Binance Futures Trading Bot
(Project Report)

Author: Adithya P.
Date: December 31, 2025
Position: Junior Python Developer
Project: BinanceBot

Table of Contents

1. Executive Summary

This Project implements a CLI-based trading bot for Binance, specifically in the futures section, trading at USDT-M. It was developed as part of the Junior Python Development assessment. The bot interacts with the Binance Futures Testnet, a developer environment made for testing order placements, account queries and error handling purposes without the risk of involving real funds.

The Primary objective of the project is to demonstrate:

- The correct usage of official Binance Futures API
- The capacity to handle a given technical task
- Producing clean and reusable python code

2. Project Overview

2.1 Objective

To develop a Python-based trading bot capable of executing various order types on Binance Futures Testnet with proper validation, logging, and error handling.

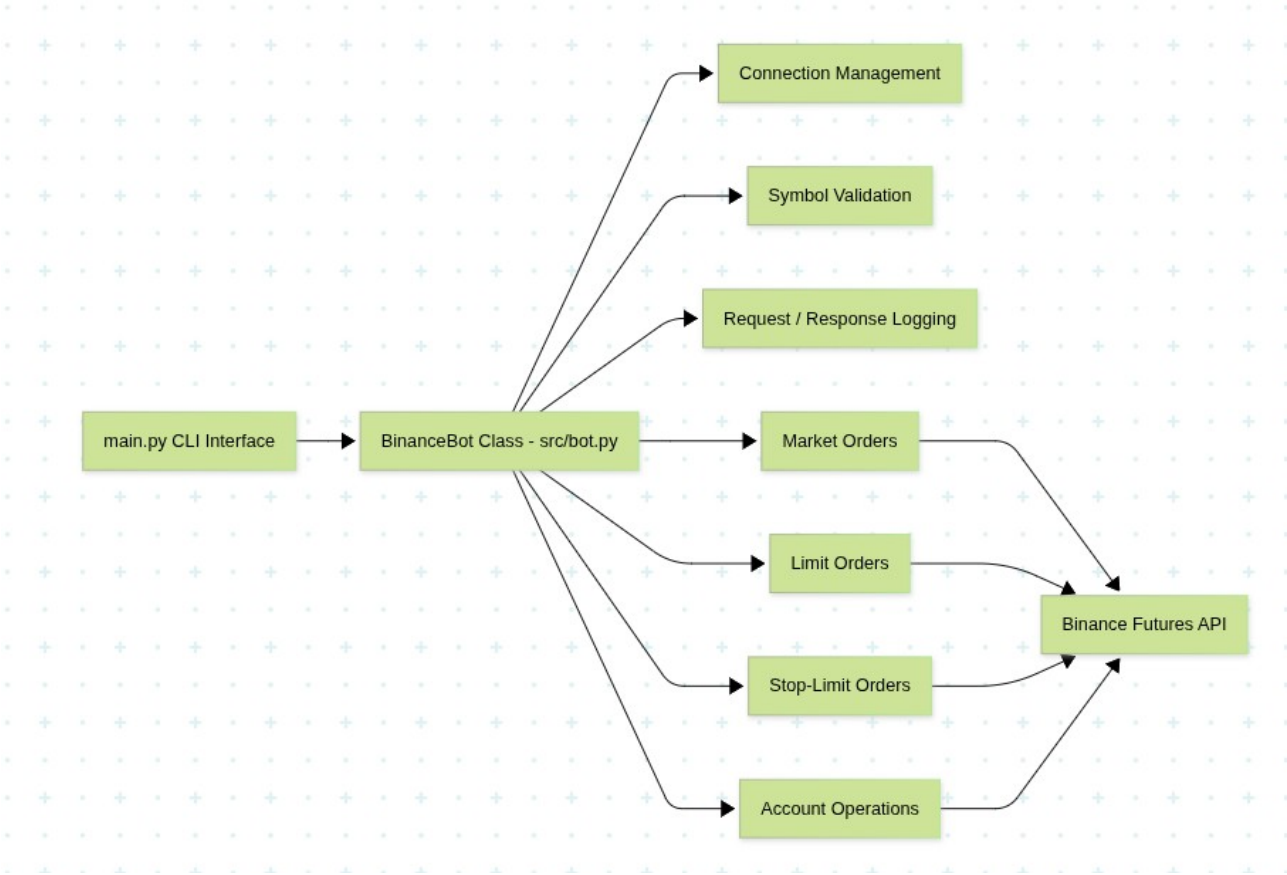
2.1 Requirements

- Place market and limit orders
- Support both BUY and SELL sides
- implement atleast one advanced order type (Stop-Limit)
- Log all important actions, API calls and errors
- Exceptional exception handling

2.2 Tech Stack

- Language: Python 3.x
- API Library: python-binance (v1.0.19)
- API Endpoint: https://testnet.binancefuture.com
- Logging: Python logging module
- Environment: Virtual environment (.venv)

3. System Architecture



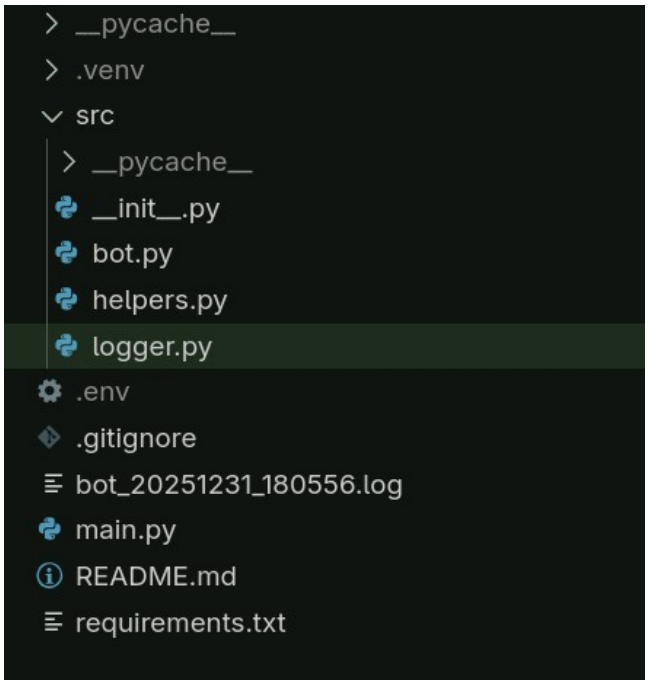
3.1 Project Structure

adithya-binance-bot/

├── **main.py**	Entry point and CLI interface
├── .env	API credentials (not committed)
├── requirements.txt	Project dependencies
├── README.md	Setup and usage documentation
├── bot.log	Generated log file
└── src/	
├── __init__.py	Package initializer
├── **bot.py**	Core BinanceBot class
├── **logger.py**	Logging configuration
├── **helpers.py**	Utility functions

3.2 File Responsibilities

- main.py – Entry point, handling the CLI interactions and menu flow
- bot.py – Core trading logic and Binance API integration
- helpers.py – CLI utility functions
- logger.py – Centralized logging config



3.3 Data Flow

1. **User Input** → CLI validates input
2. **Command Processing** → main.py routes to appropriate handler
3. **Order Validation** → BinanceBot validates symbol, quantity, price
4. **API Request** → Formatted request sent to Binance
5. **Response Handling** → Success/error logged and displayed
6. **User Feedback** → Results shown in terminal

4. Implementation

4.1 Core Bot Class (src/bot.py)

The BinanceBot class is the main interface to the Binance Futures API where the whole working of the bot is declared.

4.1.1 Initialization

```
def __init__(self, api_key: str, api_secret: str, testnet: bool = True):
    try:
        self.client = Client(api_key, api_secret, testnet=testnet)
        self.client.FUTURES_URL = "https://testnet.binancefuture.com"
        logger.info("BinanceBot is Initialized!")
        logger.info(f"Using {"TestNet" if testnet else "Main Account"}")

        self._test_connection()

    except Exception as e:
        logger.error(f"Falied to initialze BinanceBot: {str(e)}")
        raise
```

- Initializes Binance client with credentials
- Sets correct API endpoint
- Validates connection before proceeding

4.1.2 Connection Testing

```
def _test_connection(self):
    try:
        account_info = self.client.futures_account()
        logger.info("Connection to Binance Futures is Succesfull!")
        logger.info(f"Account Balance: {account_info.get("totalWalletBalance", "N/A")}")

    except Exception as e:
        logger.error(f"Connection test failed: {str(e)}")
        raise
```

- Verifies API credentials are valid
- Confirms network connectivity
- Logs initial account balance

4.1.3 Symbol Validation

```
def _validate_symbol(self, symbol: str) -> bool:
    try:
        exchange_info = self.client.futures_exchange_info()
        symbols = [s['symbol'] for s in exchange_info['symbols']]
        return symbol.upper() in symbols

    except:
        logger.error("Verification of input symbol failed: {str(e)}")
        return False
```

- Prevents invalid symbol errors
- Reduces failed API calls
- Provides immediate feedback to users

5. Core Features

5.1 Market Orders

Market orders execute immediately at the best available market price.

```
def place_market_order(self, symbol: str, side: str, quantity: float) -> Optional[Dict]:

    side = side.upper()
    symbol = symbol.upper()
    if side not in ['BUY', 'SELL']:
        logger.error(f"Invalid side: {side}. Must be BUY or SELL")
        return None

    if not self._validate_symbol(symbol):
        logger.error(f"Invalid Symbol: {symbol}")
        return None

    if quantity <= 0:
        logger.error(f"Invalid quantity: {quantity}. Must be above 0.")
        return None

    params = {
        'symbol': symbol,
        'side': side,
        'type': 'MARKET',
        'quantity': quantity
    }

    try:
        self._log_request('MARKET', params)
        order = self.client.futures_create_order(**params)
        self._log_response(order)

        logger.info(f"Market order is placed.")
        logger.info(f"Order ID: {order['orderId']}")
        logger.info(f"Status: {order['status']}")

        return order

    except BinanceAPIException as e:
        logger.error(f"Binance API Error: {e.status_code} - {e.message}")
        return None
    except BinanceRequestException as e:
        logger.error(f"Binance Request Error: {str(e)}")
        return None
    except Exception as e:
        logger.error(f"Unexpected error occured while placing market order: {str(e)}")
        return None
```

- Immediate execution needed
- Guaranteed fill (in liquid markets)
- Accept current market price

5.1.1 Market Order Test Results

```
(.venv) [dew@TheOcean BinanceBot]$ python main.py
1. Place Market Order
2. Place Limit Order
3. Place Stop-Limit Order
4. View Account Balance
5. View Open Orders
6. View Positions
7. Cancel Order
8. Exit
============================================================
Select an option: (1-8): 1

--- MARKET ORDER ---
Enter symbol: ETHUSDT
Enter side (BUY/SELL): buy
Enter quantity: 1
2025-12-31 23:10:23,443 - src.logger - INFO - API Request - Order type: MARKET
```

```
 }
 2025-12-31 23:10:23,575 - src.logger - INFO - Market order is placed.
 2025-12-31 23:10:23,575 - src.logger - INFO - Order ID: 7909916408
 2025-12-31 23:10:23,575 - src.logger - INFO - Status: NEW

 Order executed successfully!
 Order ID: 7909916408
 Status: NEW

 Press Enter to continue...
```

5.2 Limit Orders

Limit orders execute only at a specified price or better.

```python
def place_limit_order(self, symbol: str, side: str, quantity: float, price: float, time_in_force: str = "GTC") -> Optional[Dict]:

    side = side.upper()
    symbol = symbol.upper()
    time_in_force = time_in_force.upper()
    if side not in ['BUY', 'SELL']:
        logger.error(f"Invalid side: {side}. Must be BUY or SELL")
        return None

    if not self._validate_symbol(symbol):
        logger.error(f"Invalid symbol: {symbol}")
        return None

    if quantity <= 0:
        logger.error(f"Invalid quantity: {quantity}. Must be above 0")
        return None

    if price <= 0:
        logger.error(f"Invalid price: {price}. Must be above 0")
        return None

    if time_in_force not in ['GTC', 'IOC', 'FOK']:
        logger.error(f"Invalid time_in_force: {time_in_force}")
        return None

    params = {
        'symbol': symbol,
        'side': side,
        'type': 'LIMIT',
        'quantity': quantity,
        'price': price,
        'timeInForce': time_in_force
    }
```

- Price control needed
- Wait for better price
- Avoid slippage

6. Advanced Features (Bonus)

6.1 Stop-Limit Orders

Stop-limit orders combine stop and limit orders. When the stop price is reached, a limit order is triggered.

```
def place_stop_limit_order(self, symbol: str, side: str, quantity: float,
                           stop_price: float, limit_price: float,
                           time_in_force: str = 'GTC') -> Optional[Dict]:

    side = side.upper()
    symbol = symbol.upper()
    time_in_force = time_in_force.upper()

    if side not in ['BUY', 'SELL']:
        logger.error(f"Invalid side: {side}")
        return None

    if not self._validate_symbol(symbol):
        logger.error(f"Invalid symbol: {symbol}")
        return None

    params = {
        'symbol': symbol,
        'side': side,
        'type': 'STOP',
        'quantity': quantity,
        'price': limit_price,
        'stopPrice': stop_price,
        'timeInForce': time_in_force
    }

    try:
        self._log_request('STOP_LIMIT', params)
        order = self.client.futures_create_order(**params)
        self._log_response(order)

        logger.info(f"Stop-limit order placed successfully")
        if 'orderId' in order:
            logger.info(f"Order ID: {order['orderId']}")
        elif 'algoId' in order:
            logger.info(f"Algo ID: {order['algoId']}")
        else:
            logger.warning("Order placed but no orderId/algoId returned")


        return order
```

How It Works:

1. Order is placed but inactive
2. When market reaches stop_price, order activates
3. Activated order becomes a limit order at limit_price
4. Order fills at limit_price or better

7. Conclusion

The Project successfully delivers a fully functional trading bot for Binance Futures Testnet. All the Core requirements like

- Market and limit order placement
- Input validation
- Error handling
- Detailed logging system
- User-friendly CLI interface

have been met along with demonstrating my capability to deliver a given task by implementing a bonus feature of Stop-Limit orders.

The Bot also provides with,

- Account balance monitoring
- Open orders and position tracking and cancelling, etc

I have implemented an object oriented design for future modularity and each input and output of class methods are type-safe. With Minor improvements and extra added features, This assessment bot could be a viable option for the deployment at production.