Kronos Research 20251009

Backend

- Vendor provides as a service
- Cloud storage
- Push Notifications
- Logic Processing → Database matching

Frontend

- Exchange’s UI visualization
- User interactions

Restful in API

- Rest: A set of principles for designing network APIs.
- Json : imply some information by your request.

- HTTP Methods → Get, Post, Put, Delete

API = Application Programming Interface

- Allow software to interacts
- API Documents → Details of the endpoints, tutorial, library, authentication (API Key)
- We don’t need authentication for public endpoint ex: K-line data.
- Example of OKX：
    - Get Balance → Your account balance endpoint
    - HTTP Request → Get method, require parameters (query), response

HTTP = Hypertext Transfer Protocol

- Information sharing and data transfer
- Client request → Server response
- Stateless Protocol → Each request is independent; servers don’t store information.
- IP requesting → they(Server) can see your activity, sometimes they can choose their further response to your request.
- Client → SYN → Server → SYN + ACK → Client → SYN → Server

HTTP Methods

- Interaction methods with the backend server.
- nowadays, most exchange focus on GET and POST.
- Get : get what data from the server
- Post：Send data to create a resource, ex : create buy/sell order (pair, order).

- Delete : 撤單
- Put：Order update

Keep alive connection

- Connection distributions → latency and delay analysis in trading.

```
import requests
import time

url = "http://www.okx.com/api/........"
print("Testing with different connections: ")
for i in range(2):
    start_time = time.time()
    requests.get(url)
    print(f"Request {i+1} time: {time.time() - start_time:.4f} seconds")

print()
print("Testing with the same connection (Keep-Alive):")
with requests.Session() as session:
    for i in range(2):
    start_time = time.time()
    session.get(url)
    print(f"Request {i+1} time: {time.time() - start_time:.4f} seconds")
```

WebSocket

- A protocol that provides full-duplex communication over a single TCP connection.

- Unlike HTTP interactions, WebSocket interacts the clients and servers simultaneously. → Query the endpoints → Server keep updating data to client (minimize the request time and response time → latency decline)
- HTTP can only response to the client request passively.

Building WebSocket Server with Python

- Library requirement : asyncio, websockets

```
import json
import asyncio
import websockets

async def test_server(websocket, path):
    while True:

      message = {
          "type": "orderbook_update",
          "symbol": "BTC-UDT",
          "bids": [[10000, 5], [9950, 2]],
          "ask": [[10050, 3], [10100, 8]]
      }
      await websocket.send(json.dumps(message))
      await asyncio.sleep(2) # send update every 2 seconds

  start_server = websockets.serve(test_server, 'localhost', 8888) # Adjust port if needed

  asyncio.get_event_loop().run_until_complete(start_server)
  asyncio.get_event_loop().run_forever()
```

Asyncio in Python

- Allows you to write asynchronous code. → Orderbook update
- We can download 3 file simultaneously by using asyncio package. → more efficient than traditional methods

```
import asyncio
import websocket
import json

async def client():
    url = "ws://localhost:8888"
    async with websocket.connect(url) as websocket:
        while True:
            message = await websocket.recv()
            data = json.loads(message)
            print(f"Received orederbook update: {data}")
            # process the orderbook update as needed

asyncio.get_event_loop().ren_until_complete(client())
```

```
import asyncio # We can download 3 file simultaneously by using asyncio package.

async def download_file(file_name):
    print(f"Starting download: {file_name})")
    await asyncio.sleep(2) #v Simulates a time-consuming job
    print(f"Finished download: {file_name}")

async def main():
    await asyncio.gather(
        download_file("file1.txt"),
        download_file("file2.txt"),
        download_file("file3.txt"),
    )
```

```
if __name__ == "__main__":
        asyncio.run(main)
```

Public Data from Woo X

- Restful API data：Ticker, K-lines
- WebSocket API data：Orderbook snapshots (update the data instantly to the client sides), real time trades
- How to handle data missing in real time trading
    - risk control
    - The data usually saves more and more by days
    - Our HPC team focus on the computing capacity for trading simulation data → Hardware and Software tuning.

Drawback of Restful API ?

- We need instant data in real-time trading → sometimes the exchange will block the client end point request by blocking the over - query IP address → Change your IP by counter the issue ?
- WebSocket Ping-Pong test：Every time the exchange sends you a ping, you need to reply it with a pong. → Handshake process LOL

Build a good Orderbook structure

- we r building a webSocket Client that will subscribe to orderbook updates from Woo X exchange.
- Subscribe to orderbook data (bids & ask)
- Implement a mechanism to keep the connection active by sending periodic ping messages.

- Orderbook class : Help us structure and store orderbook data for easy access and display.

```
# Example

class OrderBook :
    def __init__(self, symbol):
    self.symbol = symbol
    self.asks = [] # store the ask side → [price, size]
    self.bids = [] # store the bid side → [price, size]

    def update(self, data):
        """ Updates the orderbook with new data."""
        if data["symbol"] != self.symbol:
            raise ValueError("Data symbol does not match orderbook symb
ol")

        self.asks = data["asks"]
        self.bids = data["bids"]

    def dump(self, max_level = 10):
        pass
```

Build a good BBO(Best Bid Order) structure

- The real trades that occur in the high frequency timeframe.
- Incoming data comparison → highest bid and lowest ask → market regime/trend analysis

```
class BBO:
    def __init__(self, symbol: str):
        """Initiate the BBO structure for a specific symbol"""
        self.symbol = symbol
        self.best_bid = None # Best bid price
        self.best_bid_size = 0 # Size of the best bid
        self.best_ask = None
        self.best_ask_size = 0

    def update(self, bid_price: float, bid_size: float, ask_price: float, ask_size: float):
        """Update the BBO with new bid and ask data"""
        if self.best_bid is None or bid_price > self.best_bid:
            self.best_bid = bid_price
            self.best_bid_size = bid_size

        elif bid_price == self.best_bid:
          self.best_bid_size += bid_size # Aggregate size if the bid price is the same

        if self.best_ask is None or ask_price > self.best_ask:
            self.best_ask = ask_price
            self.best_ask_size = ask_size

        elif ask_price == self.ask_bid:
          self.best_ask_size += ask_size
```

Question 1 : Compiling the Latest Orderbook

- Data arrangement from the WebSocket ? → The most naive implementation
- API endpoint → WebSocket Connection → OrderBook Class (Bid array, Ask array → append and aggregate those data together)
- Aggregate the same price and corresponding volume → sorting (the highest bid and lowest ask) → Find out the BBO

Question 2 : Leading Market Analysis

- Visualize your data to identify the highest trading volume or highest volatility.

- Utilize the OKX orderbook or ticker to identify which market is currently leading in OKX.
- We assume that someone knew the esoteric info earlier than everyone else → test your hypothesis → Gather all the market symbols , filtering and comparison
- Sometimes the signals may have only leading in 2 seconds or less, we cannot identify it by our eyes or pc. While the data in the orderbook had already detect the imbalance signals.
- What's the orderbook patterns that occur before the historical volatility spikes?
- We usually focus on the high volume symbols → indicates there are more players → the liquidity is also essential → Market depth and symbol filtering

Question 3 : Price Correlation Between BTC and ETH

- How do you predict the price action base on the historical data ?
- Ex: If BTC increase by 1%, analyze how much ETH typically increases or decreases.
- In high frequency timeframe → We can implement statistical arbitrage to capture the mispricing. → Mean Reversion trading
- Sometimes we might analyze 10+ symbols to create a delta neutral strategy, cause 2 assets usually move side-way → decrease the drawdown

Question 4 : Slippage and Orderbook Depth

- Discuss the concept of slippage and its relationship with orderbook depth.
- By creating an orderbook, we can identify the impact on the slippage costs.
- Analyzing the slippage by considering metrics such as average or quantile.
- We usually determine the slippage by analyzing fill rate of our trades → how many % of our trades that filled, sometimes we will analyze each exchanges fill rate.

- Does the market direction satisfy our expectation of our trades after being filed ?
- In theory, cancel order(queue) would usually be set priority by the exchange.
- Sometimes , the traders will look at the Defi memory pool to spot on the orders that should be occur at the exchange.

Question 5 : Predicting Price Movement with Orderbook Data

- Explain how you would use the orderbook, trade data, and BBO data to predict the price movement (momentum) in the next second.