General Architecture

Dex Demo is a Cosmos SDK application. It consists of a daemon process called DEXd, and a command line utility called Dexcli. Since these processes fulfill their standard SDK application roles, we will focus on their unique application-level architectures rather than dwell upon things that are already well-documented in the SDK.

Dexd Architecture

Components

As befits a Cosmos SDK application, Dexd manages a set of components and SDK modules that together make up the Dex Demo state machine. A list of all these components and their functions follows:

Component	Function
Asset	*SDK module* that wraps the SDK's BankKeeper to provide support for multiple tokens and finer-grained control over asset minting and burning.
Order	*SDK module* providing support for the posting and cancellation of orders.
Execution	Provides support for order matching, execution, and delivery. Not invoked directly. Further description of this functionality is provided in a later section.
Market	*SDK module* that wraps set of assets for the purposes of trading.
Embedded	Serves UI and REST endpoints.

Dexd also makes use of several SDK keepers such as the BankKeeper and AccountKeeper.

One of the guiding principles of Dexd's architecture is to keep the consensus layer minimal. As such, the components that live outside of the x/ use KSVStores that store data outside of the

Tendermint state tree and only keep their most recent state. For example, the Execution keeper will delete expired or completely-filled orders, and the Embedded component Order keeper will delete cancelled orders. To provide rich history to the UI, a separate set of keepers is used to store historical data. SDK modules keepers feed data to the non-consensus KVStores via an in-memory queue. Note that while the in-memory queue and the non-consensus stores' business logic currently exist within the Dexd process, there is no reason why they need to be. The queue could easily be swapped out to use Kafka or some similar message broker, and the non-consensus store modified to be a standalone process (or cluster of processes). We made this decision to explicitly enable the easy creation of centralized exchange-like entities that perform the service of indexing and serving historical data on behalf of users while allowing validators to skip the storage of consensus-irrelevant data.

A list of all KVStores that store data outside of Tendermint state tree and their functions follows:

Keeper	**Function**
Fill	Tracks all historical fills.
Order	Tracks all historical orders and their states.
Price	Tracks historical price ticks and candle data for all Markets.
Withdrawal	Tracks all historical withdrawals of cleared assets and their states.
Batch	Tracks all historical batch executions.

Matching Engine

Dexd uses a batch auction-based matching engine to execute orders. Batch auctions were chosen to reduce the impact of frontrunning on the exchange. According to the Flash Boys 2.0 paper on DEX frontrunning, most frontrunning on DEXes occurs as a result of priority gas auctions. Batch auctions sidestep this problem by reducing the impact of the PGA to zero. Our batch auction engine is based on this Chicago Booth paper, and operates as described below. Note that the below process occurs at the end of every block for every market on Dex:

1. All orders for the given market are collected.

2. Orders beyond their time-in-force are cancelled.

3. Orders are placed into separate lists by market side, and aggregate supply and demand curves are calculated.

4. The matching engine discovers the price at which the aggregate supply and demand curves cross, which yields the clearing price. If there is a horizontal cross - i.e., two prices for which aggregate supply and demand are equal - then the clearing price is the midpoint between the two prices.

5. If both sides of the market have equal volume, then all orders are completely filled. If one side has more volume than the other, then the side with higher volume is rationed pro-rata based on how much its volume exceeds the other side. For example, if aggregate demand is 100 and aggregate supply is 90, then every order on the demand side of the market will be matched 90%.

Orders are sorted based on their price, and order ID. Order IDs are generated at post time, and is the only part of the matching engine that is time-dependent. However, the oldest order IDs are matched first so there is no incentive to post an order ahead of someone else's.

It is important to note that because rationing happens on a pro-rata basis it is possible for the pro-rata to include fractional assets below the smallest representable amount. To handle this edge case, we round the pro-rata amount up to the nearest whole asset. To prevent matching more volume than was placed on the book, a counter keeps track of all allocated volume and ends the matching process early once it is exhausted. This is the only case where the time at which your order was placed matters: since older orders are matched first, it is possible for an order placed before yours to be filled for more than yours if there is not enough volume to fill your order completely. In practice this does not matter much, however, because the amounts involved are so small.

An example would be illustrative. Consider the following order book:

Bids			**Asks**		
ID	**Price**	**Volume**	**ID**	**Price**	**Volume**
1	10	100	2	8	100
3	12	25	4	10	100

5	12	25	6	11	100
7	14	50	8	14	100

Price	**Bid Volume**	**Ask Volume**
8	200	100
10	200	200
11	100	300
12	100	300
14	50	400

The crossover point is at price 10, which becomes the clearing price. Since both the bid volume and the ask volume are equal, all bid orders with a price greater than or equal to 10 will be matched completely, and all ask orders with a price equal to or less than 10 will be matched completely. All other orders will be returned to the order book unfilled.

Now let’s consider the following variant of the above order book:

Bids			**Asks**		
ID	**Price**	**Volume**	**ID**	**Price**	**Volume**
1	10	100	2	8	100
3	12	100	4	10	50

5	12	100	6	10	50
7	14	50	8	11	100
-	-	-	10	14	100

Price	**Bid Volume**	**Ask Volume**
8	350	100
10	350	200
11	250	300
12	250	300
14	50	400

The crossover occurs at price 11, however price 12 also preserves the same aggregate supply and demand. Thus the midpoint between the two prices - 11.50 - is chosen as the clearing price.

Finally, since aggregate supply exceeds aggregate demand, the supply side will be rationed 83.33%. It is here that the order of transactions matters. Orders 2, 4, 6, and 8 will be matched on the supply side. The order of execution is also 2, 4, 6, 8, because:

1. The first sort criteria is the price of the order.
2. The second sort criteria is the ID of the order - i.e., the time at which it hit the book.

In this case, these orders will be matched as follows:

Order ID	**Match Amount**	**Available Volume**

2	CEIL(100* 0.83333) = 84	166
4	CEIL(50 * 0.83333) = 42	124
6	CEIL(50 * 0.83333) = 42	82
8	CEIL(100 * 0.83333) = 84. 84 > Available Volume, therefore Amount = 82	0
Total:	250	

You will see in the table above that the error propagates through the orders based on the order of execution.

Dexcli Architecture

Embedded Server

Dexcli's primary use is as a daemon that runs the Dex REST API and serves the UI. Together, we call these things the "*embedded server*." The goal of the embedded server is to create a centralized-exchange-like user experience without compromising on decentralization. After installing the Dex package, they can simply visit https://localhost:1317 or some similar URL from their browser to interact with their local full node.

Under the hood, the embedded server makes RPC calls to Dexd via the standard SDK Querier pattern. Depending on the URL requested, a consensus or non-consensus keeper may be queried. The server also has the ability to initiate transactions on a user's behalf using an in-memory cache. When the user authenticates, their private key is stored encrypted in memory and decrypted using a key stored in the user's session. Thus, the user has the ability to transact with the exchange as well without confirmations, prompts, or key management.

You can find a full list of supported API endpoints at

https://docs.serverfuse.tools/uex-embedded.