DEXBot Documentation

Release 0.0.1

Fabian Schuh

Aug 02, 2018

Contents

CHAPTER 1

Basics

1.1 Setup

1.1.1 Requirements – Linux

To run in the background you need systemd and *lingering* enabled:

```
sudo loginctl enable-linger $USER
```

On some systems, such as the Raspberry Pi, you need to reboot for this to take effect.

You need to have python3 installed, including the `pip` tool, and the development tools for C extensions, and the OpenSSL libraries.

Plus for the easy configuration you need the `whiptail` command.

On Ubuntu/Debian/Raspian

Do:

```
sudo apt-get update
sudo apt-get install -y --install-recommends gcc libssl-dev python3-pip python3-dev␣
↪whiptail inetutils-ping
```

On some Ubuntu systems, it will complain about missing packages: you first need to make sure you have the `universe` repository:

```
sudo apt-get install -y software-properties-common
sudo add-apt-repository universe
```

NOTE: you *don't* need to upgrade the system: the issue here is about the *range* of packages available, not how new/old they are.

Fedora

This has been tested on Fedora 27:

```
sudo yum install -y gcc openssl-devel python3-pip python3-devel newt
```

Arch

As root, do:

```
pacman -S libnewt python-pip gcc
```

Other Distros

On other distros you need to check the documentation for how to install these packages, the names should be very similar.

1.1.2 Installation

:: sudo -H pip3 install https://github.com/Codaone/DEXBot/archive/master.zip

If you want the latest development version (which may not be tested at all), use git to download:

```
git clone git://github.com/ihaywood3/DEXBot/
cd DEXBot
sudo -H pip3 install -e .
```

Do not use the `--user` flag unless you understand its implications.

pip3 may complain about wanting to upgrade: this can be ignored.

1.1.3 Adding Keys

It is important to *install* the private key of your bot's account into a local wallet. This can be done using `uptick` which is installed as a dependency of `dexbot`:

```
uptick addkey
```

`uptick` will ask you for a passphrase to protect private keys stored in its wallet. This has no relation to any passphrase used in the web wallet.

You can get your private key from the BitShares Web Wallet: click the menu on the top right, then "Settings", "Accounts", "View keys", then tab "Owner Permissions", click on the public key, then "Show".

Look for the private key in Wallet Import Format (WIF), it's a "5" followed by a long list of letters. Select, copy and paste this into the screen where uptick asks for the key.

Check `uptick` successfully imported the key with:

```
uptick listaccounts
```

Yes, this process is a pain but for security reasons this part probably won't ever be "easy".

1.1.4 Configuration

`dexbot` can be configured using:

```
dexbot-cli configure
```

This will walk you through the configuration process. Read more about this in the *Configuration Questions*.

1.2 Configuration Questions

The configuration consists of a series of questions about the bots you wish to configure.

1. The Bot Name.

   Choose a unique name for your bot, DEXBot doesn't care what you call it. It is used to identify the bot in the logs so should be fairly short.

2. The Bot Strategy

   DEXBot provides a number of different bot strategies. They can be quite different in how they behave (i.e. spend *your* money) so it is important you understand the strategy before deploying a bot.

   (a) *Simple Echo Strategy* For testing this just logs events on a market, does no trading.

   (b) *Follow Orders Strategy* My (Ian Haywood) main bot, an extension of stakemachine's *wall*, it has been used to provide liquidity on AUD:BTS. Does function but by no mean perfect, see caveats in the docs.

3. Strategy-specific questions

   The questions that follow are determined by the strategy chosen, and each strategy will have its own questions around amounts to trade, spreads etc. See the strategy documentations linked above. But the first two strategy questions are nearly universal amongst the strategies so are documented here:

   (a) The Account.

   This is the same account name as the one where you entered the keys into `uptick` earlier on: the bot must always have the private key so it can execute trades.

   (b) The Market.

   This is the main market the bot trade on. They are specified by the quote asset, a colon (:), and the base asset, for example the market for BitShares priced in US dollars is called BTS:USD. BitShares always provides a "reverse" market so there will be a USD:BTS with the same trades, the only difference is the prices will be the inverse (1/x) of BTS:USD.

4. the Node.

   DEXBot needs to have a public node (also called "witness") that gives access to the BitShares blockchain.

   DEXBot uses `wss://status200.bitshares.apasia.tech/ws` as its default node If you run your own witness node then you can edit `config.yml` to change the node value.

5. Reporting

   DEXBot can send regular reports via e-mail of its activities. See *DEXBot E-mail Reports*

6. Systemd.

   If the configuration tool detects systemd (the process control system on most modern Linux systems) it will offer to install dexbot as a background service, this will run continuously in the background whenever you are logged in. if you enabled lingering as described, it will run whenever the computer is turned on.

7. The Passphrase

   If you select yes above, the final question will be the password you entered to protect the private key with `uptick`. Entering it here is a security risk: the configuration tool will save the password to a file on the computer. This means anyone with access to the computer can access your private key and spend the money in your account.

   There is no alternative to enable 24/7 bot trading without you being physically present to enter the password every time the bot wnats to execute a trade (which defeats the purpose of using a bot). It does mean you need to think carefully where dexbot is installed: my advice is on the computer in a secure location that you control behind a properly- configured firewall/router.

1.2.1 Manual Running

If you are not using systemd, the bot can be run manually by:

```
dexbot-cli run
```

It will ask for your wallet passphrase (that you provided when adding your private key using `uptick addkey`).

1.3 DEXBot E-mail Reports

DEXBot can send e-mail reports at regular intervals when its running in the background.

1.3.1 Configuration Questions

1. Report frequency.

   You get several options of number of days up to a week. If you want a different timeframe look in *config.yml* for the *days* option under *reporter*, can be any integer number of days. If you select "Never" then no reports are sent at all.

2. Send to address

   The address to send reports to, must be in the traditional username@server format.

3. Send from address

   The address the DEXBot e-mails will appear to be from. By default DEXBot uses the name of the user it's running as, and the name of the server it's running on. (Note this default may not work depending on your setup, a lot of e-mail servers will check the sending server name is valid from its point of view). use the same e-mail as "send to" above if you are unsure.

4. SMTP Server.

   The hostname of the e-mail server to use. Blank means the local server (so this has to be setup).

5. SMTP Port.

   Traditionally this is always "25". Some public e-mail setups (such as Gmail) require you to use the "submission" port (587): check the documentation of the e-mail service you are trying to use.

6. Login

   Use if the SMTP server requires a login name (most public ones do, but an ISP-provided or local network one may not), otherwise leave blank.

7. Password

   If you need to provide a login then a password is usually required too.

1.3.2 Reports

Reports have the subject "DEXBot Regular Report" and are HTML e-mails with a section for each bot, in each section the configuration values are quoted, then a graph is supplied by the bot.

Mst trading bots will provide a graph of the base and quote account balances and the total value (hopefully going up). These graph lines are all in the "quote" unit, using the price at the end of the reporting period (so hopefully factoring out shifts in capital value and you can actually see the effect of the bots trading).

Finally the log entries for each bot over the reporting period are supplied.

CHAPTER 2

Strategies

2.1 Simple Echo Strategy

2.1.1 API

2.1.2 Full Source Code

```
from dexbot.basestrategy import BaseStrategy


class Strategy(BaseStrategy):
    """
    Echo strategy
    Strategy that logs all events within the blockchain
    """

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        """ set call backs for events
        """
        self.onOrderMatched += self.print_orderMatched
        self.onOrderPlaced += self.print_orderPlaced
        self.onUpdateCallOrder += self.print_UpdateCallOrder
        self.onMarketUpdate += self.print_marketUpdate
        self.ontick += self.print_newBlock
        self.onAccount += self.print_accountUpdate
        self.error_ontick = self.error
        self.error_onMarketUpdate = self.error
        self.error_onAccount = self.error

    def error(self, *args, **kwargs):
        """ What to do on an error
```

(continues on next page)

(continued from previous page)

```
        """
        # Cancel all future execution
        self.disabled = True

    def print_orderMatched(self, i):
        """ Is called when an order in the market is matched

            A developer may want to filter those to identify
            own orders.

            :param bitshares.price.FilledOrder i: Filled order details
        """
        self.log.info("order matched: %s" % i)

    def print_orderPlaced(self, i):
        """ Is called when a new order in the market is placed

            A developer may want to filter those to identify
            own orders.

            :param bitshares.price.Order i: Order details
        """
        self.log.info("order placed:  %s" % i)

    def print_UpdateCallOrder(self, i):
        """ Is called when a call order for a market pegged asset is updated

            A developer may want to filter those to identify
            own orders.

            :param bitshares.price.CallOrder i: Call order details
        """
        self.log.info("call update:   %s" % i)

    def print_marketUpdate(self, i):
        """ Is called when Something happens in your market.

            This method is actually called by the backend and is
            dispatched to ``onOrderMatched``, ``onOrderPlaced`` and
            ``onUpdateCallOrder``.

            :param object i: Can be instance of ``FilledOrder``, ``Order``, or␣
↪``CallOrder``
        """
        self.log.info("marketupdate:  %s" % i)

    def print_newBlock(self, i):
        """ Is called when a block is received

            :param str i: The hash of the block

            .. note:: Unfortunately, it is currently not possible to
                      identify the block number for ``i`` alone. If you
                      need to know the most recent block number, you
                      need to use ``bitshares.blockchain.Blockchain``
        """
        self.log.info("new1 block:     %s" % i)
```

(continues on next page)

(continued from previous page)

```
        # raise ValueError("Testing disabling")

    def print_accountUpdate(self, i):
        """ This method is called when the worker's account name receives
            any update. This includes anything that changes
            ``2.6.xxxx``, e.g., any operation that affects your account.
        """
        self.log.info("account:       %s" % i)
```

2.2 Follow Orders Strategy

This strategy places a buy and a sell walls into a specific market. It buys below a base price, and sells above the base price.

It then reacts to orders filled (hence the name), when one order is completely filled, new walls are constructed using the filled order as the new base price.

2.2.1 Configuration Questions

Spread

Percentage difference between buy and sell price. So a spread of 5% means the bot sells at 2.5% above the base price and buys 2.5% below.

Wall Percent

This is the "wall height": the default amount to buy/sell, expressed as a percentage of the account balance. So for sells, its that percentage of your balance of the 'quote' asset, and for buys its that same percentage of your balance of the 'base' asset.

The advantage of this method (a change from early versions with 'fixed' wall heights) is it makes the bot 'self-correcting' if it sells to much of one asset, it will enter small orders for that asset, until the market gives it and opposing trade and it can rebalance itself.

Remember if you are using 'staggers' (see below) each order is the same: so you probably need to use quite a small percentage here.

Max

The bot will stop trading if the base price goes above this value, it will shut down until you manually restart it. (i.e. it won't restart itself if the price drops)

Remember prices are base/quote, so on AUD:BTS, that's the price of 1 AUD in BTS.

Min

Same in reverse: stop running if prices go below this value.

Start

The initial price, as a percentage of the spread between the highest bid and lowest ask. So "0" means the highest bid, "100" means the lowest ask, 50 means the midpoint between them, and so on.

Important: you need to look at your chosen market carefully and get a sense of its orderbook before setting this, justing setting "50" blind can lead to stupid prices especially in illiquid markets.

Reset

Normally the bot checks if it has previously placed orders in the market and uses those. If true, this option forces the bot to cancel any existing orders when it starts up, and re-calculate the starting price as above.

Staggers

Number of additional (or "staggered") orders to place. By default this is "1" (so no additional orders). 2 or more means multiple sell and buy orders at different prices.

Stagger Spread

The gap between each staggered order (as a percentage of the base price).

So say the spread is 5%, staggers is "2", and "stagger spread" is 4%, then there will be 2 buy orders: 2.5% and 6.5% (4% + 2.5%) below the base price, and two sells 2.5% and 6.5% above. The order amounts are all the same (see *wall percent* option).

2.2.2 Bot problems

Like all liquidity bots, the bot really works best with a "even" market, i.e. where are are counterparties both buying and selling.

With a severe "bear" market, the bot can be stuck being the sole participant that is still buying against a herd of panicked humans frantically selling. It repeatingly buys into the market and so it can run out of one asset quite quickly (although it will have bought it at lower and lower prices).

I don't have a simple good solution (and happy to hear suggestions from experienced traders)

CHAPTER 3

Developing own Strategies

3.1 Manual Configuration

The configuration of `dexbot` internally happens through a YAML formatted file. Unless you are developing or want to use a custom strategy, you don't need to edit this.

The default file name is `config.yml`, and `dexbot` only seeks the file in the current directory.

Otherwise you can specify a different config file using the `--configfile X` parameter when calling `dexbot run`.

3.1.1 The config.yml file

```
# The BitShares endpoint to talk to
node: "wss://node.testnet.bitshares.eu"

# List of bots
bots:

    # Name of the bot. This is mostly for logging and internal
    # use to distinguish different bots
    NAME_OF_BOT:

        # Python module to look for the strategy (can be custom)
        # dexbot will search in ~/bots as well as standard dirs
        module: "dexbot.strategies.echo"

        # The bot class in that module to use
        bot: Echo

        # The market to subscribe to
        market: GOLD:TEST
```

(continues on next page)

(continued from previous page)

```
        # The account to use for this bot
        account: xeroc

        # Custom bot configuration
        foo: bar
```

3.1.2 Using the configuration in custom strategies

The bot's configuration is available to in each strategy as dictionary in `self.bot`. The whole configuration is avaialable in `self.config`. The name of your bot can be found in `self.name`.

3.2 Base Strategy

All strategies should inherit `dexbot.basestrategy.BaseStrategy` which simplifies and unifies the development of new strategies.

3.2.1 API

3.3 Storage

This class allows to permanently store bot-specific data in a sqlite database (`dexbot.sqlite`) using:

```
self["key"] = "value"
```

Note: Here, `self` refers to the instance of your bot's strategy when coding yaour own strategy.

The value is persistently stored and can be access later on using:

```
print(self["key"]).
```

Note: This applies a `json.loads(json.dumps(value))`!

3.3.1 SQLite database

The user's data is stored in its OS protected user directory:

OSX:

- *~/Library/Application Support/<AppName>*

Windows:

- *C:Documents and Settings<User>Application DataLocal Settings<AppAuthor><AppName>*
- *C:Documents and Settings<User>Application Data<AppAuthor><AppName>*

Linux:

- *~/.local/share/<AppName>*

Where <AppName> is dexbot and <AppAuthor> is ChainSquad GmbH.

3.3.2 Simple example

```
from dexbot.basestrategy import BaseStrategy


class Strategy(BaseStrategy):
    """
    Storage demo strategy
    Strategy that prints all new blocks in the blockchain
    """

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.ontick += self.tick

    def tick(self, i):
        print("previous block: %s" % self["block"])
        print("new block: %s" % i)
        self["block"] = i
```

Example Output:

```
Current Wallet Passphrase:
previous block: None
new block: 008c4c2424e6394ad4bf5a9756ae2ee883b0e049
previous block: 008c4c2424e6394ad4bf5a9756ae2ee883b0e049
new block: 008c4c257a76671144fdba251e4ebbe61e4593a4
previous block: 008c4c257a76671144fdba251e4ebbe61e4593a4
new block: 008c4c2617851b31d0b872e32fbff6f8248663a3
```

3.4 Statemachine

The base strategy comes with a state machine that can be used by your strategy.

Similar to *Storage*, the methods of this class can be used in your strategy directly, e.g., via `self.get_state()`, since the class is inherited by *Base Strategy*.

3.4.1 API

class dexbot.statemachine.**StateMachine**(**args*, ***kwargs*)

Generic state machine

add_state(*state*)

Add a new state to the state machine

Parameters **state** (*str*) – Name of the state

get_state()

Return state of state machine

set_state(*state*)

Change state of the state machine

Parameters **state** (*str*) – Name of the new state

3.5 Events

The websocket endpoint of BitShares has notifications that are subscribed to and dispatched by `dexbot`. This uses python's native `Events`. The following events are available in your strategies and depend on the configuration of your bot/strategy:

- `onOrderMatched`: Called when orders in your market are matched
- `onOrderPlaced`: Called when a new order in your market is placed
- `onUpdateCallOrder`: Called if one of the assets in your market is a market-pegged asset and someone updates his call position
- `onMarketUpdate`: Called whenever something happens in your market (includes matched orders, placed orders and call order updates!)
- `ontick`: Called when a new block is received
- `onAccount`: Called when your account's statistics is updated (changes to `2.6.xxxx` with `xxxx` being your account id number)
- `error_ontick`: Is called when an error happend when processing `ontick`
- `error_onMarketUpdate`: Is called when an error happend when processing `onMarketUpdate`
- `error_onAccount`: Is called when an error happend when processing `onAccount`

3.5.1 Simple Example

```
class Simple(BaseStrategy):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        """ set call backs for events
        """
        self.onOrderMatched += print
        self.onOrderPlaced += print
        self.onUpdateCallOrder += print
        self.onMarketUpdate += print
        self.ontick += print
        self.onAccount += print
```

3.6 Wall Strategy

This strategy simply places a buy and a sell wall into a specific market using a specified account.

3.6.1 Example Configuration

```
# BitShares end point
node: "wss://node.bitshares.eu"

# List of Bots
bots:

        # Only a single Walls Bot
        Walls:

             # The Walls strategy module and class
             module: dexbot.strategies.walls
             bot: Walls

             # The market to serve
             market: HERO:BTS

             # The account to sue
             account: hero-market-maker

             # We shall bundle operations into a single transaction
             bundle: True

             # Test your conditions every x blocks
             test:
                     blocks: 10

             # Where the walls should be
             target:

                     # They relate to the price feed
                     reference: feed

                     # There should be an offset
                     offsets:
                         buy: 2.5
                         sell: 2.5

                     # We'd like to use x amount of quote (here: HERO)
                     # in the walls
                     amount:
                         buy: 5.0
                         sell: 5.0

             # When the price moves by more than 2%, update the walls
             threshold: 2
```

3.6.2 Source Code

```
from math import fabs
from collections import Counter
from bitshares.amount import Amount
from dexbot.basestrategy import BaseStrategy, ConfigElement
from dexbot.errors import InsufficientFundsError


```

(continues on next page)

(continued from previous page)

```
class Strategy(BaseStrategy):
    """
    Walls strategy
    """

    @classmethod
    def configure(cls):

        return BaseStrategy.configure() + [
            ConfigElement(
                "spread",
                "int",
                5,
                "the spread between sell and buy as percentage",
                (0,
                 100)),
            ConfigElement(
                "threshold",
                "int",
                5,
                "percentage the feed has to move before we change orders",
                (0,
                 100)),
            ConfigElement(
                "buy", "float", 0.0, "the default amount to buy", (0.0, None)),
            ConfigElement("sell", "float", 0.0,
                          "the default amount to sell", (0.0, None)),
            ConfigElement(
                "blocks",
                "int",
                20,
                "number of blocks to wait before re-calculating",
                (0,
                 10000)),
            ConfigElement(
                "dry_run",
                "bool",
                False,
                "Dry Run Mode\nIf Yes the bot won't buy or sell anything, just log␣
↪what it would do.\nIf No, the bot will buy and sell for real.",
                None)
        ]

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        # Define Callbacks
        self.onMarketUpdate += self.test
        self.ontick += self.tick
        self.onAccount += self.test

        self.error_ontick = self.error
        self.error_onMarketUpdate = self.error
        self.error_onAccount = self.error

        # Counter for blocks
        self.counter = Counter()
```

(continues on next page)

(continued from previous page)

```
64
65          # Tests for actions
66          self.test_blocks = self.worker.get("test", {}).get("blocks", 0)
67
68      def error(self, *args, **kwargs):
69          self.disabled = True
70          self.cancelall()
71          self.log.info(self.execute())
72
73      def updateorders(self):
74          """ Update the orders
75          """
76          self.log.info("Replacing orders")
77
78          # Canceling orders
79          self.cancelall()
80
81          # Target
82          target = self.worker.get("target", {})
83          price = self.getprice()
84
85          # prices
86          buy_price = price * (1 - target["offsets"]["buy"] / 100)
87          sell_price = price * (1 + target["offsets"]["sell"] / 100)
88
89          # Store price in storage for later use
90          self["feed_price"] = float(price)
91
92          # Buy Side
93          if float(self.balance(self.market["base"])
94                   ) < buy_price * target["amount"]["buy"]:
95              InsufficientFundsError(
96                  Amount(
97                      target["amount"]["buy"] *
98                      float(buy_price),
99                      self.market["base"]))
100             self["insufficient_buy"] = True
101         else:
102             self["insufficient_buy"] = False
103             self.market.buy(
104                 buy_price,
105                 Amount(target["amount"]["buy"], self.market["quote"]),
106                 account=self.account
107             )
108
109         # Sell Side
110         if float(self.balance(self.market["quote"])
111                  ) < target["amount"]["sell"]:
112             InsufficientFundsError(
113                 Amount(
114                     target["amount"]["sell"],
115                     self.market["quote"]))
116             self["insufficient_sell"] = True
117         else:
118             self["insufficient_sell"] = False
119             self.market.sell(
120                 sell_price,
```

(continues on next page)

(continued from previous page)

```
                Amount(target["amount"]["sell"], self.market["quote"]),
                account=self.account
            )

        self.log.info(self.execute())

    def getprice(self):
        """ Here we obtain the price for the quote and make sure it has
            a feed price
        """
        target = self.worker.get("target", {})
        if target.get("reference") == "feed":
            assert self.market == self.market.core_quote_market(
            ), "Wrong market for 'feed' reference!"
            ticker = self.market.ticker()
            price = ticker.get("quoteSettlement_price")
            assert abs(price["price"]) != float(
                "inf"), "Check price feed of asset! (%s)" % str(price)
        return price

    def tick(self, d):
        """ ticks come in on every block
        """
        if self.test_blocks:
            if not (self.counter["blocks"] or 0) % self.test_blocks:
                self.test()
            self.counter["blocks"] += 1

    def test(self, *args, **kwargs):
        """ Tests if the orders need updating
        """
        orders = self.orders

        # Test if still 2 orders in the market (the walls)
        if len(orders) < 2 and len(orders) > 0:
            if (
                not self["insufficient_buy"] and
                not self["insufficient_sell"]
            ):
                self.log.info("No 2 orders available. Updating orders!")
                self.updateorders()
        elif len(orders) == 0:
            self.updateorders()

        # Test if price feed has moved more than the threshold
        if (
            self["feed_price"] and
            fabs(1 - float(self.getprice()) / self["feed_price"]) > self.worker[
↪"threshold"] / 100.0
        ):
            self.log.info(
                "Price feed moved by more than the threshold. Updating orders!")
            self.updateorders()
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Index