setup
- Placing orders
- Matching orders
- Consuming Events
- Settling funds
- Cancelling

Interacting with the DEX
- Libraries for programmatic access
- Using a GUI
- Fees, (M)SRM, and Nodes

What's next?

Solana, in a nutshell

The Openbook DEX is a program on the Solana blockchain. Solana was selected because of its unprecedented performance: it introduces many technical innovations, allowing it to offer extremely low transaction latency (by blockchain standards), high throughput, and low fees.

As a ballpark: Solana has block times of 400ms, and can currently process roughly 50,000 transactions per second. This allows Openbook to handle hundreds of orders per second per market.

Check out their site for more information, and dig into their docs (which are actually pretty digestible!) to understand the fundamentals.

Throughout this doc we'll reference some important concepts about and features of the Solana blockchain. Here's the first: SPL tokens.

SPL tokens, in a nutshell

Tokens on Solana are instrumented as instances of the SPL Token Program. An SPL token on Solana is analogous to an ERC20 token on Ethereum. Solana programs, more generally, are analogous to smart contracts.

There are a number of open-source wallets that make managing token holdings easy! So in practice you needn't worry about these details. But if you're curious:

To have balances in an SPL token, you must have the private key to the Owner of an SPL token account whose Mint (akin to Contract Address) corresponds to that SPL token.

Here's an example of what that looks like: The Owner (BpU4BSqq8FVcZYrtparzH2gqx2CPx1TbXpTG3UAoKb2M) holds 99927 tokens for the Mint (AXqqqfBm33bk4tAfv6hYQS5L49vHcvMUXLuGJWg2ujV7), held in a special program-controlled account (Dp2J3c1pWS1wZZuihZkWwE9tK9cQZW9ykP6uFgte95Er).

The SPL Token program grants special rights to the Owner of this account: to withdraw the funds in that account, one must issue instructions on-chain to the SPL token program, signing with the Owner's keypair.

The DEX, at some length

The Openbook DEX is a fully trustless, fully decentralized exchange. Crucially, it supports full limit orderbooks and blazing fast order placement, cancellation, and fund management. Its goal is to bring the best features of centralized exchanges to DeFi while remaining fully trustless and transparent. It is, as far as we know, the first high-throughput low-latency exchange that has a fully on-chain orderbook and matching engine.

Like SPL Tokens, the Openbook DEX is a program on Solana. We'll dive into what that program does, mechanically, with brief digressions into why it's even possible for the DEX to be as performant as it is.

The general setup

Trade lifecycle

The high level lifecycle of a trade is:

- Placing orders: A user funds an intermediary account (their OpenOrders account) from their SPL token account (wallet) and adds an order placement request to the Request Queue
- Matching orders: The request is popped off of the Request Queue and processed: it's placed on the Orderbook. Any resulting trades get reported in the Event Queue.
- Consuming events: Events are popped off of the Event Queue and processed: OpenOrders account balances are updated as the result of the trade
- Settlement: Users can settle free funds from their OpenOrders back to their SPL token account (wallet) at any time

This whole cycle takes just a second or two.

Important accounts

The DEX operates by coordinating interactions between many Solana accounts. Under the hood, a Solana account is a fixed length byte sequence that's durable on the blockchain, each with a unique address.

Some of the Solana accounts involved in the DEX are user specific, and others are global singletons.

Note that, while the funds go through a few accounts, none of these have an outside admin that could confiscate the funds or choose trades; they are all programmatically controlled and only able to process orders and then send the funds to the correct owners. Openbook is fully non-custodial.

Here are the essential and interesting global accounts:

- Request Queue: this account maintains all submitted but unprocessed order placement and cancellation requests.
- Orderbook: speaks for itself. There's one account for bids and another for asks, but for simplicity we'll refer to both of them collectively as Orderbook going forward.
- Event Queue: reports the list of outputs from order matching: trades, for instance

… and the essential but uninteresting global accounts:

- Market: holds metadata about the market (e.g. important constants like tick size and references to the other accounts below)
- Base Currency Vault: an SPL token account that holds base currency balances
- Quote Currency Vault: an SPL tokena account that holds quote currency balances

And the user-specific accounts: to interact with the DEX, a given user must create an OpenOrders account. This account stores the following:

- How much of the base and quote currency that user has locked in open orders or settleable
- A list of open orders for that user on that market

Some of the details in the following sections might evolve and grow, but a lot of the core design is likely to remain similar. For the latest, refer to open-source resources and docs.

Placing orders

To place an order, a user submits a Place Order instruction to the DEX program, passing:

- Order details: the market, size, price, order type, side
- Their OpenOrders account on that market
- Their SPL token account for the market's base currency (if selling) or quote currency (if buying)

The DEX program then:

- Transfers funds:
    - Determines the max funds required for this order (the size if selling, or size * price if buying)
    - Notes the free balance for the corresponding currency indicated in the OpenOrders account
    - Transfers the difference between the required amount and free OpenOrders balance from the SPL token account to the Base Currency Vault or Quote Currency Vault.
    - Increments the OpenOrders account's total balance for the corresponding currency by the amount transferred
- Increments and retrieves a sequence number from the Request Queue: this, along with the order's price, determines the new order's ID.
- Adds an item to the Request Queue specifying the order details (size, price, order type, side)
- Adds an item representing the new order to an array in the OpenOrders account in an available slot

Some particulars of Solana that govern this process:

- If any individual step listed above fails, the whole transaction is dropped and has no effect
- The step that involves transferring from the user's SPL token account uses a cross-program invocation to the SPL Token program: the DEX program asks

the SPL token program to do the transfer, given that the SPL token account holder has given permission to do so; it can't do the transfer itself
- The transaction must therefore be signed by the owner of the SPL token account (which also needs to be the owner of the OpenOrders account!). Note that the meaning of "owner" here is subtle: it's not that the user needs to hold the private keys to each of those accounts. Instead, they need to sign with the single keypair whose public key is stored in each Account's data bytes as its "owner" (all semantic!). The DEX program enforces that the OpenOrders account "owner" signed the transaction, and the SPL token program enforces that the SPL token account's "owner" did as well.

If you visit the example client code for placing orders also linked above, you'll notice that the instruction passes in all of the mentioned accounts, each with two further bits: if the account is a signer, and if it is writable (if its data might be modified as a part of the transaction). The first is just enforced, and the latter reflects a key design choice that enables Solana's performance: To run the transaction, Solana nodes know exactly what they need to load into memory (only the data from these accounts needs to be used, read, modified, etc), and can parallelize transaction processing by being aware of what accounts might be modified (those that are writable).

And a fun fact: the Order ID is a unique identifier when combined with the order side and is constructed such that the result of comparing it to other Order IDs reflects its relative price-time priority: it's a 128-bit number, where the first 64 bits are the price, and the second 64 bits are the sequence number mentioned above (with all bits flipped if the order is buying).

Matching orders

This step removes and processes requests from the Request Queue, updates orders on the Orderbook, and puts information about resulting trades in the Event Queue.

Anybody can submit this instruction – there's nothing about it that requires that the users whose orders are involved to sign! All of the involved accounts owned by the program.

So why is this an instruction at all? The alternative is that it'd happen automatically, and there isn't a great blockchain mechanism for cycling some instruction at some frequency: the chain needs prompting. It's also totally safe and adjusts state in a deterministic, knowable direction: each item must be processed in order exactly once and has a deterministic effect. So there is exactly one way the matching engine can possibly proceed, and it just waits for someone to prompt that. There is no ability for people to influence trades with this.

For all these reasons, the job of matching orders is often referred to as "crank turning". Not that it's a terribly costly job, but users who pay blockchain transaction fees to turn this crank will be compensated so it's worth their while.

In a Match Orders transaction, clients (crank turners) provide a limit parameter, which specifies the number of requests to process from the Request Queue. Each request is processed as follows:

- It is popped off of the RequestQueue account
- The corresponding cancellation or order placement is made on the Orderbook (the orderbook consists of one tree per side, and so finding the relevant part of the appropriate tree is made easier by the comparison property of Order IDs mentioned earlier)
- In the case of a new order that stays on the book, the Orderbook stores details about it, including the obvious specs (side, price, remaining size), the public key of the order placer's OpenOrders account, and the index of this order in that OpenOrders account's order array (the slot number).
- Order types like IOC and post-only are enforced
- For any trades, two corresponding Fill items are added to the Event Queue
- In the case of a trade, cancel, or IOC order that missed, Out items are added to the Event Queue

The two types of event queue objects store the following information, which will be useful for the Consume Events step:

- Fill:
- Side
- If this side of the trade was the maker
- Quantity paid by this counterparty (the currency is inferable from the Side: if buying, this quantity is always denominated in the quote currency)
- Quantity received by this counterparty (if buying, this quantity is always denominated in the base currency)

    - Quantity of any fees paid (always in the quote currency)

- Out:
- Side
- Quantity unlocked
- Quantity still locked in the order (0 in the case of a cancel or full fill, nonzero in the case of a partial fill)

And they both also store: the order's ID and slot number, the public key of the corresponding OpenOrders account

Here's some client-code for running this instruction.

Also note that clients can read directly off the Orderbook account to get its state, the Event Queue account to see their trades (and others'), and off of OpenOrders accounts to get their open orders (and others').

Note that the Request Queue and Event Queue are both ring buffers: we've been referring to items getting popped off as they're processed, but in reality it's just a queue head reference that updates, and the actual events stick around to be read by whoever wants to until the ring buffer wraps around.

Consuming Events

This is another crank-turning instruction: its primary job is to make sure that OpenOrders accounts are updated according to events emitted by Match Orders.

Submitting Consume Events instructions (that do anything) does involve a bit of work for the client: they have to read some prefix of the event log, gather and sort the public keys of the affected OpenOrders accounts, and submit those as writable along with other relevant global singleton accounts.

The processing of the instruction is as follows:

- The next event in the Event Queue is considered. If its OpenOrders public key is found (via binary search – hence the sorting requirement) in the account list submitted by the client as writable, then continue. Otherwise abort.
- That event is popped off the queue and processed

- Fill:
- The OpenOrders account's total balances are decremented by quantity paid The OpenOrders account's total and free balances are incremented by quantity received

    - The Market's total fees paid counter increments by the quantity of fees paid

- Out:
    - The OpenOrders account's free balance increments by quantity unlocked
    - If the event's specified quantity still locked is zero, then remove the order from the OpenOrders account's account list. Note that this can happen in constant time because the slot number is provided in the event

Settling funds

This is the step in which users can withdraw freed or received funds back to their SPL token wallets.

Here is some client code for this instruction.

This instruction is relatively simple: the user passes their OpenOrders account and their SPL token accounts for the base and quote currency, signs with the "owner" of all of them (the same keypair that signed for placing orders using these same accounts), and the runtime does the following:

- Using a cross program invocation to the SPL token program, moves funds from the Base Currency Vault and Quote Currency Vault to the provided SPL token accounts equal to the free balance amount in the OpenOrders for each currency.

In one sense this step is optional: funds that remain accounted for as free in the OpenOrders account can be used for sending more orders; and like every part of Openbook, there is no third party who can confiscate or use them. But it's also totally harmless to withdraw, and doing so means that your funds don't sit around in the DEX: they only need remain in the DEX when they're locked up in orders, and can otherwise stay in your own SPL token accounts.

Canceling

We skipped this one earlier since it's not terribly different from placing an order: it takes in an order (which includes the slot number and Order ID) adds an event to cancel it to the Event Queue.

Here's some client code for it.

An exercise to the reader: see if you can walk through the above steps to trace through what happens with this request. In particular, do you see how it ends up modifying the Orderbook and your OpenOrders account, and letting you withdraw unlocked funds?

Interacting with the DEX

Sure, the theory's very nice, but how does one actually interact with the DEX in practice?

There are a few ways!

Libraries for programmatic access

You can use a client-side library like Openbook-js: load it up with your private keys (passed into a solana/web3.js Account) and get cracking.

We'd be thrilled to see further open source client-side tooling, including improving on Openbook-js or implementing libraries like it for other languages.

Using a GUI

You can trade on Openbook via a web-based GUI, just like you can for any centralized exchange. For reference, here's the picture of a demo GUI again.

If you're curious, this isn't a mockup; it's fully functional! You can see the open source code here.

This UI also uses Openbook-js, but without requiring explicit access to your private keys. And there's no backing server: it's all trustless and in your browser.

The trick? It asks the user to connect to a wallet provider (like sollet.io) using a secure, sol-wallet-adapter interface, through which it will ask the wallet to sign transactions it wants to send. The wallet asks the user to authorize each transaction being signed.

Note that any wallet and any decentralized app can implement the sol-wallet-adapter interface (or others like it – this is just one open source solution) to provide a similar secure signing solution for users.

But back to the DEX – what does this look like from the user's perspective?

First, they connect their wallet by clicking the button in the top right corner and clicking ‘Connect’ on the popup provided by the wallet:

Once connected, the DEX learns the public key of the wallet, and using Openbook-js can read on-chain state and display data specific to that user, like their user-specific trade history, balances, open orders, or base and quote currency deposit addresses.

If the user wants to interact with the DEX in a way that requires signing a request (say, like placing an order), then they can fill out the order form in the top right and click Buy or Sell, and either Cancel or Approve the corresponding transaction the DEX generates and asks the wallet to sign in a pop-up similar to the one shown above.

For known transaction types, wallets can parse the transaction passed from the DEX and display the details of what’s being attempted to the user.

To all Solana wallet providers out there: we highly recommend you implement SPL token and sol-wallet-adapter support so that your project can be hooked up to DEX GUIs like the one above!

Fees

There are two types of fees when using Openbook: trading fees and blockchain fees.

Because of Solana’s efficient design, blockchain fees are a tiny fraction of that of most networks. As an example, I just sent some SOL on the mainnet, paying 0.000000001 SOL in fees (about $0.0000003!).

In addition, Openbook charges trading fees. Those fees may be modified from time to time.

Current fees

- stable 0.01% taker fee 0.005% maker rebate
- all other markets 0.04% taker fee 0.02% maker rebate, 0.02% UI rebate

What’s next?

The Solana and Openbook ecosystems are growing rapidly and with a strong open-source focus. There are lots of ways to get involved!
Visit the Project Openbook site for updates, check out the ecosystem’s developer tooling, and make sure to join the discord group.

For more information on Solana, visit their website, discord, and technical docs.