Project Overview

This project is a command-line interface (CLI) trading bot designed for the Binance USDT-M Futures market. It provides a powerful and flexible tool for traders to automate their order placement on the Binance testnet. The bot is built with a modular architecture, making it easy to understand, maintain, and extend with new features.

The core functionalities of the bot include:

- Multiple Order Types: The bot supports a variety of order types, from basic MARKET and LIMIT orders to more advanced strategies like STOP_LIMIT, OCO (One-Cancels-the-Other), and TWAP (Time-Weighted Average Price).
- Input Validation: A robust validation system is in place to check all user inputs (symbol, quantity, price, etc.) against the exchange's trading rules before placing an order. This significantly reduces the risk of errors.
- Structured Logging: All actions, including order placements, errors, and successful executions, are logged to a file (logs/bot.log) for easy debugging and auditing.
- Direct API Interaction: The bot communicates directly with the Binance API through the python-binance library, ensuring efficient and reliable order execution.

How it Benefits a Trader
This bot offers several advantages for a trader:

- Automation and Efficiency: By automating the order placement process, the bot saves traders valuable time and allows them to execute trades with speed and precision.
- Advanced Order Strategies: The inclusion of advanced order types like OCO and TWAP empowers traders to implement sophisticated trading strategies. For instance, an OCO order can be used to set both a take-profit and a stop-loss order simultaneously, while a TWAP order can help to minimize the market impact of a large trade.
- Reduced Risk of Errors: The pre-emptive input validation is a critical feature that protects traders from costly mistakes. By catching invalid order parameters before they are sent to the exchange, the bot helps to prevent unexpected losses.
- Flexibility and Extensibility: The modular design of the bot makes it easy to add new order types or even integrate it into a larger automated trading system.
- Complete Transparency: The detailed logging provides a full audit trail of all trading activities, giving the trader a clear understanding of the bot's behavior.

Key Code Modules and Highlights

Here is a breakdown of the key modules and code sections that are central to the bot's functionality:

`src/bot.py`: The Core Engine

This is the main entry point of the application.

Argument Parsing: The main() function uses Python's argparse library to handle command-line arguments. This makes the bot highly configurable and user-friendly.

```python
def main():
    p = argparse.ArgumentParser(description='Binance Futures Testnet Trading Bot')
    p.add_argument('symbol', help='e.g., BTCUSDT')
    p.add_argument('side', help='BUY or SELL')
    p.add_argument('order_type', help='MARKET, LIMIT, STOP_LIMIT, TWAP')
    p.add_argument('quantity', type=float, help='Order quantity')
    p.add_argument('--price', type=float, help='Limit price (LIMIT/STOP_LIMIT/TWAP)')
    p.add_argument('--stop-price', type=float, help='Stop trigger price (STOP_LIMIT)')
    p.add_argument('--tif', default='GTC', help='Time in force (default GTC)')
    p.add_argument('--chunks', type=int, default=1, help='TWAP chunks')
    p.add_argument('--interval', type=int, default=30, help='TWAP interval seconds')
    args = p.parse_args()

    client = BinanceClient(testnet=True)

    qty, px, stop_px = validate_and_normalize(client, args.symbol, args.side, args.order_type, args.quantity, args.price, args.stop_price)

    if args.order_type.upper() == 'MARKET':
        order = place_market_order(client, args.symbol, args.side, qty)
    elif args.order_type.upper() == 'LIMIT':
        order = place_limit_order(client, args.symbol, args.side, qty, px, args.tif)
    elif args.order_type.upper() == 'STOP_LIMIT':
        order = place_stop_limit_order(client, args.symbol, args.side, qty, px, stop_px, args.tif)
    elif args.order_type.upper() == 'TWAP':
        order = place_twap_orders(client, args.symbol, args.side, qty, px, args.chunks, args.interval, args.tif)
    else:
        raise SystemExit('Unsupported order_type')

    print('Order Result:', order)
```

Order Dispatching: Based on the order_type provided by the user, the bot dynamically calls the appropriate function to place the order. This is a clean and efficient way to handle different order types.

```
1    if args.order_type.upper() == 'MARKET':
2        order = place_market_order(client, args.symbol, args.side, qty)
3    elif args.order_type.upper() == 'LIMIT':
4        order = place_limit_order(client, args.symbol, args.side, qty, px, args.tif)
5    # ... and so on for other order types
```

`src/client.py`: The API Connector

```python
class BinanceClient:
    def __init__(self, testnet: bool = True):
        self.logger = setup_logger('BinanceClient')
        self.client = Client(BINANCE_API_KEY, BINANCE_API_SECRET)
        # Point the futures base URL to testnet (UM: /fapi)
        if testnet:
            self.client.FUTURES_URL = TESTNET_BASE_URL.rstrip('/') + '/fapi'
            self.client.FUTURES_DATA_URL = self.client.FUTURES_URL
        # Ensure one-way mode (not dual)
        try:
            self.client.futures_change_position_mode(dualSidePosition=False)
        except Exception as e:
            self.logger.debug(f"Could not change position mode (maybe already set): {e}")

    def exchange_info(self):
        return self.client.futures_exchange_info()

    def symbol_info(self, symbol: str):
        info = self.exchange_info()
        for s in info['symbols']:
            if s['symbol'].upper() == symbol.upper():
                return s
        return None

    def place_order(self, symbol: str, side: str, order_type: str, **kwargs):
        try:
            # Ensure workingType is set for stop/tp orders where relevant
            if 'workingType' not in kwargs:
                kwargs['workingType'] = DEFAULT_WORKING_TYPE
            order = self.client.futures_create_order(
                symbol=symbol.upper(),
                side=side.upper(),
                type=order_type.upper(),
                **kwargs
            )
            return order
```

This module is responsible for all communication with the Binance API.
BinanceClient Class: This class acts as a wrapper around the python-binance library. It simplifies the API interaction and encapsulates the
logic for connecting to the testnet and handling API keys.

```
1   class BinanceClient:
2       def __init__(self, testnet: bool = True):
3           # ... initialization logic ...
4           self.client = Client(BINANCE_API_KEY, BINANCE_API_SECRET)
5           if testnet:
6               self.client.FUTURES_URL = TESTNET_BASE_URL.rstrip('/') + '/fapi'
```

`src/validator.py`: The Guardian
This is one of the most critical modules in the project, responsible for preventing invalid orders. `validate_and_normalize()` Function: This function performs a series of checks to ensure that the order parameters are valid before they are sent to the exchange. This includes checking for minimum quantity, correct price precision, and, importantly, the minimum notional value.

```python
class SymbolFilters:
    def __init__(self, lot_step, min_qty, max_qty, price_tick, min_price, max_price, min_notional):
        self.lot_step = float(lot_step)
        self.min_qty = float(min_qty)
        self.max_qty = float(max_qty)
        self.price_tick = float(price_tick)
        self.min_price = float(min_price)
        self.max_price = float(max_price)
        self.min_notional = float(min_notional)

    def round_qty(self, qty: float) -> float:
        return math.floor(qty / self.lot_step) * self.lot_step

    def round_price(self, price: float) -> float:
        # Round down to nearest tick
        return math.floor(price / self.price_tick) * self.price_tick

def extract_filters(symbol_info) -> SymbolFilters:
    if not symbol_info:
        raise ValueError("Symbol not found in exchange info.")
    price_filter = next(f for f in symbol_info['filters'] if f['filterType'] == 'PRICE_FILTER')
    lot_filter = next(f for f in symbol_info['filters'] if f['filterType'] in ('LOT_SIZE','MARKET_LOT_SIZE'))
    notional_filter = next(f for f in symbol_info['filters'] if f['filterType'] == 'MIN_NOTIONAL')
    return SymbolFilters(
        lot_step=lot_filter['stepSize'],
        min_qty=lot_filter['minQty'],
        max_qty=lot_filter['maxQty'],
        price_tick=price_filter['tickSize'],
        min_price=price_filter['minPrice'],
        max_price=price_filter['maxPrice'],
        min_notional=notional_filter['notional']
    )
```

Highlight: The addition of the MIN_NOTIONAL check was a key improvement that addresses a common API error and provides a much better user experience.
`src/orders/` and `src/orders/advanced/`: Modular Order Logic

The separation of each order type into its own file is a great example of a clean and modular design.

* `src/orders/advanced/oco.py`: The implementation of the OCO order demonstrates how the bot can be extended with advanced trading features.

Highlight: This function showcases how to leverage the python-binance library to execute complex order types.

`src/logger_config.py`: The Scribe

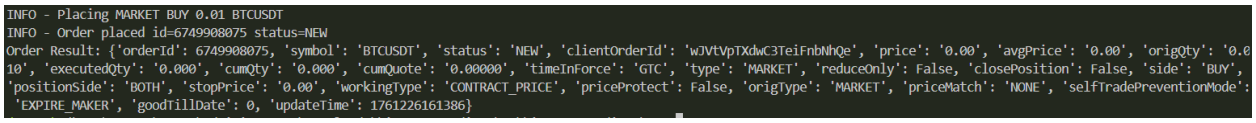
This module sets up a robust logging system.

setup_logger()` Function: This function configures a logger that writes to both the console and a rotating file. This ensures that all trading activities are recorded for later analysis and debugging, without letting the log file grow indefinitely.

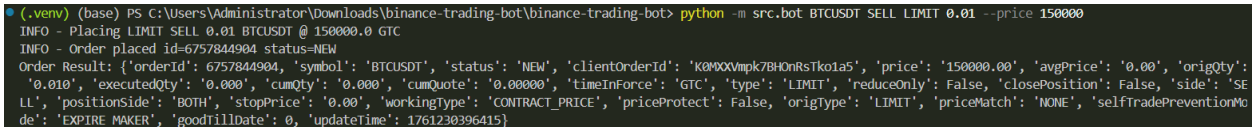
All these modules work together to create a powerful and reliable trading bot. The modular architecture and the focus on validation and logging are key to the project's success.

Results:

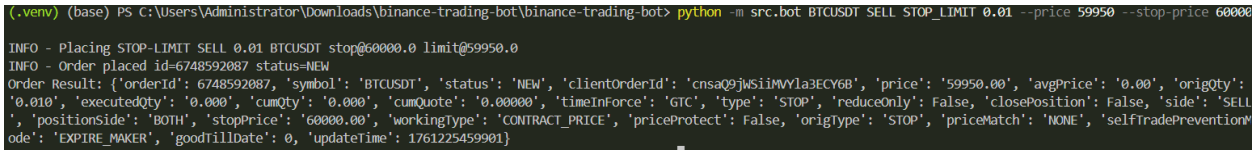
Placing a basic market order:



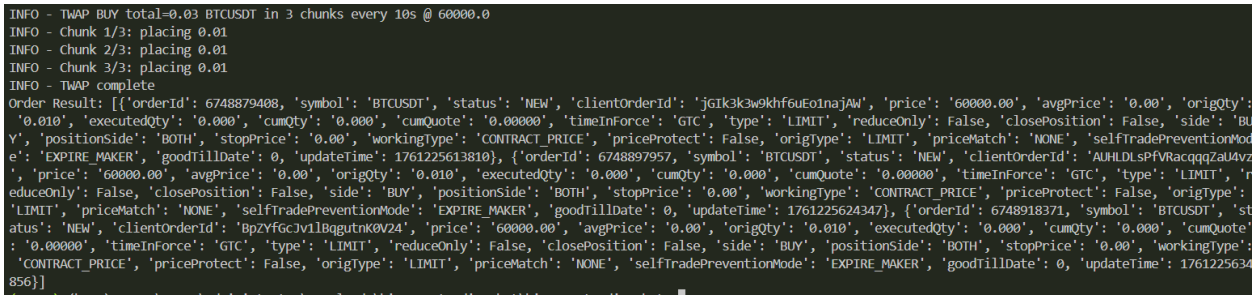
Placing a limit order



SELL_STOP_LIMIT



TWAP



python -m src.bot BTCUSDT BUY TWAP 0.03 --price 60000 --chunks 3 --interval