# LedgerX API Python SDK Documentation

*Release 0.0.1*

**Amr Ali**

**Sep 26, 2017**

# Contents

This Python API SDK allows you to communicate with the LedgerX platform. If you'd like to build your own SDK please refer to our ZMQ API documentation.

Python 3.5 is required.

For API SDK usage documentation, please visit API SDK documentation

CHAPTER 1

# Easy Installation

You could also use pip to install LedgerX API Python SDK from our PyPI repository:

```
# pip3 install ledgerx-python
```

Need more help with installation? See *Installation*.

CHAPTER 2

# User Guide

## Installation

LedgerX API Python SDK can be installed in a couple of ways. You could either install it directly from the code repository or from the PyPI repository.

### Installation from the PyPI repository

You need to install pip to be able to install LedgerX API Python SDK from the PyPI repository. If you are using a Debian based system, please execute the following commands:

```
$ sudo apt-get install python3-pip python3-virtualenv
```

Or any other way to make virtualenv and pip available on your system. If you have ZeroMQ installed then the system will detect that and use your installed version. After that you'll need to install the client either inside your virtualenv or globally via:

```
$ pip3 install ledgerx-python
```

**Note:** If you opted for installing your own 0MQ version, please make sure that the versions you install are compatiable with PyZMQ.

### Installation from the code repository

You will need to install git and virtualenv, and of course you'll have to have Python v3.5.x installed. After cloning the code repository you'll need to run a similar command as the above for installing the client:

```
$ pip3 install cloned_ledgerx_client_directory/
```

# Protocol Expected Behavior

The exchange expects a certain behavior from clients. All communication is asynchronous and requires clients to implement basic message state tracking.

Acknowledge `ACK` messages are always the first response to any message submitted to the exchange. `ACK` messages will contain a generated Message ID `MID` which will be used to track the state of each message through its life cycle but also acts as an entry ID if the submitted message is an order type. After a message is submitted the context blocks for a certain period/timeout for expected `ACK` message(s). these message can be processed via your processor.

# Exchange Unavailability Scenarios

- Client sends a message and doesn't receive an `ACK` (`S1`)
- Client sends a message and receives an `ACK` but not a response (`S2`)
- Client sends a message and receives both an `ACK` and a response (`S3`)

## Scenario 1

In this scenario it is the client's responsibility to resend a message until it receives an `ACK`.

## Scenario 2

In this scenario it is the client's responsibility to maintain a queue of messages that received an `ACK`.

Once the exchange is back online a client could query the exchange for a set of messages they might have missed through `MessageReplay`.

## Scenario 3

Once a response is received that contains the same `MID` for that particular message, the client is now free to discard the 3 messages (i.e., the message sent, the `ACK` received, and the final response).

Trade requests (e.g., limit order, quotes) are an exception in this scenario as the response to these requests is an unpredictable number of action reports as long as the client is connected to the exchange. Therefore the client must not discard the trade request's `MID` until the last action report received reports a size of **zero**. If the connection is severed before the action report's size reaches zero the client must query the exchange for outstanding trade notifications through `MessageReplay` to synchronize their internal CLOB with the exchange's.

# Batch Messages

Batch messages provides the ability to send up to 200 messages in a single call. These messages are executed in the order provided by the client. Unlike stream messages, Batch operations have different behaviour.

Once a batch message is sent to the exchange, an `ACK` is received holding the `MID` of the batch message sent. A Batch message should be received after the `ACK` that contains a list of `ACK` for each message contained within the client Batch message. At this point, it's the client's responsility to maintain these `ACK` replies. Shortly, responses for messages sent within the client's Batch message follow each with the `MID` of its corresponding client's message.

## Exchange Run IDs and Run Change

All messages received contain a `run_id` field that contains an identifier of the current exchange run. In case the exchange is restarted, a different id is generated and set to all messages `run_id` field. At this point all open orders that existed before exchange restart are canceled. So, you should make sure you know which orders during the old run.

`run_id` is likely to first change on either `ACK` messages or heartbeats. In case an `ACK` is received with the newer `run_id` it means the message sent executed within the new run.

## Usage Examples

LedgerX API Python SDK is very simple to use. But let's discover a few corner stone concepts about the API first.

### Library Context

The `Context` class is where the main I/O loop resides and also where message dispatch takes place. This class represents the context for the entire library, it needs to be initialized with a subclass of `MessageProcessor` that implements all of the abstract methods to respond to received messages from the exchange.

This class implements an interface similar to Python's `Thread`. `Start`, `Stop`, and `Join` behave as expected in `Thread`.

Some functions on `Context` such as `submit` are there to be extensible but are not supposed to be used directly.

`Context` expects the FQDN of the exchange endpoint to be specified as its first argument. You can specify *ledgerx.sdk.settings.EXCHANGE_CTE_FQDN* to have the context connect to the Customer Test Environment (CTE), or use *ledgerx.sdk.settings.EXCHANGE_TRADE_FQDN* to trade live on the market.

### Message Processing

The `MessageProcessor` class is an abstract class to provide an interface contract for subclasses to implement all the abstract methods for handling messages and high level socket state events.

### Logging

By default the SDK won't show logs to stdout. you must enable stream logging first to see any logs from the SDK. In your application add a `logging.StreamHandler` to your logger.

```
import sys
import logging

logger = logging.getLogger()
logger.addHandler(logging.StreamHandler(sys.stdout))
```

### Key Pair Object

The `KeyPair` class is used to represent an EC key pair object that stores the private and public key pair of 0MQ ZAP/CURVE mechanism.

This class also supports generation of new key pairs through the `generate()`, saving and loading key pair certificates through `save_certificate()` and `load_certificate()`.

## Fetching Exchange's Public Key

To fetch the Customer Test Environment exchange public key you can issue the following command:

```
$ pyledgerx --fetch-exchange-key test.ledgerx.com
```

Specifying a different domain name (e.g., `trade.ledgerx.com`) will fetch a different exchange's public key.

## Initializing Context Example

```
from ledgerx.protocol.crypto import KeyPair

from ledgerx.sdk import settings
from ledgerx.sdk.context import Context
from ledgerx.sdk.processor import MessageProcessor

class MyMessageProcessor(MessageProcessor):
    pass

client_keypair = KeyPair.generate()
server_public_key = b'somepublickey'
context = Context(settings.EXCHANGE_CTE_FQDN, client_keypair.private,
    client_keypair.public, server_public_key, MyMessageProcessor)

context.start()
```

## Submitting and Handling Messages Example

```
from ledgerx.protocol.crypto import KeyPair

from ledgerx.sdk import settings
from ledgerx.sdk.context import Context
from ledgerx.sdk.commands import get_cmd_version
from ledgerx.sdk.processor import MessageProcessor

class MyMessageProcessor(MessageProcessor):
    def handle_run_change(self, run_id, prev_run_id):
        # If this handler is invoked, open orders
        # are considered canceled and should be
        # purged from your local state
        pass
    def handle_action_report(self, msg, prev_msg):
        # If this is the first action report, prev_msg will be
        # the limit_order command's message.
        pass
    def handle_contract_detail(self, msg, prev_msg): pass
    def handle_status_message(self, msg, prev_msg):
        # You can handle all status messages here (e.g., ACK messages).
        pass
    def handle_batch_message(self, msg, prev_msg):
```

```
        # You can handle all batch messages here
        # a batch message is received when
        # it is a batch of acks for a sent batch
        pass
    def handle_book_top(self, msg, prev_msg):
        # This handler is for the book top message for the limit order.
        pass
    def handle_book_state_snapshot(self, msg, prev_msg): pass
    def handle_heartbeat(self, msg, prev_msg): pass

client_keypair = KeyPair.generate()
server_public_key = b'somepublickey'
context = Context(settings.EXCHANGE_CTE_FQDN, client_keypair.private,
    client_keypair.public, server_public_key, MyMessageProcessor)

context.start()

cmd = get_cmd_version('0.0.1').Commands(context)
cmd.limit_order()

context.join()
```

## Trading Session Example

```
import os
import time
from random import randint

from ledgerx.protocol.crypto import KeyPair

from ledgerx.sdk import settings
from ledgerx.sdk.context import Context
from ledgerx.sdk.commands import get_cmd_version
from ledgerx.sdk.processor import MessageProcessor

# A list of contracts on the exchange
contracts = []

class MyMessageProcessor(MessageProcessor):

    def handle_run_change(self, run_id, prev_run_id):
        # If this handler is invoked, open orders
        # are considered canceled and should be
        # purged from your local state
        pass

    def handle_action_report(self, msg, prev_msg):
        # If this is the first action report, prev_msg will be
        # the limit_order command's message.
        print(msg, prev_msg)

    def handle_contract_detail(self, msg, prev_msg):
        global contracts; contracts.append(msg)

    def handle_status_message(self, msg, prev_msg):
        # You should receive an ACK status message for the get_contract
```

```
        # call then another ACK message for the limit order submitted. You
        # might receive other status messages depending on the parameters
        # passed to the commands you issued.
        print(msg.status, msg.message, prev_msg)

    def handle_batch_message(self, msg, prev_msg):
        pass

    def handle_book_top(self, msg, prev_msg):
        print(msg, prev_msg)

    def handle_book_state_snapshot(self, msg, prev_msg):
        # This will receive the MessageBookStateSnapshot that contains all
        # resting limit orders on a contract. prev_msg will be the
        # MessageGetBookState request message.
        print(msg, prev_msg)

    def handle_heartbeat(self, msg, prev_msg):
        # Upon receiving a heartbeat, note the `interval_ms` property.
        # This value is the number of milliseconds after which you should
        # have received the next heartbeat message. If you do not receive
        # another heartbeat after this interval has elapsed, you may assume
        # the exchange went offline.
        print(msg, prev_msg)

class StoreMIDCallback(object):
    """\
    A callback to store the generated Message ID (MID) which is also the
    ID that points to an entry on the exchange's CLOB.
    """
    def __init__(self):
        self._mid = None

    def __call__(self, msg, prev_msg):
        self._mid = msg.mid

    @property
    def entry_id(self):
        return self._mid

# You have to upload your public key to your account
client_keypair = KeyPair.generate()

# Acquire server's public key and set it here
server_public_key = b'somepublickey'

# Establish context
context = Context(settings.EXCHANGE_CTE_FQDN, client_keypair.private,
    client_keypair.public, server_public_key, MyMessageProcessor)

context.start()

cmd = get_cmd_version('0.0.1').Commands(context)

# Acquire a list of contracts on the exchange
cmd.get_contract(all_contracts=True)
time.sleep(3) # allow enough time to build the contracts list
```

```
if not len(contracts):
    print("loading contracts failed")
    # Timeout after 1 second in case 0MQ lingering option was set and there
    # are messages waiting to be processed. We don't want those in case we
    # need to exit.
    context.join(1)
    raise SystemExit

# Pick a contract at random
contract = contracts[randint(0, len(contracts) - 1)]

# Submit a limit order
sm_cb = StoreMIDCallback()
item = cmd.limit_order(size=1, price_in_cents=5000,
    contract_id=contract.contract_id)
item.callback = sm_cb
time.sleep(3) # allow enough time to get a reply with the Entry ID

print("Order's ID: {}".format(sm_cb.entry_id))

# Timeout after 3 seconds just in case there are lingering messages that
# need to be consumed from 0MQ internal queues.
context.join(3) # Timeout after 3 seconds
```

## Orders Batch Example

```
import os
import time
from random import randint

from ledgerx.protocol.crypto import KeyPair

from ledgerx.sdk import settings
from ledgerx.sdk.context import Context
from ledgerx.sdk.commands import get_cmd_version
from ledgerx.sdk.processor import MessageProcessor
from ledgerx.api.client import v0_0_1 as api

# A list of contracts on the exchange
contracts = []

class MyMessageProcessor(MessageProcessor):

    def handle_run_change(self, run_id, prev_run_id):
        # If this handler is invoked, open orders
        # are considered canceled and should be
        # purged from your local state
        pass

    def handle_action_report(self, msg, prev_msg):
        # This will receive action reports if any orders
        # were placed.
        # Even though you will receive action reports in
        # the callback for the order message (if any exist)
        pass
```

```
    def handle_contract_detail(self, msg, prev_msg):
        global contracts; contracts.append(msg)

    def handle_status_message(self, msg, prev_msg):
        # You should receive an ACK status message for the get_contract
        # call then another ACK message for batch message sent.
        # You might receive other status messages depending on
        # the parameters passed to the commands you issued.
        pass

    def handle_batch_message(self, msg, prev_msg):
        # You should receive a batch message of acks for the
        # batch message sent.
        for ack in msg.messages:
            # do something with an ack
            continue

    def handle_book_top(self, msg, prev_msg):
        pass

    def handle_book_state_snapshot(self, msg, prev_msg):
        # This will receive the MessageBookStateSnapshot that contains all
        # resting limit orders on a contract. prev_msg will be the
        # MessageGetBookState request message.
        pass

    def handle_heartbeat(self, msg, prev_msg):
        # Upon receiving a heartbeat, note the `interval_ms` property.
        # This value is the number of milliseconds after which you should
        # have received the next heartbeat message. If you do not receive
        # another heartbeat after this interval has elapsed, you may assume
        # the exchange went offline.
        pass

class ProcessOrderResponses(object):
    """\
    A callback to process order responses
    Responses might be:
    - MessageStatus
    - MessageActionReport
    """
    def __init__(self):
        self._mid = None

    def __call__(self, msg, prev_msg):
        if isinstance(msg, api.MessageStatus):
            self.process_status(msg)
        else:
            self.process_ar(msg)

    def process_ar(self, msg):
        """
        process an action report
        """
        pass

    def process_status(self, msg):
        """
```

```
        process a status message
        """
        if msg.status == api.MessageStatus.STATUS_ORDER_SUCCESS:
            print("Order {} placed successfully".format(msg.mid))
        else:
            print("Order {} failed with code {}".format(msg.mid, msg.status))

# You have to upload your public key to your account
client_keypair = KeyPair.generate()

# Acquire server's public key and set it here
server_public_key = b'somepublickey'

# Establish context
context = Context(settings.EXCHANGE_CTE_FQDN, client_keypair.private,
    client_keypair.public, server_public_key, MyMessageProcessor)

context.start()

cmd = get_cmd_version('0.0.1').Commands(context)

# Acquire a list of contracts on the exchange
cmd.get_contract(all_contracts=True)
time.sleep(3) # allow enough time to build the contracts list

if not len(contracts):
    print("loading contracts failed")
    # Timeout after 1 second in case 0MQ lingering option was set and there
    # are messages waiting to be processed. We don't want those in case we
    # need to exit.
    context.join(1)
    raise SystemExit

# Pick a contract at random
random_contract = lambda : contracts[randint(0, len(contracts) - 1)]

# create a set of limit orders
# note: a batch message can contain different types of messages
# and not necessarily of the same type
orders = []
for _ in range(10):
    contract = random_contract()
    order = api.MessageLimitOrder()
    order.size = 1
    order.price_in_cents = 100
    order.contract_id=contract.contract_id
    orders.append(order)

(batch_item, items) = cmd.batch(messages=orders)
for idx, item in enumerate(items):
    item.callback = ProcessOrderResponses()
    print("Order no.{} ID: {}".format(idx, item.msg.mid))

time.sleep(2) # Wait for replies

# Timeout after 3 seconds just in case there are lingering messages that
# need to be consumed from 0MQ internal queues.
context.join(3) # Timeout after 3 seconds
```

## Run Change example

```
import os
import time
from random import randint

from ledgerx.protocol.crypto import KeyPair

from ledgerx.sdk import settings
from ledgerx.sdk.context import Context
from ledgerx.sdk.commands import get_cmd_version
from ledgerx.sdk.processor import MessageProcessor

# A list of contracts on the exchange
contracts = []
orders = []

class MyMessageProcessor(MessageProcessor):

    def handle_run_change(self, run_id, prev_run_id):
        # If this handler is invoked, open orders
        # are considered canceled and should be
        # purged from your local state
        global orders; orders = []

    def handle_action_report(self, msg, prev_msg):
        # If this is the first action report, prev_msg will be
        # the limit_order command's message.
        print(msg, prev_msg)

    def handle_contract_detail(self, msg, prev_msg):
        global contracts; contracts.append(msg)

    def handle_status_message(self, msg, prev_msg):
        # You should receive an ACK status message for the get_contract
        # call then another ACK message for the limit order submitted. You
        # might receive other status messages depending on the parameters
        # passed to the commands you issued.
        print(msg.status, msg.message, prev_msg)

    def handle_batch_message(self, msg, prev_msg):
        pass

    def handle_book_top(self, msg, prev_msg):
        print(msg, prev_msg)

    def handle_book_state_snapshot(self, msg, prev_msg):
        # This will receive the MessageBookStateSnapshot that contains all
        # resting limit orders on a contract. prev_msg will be the
        # MessageGetBookState request message.
        print(msg, prev_msg)

    def handle_heartbeat(self, msg, prev_msg):
        # Upon receiving a heartbeat, note the `interval_ms` property.
        # This value is the number of milliseconds after which you should
        # have received the next heartbeat message. If you do not receive
        # another heartbeat after this interval has elapsed, you may assume
        # the exchange went offline.
```

```
        print(msg, prev_msg)

class StoreMIDCallback(object):
    """\
    A callback to store the generated Message ID (MID) which is also the
    ID that points to an entry on the exchange's CLOB.
    """
    def __init__(self):
        self._mid = None

    def __call__(self, msg, prev_msg):
        self._mid = msg.mid
        global orders; orders.append(msg)

    @property
    def entry_id(self):
        return self._mid

# You have to upload your public key to your account
client_keypair = KeyPair.generate()

# Acquire server's public key and set it here
server_public_key = b'somepublickey'

# Establish context
context = Context(settings.EXCHANGE_CTE_FQDN, client_keypair.private,
    client_keypair.public, server_public_key, MyMessageProcessor)

context.start()

cmd = get_cmd_version('0.0.1').Commands(context)

# Acquire a list of contracts on the exchange
cmd.get_contract(all_contracts=True)
time.sleep(3) # allow enough time to build the contracts list

if not len(contracts):
    print("loading contracts failed")
    # Timeout after 1 second in case 0MQ lingering option was set and there
    # are messages waiting to be processed. We don't want those in case we
    # need to exit.
    context.join(1)
    raise SystemExit

# Pick a contract at random
contract = contracts[randint(0, len(contracts) - 1)]

# Submit a limit order
sm_cb = StoreMIDCallback()
item = cmd.limit_order(size=1, price_in_cents=5000,
    contract_id=contract.contract_id)
item.callback = sm_cb
time.sleep(3) # allow enough time to get a reply with the Entry ID

print("Order's ID: {}".format(sm_cb.entry_id))

# Timeout after 3 seconds just in case there are lingering messages that
# need to be consumed from 0MQ internal queues.
```

```
context.join(3) # Timeout after 3 seconds
```

# API Reference

## CLI Utilities

## Exchange Commands

**module** ledgerx.sdk.commands

**synopsis** Commands module to issue orders or quotes to the exchange.

**author** Amr Ali <amr@ledgerx.com>

ledgerx.sdk.commands.**get_cmd_version**(*version*)
Get a specific commands module that corresponds to a particular protocol version.

**Parameters** **version** – A version string in the format major.minor.build.

**Returns** None or a commands module corresponding to the version supplied.

## Library Context

## Mixin Helpers

**module** ledgerx.sdk.mixin

**synopsis** A module containing useful utilities.

**author** Amr Ali <amr@ledgerx.com>

ledgerx.sdk.mixin.**public**(*obj*)
A decorator to avoid retyping function/class names in __all__.

## Message Processing

## Setting Variables and Configurations

**module** ledgerx.sdk.settings

**synopsis** A module that lets clients specify settings.

## Memory Store Facilities

**module** ledgerx.sdk.store

**synopsis** Memory stores facilities.

**author** Amr Ali <amr@ledgerx.com>

**class** ledgerx.sdk.store.**MemoryStore**(*shared=False*)
An inter-process in-memory store to provide common facilities for loading data from different sources. Namely from another process. This container can be constructed to not be shared among forked processes.

**class** ledgerx.sdk.store.**IDStoreItem**(*msg=None*, *cb=<function IDStoreItem.<lambda>>*)
A class to represent a MessageIDStore's item.

**Parameters**

- **msg** – A property that holds a message object.
- **callback** – A callable that accepts two arguments, the first is the just received message object and the second is the previous message with the same message ID.

**class** ledgerx.sdk.store.**MessageIDStore**
A store to keep track of the IDs of in-progress messages.

## Miscellaneous Utilities

**module** ledgerx.sdk.utils

**synopsis** Miscellaneous tools.

**author** Amr Ali <amr@ledgerx.com>

### Metaclass Patterns

**module** ledgerx.utils.meta

**synopsis** Module for metaclass patterns.

**author** Amr Ali <amr@ledgerx.com>

**class** ledgerx.sdk.utils.meta.**KindSingletonMeta**(*name*, *bases*, *attrs*)
A singleton pattern metaclass that uses the underlying class' name as the unique constraint.

**class** ledgerx.sdk.utils.meta.**SingletonAbstractMeta**(*name*, *bases*, *attrs*)
A Singleton pattern abstract metaclass.

ledgerx.sdk.utils.**import_versions**(*basefile*, *package*)
Iterate over version files and import all version modules.

**Returns** A dictionary of versions *x.x.x* and imported version modules.

ledgerx.sdk.utils.**reraise**(*ex*, *message=None*)
Reraise the exception last happened with the original traceback.

**Parameters**

- **ex** (*A subclass of Exception.*) – The exception to be raised instead of the original one.
- **message** (*str*) – An optional message to replace the old exception's message.

**Raises** ex

## Version Utilities

**module** ledgerx.sdk.version

**synopsis** A module that contain version details for the package.

**author** Amr Ali <amr@ledgerx.com>

Update version information here only. There's no need to update any other file in the package.

Release of a hotfix/bugfix: - Increment version's `BUILD` field.

Release of an update that modifies the API but maintains backward compatibility: - Increment version's `MINOR` field and reset the `BUILD` field.

Release of an update that that modifies the API and breaks backward compatibility: - Increment version's `MAJOR` field and reset both the `MINOR` and `BUILD` fields.

CHAPTER 3

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

## l

# Index