StakeMachine Documentation

Release 0.0.1

Fabian Schuh

Jun 03, 2018

Contents

CHAPTER 1

Basics

1.1 Setup

1.1.1 Installation

```
pip3 install stakemachine     [--user]
```

If you install using the `--user` flag, the binaries of `stakemachine` and `uptick` are located in `~/.local/bin`. Otherwise they should be globally reachable.

1.1.2 Adding Keys

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

```
uptick addkey
```

1.1.3 Configuration

You will need to create configuration file in YAML format. The default file name is `config.yml`, otherwise you can specify a different config file using the `--configufile X` parameter of `stakemachine`.

Read more about the *Configuration*.

1.1.4 Running

The bot can be run by:

```
stakemachine run
```

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

If you want to prevent the password dialog, you can predefine an environmental variable `UNLOCK`, if you understand the security implications.

1.2 Configuration

The configuration of `stakemachine` happens through a YAML formated file and takes the following form:

```
# 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)
        module: "stakemachine.strategies.echo"

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

        # The market to subscribe to
        market: GOLD:TEST

        # The account to use for this bot
        account: xeroc

        # Custom bot configuration
        foo: bar
```

1.2.1 Usig 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`.

CHAPTER 2

Strategies

2.1 Wall Strategy

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

2.1.1 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: stakemachine.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
```

(continues on next page)

(continued from previous page)

```
            # 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
```

2.1.2 Source Code

```
from math import fabs
from pprint import pprint
from collections import Counter
from bitshares.amount import Amount
from stakemachine.basestrategy import BaseStrategy
from stakemachine.errors import InsufficientFundsError
import logging
log = logging.getLogger(__name__)


class Walls(BaseStrategy):
    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()

        # Tests for actions
        self.test_blocks = self.bot.get("test", {}).get("blocks", 0)

    def error(self, *args, **kwargs):
        self.disabled = True
        self.cancelall()
```

(continues on next page)

(continued from previous page)

```
        pprint(self.execute())

    def updateorders(self):
        """ Update the orders
        """
        log.info("Replacing orders")

        # Canceling orders
        self.cancelall()

        # Target
        target = self.bot.get("target", {})
        price = self.getprice()

        # prices
        buy_price = price * (1 - target["offsets"]["buy"] / 100)
        sell_price = price * (1 + target["offsets"]["sell"] / 100)

        # Store price in storage for later use
        self["feed_price"] = float(price)

        # Buy Side
        if float(self.balance(self.market["base"])) < buy_price * target["amount"]["buy"]:
            InsufficientFundsError(Amount(target["amount"]["buy"] * float(buy_price), self.market["base"]))
            self["insufficient_buy"] = True
        else:
            self["insufficient_buy"] = False
            self.market.buy(
                buy_price,
                Amount(target["amount"]["buy"], self.market["quote"]),
                account=self.account
            )

        # Sell Side
        if float(self.balance(self.market["quote"])) < target["amount"]["sell"]:
            InsufficientFundsError(Amount(target["amount"]["sell"], self.market["quote"]))
            self["insufficient_sell"] = True
        else:
            self["insufficient_sell"] = False
            self.market.sell(
                sell_price,
                Amount(target["amount"]["sell"], self.market["quote"]),
                account=self.account
            )

        pprint(self.execute())

    def getprice(self):
        """ Here we obtain the price for the quote and make sure it has
            a feed price
        """
        target = self.bot.get("target", {})
        if target.get("reference") == "feed":
            assert self.market == self.market.core_quote_market(), "Wrong market for 'feed' reference!"
```

(continues on next page)

(continued from previous page)

```
            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"]
            ):
                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.bot[
↪"threshold"] / 100.0
        ):
            log.info("Price feed moved by more than the threshold. Updating orders!")
            self.updateorders()
```

CHAPTER 3

Developing own Strategies

3.1 Base Strategy

All strategies should inherit *stakemachine.basestrategy.BaseStrategy* which simplifies and unifies the development of new strategies.

3.1.1 API

class stakemachine.basestrategy.**BaseStrategy**(*config*, *name*, *onAccount=None*, *onOrderMatched=None*, *onOrderPlaced=None*, *onMarketUpdate=None*, *onUpdateCallOrder=None*, *ontick=None*, *bitshares_instance=None*, **args*, ***kwargs*)

Base Strategy and methods available in all Sub Classes that inherit this BaseStrategy.

BaseStrategy inherits:

- `stakemachine.storage.Storage`
- *stakemachine.statemachine.StateMachine*
- `Events`

Available attributes:

- `basestrategy.bitshares`: instance of ´‘bitshares.BitShares()‘‘
- `basestrategy.add_state`: Add a specific state
- `basestrategy.set_state`: Set finite state machine
- `basestrategy.get_state`: Change state of state machine
- `basestrategy.account`: The Account object of this bot
- `basestrategy.market`: The market used by this bot
- `basestrategy.orders`: List of open orders of the bot's account in the bot's market

- `basestrategy.balance`: List of assets and amounts available in the bot's account

Also, Base Strategy inherits `stakemachine.storage.Storage` which allows to permanently store data in a sqlite database using:

```
basestrategy["key"] = "value"
```

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

account
Return the full account as `bitshares.account.Account` object!

Can be refreshed by using `x.refresh()`

balance (*asset*)
Return the balance of your bot's account for a specific asset

balances
Return the balances of your bot's account

cancelall()
Cancel all orders of this bot

execute()
Execute a bundle of operations

market
Return the market object as `bitshares.market.Market`

orders
Return the bot's open accounts in the current market

3.2 Storage

This class allows to permanently store bot-specific data in a sqlite database (`stakemachine.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))`!

3.2.1 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 `stakemachine` and `<AppAuthor>` is `ChainSquad GmbH`.

3.2.2 Simple example

```
from stakemachine.basestrategy import BaseStrategy


class StorageDemo(BaseStrategy):
    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
```

3.3 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*.

3.3.1 API

class stakemachine.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

3.4 Events

The websocket endpoint of BitShares has notifications that are subscribed to and dispatched by `stakemachine`. 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`

3.4.1 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
```

3.5 Simple Echo Strategy

3.5.1 API

class stakemachine.strategies.echo.**Echo**(**args*, ***kwargs*)

error(**args*, ***kwargs*)
What to do on an error

print_UpdateCallOrder(*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.

Parameters **i** (*bitshares.price.CallOrder*) – Call order details

print_accountUpdate(*i*)
This method is called when the bot's account name receives any update. This includes anything that changes `2.6.xxxx`, e.g., any operation that affects your account.

print_marketUpdate(*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`.

Parameters **i** (*object*) – Can be instance of `FilledOrder`, `Order`, or `CallOrder`

print_newBlock(*i*)
Is called when a block is received

Parameters **i** (*str*) – 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`

---

print_orderMatched(*i*)
Is called when an order in the market is matched

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

Parameters **i** (*bitshares.price.FilledOrder*) – Filled order details

print_orderPlaced(*i*)
Is called when a new order in the market is placed

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

Parameters **i** (*bitshares.price.Order*) – Order details

3.5.2 Full Source Code

```
from stakemachine.basestrategy import BaseStrategy
import logging
log = logging.getLogger(__name__)


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

        """ set call backs for events
        """
        self.onOrderMatched += self.print_orderMatched
        self.onOrderPlaced += self.print_orderPlaced
```

(continues on next page)

(continued from previous page)

```
        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
        """
        # 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
        """
        print("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
        """
        print("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
        """
        print("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``
        """
        print("marketupdate:  %s" % i)

    def print_newBlock(self, i):
```

(continues on next page)

(continued from previous page)

```
        """ 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``
        """
        print("new block:     %s" % i)
        # raise ValueError("Testing disabling")

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

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Index