# PythonTradingClient

*Release 2.2.1*

**Adhish Aggarwal**

**Dec 19, 2023**

# CONTENTS:

CHAPTER

# ONE

# CONFIG.JSON DOCUMENTATION

Listing 1: Trading Config

```
{
  "MODE": "PaperTrading",

  "DB_CONFIG": {
      "USER": "sec_master_read",
      "PASSWORD": "aargo.trade@123",
      "HOST": "103.177.180.4",
      "PORT": "5432",
      "DATABASE": "coresystem"
  },

  "BK_SERVER_ROUTER_LIVE": "<IP>:<PORT>",
  "BK_SERVER_ROUTER_PAPER": "<IP>:<PORT>",
  "STRATEGY_ID": -999,
  "CLIENT_ID": -999,

  "REGION_IP_CONFIG":{
      "OHIO": {
          "MARKET_DATA_SERVER_ROUTER": "<IP>:<PORT>",
          "MARKET_DATA_SERVER_PUBSUB": "<IP>:<PORT>",
          "TRADING_SERVER_ROUTER": "<IP>:<PORT>"
      },
      "LONDON": {
          "MARKET_DATA_SERVER_ROUTER": "<IP>:<PORT>",
          "MARKET_DATA_SERVER_PUBSUB": "<IP>:<PORT>",
          "TRADING_SERVER_ROUTER": "<IP>:<PORT>"
      },
      "OSAKA": {
          "MARKET_DATA_SERVER_ROUTER": "<IP>:<PORT>",
          "MARKET_DATA_SERVER_PUBSUB": "<IP>:<PORT>",
          "TRADING_SERVER_ROUTER": "<IP>:<PORT>"
      },

      "PAPER": {
          "TRADING_SERVER_ROUTER": "<IP>:<PORT>"
      }
  },
```

(continues on next page)

(continued from previous page)

```
  "ACCOUNT_ID_CONFIG": {
      "CME": -999,
      "EUREX": -999,
      "EURONEXT": -999
  },

  "CONNECTION_TIMEOUT": 10,
  "HEARTBEAT_INTERVAL": 0.5,
  "HEARTBEAT_TIMEOUT": 1.5,
  "RESPONSE_TIMEOUT": 10,
  "SYMBOL_COLNAME": "<SYMBOL_COLNAME>",
  "LOGGING_CONFIG": {
      "LOGGING_FILEPATH": "<FILEPATH_LOGFILE>",
      "SOCKET_FILEPATH": "<FILEPATH_LOGFILE>",
      "FILEHANDLER_LOGGING_LEVEL": "DEBUG",
      "STREAMHANDLER_LOGGING_LEVEL": "DEBUG"
  }
}
```

## 1.1 Explanation of Fields

- `MODE`: The mode in which the system should run. Set this to “PaperTrading” or “Live”.
- `BK_SERVER_ROUTER_*`: The IP address and port number for the live broker server router. Replace <*IP*> and <*PORT*> with the actual IP address and port number.
- `MARKET_DATA_SERVER_ROUTER`: The IP address and port number for the market data server router. Replace <*IP*> and <*PORT*> with the actual IP address and port number.
- `MARKET_DATA_SERVER_PUBSUB`: The IP address and port number for the market data server Pub/Sub. Replace <*IP*> and <*PORT*> with the actual IP address and port number.
- `TRADING_SERVER_ROUTER`: The IP address and port number for the trading server router. Replace <*IP*> and <*PORT*> with the actual IP address and port number.
- `GUI_SERVER_ROUTER`: The IP address and port number for the GUI server router. Replace <*IP*> and <*PORT*> with the actual IP address and port number.
- `CONNECTION_TIMEOUT`: The time, in seconds, that the system should wait while trying to establish a connection before timing out. In this example, it’s set to 10 seconds.
- `RESPONSE_TIMEOUT`: The time, in seconds, that the system should wait while waiting for a response before timing out. In this example, it’s set to 10 seconds.
- `CLIENT_ID`: The unique identifier for the client. In this example, it’s set to -999, but should be replaced with the actual client ID.
- `STRATEGY_ID`: The unique identifier for the strategy. In this example, it’s set to -999, but should be replaced with the actual strategy ID.
- `ACCOUNT_ID`: The unique identifier for the account. In this example, it’s set to -999, but should be replaced with the actual account ID.
- `SYMBOL_ID_COLNAME`: The column name that represents the symbol ID in the data. Replace <*symbol_id*> with the actual column name.
- `LOGGING_CONFIG`: A nested JSON object containing configuration settings for logging.

CHAPTER

TWO

# STATIC VARIABLES DOCUMENTATION

This document provides details on various variables used across the project. Each variable is listed with its possible values and descriptions.

## 2.1 exchange

Exchanges currently covered in the trading client. This is used to determine which exchange to use for a given trade.

**Possible Values:**

- CME : Chicago Mercantile Exchange
- ICE_L : ICE Futures Europe
- EUREX : Eurex Exchange
- EURONEXT : Euronext
- JPX : Osaka Exchange
- HKEX : Hong Kong Exchange

**Note:** **KINDLY** confirm the sub exchange before placing the order. For example, for CME, the sub exchange can be CME, CBOT, NYMEX, COMEX, etc. and trade server only has access to CBOT. So, if you place an order for CME, it will be rejected.

## 2.2 priceType

The type of price to use for a given trade. This is used to determine which price to use for a given trade.

**Possible Values:**

- LIMIT : Limit price
- MARKET : Market price
- STOP : Stop price
- STOP_LIMIT : Stop limit price

## 2.3 timeInForce

The time in force for a given trade. This is used to determine how long a trade should be active.

**Possible Values:**

- DAY : Day order
- GTC : Good 'til cancelled
- GTD : Good 'til date
- FOK : Fill or kill
- IOC : Immediate or cancel

## 2.4 symbolSource

Symbol sources currently covered in the trading client. This is used to determine which symbol source to use for a given trade.

**Possible Values:**

- SYMBOL_RIC : Reuters Instrument Code
- SYMBOL_ID : Symbol ID (aargo_security_master_3) table
- BLOOMBERG_CODE : Bloomberg Code

## 2.5 OrderStatus

The status of an order. This is used to determine the status of a given order.

**Possible Values:**

- UNACKED : Order has not been acknowledged by the exchange but has been received by the Trade Server
- ACKED : Order has been acknowledged by the exchange
- PARTIALLY_FILLED : Order has been partially filled
- FILLED : Order has been filled
- CANCELLED : Order has been cancelled
- REJECTED : Order has been rejected. Look at *rejection_reason* for more details

## 2.6 InstrumentStatus

The status of an instrument. This is used to determine the status of a given instrument.

**Possible Values:**

- TRADING_HALTED : Trading has been halted for the instrument
- AVAILABLE_TO_TRADE : Instrument is available to trade
- UNAVAILABLE_TO_TRADE : Instrument is unavailable to trade

- PRE_OPEN : Instrument is in pre-open state
- FAST_MARKET : Instrument is in fast market state
- POST_CLOSE : Instrument is in post-close state
- PRE_TRADE : Instrument is in pre-trade state
- UNKNOWN : Instrument status is unknown

**Note:** The values and descriptions provided here are subject to change as the project evolves. Always refer to the most recent version of this document for up-to-date information.

CHAPTER

THREE

# GUISERVERCLIENT MODULE

**class** pythonclient.GUIServerClient.**GUIServerClient**(*config_path: str*)

A client class for communicating with a GUI server, handling strategy state updates, and managing GUI-related data.

This class includes methods for initializing connections with the GUI server, handling login and logout processes, updating strategy states and parameters, and processing incoming GUI messages.

**gui_config**

Configuration object containing GUI server settings.

**Type**
ConfigObjectClass

**SYMBOL_ID_DICT**

Dictionary mapping symbol IDs to GUI row IDs.

**Type**
dict

**STRATEGY_STATE_DICT**

Dictionary mapping GUI row IDs to strategy states.

**Type**
dict

**SEND_QUEUE**

Queue for sending messages to the GUI server.

**Type**
Queue

**gui_server_dealer_socket**

Dealer socket for communication with the GUI server.

**Type**
zmq.Socket

**gui_server_monitor_socket**

Monitor socket for observing the dealer socket's events.

**Type**
zmq.Socket

**__init__**(*self*, *config_path: str*)

Initializes the GUIServerClient instance.

`guiserver_init`(*self*)

Initializes and sets up the connection with the GUI server.

`init_symbols`(*self*, *trading_symbols*)

Initializes trading symbols for GUI representation.

`gui_login`(*self*)

Sends a login request to the GUI server and handles the response.

`gui_logout`(*self*)

Logs out from the GUI server and closes the connection.

`update_state`(*self*, *row_indx*, *state*)

Sends a state update request for a specific strategy row.

`on_state_update`(*self*, *row_index*, *state*, *strategy_id*, *client_id*, *error_code*, *error_reason*)

Abstract method to be implemented for handling state updates.

`on_parameter_update`(*self*, *row_index*, *column*, *value*)

Abstract method to be implemented for handling parameter updates.

`__process_gui_message`(*self*, *message*)

Processes incoming messages from the GUI server.

`get_state`(*self*, *row_indx*)

Retrieves the state of a strategy at a given row index.

`get_state_symbol`(*self*, *symbol*)

Retrieves the state of a strategy associated with a given symbol.

`handle_gui_data`(*self*)

Handles incoming GUI data and processes messages.

`update_gui_parameter`(*self*, *indx*, *column*, *value*, *dtype*, *configurable*)

Sends a request to update a GUI parameter and updates local state.

`get_state`(*row_indx: int*)

Get the state of the strategy at the specified row index.

**Parameters**

`row_indx` (`int`) – Row index of the strategy whose state needs to be retrieved.

**Returns**

**state** – State of the row with the specified row index. 'START' if the row is in the "START" state, 'STOP' otherwise.

**Return type**

'START' | 'STOP' | None

`get_state_symbol`(*symbol*)

Gets the state of the specified symbol.

**Parameters**

`symbol` (`int | str`) – Symbol of the strategy whose state needs to be retrieved.

**Returns**

**state** – State of the strategy with the specified symbol. 'START' if the strategy is in the "START" state, 'STOP' otherwise.

**Return type**
‘START’ | ‘STOP’

`gui_login()`

Sends a login request to the GUI server and processes the response.

Constructs and sends a login request using FlatBuffers, waits for the server’s reply, and handles the login status. On failure, raises an exception with the error code.

`guiserver_init()`

Initializes the connection with the GUI server.

Sets up the dealer and monitor sockets for communication with the GUI server, registers callbacks, and logs in to the GUI server.

`handle_gui_data()`

Listens for and processes messages received from the GUI server.

`init_symbols`(*trading_symbols*)

Initializes trading symbols for GUI representation.

Maps trading symbols to GUI row IDs and initializes their state in the strategy state dictionary.

**Parameters**
`trading_symbols` (*list*) – List of trading symbols to be initialized.

**Raises**
`Exception` – If a duplicate symbol ID is encountered in the trading_symbols list.

`abstract on_parameter_update`(*parameter_ls*)

Abstract method to handle parameter updates.

This method should be implemented by the user to define behavior for strategy parameter updates.

**Parameters**
`parameter_ls` (*list*) – List of tuples containing the row index, column name, and new value of the updated parameter. - row index : int - column name : str - new value : Any

`abstract on_state_update`(*row_index*, *state*, *strategy_id*, *client_id*, *error_code*, *error_reason*)

This method should be implemented by the user to define behavior for strategy state updates.

This method is called when a strategy state update is received from the GUI server.

**Parameters**

- `row_index` (*int*) – The row index in the GUI where the state update occurred.
- `state` (*str*) –
  - ‘START’ if the strategy was started.
  - ’STOP’ if the strategy was stopped.
- `strategy_id` (*int*) – The ID of the strategy that was updated.
- `client_id` (*int*) – The client ID associated with the update.
- `error_code` (*int*) – Error code received in response to the state update request.
- `on_state_update` (*str*) – Error reason received in response to the state update request.

`update_gui_parameter`(*indx*, *column*, *value*, *dtype*, *configurable*)

Updates a parameter in the GUI.

Sends a parameter update request to the GUI server and optionally waits for an acknowledgment.

**Parameters**

- `indx` (*int*) – Row index of the parameter to be updated.
- `column` (*str*) – Column name of the strategy parameter to be updated.
- `value` (*Any*) – New value of the strategy parameter.
- `dtype` (*str*) – Data type of the value parameter (one of 'str', 'int', 'float').
- `configurable` (*bool*) – Indicates if the parameter is configurable from the GUI.

**Raises**

`AssertionError` – If dtype is not one of 'str', 'int', 'float', or if configurable is not a boolean.

`update_state`(*row_indx*, *state*)

Sends a request to update the state of a specific strategy row in the GUI.

Constructs a state update request using FlatBuffers and adds it to the send queue.

**Parameters**

- `row_indx` (*int*) – The row index in the GUI corresponding to the strategy.
- `state` (*str*) – Desired state to set for the strategy, either 'START' or 'STOP'.

**Raises**

`AssertionError` – If the provided state is neither 'START' nor 'STOP'.

CHAPTER

FOUR

# MARKETDATASERVERCLIENT MODULE

`class pythonclient.marketDataServerClient.`**`InnerDict`**

`class pythonclient.marketDataServerClient.`**`MarketDataServerClient`**`(`*config_path*`)`

A client class for communicating with a market data server, handling market data updates, and managing market data-related operations.

This class includes methods for initializing connections with the market data server, handling login and logout processes, processing various types of market data, and sending data requests to the market data server.

**`md_config`**

Configuration object containing market data server settings.

**Type**
ConfigObjectClass

**`SEND_QUEUE`**

Queue for sending messages to the market data server.

**Type**
Queue

**`SYMBOL_ID_MAPPING`**

Dictionary mapping symbol names to their respective IDs.

**Type**
dict

**`SYMBOL_NAME_MAPPING`**

Dictionary mapping symbol IDs to their names.

**Type**
dict

**`SYMBOL_EXCHANGE_MAPPING`**

Dictionary mapping symbol names to their respective exchanges.

**Type**
dict

**`__init__`**`(`*self*, *config_path*`)`

Initializes the MarketDataServerClient instance.

**`marketdata_server_init`**`(`*self*`)`

Initializes and sets up the connection with the market data server.

`marketdata_logout`(*self*, *dealer_socket*, *sub_socket*)
: Logs out from the market data server and closes the connections.

`on_trade_data`(*self*, *timestamp*, *symbol*, *price*, *size*, *open_interest*)
: Abstract method to process trade data received from the market data server.

`on_ohlcv_data`(*self*, *start_ts*, *end_ts*, *symbol*, *open*, *high*, *low*, *close*, *volume*, *open_interest*)
: Abstract method to process OHLCV data received from the market data server.

`on_bidask_data`(*self*, *symbol*, *status*, *timestamp*, *price*, *size*, *data_action*, *side*, *data_type*, *seq_num*)
: Abstract method to process bid/ask data received from the market data server.

`on_instrument_status`(*self*, *timestamp*, *symbol*, *symbol_status*, *symbol_status_code*, *req_status*)
: Abstract method to process instrument status data received from the market data server.

`__process_bid_ask`(*self*, *flatbuff_msg*, *type_*, *dt_ts*, *symbol*, *req_status*, *dtype_*)
: Processes bid/ask data and calls the appropriate method to handle the data.

`__process_marketdata_admin_reply`(*self*, *msg*)
: Processes the market data admin reply received from the market data server.

`handle_marketsub_data`(*self*, *socket*)
: Handles data received on the subscription socket for market data updates.

`handle_marketserver_data`(*self*, *socket*)
: Handles data received on the dealer socket for market data updates.

`marketdata_login`(*self*, *socket*)
: Sends a login request to the market data server.

`query_bidask_snapshot`(*self*, *symbol*)
: Sends a request for a bid/ask snapshot for a specific symbol.

`query_instrument_status`(*self*, *symbol*)
: Sends a request for the instrument status of a specific symbol.

`handle_marketserver_data`(*socket: Socket*)
: Handles the market data received from the dealer socket.

  Processes data received on the dealer socket for market data updates.

  **Parameters**
  `socket` (*zmq.Socket*) – The ZeroMQ dealer socket used for sending requests to the market data server.

`handle_marketsub_data`(*socket: Socket*)
: Handles the market data received from the subscribed socket.

  Processes data received on the subscription socket for market data updates.

  **Parameters**
  `socket` (*zmq.Socket*) – The ZeroMQ subscription socket used for receiving market data updates.

`marketdata_login`(*socket: Socket*)
: Sends a login request to the market data server and waits for a login reply.

  **Parameters**
  `socket` (*zmq.sugar.socket.Socket*) – The ZeroMQ dealer socket used for sending requests to the market data server.

`abstract on_bidask_data`(*timestamp*, *symbol*, *status*, *price*, *size*, *data_action*, *side*, *data_type*)

Abstract method to be implemented by user for processing bid/ask data received from the market data server.

**Parameters**

- **timestamp** (*datetime.datetime.utcfromtimestamp*) – The UTC timestamp of the data.
- **symbol** (*str | int*) – The trading symbol for which the data is received.
- **status** (*str*) – The status of the request. Either 'SUCCESS' or 'FAILURE'.
- **price** (*float*) – The price of the data.
- **size** (*float*) – The size of the data.
- **data_action** (*str*) – The action of the data. Either 'NEW', 'CHANGE', or 'DELETE'.
- **side** (*str. "BID" | "ASK"*) – The side of the order book. Either 'BID' or 'ASK'.
- **data_type** (*str. "INCREMENTAL" | "SNAPSHOT"*) – The type of the data. Either 'INCREMENTAL' or 'SNAPSHOT'.

`abstract on_instrument_status`(*timestamp*, *symbol*, *symbol_status*, *symbol_status_code*, *req_status*)

Abstract method for processing instrument status data received from the market data server.

**Parameters**

- **timestamp** (*datetime.datetime.utcfromtimestamp*) – The UTC timestamp of the data.
- **symbol** (*str | int*) – The trading symbol for which the data is received.
- **symbol_status** (*str*) – The status of the symbol. Either 'TRADING_HALTED', 'AVAILABLE_TO_TRADE', 'UNAVAILABLE_TO_TRADE', 'PRE_OPEN', 'FAST_MARKET', 'POST_CLOSE', 'PRE_TRADE', or 'UNKNOWN'.
- **symbol_status_code** (*int*) – The code of the symbol status.
- **req_status** (*str*) – The status of the request. Either 'SUCCESS' or 'FAILURE'.

`abstract on_ohlcv_data`(*start_ts*, *end_ts*, *symbol*, *open*, *high*, *low*, *close*, *volume*, *open_interest*)

Abstract method to be implemented by user for processing OHLCV data received from the market data server.

**Parameters**

- **start_ts** (*datetime.datetime.utcfromtimestamp*) – The UTC timestamp of the start of the data.
- **end_ts** (*datetime.datetime.utcfromtimestamp*) – The UTC timestamp of the end of the data.
- **symbol** (*str | int*) – The trading symbol for which the data is received.
- **open** (*float*) – The open price of the data.
- **high** (*float*) – The high price of the data.
- **low** (*float*) – The low price of the data.
- **close** (*float*) – The close price of the data.
- **volume** (*float*) – The volume of the data.
- **open_interest** (*float*) – The open interest of the data.

`abstract on_trade_data`(*timestamp*, *symbol*, *price*, *size*, *open_interest*)

Abstract method to be implemented by user for processing trade data received from the market data server.

**Parameters**

- **`timestamp`** (*`datetime.datetime.utcfromtimestamp`*) – The UTC timestamp of the data.
- **`symbol`** (*`str | int`*) – The trading symbol for which the data is received.
- **`price`** (*`float`*) – The price of the data.
- **`size`** (*`float`*) – The size of the data.
- **`open_interest`** (*`float`*) – The open interest of the data.

`query_bidask_snapshot`(*symbol*)

Sends a bid ask snapshot request to the market data server for a specific symbol.

**Parameters**

**`symbol`** – Symbol for which the snapshot is requested. But it should in *trading_universe* list.

`query_instrument_status`(*symbol*)

Sends a request to the market data server to get the status of a specified instrument.

**Parameters**

**`symbol`** (*`str`*) – The trading symbol for which the instrument status is requested.

# PYTHONTRADINGCLIENTBASE MODULE

**class** pythonclient.tradingClientBase.**TradingClientBase**(*config_path: str*, *trading_universe: list*, *symbol_source: str*)

A base class for trading clients, integrating functionalities from market data, trading, and GUI server clients.

**config**

Configuration object containing various settings.

**Type**
ConfigObjectClass

**SEND_QUEUE**

Queue for sending messages.

**Type**
Queue

**CONN**

Database connection object.

**Type**
connection

**CURSOR**

Database cursor for executing queries.

**Type**
cursor

**SYMBOL_ID_MAPPING**

Mapping from symbol names to symbol IDs.

**Type**
dict

**SYMBOL_EXCHANGE_MAPPING**

Mapping from symbol names to their respective exchanges.

**Type**
dict

**SYMBOL_NAME_MAPPING**

Mapping from symbol IDs to symbol names.

**Type**
dict

`__init__`(*self*, *config_path: str*, *trading_universe: list*, *symbol_source: str*)
: Initializes the TradingClientBase instance.

`__request_open_orders`(*self*)
: Requests all open orders from the trading server for all exchanges.

`__checks_trading_universe`(*self*, *trading_universe: list*, *symbol_source: str*)
: Validates the trading universe and symbol source.

`connect_postgre`(*self*)
: Connects to the PostgreSQL database specified in the configuration.

`shutdown`(*self*, *signum*, *frame*)
: Handles shutdown procedure and exits the program when SIGINT is received.

`subscribe_data`(*self*, *symbol_id_ls*)
: Subscribes to data based on the provided symbol ID list.

`register_poller`(*self*)
: Registers sockets with the ZMQ Poller for event polling.

`send_msg`(*self*)
: Sends messages from the send queue.

`recv_msg`(*self*)
: Receives messages from registered sockets.

`run`(*self*)
: Processes and sends messages from the send queue.

`connect_postgre`()
: Connects to the PostgreSQL database specified in the configuration.

  **Returns**
  A tuple containing the connection and cursor objects for the database.

  **Return type**
  tuple

`recv_msg`()
: Receives a message from the server.

  **Parameters**
  **socket** (*zmq.Socket*) – Socket to receive message from.

  **Returns**
  Received message.

  **Return type**
  bytes

`register_poller`()
: Registers all relevant sockets with the ZMQ Poller.

  This method is used to set up the poller for receiving messages from various sockets that the client interacts with.

`run`()
: This is the main loop of the client. It keeps running until the user presses Ctrl-C or *shutdown* is called

  It will block the main thread.

`send_msg()`

Sends a message to the server.

`shutdown`(*signum=None*, *frame=None*)

Handles shutdown procedure and exits the program.

This method is triggered when a SIGINT signal is received (e.g., Ctrl+C). It safely logs out from various servers, closes connections, and exits the program.

**Parameters**

- **`signum`** (*`int`*) – Signal number.
- **`frame`** (*`Frame`*) – Frame object.

`subscribe_data`(*symbol_id_ls*)

Subscribes to data based on the provided symbol ID list.

**Parameters**

**`symbol_id_ls`** (*`list`*) – List of symbol IDs to subscribe or unsubscribe.

**Raises**

- **`ValueError`** – If *symbol_id_ls* is empty.
- **`AssertionError`** – If *symbol_id_ls* is not a list.

CHAPTER

SIX

# TRADINGSERVERCLIENT MODULE

`class pythonclient.tradingServerClient.InnerDict`

`class pythonclient.tradingServerClient.TradeServerClient(`*config_path*`)`

A client class for interacting with a trade server, managing orders, and handling trade-related operations.

This class includes methods for initializing connections with the trade server, processing trade orders and updates, and sending various trade-related requests to the trade server.

`ts_config`

Configuration object containing trade server settings.

**Type**

ConfigObjectClass

`SEND_QUEUE`

Queue for sending messages to the trade server.

**Type**

Queue

`SYMBOL_ID_MAPPING`

Dictionary mapping symbol names to their respective IDs.

**Type**

dict

`SYMBOL_EXCHANGE_MAPPING`

Dictionary mapping symbol names to their respective exchanges.

**Type**

dict

`SYMBOL_NAME_MAPPING`

Dictionary mapping symbol IDs to their names.

**Type**

dict

`SYMBOL_ID_DICT`

Dictionary holding symbol IDs.

**Type**

dict

`STRATEGY_STATE_DICT`
Dictionary holding strategy states.

**Type**
dict

`__init__`(*self*, *config_path*)
Initializes the TradeServerClient instance.

`tradeserver_init`(*self*)
Initializes and sets up the connection with the trade server.

`tradeserver_logout`(*self*, *socket*)
Logs out from the trade server and closes the connection.

`handle_tradeserver_data`(*self*, *socket*)
Handles updates received from the trade server.

`__process_admin_reply`(*self*, *msg*)
Processes an admin reply message received from the trade server.

`on_position_update`(*self*, *account_id*, *strategy_id*, *client_id*, *symbol*, *position*, *status*)
Abstract method to handle position updates.

`on_pre_acknowledge`(*self*, *req_id*, *uuid*)
Abstract method to handle pre-acknowledged orders.

`on_acknowledge`(*self*, *ts*, *account_id*, *strategy_id*, *client_id*, *symbol*, *uuid*, *side*, *sent_price*, *sent_size*, *order_state*, *is_third_party_manual*)
Abstract method to handle order acknowledgments.

`on_fill`(*self*, *ts*, *account_id*, *strategy_id*, *client_id*, *symbol*, *uuid*, *side*, *sent_price*, *sent_size*, *stop_price*, *exec_price*, *exec_size*, *order_state*, *is_third_party_manual*)
Abstract method to handle order fills.

`on_cancel`(*self*, *ts*, *account_id*, *strategy_id*, *client_id*, *symbol*, *uuid*, *sent_price*, *sent_size*, *stop_price*, *remaining_size*, *order_state*, *is_third_party_manual*)
Abstract method to handle order cancellations.

`on_reject`(*self*, *ts*, *account_id*, *strategy_id*, *client_id*, *symbol*, *uuid*, *sent_price*, *stop_price*, *order_state*, *rejection_reason*, *is_third_party_manual*)
Abstract method to handle order rejections.

`on_order_details`(*self*, *ts*, *account_id*, *strategy_id*, *client_id*, *symbol*, *uuid*, *side*, *sent_price*, *sent_size*, *stop_price*, *order_state*, *exec_price*, *exec_size*, *price_type*, *time_in_force*)
Abstract method to handle detailed order information.

`__extract_order_details`(*self*, *msg*, *extract_ls*)
Extracts order details from a message.

`__process_trade_reply`(*self*, *msg*)
Processes trade reply messages from the trade server.

`tradeserver_login`(*self*, *socket*)
Logs into the trade server.

`__exchange_check`(*self*, *exchange*)
Validates the exchange name.

`__place_order_checks`(*self*, *price_type*, *time_in_force*, *exchange*, *price*, *stop_price*)

Performs validation checks for order placement.

`place_order`(*self*, *symbol*, *side*, *size*, *price_type*, *time_in_force*, *req_id*, *price*, *stop_price*, *auto_cancel*, *is_manual*)

Sends a new order to the trade server.

`query_order`(*self*, *req_id*, *uuid*, *exchange*, *is_manual*)

Queries the status of a specific order.

`cancel_order`(*self*, *req_id*, *uuid*, *exchange*, *is_manual*)

Sends a cancellation request for a specific order.

`modify_order`(*self*, *uuid*, *side*, *price*, *size*, *time_in_force*, *is_manual*)

Sends a request to modify an existing order.

`query_position_clientid`(*self*, *symbol*)

Queries the position details for a specific symbol based on client ID.

`query_position_accountid`(*self*, *symbol*)

Queries the position details for a specific symbol based on account ID.

`cancel_all_orders`(*self*, *req_id*, *exchange*, *is_manual*)

Requests cancellation of all active orders for a specific exchange.

`query_all_active_orders`(*self*, *req_id*, *exchange*, *isManual*)

Queries details of all active orders for a specific exchange.

`cancel_all_orders`(*req_id*, *exchange*, *is_manual=False*)

Sends a request to the trading server to cancel all active orders for a specific exchange.

**Parameters**

- **req_id** (`int`) – Unique identifier of the request. This identifier is used to get *uuid* of the request in *on_pre_acknowledge* callback.
- **exchange** (`str`) – The exchange for which to retrieve active orders. For detailed information on possible values, see *Static Variables Documentation*.
- **is_manual** (`bool`) – Flag indicating if the order is manually placed using python code.

`cancel_order`(*req_id*, *uuid*, *exchange*, *is_manual=False*)

Sends a request to the trading server to cancel a specific order. Perform initial checks on the order parameters before sending the request.

**Parameters**

- **req_id** (`int`) – Unique identifier of the request. This identifier is used to get *uuid* of the request in *on_pre_acknowledge* callback.
- **uuid** (`int`) – Unique identifier of the order to be cancelled.
- **exchange** (`str`) – The exchange where the order was placed. For detailed information on possible values, see *Static Variables Documentation*.
- **is_manual** (`bool`) – Flag indicating if the order is manually placed using python code.

**Raises**

`AssertionError` – If order_id is not an integer or is_manual is not a boolean.

`handle_tradeserver_data`(*socket: Socket*)

Handles updates received from a trading server.

This method waits for a message from the trading server, parses it, and then processes it based on its type. There are two types of messages that can be processed: *trade reply* messages and *admin reply* messages. Each type is handled by a dedicated method. If the message type is unsupported, the method logs a critical error and raises a ValueError.

**Parameters**

`socket` (*zmq.Socket*) – The ZeroMQ socket instance used for communication with the trading server.

**Raises**

`ValueError` – If the received message type from the trading server is unsupported, a ValueError is raised to indicate this issue.

`modify_order`(*uuid*, *side*, *price*, *size*, *time_in_force*, *is_manual=False*)

Sends a request to replace an existing order with new details.

**Parameters**

- **msg_id** (*int*) – Unique identifier of this message.
- **order_id** (*int*) – Identifier of the order to be replaced.
- **side** (*str*) – New order side, must be 'BUY' or 'SELL'.
- **price** (*float*) – New order price.
- **size** (*int*) – New order size.
- **time_in_force** (*str*) – New time-in-force policy of the order, e.g., 'IOC', 'FOK', 'DAY', 'GTD', 'GTC', 'OPG'
- **is_manual** (*bool*) – Flag indicating if the order is manually placed using python code.

**Raises**

`AssertionError` – If parameters do not conform to the expected types or values.

`abstract on_acknowledge`(*ts*, *account_id*, *strategy_id*, *client_id*, *symbol*, *uuid*, *side*, *sent_price*, *sent_size*, *order_state*, *is_third_party_manual*)

Processes order acknowledgment from the trade server.

This is an abstract method that should be implemented in a subclass by the user.

**Parameters**

- **ts** (*utc datetime*) – Timestamp of the acknowledgment.
- **account_id** (*int*) – The account ID associated with the order.
- **strategy_id** (*int*) – The strategy ID associated with the order.
- **client_id** (*int*) – The client ID associated with the order.
- **symbol** (*str or int*) – The trading symbol of the order.
- **uuid** (*int*) – The UUID of the order.
- **side** (*str*) – The side of the order ('BUY' or 'SELL').
- **sent_price** (*float*) – The price at which the order was placed.
- **sent_size** (*float*) – The size of the order.

- **order_state** (*str*) – The current state of the order. For detailed information on possible values, see *Static Variables Documentation*.
- **is_third_party_manual** (*bool*) – Indicates if the order was manually placed by a third party.

**abstract on_cancel**(*ts*, *account_id*, *strategy_id*, *client_id*, *symbol*, *uuid*, *sent_price*, *sent_size*, *stop_price*, *remaining_size*, *order_state*, *is_third_party_manual*)

Processes order cancellation updates from the trade server.

This is an abstract method that should be implemented in a subclass by the user.

**Parameters**

- **ts** (*UTC datetime*) – Timestamp of the cancellation.
- **account_id** (*int*) – The account ID associated with the order.
- **strategy_id** (*int*) – The strategy ID associated with the order.
- **client_id** (*int*) – The client ID associated with the order.
- **symbol** (*str or int*) – The trading symbol of the order.
- **uuid** (*int*) – The UUID of the order.
- **sent_price** (*float*) – The price at which the order was placed.
- **sent_size** (*float*) – The size of the order.
- **stop_price** (*float*) – The stop price of the order, if applicable.
- **remaining_size** (*float*) – The remaining size of the order after cancellation.
- **order_state** (*str*) – The current state of the order. For detailed information on possible values, see *Static Variables Documentation*.
- **is_third_party_manual** (*bool*) – Indicates if the order was manually placed by a third party.

**abstract on_fill**(*ts*, *account_id*, *strategy_id*, *client_id*, *symbol*, *uuid*, *side*, *sent_price*, *sent_size*, *stop_price*, *exec_price*, *exec_size*, *order_state*, *is_third_party_manual*)

Handles order fill updates from the trade server.

This is an abstract method that should be implemented in a subclass by the user.

**Parameters**

- **ts** (*utc datetime*) – Timestamp of the fill.
- **account_id** (*int*) – The account ID associated with the order.
- **strategy_id** (*int*) – The strategy ID associated with the order.
- **client_id** (*int*) – The client ID associated with the order.
- **symbol** (*str or int*) – The trading symbol of the order.
- **uuid** (*int*) – The UUID of the order.
- **side** (*str*) – The side of the order (‘BUY’ or ‘SELL’).
- **sent_price** (*float*) – The price at which the order was placed.
- **sent_size** (*float*) – The size of the order.
- **stop_price** (*float*) – The stop price of the order, if applicable.

- **exec_price** (*float*) – The execution price of the order.
- **exec_size** (*float*) – The executed size of the order.
- **order_state** (*str*) – The current state of the order. For detailed information on possible values, see *Static Variables Documentation*.
- **is_third_party_manual** (*bool*) – Indicates if the order was manually placed by a third party.

**abstract on_order_details**(*ts*, *account_id*, *strategy_id*, *client_id*, *symbol*, *uuid*, *side*, *sent_price*, *sent_size*, *stop_price*, *order_state*, *exec_price*, *exec_size*, *price_type*, *time_in_force*)

Receives detailed information about a specific order from the trade server. Usually called in response to a query order request.

This is an abstract method that should be implemented in a subclass.

**Parameters**

- **ts** (*utc datetime*) – Timestamp of the order detail.
- **account_id** (*int*) – The account ID associated with the order.
- **strategy_id** (*int*) – The strategy ID associated with the order.
- **client_id** (*int*) – The client ID associated with the order.
- **symbol** (*str or int*) – The trading symbol of the order.
- **uuid** (*int*) – The UUID of the order.
- **side** (*str*) – The side of the order ('BUY' or 'SELL').
- **sent_price** (*float*) – The price at which the order was placed.
- **sent_size** (*float*) – The size of the order.
- **stop_price** (*float*) – The stop price of the order, if applicable.
- **order_state** (*str*) – The current state of the order. For detailed information on possible values, see *Static Variables Documentation*.
- **exec_price** (*float*) – The execution price of the order.
- **exec_size** (*float*) – The executed size of the order.
- **price_type** (*str*) – The type of the order price. For detailed information on possible values, see *Static Variables Documentation*.
- **time_in_force** (*str*) – The time-in-force policy of the order. For detailed information on possible values, see *Static Variables Documentation*.

**abstract on_position_update**(*account_id*, *strategy_id*, *client_id*, *symbol*, *position*, *status*)

Handles position updates from the trade server.

This is an abstract method that should be implemented by the user in a subclass.

**Parameters**

- **account_id** (*int*) – The account ID associated with the position.
- **strategy_id** (*int*) – The strategy ID associated with the position.
- **client_id** (*int*) – The client ID associated with the position.
- **symbol** (*str or int*) – The symbol associated with the position.

- **position** (*float*) – The current position quantity.
- **status** (*str*) – The status of the position update.

**abstract on_pre_acknowledge**(*req_id*, *uuid*)

Processes pre-acknowledgment of an order from the trade server.

This is an abstract method that should be implemented in a subclass.

**Parameters**

- **req_id** (*int*) – The request ID associated with the request.
- **uuid** (*int*) – The UUID of the request.

**abstract on_reject**(*ts*, *account_id*, *strategy_id*, *client_id*, *symbol*, *uuid*, *sent_price*, *stop_price*, *order_state*, *rejection_code*, *rejection_reason*, *is_third_party_manual*)

Handles order rejection notifications from the trade server.

This is an abstract method that should be implemented in a subclass by the user.

**Parameters**

- **ts** (*utc datetime*) – Timestamp of the rejection.
- **account_id** (*int*) – The account ID associated with the order.
- **strategy_id** (*int*) – The strategy ID associated with the order.
- **client_id** (*int*) – The client ID associated with the order.
- **symbol** (*str or int*) – The trading symbol of the order.
- **uuid** (*int*) – The UUID of the order.
- **sent_price** (*float*) – The price at which the order was placed.
- **stop_price** (*float*) – The stop price of the order, if applicable.
- **order_state** (*str*) – The current state of the order. For detailed information on possible values, see *Static Variables Documentation*.
- **rejection_code** (*int*) – The rejection code for the order.
- **rejection_reason** (*str*) – The reason for order rejection.
- **is_third_party_manual** (*bool*) – Indicates if the order was manually placed by a third party.

**place_order**(*symbol*, *side*, *size*, *price_type*, *time_in_force*, *req_id*, *price=None*, *stop_price=None*, *auto_cancel=True*, *is_manual=False*)

Sends a new order to the trading server.

Places an order with the specified details by sending a request to the trading server. Performs initial checks on the order parameters before sending the request.

**Parameters**

- **symbol** (*str | int*) – Unique identifier of the trading symbol.
- **side** (*str*) – The side of the order (‘BUY’ or ‘SELL’).
- **size** (*int*) – Order size.
- **price_type** (*str*) – Type of the order price. For detailed information on possible values, see *Static Variables Documentation*.

- **time_in_force** (*str*) – Time-in-force policy of the order. For detailed information on possible values, see *Static Variables Documentation*.
- **req_id** (*int*) – Unique identifier of the order provided by the client. This identifier is used to track the request in the *Pre-Acknowledged* state.
- **price** (*float | None*) – Order price.
- **stop_price** (*float | None*) – Stop Order price.
- **auto_cancel** (*bool*) – Flag indicating if the order should be automatically cancelled if client is disconnected from the trading server or python crashes/errors out.
- **is_manual** (*bool*) – Flag indicating if the order is manually placed using python code.

**Raises**

- **AssertionError** – If any of the parameters do not conform to the expected type or set of values.
- **ValueError** – If the symbol is not in the trading universe or if the strategy state is not started.

**Returns**
**order_id** – Identifier of the order that was sent. Use this identifier to get *UUID*.

**Return type**
int

`query_all_active_orders`(*req_id*, *exchange*, *isManual=False*)

Sends a request to the trading server to retrieve details of all active orders for a specific exchange.

**Parameters**

- **req_id** (*int*) – Unique identifier of the request. This identifier is used to get *uuid* of the request in *on_pre_acknowledge* callback.
- **exchange** (*str*) – The exchange for which to retrieve active orders. For detailed information on possible values, see *Static Variables Documentation*.
- **isManual** (*bool, optional*) – Indicates whether the request is manual. Default is False.

`query_order`(*req_id*, *uuid*, *exchange*, *is_manual=False*)

Sends a request to the trading server to query the status of a specific order.

**Parameters**

- **req_id** – Unique identifier of the request. This identifier is used to get *uuid* of the request in *on_pre_acknowledge* callback.
- **uuid** (*int*) – Unique identifier of the order.
- **exchange** (*str*) – The exchange where the order was placed. For detailed information on possible values, see *Static Variables Documentation*.
- **is_manual** (*bool*) – Flag indicating if the order is manually placed using python code.

**Raises**
**AssertionError** – If order_id is not an integer or is_manual is not a boolean.

`query_position_accountid`(*symbol*)

Sends a request to the trading server for position details of a specific symbol associated with the account id.

**Parameters**
`symbol` (*int | str*) – Unique identifier of the trading symbol to query. It must be in trading universe.

**Raises**
`AssertionError` – If symbol_id is not an integer.

`query_position_clientid`(*symbol*)

Sends a request to the trading server for position details of a specific symbol associated with the client id.

**Parameters**
`symbol` (*int | str*) – Unique identifier of the trading symbol to query. It must be in trading universe.

**Raises**
`AssertionError` – If symbol_id is not an integer.

`tradeserver_init()`

Loads the configuration and initializes the connection with all the trading server in the config.

`tradeserver_login`(*socket: Socket*)

Logs into the trading server.

Sends a login request to the trading server and handles the response, updating the socket state accordingly.

**Parameters**
`socket` (*zmq.Socket*) – The ZeroMQ socket instance used for communication with the trading server.

**Raises**
`ValueError` – If login to the trading server fails.

`tradeserver_logout`(*socket: Socket*)

Logs out from the trading server and closes the socket connection.

Sends a logout request to the trading server and closes the connection based on the reply. The state of the socket connection is updated accordingly.

**Parameters**
`socket` (*zmq.Socket*) – The ZeroMQ socket instance used for communication with the trading server.

# CHAPTER SEVEN

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

# INDEX