# DeepBook Orders

DeepBook V2 is now deprecated. Update your integrations to use DeepBook V3 instead.

DeepBook adopts a hyper-efficient approach to store orders. Each pool stores the unfilled maker orders. Taker orders are filled instantaneously within the same transaction the order is submitted. Bid and ask orders are stored separately, each with a two-level nested crit-bit tree. The first level crit-bit tree is ordered using the price of the maker order, and the second level crit-bit tree is ordered using the order id of the maker order.

DeepBook supports the placement of market orders and limit orders. DeepBook also supports a few configurations of limit orders, which is covered in the API section. When users submit a market order, it is matched against the existing maker orders on the order book instantaneously in the same transaction upon submission of the market order. When users submit a limit order, the limit order is first matched against the existing maker orders as a taker order. If the order cannot be fully filled, the remaining quantity can either be injected as a maker order or be dropped, depending on the configuration of the limit order. See the API section for further details.

Order matching occurs when a taker order is submitted to the CLOB without a centralized entity or crank involvement. Taker orders are matched against the existing maker orders in the CLOB in the same transaction the taker order is submitted.

DeepBook supports efficient tracking of maker orders. Each unfilled maker order is associated with a unique u64 order id, and users can query the order status using the order id together with the Sui RPC call. Users can also subscribe to the event stream emitted by DeepBook related to changes in order status. DeepBook currently supports the following events, OrderPlaced , OrderFilled and OrderCanceled .

The following example uses the Sui TypeScript SDK to connect via websocket to listen to events.

## Order book structure

DeepBook adopts a hyper-efficient approach to store orders. Each pool stores the unfilled maker orders. Taker orders are filled instantaneously within the same transaction the order is submitted. Bid and ask orders are stored separately, each with a two-level nested crit-bit tree. The first level crit-bit tree is ordered using the price of the maker order, and the second level crit-bit tree is ordered using the order id of the maker order.

DeepBook supports the placement of market orders and limit orders. DeepBook also supports a few configurations of limit orders, which is covered in the API section. When users submit a market order, it is matched against the existing maker orders on the order book instantaneously in the same transaction upon submission of the market order. When users submit a limit order, the limit order is first matched against the existing maker orders as a taker order. If the order cannot be fully filled, the remaining quantity can either be injected as a maker order or be dropped, depending on the configuration of the limit order. See the API section for further details.

Order matching occurs when a taker order is submitted to the CLOB without a centralized entity or crank involvement. Taker orders are matched against the existing maker orders in the CLOB in the same transaction the taker order is submitted.

DeepBook supports efficient tracking of maker orders. Each unfilled maker order is associated with a unique u64 order id, and users can query the order status using the order id together with the Sui RPC call. Users can also subscribe to the event stream emitted by DeepBook related to changes in order status. DeepBook currently supports the following events, OrderPlaced , OrderFilled and OrderCanceled .

The following example uses the Sui TypeScript SDK to connect via websocket to listen to events.

## Placing orders

DeepBook supports the placement of market orders and limit orders. DeepBook also supports a few configurations of limit orders, which is covered in the API section. When users submit a market order, it is matched against the existing maker orders on the order book instantaneously in the same transaction upon submission of the market order. When users submit a limit order, the limit order is first matched against the existing maker orders as a taker order. If the order cannot be fully filled, the remaining quantity can either be injected as a maker order or be dropped, depending on the configuration of the limit order. See the API section for further details.

Order matching occurs when a taker order is submitted to the CLOB without a centralized entity or crank involvement. Taker orders are matched against the existing maker orders in the CLOB in the same transaction the taker order is submitted.

DeepBook supports efficient tracking of maker orders. Each unfilled maker order is associated with a unique u64 order id, and users can query the order status using the order id together with the Sui RPC call. Users can also subscribe to the event stream emitted by DeepBook related to changes in order status. DeepBook currently supports the following events, OrderPlaced ,

OrderFilled and OrderCanceled .

The following example uses the Sui TypeScript SDK to connect via websocket to listen to events.

## Order matching

Order matching occurs when a taker order is submitted to the CLOB without a centralized entity or crank involvement. Taker orders are matched against the existing maker orders in the CLOB in the same transaction the taker order is submitted.

DeepBook supports efficient tracking of maker orders. Each unfilled maker order is associated with a unique u64 order id, and users can query the order status using the order id together with the Sui RPC call. Users can also subscribe to the event stream emitted by DeepBook related to changes in order status. DeepBook currently supports the following events, OrderPlaced , OrderFilled and OrderCanceled .

The following example uses the Sui TypeScript SDK to connect via websocket to listen to events.

## Order tracking

DeepBook supports efficient tracking of maker orders. Each unfilled maker order is associated with a unique u64 order id, and users can query the order status using the order id together with the Sui RPC call. Users can also subscribe to the event stream emitted by DeepBook related to changes in order status. DeepBook currently supports the following events, OrderPlaced , OrderFilled and OrderCanceled .

The following example uses the Sui TypeScript SDK to connect via websocket to listen to events.