SimpleCoin Documentation

Release 0.8.1-dev

Isaac Cook

February 28, 2017

Contents

SimpleCoin is an open source mining pool frontend, it performs many of the same functions as software like MPOS. It is currently under active development for use in the SimpleMulti mining pool, although we are gradually adding documentation so it can be set up more easily by others as well.

CHAPTER 1

Features

- Multi-currency support. One instance of this pool software supports mining and paying out many currencies.
- Multi-algo support. Currently configured for X11, Scrypt, and Scrypt-N, but designed so others could be added.
- Merge mining enabled. We support merge mining as many currencies as you'd like.
- Multi-chain support. Some miners can mine with PPLNS, while others use PROP payout, but they work together to solve blocks faster for each "chain".
- Translation support. Thanks to sbwdlihao's contributions we support internationalization.

CHAPTER 2

Table of Contents

Getting Setup

SimpleCoinMulti uses PostgreSQL as its primary database, although SCM is configurable and allows using pretty much any database supported via SQLAlchemy. Setup is tested running on Ubuntu 12.04. If you're doing development you'll also want to install Supervisor for convenience.

Installation

To install thse packages on Ubuntu 12.04:

```
apt-get install redis-server postgresql-contrib-9.3 postgresql-9.3 postgresql-server-dev-9.3
# to install supervisor as well
apt-get install supervisor
```

Now you'll want to setup a Python virtual enviroment to run the application in. This isn't stricly necessary, but not using virtualenv can cause all kinds of headache, so it's *highly* recommended. You'll want to setup virtualenvwrapper to make this easier.

```
# make a new virtual environment for simplecoin multi
mkvirtualenv scm
# clone the source code repo
git clone git@github.com:simplecrypto/simplecoin_multi.git
cd simplecoin_multi
pip install -e .
# install all python dependencies
pip install -r requirements.txt
pip install -r dev-requirements.txt
```

SimpleCoin uses environmental variables to know where to look for its configuration files, so we need to set those before we attempt to start the server or run any management commands.

```
# If we'll just be using the dev server in the same directory, this is fine
export SIMPLECOIN_CONFIG=example.toml
# For production, best to use full paths
export SIMPLECOIN_CONFIG=/home/limpit/simplecoin/example.toml

# You can also define multiple configs that will be applied in numberical order
export SIMPLECOIN_CONFIG_1=/home/limpit/simplecoin/base.toml
export SIMPLECOIN_CONFIG_2=/home/limpit/simplecoin/specific.toml
# This is useful for staging servers, etc
```

Initialize an empty database & add tables

```
# creates a new user with password 'testing', creates the database
./util/reset_db.sh
# creates the database schema for simpledoge
python manage.py init_db
```

Now everything should be ready for running the server. This project uses supervisor in development to watch for file changes and reload the server.

```
supervisord -c supervisor.conf
```

This should successfully start both the development server and the task scheduler if all is well. If not, carefully reading the output from supervisor should give good hints on what's not working right.

Mining

If you want to start sending shares and solved blocks to Redis for this dev instance to process, head over to powerpool and read it's setup instructions. You'll likely want to run a local testnet in a box, or on the live testnet. We recommend the litecoin testnet for testing.

Payouts & Manual Exchanging

The RPC client works with SimpleCoin Multi's RPC views. This can be run on a secure server to pull payout and trade data. This client is what actually makes the payouts, and the `simplecoin_rpc_client` allows manually managing exchanging.

simplecoin_rpc_client

Unfortunately docs for how to use this (especially in a production setting) are very lacking at the moment.

Autoexchanging

We currently offer no code to perform automatic exchanging, although you could expand the RPC client to do it, or write your own app to handle it. A first class autoexchanging service may be offered by us at some point in the future.

Production Installation

Currently we have very limited documentation on how to properly get setup for production. SimpleCoin Multi is still a pretty green package. By reading through the "Getting Setup" section you should be able to get an idea of how you might set it up in production, but we take no responsibilities for problems that arise. If you have questions you're welcome to ask in our IRC #simplecrypto but we won't always have time to help.

General Tips

- It is a good idea to use a process control package like Supervisor or Upstart to run & manage gunicorn, nginx, the scheduler, and any other mission critical processes (ie, PowerPool, coin daemons, etc). This allows easier log management, user control, automatically restarts, etc.
- Optimal scheduled task intervals may be different than the default, so reading through the different tasks and understanding what they do and tweaking their schedules accordingly is a good idea.

- Gunicorn is the webserver that runs SimpleCoin, but it is lacking in many desirable features like caching and rate limiting. Because of this it's highly recommended that you put a HTTP reverse proxy server in front of it. NGINX is what we run in production, but Apache and others could be used as well.

Configuration

The configuration file has a lot of options, and at the moment they're not documented the best. It's recommended that you read through the `example.toml` and `defaults.toml` to get handle on it. We hope to have sections explaining sharechains and mining server configuration in detail soon, but at the moment these topics are a bit deep and will likely be confusing without reading quite a bit of code.

Scheduler

The scheduler handles things like pulling PowerPool's share data out of redis, generating various cached values, creating payouts & trade requests, and many other vital tasks. To get it running you'll first need to set an environment variable with the path to the config file. It should look something like this:

```
export SIMPLECOIN_CONFIG=/home/$USER/simplecoin_multi/config.toml
python simplecoin/scheduler.py
```

A simple upstart script would look something like this:

```
start on (filesystem)
stop on runlevel [016]
respawn
console log
setuid multi
setgid multi
env SIMPLECOIN_CONFIG=/home/multi/web/config.toml

exec /home/multi/web_venv/bin/simplecoin_scheduler
```

Webserver

The webserver handler user requests. An example upstart configuration:

```
start on (filesystem)
stop on runlevel [016]

respawn
console log
setuid multi
setgid multi
chdir /home/multi/web
env SIMPLECOIN_CONFIG=/home/multi/web/config.toml

exec /home/multi/web_venv/bin/gunicorn simplecoin.wsgi_entry:app -b 127.0.0.1:8080 --timeout 270
```

Adding A Currency

Currency Configuration

SimpleCoin has configuration sections that specify certain coin specific parameters. Several common coins are already defined in `defaults.toml`. If your currency is not defined in the defaults file then you will have to add it. It's best practice to add this information to your configuration, and not `defaults.toml`.

- *name* - the currency name as displayed throughout the site. This is purely cosmetic.
- *algo* - the name of a configured algo. These are also defined in the defaults.toml. If you're using a new algorithm then you'll have to add an algorithm entry to the configs as well. Contact us or the coin creator to determine these details.
- *address_version* - This is a number that is a valid prefix for an address on this coin network. See below for figuring this out.
- *merged* - is this a merge mined coin?
- *block_mature_confirms* - the number of blocks required to have passed before you can spend a coinbase transaction. Usually defined by COINBASE_MATURITY in the main.h of the core client.
- *block_time* - the target block time in seconds.
- *block_explore* - a url prefix to which a block hash can be looked up
- *tx_explore* - url prefix to which a transaction hash can be looked up

After you've added this information you'll now also need to define coinserver information and exchangeability. If you're not exchanging coins than nothing is buyable or sellable except the "pool payout" currency. This can be seen in the `example.toml` and is quite straightforward.

Finding valid address versions can be done using cryptokit on the python command line. Assuming you've insalled cryptokit with pip, then the following will print the address version. Keep in mind that most currencies support a few different address versions, so consulting the authors or looking at source code would be good to ensure you've adding them all. Notice that the parameter is a list, allowing for multiple valid versions.

```
>>> from cryptokit import base58
>>> base58.get_bcaddress_version("zkFuhTiU8f5gRFdwDqwWN4QsU4MwE8134J") # enter your example address h
143
```

Chain Configuration

Chains are a little complicated to explain succinctly. They are essentially a way by which profit switching mining servers and dedicated currency mining servers could work together to mine blocks, even with different payout methods defined on each (PPLNS or PROP, etc). This is quite powerful, but also a bit complex.

If you're simply doing dedicated currency mining servers with no profit switching or merged mining of any kind then each currency will be on its own chain. So to setup your new currency you must add a new chain to the configuration file. Make sure the chain number is unique, and never overlap them!

- *title* - a description to display on the web interface
- *currencies* - a list of currencies that can be mined on the chain. So just the currency code you defined above.
- *algo* - the hashing algorithm of all currencies on this chain
- *type* - the method used to determine payouts for shares on the chain. Usually "pplns" is best as "prop" is poorly tested.

- *last_n* - pplns configuration
- *fee_perc* - a percentage to extract from earnings on this chain. given as a string such as "0.01" for 1%

Mining Server

This block should tie to a mining port on a powerpool instance. Keep in mind that a single powerpool instance may contain multiple stratum ports, and each stratum port should have it's own "mining_servers" configuration block in the configuration.

- *address* - A stratum address that users can point at, minus the port
- *monitor_address* - a url for the internal JSON monitor of this stratum port. Don't confuse this with powerpool's monitor URL. Each component within powerpool has it's own sub-address, and each mining port in powerpool is a component, so this should look something like `http://localhost:[monitor port]/[name of stratum component]`.
- *port* - the stratum addresses port
- *location* - the location configuration information. This needs to correspond to a location configuration block. Sorry, this seemed cool when we added it....
- *diff* - a text representation of the difficulty for this port. If it's vardiff, represent as a range.
- *chain* - the most important bit, which mining chain will this be on

API

Utils

class simplecoin.utils.**Benchmark**(*name*)

class simplecoin.utils.**ShareTracker**(*algo*)

accepted

algo

count_slice(*slc*)

dup_efficiency

efficiency

hashrate(*typ='acc'*)

low_efficiency

rejected

stale_efficiency

total

class simplecoin.utils.**ShareTypeTracker**(*share_type*)

simplecoin.utils.**anon_users**(**args*, ***kwargs*)

simplecoin.utils.**collect_pool_stats**()
Collects the necessary data to render the /pool_stats view or the API

simplecoin.utils.**collect_user_stats**(*user_address*)
Accumulates all aggregate user data for serving via API or rendering into main user stats page

simplecoin.utils.**get_alerts**(**args*, ***kwargs*)

simplecoin.utils.**get_past_chain_profit**()

simplecoin.utils.**get_pool_hashrate**(**args*, ***kwargs*)
Retrieves the pools hashrate average for the last 10 minutes.

simplecoin.utils.**last_block_time**(*algo*, *merged=False*)
Retrieves the last time a block was solved using progressively less accurate methods. Essentially used to calculate round time. TODO XXX: Add pool selector to each of the share queries to grab only x11, etc

simplecoin.utils.**orphan_percentage**(**args*, ***kwargs*)

simplecoin.utils.**pool_share_tracker**(**args*, ***kwargs*)
Get accepted and rejected share count totals for the last month

simplecoin.utils.**resort_recent_visit**(*recent*)
Accepts a new dictionary of recent visitors and calculates what percentage of your total visits have gone to that address. Used to dim low percentage addresses. Also sortes showing most visited on top.

simplecoin.utils.**time_format**(*seconds*)

simplecoin.utils.**validate_message_vals**(*address*, ***kwargs*)

simplecoin.utils.**validate_str_perc**(*perc*, *round=Decimal('0.01')*)
Tries to convert a var representing an 0-100 scale percentage into a mathematically useful Python Decimal. Default is rounding to 0.01%

Then checks to ensure decimal is within valid bounds

simplecoin.utils.**verify_message**(*address*, *curr*, *message*, *signature*)

Models

class simplecoin.models.**Block**(***kwargs*)
This class stores metadata on all blocks found by the pool

algo

algo_obj

average_hashrate

chain_distrib()

chain_profitability()
Creates a dictionary that is keyed by chainid to represent the BTC earned per number of shares for every share chain that helped solve this block

confirms_remaining

contributed
Total fees + donations associated with this block

currency

currency_obj

difficulty

duration

explorer_link

found_at

hash

hashes_to_solve

height

id

luck

mature

merged

orphan

shares_to_solve
Total shares that were required to solve the block

standard_join = ['status', 'merged', 'currency', 'worker', 'explorer_link', 'luck', 'total_value', 'difficulty', 'duration',

status

time_started

timestamp

total_value

transaction_fees

user

worker

class simplecoin.models.**ChainPayout**(***kwargs*)

amount

block

block_id

chain_shares

chainid

config_obj

distribute()

donations

fees

hashes

make_credit_obj(*user*, *address*, *currency*, *shares*)
Makes the appropriate credit object given a few details. Payout amount too be calculated.

mhashes

payout_shares

solve_slice

class simplecoin.models.**Credit**(***kwargs*)

A credit for currency directly crediting a users balance. These have no intermediary exchanges.

address

amount

block

block_id

currency

currency_obj

cut_perc

fee_perc

height

hr_fee_perc

hr_pd_perc

id

classmethod make_credit(*currency*, *block*, ***kwargs*)

mined

payable

payable_amount

payout

payout_id

pd_perc

perc_applied

sharechain_id

sharechain_title

source

standard_join = ['status', 'created_at', 'explorer_link', 'text_perc_applied', 'mined', 'height', 'transaction_id']

status

text_perc_applied

type

user

class simplecoin.models.**CreditExchange**(***kwargs*)

A credit that needs a sale and a buy to get to the correct currency

address

amount

block

block_id

buy_amount

buy_req

buy_req_id

currency

fee_perc

final_amount

id

payable

payable_amount

payout

payout_id

pd_perc

sell_amount

sell_req

sell_req_id

sharechain_id

source

status

type

user

class simplecoin.models.**DeviceSlice**(***kwargs*)

An data sample that pertains to a single workers device. Currently used to temperature and hashrate.

classmethod combine(**lst*)

Takes an iterable and combines the values. Usually either returns an average or a sum. Can assume at least one item in list

device

from_db = {0: 'hashrate', 1: 'temperature'}

get_stat(*stat*)

key

alias of Key

keys = ['user', 'worker', 'device', 'stat_val']

set_stat(*stat*)

span

span_config = [{'window': datetime.timedelta(0, 3600), 'slice': datetime.timedelta(0, 60)}, {'window': datetime.timede

stat

stat_val

time

to_db = {'temperature': 1, 'hashrate': 0}

user

value

worker

class simplecoin.models.**Payout**(***kwargs*)

address

amount

count

created_at

currency

currency_obj

id

payout_currency

status

timestamp

transaction

transaction_id

user

class simplecoin.models.**PayoutAddress**(***kwargs*)

address

classmethod create(*address*, *currency*)

currency

exchangeable

user

class simplecoin.models.**ShareSlice**(***kwargs*)

SHARE_TYPES = ['acc', 'low', 'dup', 'stale']

algo

classmethod combine(**lst*)
Takes a query list and combines the values. Usually either returns an average or a sum. Can assume at least one item in ql

key
alias of Key

keys = ['user', 'worker', 'algo', 'share_type']

share_type

span

span_config = [{'window': datetime.timedelta(0, 3600), 'slice': datetime.timedelta(0, 60)}, {'window': datetime.timede

time

user

value

worker

class simplecoin.models.**TimeSlice**
An time abstracted data sample that pertains to a single worker. Currently used to represent accepted and rejected shares.

classmethod add_value(*user*, *value*, *time*, *worker*)

classmethod compress(*span*, *delete=True*)

classmethod create(*user*, *worker*, *algo*, *value*, *time*)

end_time

classmethod floor_time(*time*, *span*, *stamp=False*)

classmethod get_span(*lower=None*, *upper=None*, *stamp=False*, *ret_query=False*, *slice_size=None*, ***kwargs*)
A utility to grab a group of slices and automatically compress smaller slices into larger slices

address, worker, and algo are just filters. They may be a single string or list of strings.

upper and lower are datetimes.

item_key

class simplecoin.models.**TradeRequest**(***kwargs*)
Used to provide info necessary to external applications for trading currencies

Created rows will be checked + updated externally

classmethod create(*currency*, *quantity*)

created_at

credits

currency

distribute()

exchanged_quantity

fees

id

quantity

status

type

class simplecoin.models.**Transaction**(***kwargs*)

confirmed

created_at

currency

currency_obj

id

network_fee

standard_join = ['txid', 'confirmed', 'created_at', 'currency', '__dont_mongo']

status

timestamp

txid

url_for

class simplecoin.models.**UserSettings**(***kwargs*)

addresses

anon

apply(*shares*, *user_currency*, *block_currency*, *valid_currencies*)
Given a share amount, a currency we're paying out, and the valid exchangeable currencies we return a new distribution of shares among some number of addresses.

classmethod create(*user*, *pdonate_perc*, *spayout_perc*, *spayout_addr*, *spayout_curr*, *del_spayout_addr*, *anon*, *set_addrs*)

exchangeable_addresses

hr_pdonation_perc

hr_perc

hr_spayout_perc

pdonation_perc

spayout_addr

spayout_curr

spayout_perc

unexchangeable_addresses

classmethod update(*address*, *set_addrs*, *del_addrs*, *pdonate_perc*, *spayout_perc*, *spayout_addr*, *spayout_curr*, *del_spayout_addr*, *anon*)

user

simplecoin.models.**average_combine**
classmethod(function) -> method

Convert a function to be a class method.

A class method receives the class as implicit first argument, just like an instance method receives the instance. To declare a class method, use this idiom:

class C: def f(cls, arg1, arg2, ...): ... f = classmethod(f)

It can be called either on the class (e.g. C.f()) or on an instance (e.g. C().f()). The instance is ignored except for its class. If a class method is called for a derived class, the derived class object is passed as the implied first argument.

Class methods are different than C++ or Java static methods. If you want those, see the staticmethod builtin.

simplecoin.models.**make_upper_lower**(*trim=None*, *span=None*, *offset=None*, *clip=None*, *fmt='dt'*)
Generates upper and lower bounded datetime objects.

simplecoin.models.**sum_combine**
classmethod(function) -> method

Convert a function to be a class method.

A class method receives the class as implicit first argument, just like an instance method receives the instance. To declare a class method, use this idiom:

class C: def f(cls, arg1, arg2, ...): ... f = classmethod(f)

It can be called either on the class (e.g. C.f()) or on an instance (e.g. C().f()). The instance is ignored except for its class. If a class method is called for a derived class, the derived class object is passed as the implied first argument.

Class methods are different than C++ or Java static methods. If you want those, see the staticmethod builtin.

Scheduled Tasks

simplecoin.scheduler.**cache_profitability**()
Calculates the profitability from recent blocks

simplecoin.scheduler.**cache_user_donation**()
Grab all user donations and loop through them then cache donation %

simplecoin.scheduler.**chain_cleanup**(*chain*, *dont_simulate*)
Handles removing all redis share slices that we are fairly certain won't be needed to credit a block if one were to be solved in the future.

simplecoin.scheduler.**collect_minutes**()
Grabs all the pending minute shares out of redis and puts them in the database

simplecoin.scheduler.**collect_ppagent_data**()
Grabs all the pending ppagent data points

simplecoin.scheduler.**compress_five_minute**()

simplecoin.scheduler.**compress_minute**()

simplecoin.scheduler.**compress_slices**()

simplecoin.scheduler.**create_payouts**()
Groups payable payouts at the end of the day by currency for easier paying out and database compaction, allowing deletion of regular payout records.

simplecoin.scheduler.**create_trade_req**(*typ*)
Takes all the credits in need of exchanging (either buying or selling, not both) and attaches them to a new trade request.

simplecoin.scheduler.**credit_block**(*redis_key*, *simulate=False*)
Calculates credits for users from share records for the latest found block.

simplecoin.scheduler.**credit_cleanup**(*days_ago=7*, *batch_size=10000*, *sleep=1*, *dont_simulate=True*)

simplecoin.scheduler.**crontab**(*func*)
Handles rolling back SQLAlchemy exceptions to prevent breaking the connection for the whole scheduler. Also records timing information into the cache

simplecoin.scheduler.**distributor**(**args*, ***kwargs*)

simplecoin.scheduler.**generate_credits**(*dont_simulate=True*)
Loops through all the blocks that haven't been credited out and attempts to process them

simplecoin.scheduler.**leaderboard**()

simplecoin.scheduler.**main**()

simplecoin.scheduler.**reload_cached**()
Recomputes all the cached values that normally get refreshed by tasks. Good to run if celery has been down, site just setup, etc.

simplecoin.scheduler.**server_status**()
Periodically poll the backend to get number of workers and other general status information.

simplecoin.scheduler.**share_cleanup**(*dont_simulate=True*)
Runs chain_cleanup on each chain.

simplecoin.scheduler.**update_block_state**(*block_id=None*)
Loops through blocks (default immature and non-orphaned blocks)

If *block_id* is passed, instead of the checking the default blocks, all blocks of the same currency of a >= id will be updated.

First checks to see if blocks are orphaned, then it checks to see if they are now matured.

simplecoin.scheduler.**update_network**()
Queries the RPC servers confirmed to update network stats information.

simplecoin.scheduler.**update_online_workers**()

Grabs data on all currently connected clients. Forms a dictionary of this form:
dict(address=dict(worker_name=dict(powerpool_id=connection_count)))

And caches each addresses connection summary as a single cache key.

Views

RPC Views

simplecoin.rpc_views.**api_error_handler**(*exc*)

simplecoin.rpc_views.**associate_payouts**()
Used to update a SC Payout with a network transaction. This will create a new CoinTransaction object and link it to the transactions to signify that the transaction has been processed.

simplecoin.rpc_views.**check_signature**()

simplecoin.rpc_views.**confirm_transactions**()
Used to confirm that a transaction is now complete on the network.

simplecoin.rpc_views.**get_payouts**()
Used by remote procedure call to retrieve a list of payout amounts to be processed. Transaction information is signed for safety.

simplecoin.rpc_views.**get_trade_requests**()
Used by remote procedure call to retrieve a list of sell requests to be processed. Transaction information is signed for safety.

simplecoin.rpc_views.**sign**(*data*, *code=200*)

simplecoin.rpc_views.**update_trade_requests**()
Used as a response from an rpc sell request system. This will update the amount received for a sell request and its status. Both request and response are signed.

Config Objects

class simplecoin.config.**Algo**(*bootstrap*)

defaults = {'enabled': True}

class simplecoin.config.**AlgoKeeper**(*configs*)

active_algos()

type_map = {'default': <class 'simplecoin.config.Algo'>}

class simplecoin.config.**Chain**(*bootstrap*)

algo

calc_shares(*block_payout*)
Pass a block_payout object with only chain ID and blockhash populated and compute share amounts

currencies

defaults = {'safety_margin': 2, 'currencies': [], 'block_bonus': '0'}

max_indexes = 1000

min_index = 0

requires = ['type', 'fee_perc', '_algo', '_currencies', 'safety_margin']

class simplecoin.config.**ChainKeeper**(*configs*)

type_map = {'ppln': <class 'simplecoin.config.PPLNSChain'>, 'prop': <class 'simplecoin.config.PropChain'>}

class simplecoin.config.**ConfigChecker**(*cfg*, *app*)
This class provides various methods for validating config values and checks configuration values and makes sure they're properly filled out.

It needs a lot of expansion :/

Currently validates the following keys: pool_payout_addr currencies

check_is_bcaddress(*val*)
Helper method: Checks a value for truthiness

check_truthiness(*val*)
Helper method: Checks a value for truthiness

check_type(*val*, *obj_type*)
Helper method: Checks a value to make sure its the correct type

lookup_key(*key*, *nested=None*)
Helper method: Checks the config for the specified key and raises an error if not found

class simplecoin.config.**ConfigObject**(*bootstrap*)

defaults = {}

requires = []

class simplecoin.config.**Currency**(*bootstrap*)

algo

defaults = {'tx_explore': None, 'coinserv': {}, 'sellable': False, 'block_explore': None, 'minimum_payout': '0.0000000

pool_payout

requires = ['_algo', 'name', 'address_version', 'trans_confirmations', 'block_time', 'block_mature_confirms']

class simplecoin.config.**CurrencyKeeper**(*configs*)

available_versions

buyable_currencies

lookup_payable_addr(*address*)

Checks an address to determine if its a valid and payable(buyable) address. Typically used to validate a username address. Returns the payable currency object for that version.

When calling this function you should always expect exceptions to be raised.

!!! This function assumes that a currency will not be configured as buyable if there is a version conflict with another currency.

Although it makes this assumption - it should return a consistent Currency obj even if configuration is incorrect

sellable_currencies

type_map = {'default': <class 'simplecoin.config.Currency'>}

unbuyable_currencies

unmineable_currencies

unsellable_currencies

validate_bc_address(*bc_address_str*)

The go-to function for all your bitcoin style address validation needs.

Expects to receive a string believed to represent a bitcoin address Raises appropriate errors if any checks are failed, otherwise returns a list of Currency objects that have the same addr version.

class simplecoin.config.**Keeper**(*configs*)

class simplecoin.config.**Location**(*bootstrap*)

required = ['location_acronym', 'location', 'country_flag', 'address']

stratums_by_algo()

class simplecoin.config.**LocationKeeper**(*configs*)

type_map = {'default': <class 'simplecoin.config.Location'>}

class simplecoin.config.**PPLNSChain**(*bootstrap*)

calc_shares(*block_payout*)

requires = ['type', 'fee_perc', '_algo', '_currencies', 'safety_margin', 'last_n']

class simplecoin.config.**PowerPool**(*bootstrap*)

chain

display_text

full_info()

location

request(*url*, *method='GET'*, *max_age=None*, *signed=True*, ***kwargs*)

requires = ['_chain', 'port', 'address', 'monitor_address', '_location']

stratum_address

timeout = 10

class simplecoin.config.**PowerPoolKeeper**(*configs*)

type_map = {'default': <class 'simplecoin.config.PowerPool'>}

class simplecoin.config.**PropChain**(*bootstrap*)

calc_shares(*block_payout*)

S

A

B

C

D

E

F

Q

R

S

T

U

V

W