cryptoassets.core Documentation

Release 0.1

Mikko

July 10, 2015

Contents

The ultimate solution to build cryptoassets services

cryptoassets.core is Bitcoin, cryptocurrency and cryptoassets database and accounting library for Python.

CHAPTER 1

Community and resources

- Project homepage
- Documentation
- Discussion group and mailing list
- Source code and issue tracker
- Continuos integration service
- Code coverage report

CHAPTER 2

Installation

2.1 Installing cryptoassets.core package

2.1.1 Requirements

You need at least Python version 3.4.

- Ubuntu 14.04 comes with Python 3.4. Install Python 3.4 on older versions of Ubuntu
- Install Python 3.4 on OSX
- Microsoft Windows is unsupported at the moment as the authors do not have access to Microsoft Windows development tools

2.1.2 Create a virtualenv

`cryptoassets.core` is distributed as a Python package. To use packages in your application, follow the Python community best practices and create a virtualenv where you to install the third party packages and their dependencies.

OSX

For Homebrew with Python 3.4 installed:

```
mkdir myproject
cd myproject
virtualenv python3.4 -m venv venv
source venv/bin/activate
```

Ubuntu / Debian

First get a virtualenv which is not old and horribly broken, like the one in the system default installation:

```
sudo pip install -U virtualenv
```

This creates `/usr/local/bin/virtualenv`. Then:

```
mkdir myproject
cd myproject
virtualenv -p python3.4 venv
source venv/bin/activate
```

Note: Ubuntu and Debian have an open issue regarding Python 3.4 virtualenv support. Thus, check the link below for how to workaround installation issues if you are using a non-conforming distribution.

- Issues with Ubuntu / Debian, Python 3.4 and virtualenv - see workaround

2.1.3 Installing cryptoassets package

Installing the release version

After virtualenv is created and active you can run:

```
# Install the known good versions of dependencies
curl -o requirements.txt "https://bitbucket.org/miohtama/cryptoassets/raw/9bfbe5e16fd878cbb8231f06d88

# Install the latest release version of cryptoassets.core package
pip install cryptoassets.core
```

This will use the latest package pindowns of known good version set.

Installing the development version

You can install git checkout if you want to develop / bug fix *cryptoassets.core* itself.

First install dependencies.

Checkout and install from Bitbucket:

```
git clone https://miohtama@bitbucket.org/miohtama/cryptoassets.git
cd cryptoassets
python setup.py develop
```

CHAPTER 3

Getting started

3.1 Introduction

This tutorial introduces *cryptoassets.core*: what it does for you and how to set up a trivial Bitcoin wallet command line application on the top of it.

cryptoassets.core is a Python framework providing safe, scalable and future-proof cryptocurrency and cryptoassets accounting for your Python application. You can use it to accept cryptocurrency payments, build cryptoasset services and exchanges.

- See PyPi download page
- See project homepage

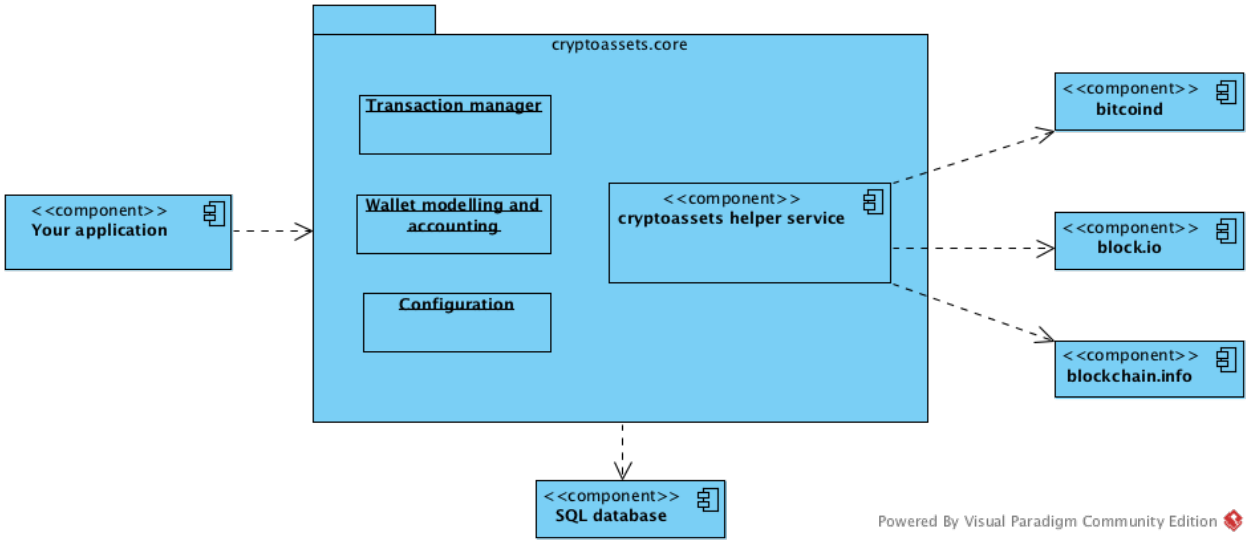
3.2 Benefits

cryptoassets.core is built on the top of Python programming language, community ecosystem and best practices. Python is proven tool for building financial applications and is widely used to develop cryptoassets software and Bitcoin exchanges. *cryptoassets.core* is

- Easy: Documented user-friendly APIs.

- Extensible: Any cryptocurrency and cryptoassets support.
- Safe: Secure and high data integrity.
- Lock-in free: Vendor independent and platform agnostics.
- Customizable: Override and tailor any part of the framework for your specific needs.

3.3 Basics



- You can use *cryptoassets.core* framework in any Python application, including Django applications. Python 3 is required.
- *cryptoassets.core* is designed to be extendable to support altcoins and different cryptoassets.
- *cryptoassets.core* works with API services (block.io, blockchain.info) and daemons (*bitcoind*, *dogecoind*). The framework uses term *backend* to refer these. You either sign up for an account on the API service or run the daemon on your own server (*)
- Basic SQLAlchemy knowledge is required for using the models API.
- A separate a cryptoassets helper service process is responsible for communicating between your application and cryptoasset networks. This process runs on background on your server.
- *cryptoassets.core* framework is initialized from a configuration, which can be passed in as a Python dictionary or a YAML configuration file.
- For data integrity reasons, *cryptoassets.core* database connection usually differs from the default application database connection.
- At the moment *cryptoassets.core* is in initial version 0.1 release. Expect the scope of the project to expand to support other cryptoassets (Counterparty, Ethereum, BitShares-X) out of the box.

Note: Please note that running *bitcoind* on a server requires at least 2 GB of RAM and 25 GB of disk space, so low end box hosting plans are not up for the task.

3.3.1 Interacting with cryptoassets.core

The basic programming flow with *cryptoassets.core* is

- You set up `cryptoassets.core.app.CryptoAssetsApp` instance and configure it inside your Python code.
- You set up a channel how cryptoassets helper service process calls backs your application. Usually this happens over HTTP web hooks.
- You put your cryptoassets database accessing code to a separate function and decorate it with `cryptoassets.core.app.CryptoAssetsApp.conflict_resolver` to obtain transaction conflict aware SQLAlchemy session.
- In your cryptoasset application logic, you obtain an instance to `cryptoassets.core.models.GenericWallet` subclass. Each cryptoasset has its own set of SQLAlchemy model classes. The wallet instance contains the accounting information: which assets and which transactions belong to which users. Simple applications usually require only one default wallet instance.
- After having set up the wallet, call various wallet model API methods like `cryptoassets.core.models.GenericWallet.send()`.
- For receiving the payments you need to create at least one receiving address (see `cryptoassets.core.models.GenericWallet.create_receiving_address()`). *Cryptoassets helper service* triggers events which your application listens to and then performs application logic when a payment or a deposit is received.

3.4 Example command-line application

Below is a simple Bitcoin wallet terminal application using block.io API service as the backend. It is configured to work with Bitcoin Testnet. Testnet Bitcoins are worthless, free to obtain and thus useful for application development and testing.

The example comes with pre-created account on block.io. It is recommended that you sign up for your own block.io account and API key and use them instead of ones in the example configuration.

3.4.1 Install cryptoassets.core

First make sure you have created a virtualenv and installed cryptoassets.core and its dependencies.

3.4.2 Application code

Note: The example is tested only for UNIX systems (Linux and OSX). The authors do not have availability of Microsoft development environments to ensure Microsoft Windows compatibility.

Here is an example walkthrough how to set up a command line application.

Save this as `example.py` file.

```
"""cryptoassets.core example application"""

import os
import warnings
from decimal import Decimal
```

```
import datetime


from sqlalchemy.exc import SAWarning

from cryptoassets.core.app import CryptoAssetsApp
from cryptoassets.core.configure import Configurator
from cryptoassets.core.utils.httpeventlistener import simple_http_event_listener
from cryptoassets.core.models import NotEnoughAccountBalance

# Because we are using a toy database and toy money, we ignore this SQLLite database warning
warnings.filterwarnings(
    'ignore',
    r"^Dialect sqlite\+pysqlite does \*not\* support Decimal objects natively\, "
    "and SQLAlchemy must convert from floating point - rounding errors and other "
    "issues may occur\. Please consider storing Decimal numbers as strings or "
    "integers on this platform for lossless storage\.$",
    SAWarning, r'^sqlalchemy\.sql\.type_api$')


assets_app = CryptoAssetsApp()

# This will load the configuration file for the cryptoassets framework
# for the same path as examply.py is
conf_file = os.path.join(os.path.dirname(__file__), "example.config.yaml")
configurer = Configurator(assets_app)
configurer.load_yaml_file(conf_file)

# This will set up SQLAlchemy database connections, as loaded from
# config. It's also make assets_app.conflict_resolver available for us
assets_app.setup_session()


# This function will be run in its own background thread,
# where it runs mini HTTP server to receive and process
# any events which cryptoassets service sends to our
# process
@simple_http_event_listener(configurer.config)
def handle_cryptoassets_event(event_name, data):
    if event_name == "txupdate":
        address = data["address"]
        confirmations = data["confirmations"]
        txid = data["txid"]
        print("")
        print("")
        print("Got transaction notification txid:{} addr:{}, confirmations:{}".
            format(txid, address, confirmations))
        print("")


def get_wallet_and_account(session):
    """Return or create instances of the default wallet and accout.

    :return: Tuple (BitcoinWallet instance, BitcoinAccount instance)
    """

    # This returns the class cryptoassets.core.coins.bitcoin.models.BitcoinWallet.
    # It is a subclass of cryptoassets.core.models.GenericWallet.
```

```
    # You can register several of cryptocurrencies to be supported within your application,
    # but in this example we use only Bitcoin.
    WalletClass = assets_app.coins.get("btc").wallet_model

    # One application can have several wallets.
    # Within a wallet there are several accounts, which can be
    # user accounts or automated accounts (like escrow).
    wallet = WalletClass.get_or_create_by_name("default wallet", session)
    session.flush()

    account = wallet.get_or_create_account_by_name("my account")
    session.flush()

    # If we don't have any receiving addresses, create a default one
    if len(account.addresses) == 0:
        wallet.create_receiving_address(account, automatic_label=True)
        session.flush()

    return wallet, account


# Every time you access cryptoassets database it must happen
# in sidea managed transaction function.
#
# Use ConflictResolevr.managed_transaction decoreator your function gets an extra
# ``session`` argument as the first argument. This is the SQLAlchemy
# database session you should use to  manipulate the database.
#
# In the case of a database transaction conflict, ConflictResolver
# will rollback code in your function and retry again.
#
# For more information see
# http://cryptoassetscore.readthedocs.org/en/latest/api/utils.html#transaction-conflict-resolver
#
@assets_app.conflict_resolver.managed_transaction
def create_receiving_address(session):
    """Create new receiving address on the default wallet and account."""
    wallet, my_account = get_wallet_and_account(session)

    # All addresses must have unique label on block.io.
    # Note that this is not a limitation of Bitcoin,
    # but block.io service itself.
    wallet.create_receiving_address(my_account, automatic_label=True)


@assets_app.conflict_resolver.managed_transaction
def send_to(session, address, amount):
    """Perform the actual send operation within managed transaction."""
    wallet, my_account = get_wallet_and_account(session)
    friendly_date = datetime.datetime.utcnow().strftime("%Y-%m-%dT%H:%M:%S")
    transaction = wallet.send(my_account, address, amount, "Test send at {}".format(friendly_date))
    print("Created new transaction #{}".format(transaction.id))


def send():
    """Ask how many BTCTEST bitcoins to send and to which address."""
    address = input("Give the bitcoin TESTNET address where you wish to send the bitcoins:")
    amount = input("Give the amount in BTC to send:")
```

```
    try:
        amount = Decimal(amount)
    except ValueError:
        print("Please enter a dot separated decimal number as amount.")
        return

    try:
        send_to(address, amount)
    except NotEnoughAccountBalance:
        print("*" * 40)
        print("Looks like your wallet doesn't have enough Bitcoins to perform the send. Please top up
        print("*" * 40)


@assets_app.conflict_resolver.managed_transaction
def print_status(session):
    """Print the state of our wallet and transactions."""
    wallet, account = get_wallet_and_account(session)

    # Get hold of classes we use for modelling Bitcoin
    # These classes are defined in cryptoassets.core.coin.bitcoind.model models
    Address = assets_app.coins.get("btc").address_model
    Transaction = assets_app.coins.get("btc").transaction_model

    print("-" * 40)
    print("Account #{}, confirmed balance {:.8f} BTC, incoming BTC {:.8f}". \
        format(account.id, account.balance, account.get_unconfirmed_balance()))

    print("")
    print("Receiving addresses available:")
    print("(Send Testnet Bitcoins to them to see what happens)")

    for address in session.query(Address).filter(Address.account == account):
        print("- {}: confirmed received {:.8f} BTC".format(address.address, address.balance))
    print("")
    print("Wallet transactions:")
    for tx in session.query(Transaction):
        if tx.state in ("pending", "broadcasted"):

            # This transactions might not be broadcasted out by
            # cryptoassets helper service yet, thus it
            # does not have network txid yet
            txid = "(pending broadcast)" if tx.state == "pending" else tx.txid

            print("- OUT tx:{} to {} amount:{:.8f} BTC confirmations:{}".format(
                txid, tx.address.address, tx.amount, tx.confirmations))
        elif tx.state in ("incoming", "processed"):
            print("- IN tx:{} to:{} amount:{:.8f} BTC confirmations:{}".format(
                tx.txid, tx.address.address, tx.amount, tx.confirmations))
        else:
            print("- OTHER tx:{} {} amount:{:.8f} BTC".format(
                tx.id, tx.state, tx.amount))

    print("")
    print("Available commands:")
    print("1) Create new receiving address")
    print("2) Send bitcoins to other address")
    print("3) Quit")
```

```
print("Welcome to cryptoassets example app")
print("")

running = True
while running:

    print_status()
    command = input("Enter command [1-3]:")
    print("")
    if command == "1":
        create_receiving_address()
    elif command == "2":
        send()
    elif command == "3":
        running = False
    else:
        print("Unknown command!")
```

3.4.3 Example configuration

Save this as `example.config.yaml` file.

```
---

# Cryptoassets.core configuration for example application

database:
    url: sqlite:////tmp/cryptoassets.example.sqlite

# What services we use to run talk to the cryptocurrency networks.
# This will configure us to use pre-defined block.io API service
# testnet accounts for BTC and Doge (coins are worthless)
coins:
    btc:
        backend:
            class: cryptoassets.core.backend.blockio.BlockIo
            api_key: b2db-c8ad-29d2-c611
            pin: ThisIsNotVerySecret1
            network: btctest
            # walletnotify section tells how we receive
            # transaction updates from the  the backend
            # (new deposits to the backend wallet)
            walletnotify:
                class: cryptoassets.core.backend.blockiowebsocket.BlockIoWebsocketNotifyHandler

# This section tells how cryptoassets helper process will
# notify your app from events like new incoming transactions
# and outgoing transaction confirmation updates
events:
    # For each event, we send a HTTP webhook notification
    # to your app. Your app should be listening HTTP at localhost:10000
    example_app:
        class: cryptoassets.core.event.http.HTTPEventHandler
        url: http://localhost:10000
```

3.4.4 Creating the database structure

The example application uses SQLite database as a simple self-contained test database.

Run the command to create the database tables:

```
cryptoassets-initialize-database example.config.yaml
```

This should print out:

```
[11:49:16] cryptoassets.core version 0.0
[11:49:16] Creating database tables for sqlite:////tmp/cryptoassets.example.sqlite
```

3.4.5 Running the example

The example application is fully functional and you can start your Bitcoin wallet business right away. Only one more thing to do...

...the communication between cryptoasset networks and your application is handled by the cryptoassets helper service background process. Thus, nothing comes in or goes out to your application if the helper service process is not running. Start the helper service:

```
cryptoassets-helper-service example.config.yaml
```

You should see something like this:

```
...
[00:23:09] [cryptoassets.core.service.main splash_version] cryptoassets.core version 0.0
```

You might get some connection refused errors, because the app is not up yet. Please ignore those now.

Now leave *cryptoassets helper service* running and start the example application **in another terminal**:

```
python example.py
```

You should see something like this:

```
Welcome to cryptoassets example app

Receiving addresses available:
(Send testnet Bitcoins to them to see what happens)
- 2MzGzEUyHgqBXzbuGCJDSBPKAyRxhj2q9hj: total received 0.00000000 BTC

We know about the following transactions:

Give a command
1) Create new receiving address
2) Send Bitcoins to other address
3) Quit
```

You will get some *Rescanned transactions* log messages on the start up if you didn't change the default block.io credentials. These are test transactions from other example users.

Now you can send or receive Bitcoins within your application. If you don't start the helper service the application keeps functioning, but all external cryptoasset network traffic is being buffered until the *cryptoassets helper service* is running again.

If you want to reset the application just delete the database file `/tmp/cryptoassets.test.sqlite`.

3.4.6 Obtaining testnet Bitcoins and sending them

The example runs on testnet Bitcoins which are not real Bitcoins. Get some testnet coins and send them from the faucet to the receiving address provided by the application.

List of services providing faucets giving out Testnet Bitcoins.

No more than **0.01** Bitcoins are needed for playing around with the example app.

After sending the Bitcoins you should see a notification printed for an incoming transaction in ~30 seconds which is the time it takes for the Bitcoin transaction to propagate through testnet:

```
Got transaction notification txid:512a082c2f4908d243cb52576cd5d22481344faba0d7a837098f9af81cfa8ef3 ac
```

3.5 After completing the example

Explore model API documentation, configuration and what tools there are available.

You can also study Liberty Music Store open source application, built on the top of Django and Bitcoin.

3.5.1 Django integration

If you are using Django see cryptoassets.django package.

3.5.2 More about SQLAlchemy

Please see these tutorials

- Official SQLAlchemy tutorial
- SQLAlchemy ORM Examples by Xiaonuo Gantan

3.5.3 Questions?

See the community resources how to contact the developers.

CHAPTER 4

Service and commands

4.1 Introduction

Cryptoassets helper service is a standalone process responsible for communication between cryptocurrency networks, cryptocurrency API providers and your application.

Primarily Cryptoassets helper service

- Broadcasts new outgoing transactions to the cryptocurrency network
- Gets transaction notifications from cryptocurrency daemons and APIs and then notifies your application about the transaction updates

Even if network connections go down, you lose connection to APIs or cryptocurrency networks, cryptoassets library continuous to work somewhat indepedently. The users can send and receive transactions, which are buffered until the network becomes available again. Some functions, which are synchronous by nature, like creating new addresses, might not be available.

Besides providing a daemon for communications additional helping commands are available

- Initialize database
- Rescan wallet for missed transations

4.2 cryptoassets-helper-service

This command is the service helper process. The service process must be running on the background for your application to receive external deposit transactions and broadcast outgoing transctions.

4.2.1 Running the service with Python project

After installing *cryptoassets.core* to your virtualenv you should be able to run the cryptoassets helper service as following:

```
cryptoassets-helper-service <your YAML config file>
```

4.2.2 Running the service with Django

If you are running a Django application, a special Django management command is provided by cryptoassets.django library.

4.2.3 Status server

- Cryptoassets helper service* comes with a built-in mini status server. You can use this to
- Diagnose to see that *cryptoassets helper service* process is alive and runnign well
- Debug incoming and outgoing transaction issues

By default the status server listens to http://localhost:18881. See configuration how to include a status server in cryptoassets helper service.

Note: Status server is designed only for testing and diagnostics purpose and does not scale to production use.

Warning: It is not safe to expose status server to the Internet. Make sure you have authenticating proxy set up or only expose this to localhost.

4.2.4 System service integration

To have automatic start/stop and other functionality for cryptoassets helper service, use something akin *systemd* or supervisord.

Send `SIGTERM` signal to the service for graceful shutdown. Give the service 45 seconds to gracefully shutdown its own threads until sending `SIGKILL` if the service has not terminated itself.

The clean shutdown is indicated by exit code `0`.

In the case of any of the service threads dying the service will shutdown itself with exit code `2`.

You can configure logging using Python logging best practices.

Note: Further system exit codes coming in upcoming releases.

4.3 cryptoassets-initialize-database

This command will create database tables for different cryptocurrencies as described in the configuration file. Usually you need to do this only once when setting up the database.

4.4 cryptoassets-scan-received

Rescan all receiving addresses for missed deposit transactions.

This is also performed automatically on startup of *cryptoassets helper service*.

For more information see *cryptoassets.core.tools.receivescan*.

Note: At the moment it is not recommended to run this command while cryptoassetshelper is running on background.

CHAPTER 5

Configuration

5.1 Introduction

cryptoassets.core must know about cryptocurrencies, databases and backends you use in your application.

5.2 Creating application object and configuring it

Most of interaction with *cryptoassets.core* is done through `cryptoassets.core.app.CryptoAssetsApp` application object. Create one singleton instance within your application:

```
from cryptoassets.core.app import CryptoAssetsApp

assets_app = CryptoAssetsApp()
```

5.2.1 Configuring using YAML configuration file

Use `cryptoassets.configuration.Configuraror.load_yaml_file()` to load YAML syntax config file:

```
from cryptoassets.core.app import CryptoAssetsApp
from cryptoassets.core.configuration import Configurator

assets_app = CryptoAssetsApp()

# This will load the configuration file for the cryptoassets framework
configurer = Configurator(assets_app)
configurer.load_yaml_file("my-cryptoassets-settings.yaml")
```

Example YAML configuration file:

```
---

# Cryptoassets.core configuration for example application

database:
    url: sqlite:////tmp/cryptoassets.example.sqlite

# What services we use to run talk to the cryptocurrency networks.
# This will configure us to use pre-defined block.io API service
# testnet accounts for BTC and Doge (coins are worthless)
coins:
    btc:
        backend:
            class: cryptoassets.core.backend.blockio.BlockIo
            api_key: b2db-c8ad-29d2-c611
            pin: ThisIsNotVerySecret1
            network: btctest
            # walletnotify section tells how we receive
            # transaction updates from the  the backend
            # (new deposits to the backend wallet)
            walletnotify:
                class: cryptoassets.core.backend.blockiowebsocket.BlockIoWebsocketNotifyHandler

# This section tells how cryptoassets helper process will
# notify your app from events like new incoming transactions
# and outgoing transaction confirmation updates
events:
    # For each event, we send a HTTP webhook notification
    # to your app. Your app should be listening HTTP at localhost:10000
    example_app:
        class: cryptoassets.core.event.http.HTTPEventHandler
        url: http://localhost:10000
```

5.2.2 Configuring using Python dict

You can give your settings as Python dictionary:

```
CRYPTOASSETS_SETTINGS = {

    # You can use a separate database for cryptoassets,
    # or share the Django database. In any case, cryptoassets
    # will use a separate db connection.
```

```
    "database": {
        "url": "postgresql://localhost/cryptoassets",
        "echo": True,
    },

    # Locally running bitcoind in testnet
    "coins": {
        "btc": {
            "backend": {
                "class": "cryptoassets.core.backend.bitcoind.Bitcoind",
                "url": "http://x:y@127.0.0.1:9999/",

                # bitcoind has 60 seconds to get back to us
                "timeout": 60,

                # Cryptoassets helper process will use this UNIX named pipe to communicate
                # with bitcoind
                "walletnotify": {
                    "class": "cryptoassets.core.backend.httpwalletnotify.HTTPWalletNotifyHandler",
                    "ip": "127.0.0.1",
                    "port": 28882
                },
            },

            # We run in testnet mode
            "testnet": True
        },
    },
}

configurator.load_from_dict(CRYPTOASSETS_SETTINGS)
```

5.3 Configuration sections

5.3.1 database

Configure usd SQLAlchemy database connection.

Example:

```
"database": {
    "url": "postgresql://localhost/cryptoassets",
    "echo": true,
}
```

Note: The database connection will always use Serializable transaction isolation level.

For more information see

- Data integrity
- SQLAlchemy isolation_level

url

SQLAlchemy connection URL.

echo

Set to `true` (or in Python to `True`) and executed SQL statements will be logged via Python logging.

5.3.2 coins

Configure database models and used backends for cryptocurrencies and assets enabled in your application.

This dictonary contains a list of subentries.

- Name of each entry is acronym of the cryptoasset in lowercase (`btc`, `doge`)

Example:

```
"coins": {
    # AppleByte using applebyted (bitcoind-like) as the backend
    "aby": {
        "backend": {
            "class": "cryptoassets.core.backend.bitcoind.Bitcoind",
            "url": "http://x:y@127.0.0.1:8607/",
            "walletnotify": {
                "class": "cryptoassets.core.backend.httpwalletnotify.HTTPWalletNotifyHandler",
                "ip": "127.0.0.1",
                "port": 28882
            },
        },
    },

    "bitcoin": {
        "backend": {
            "class": "cryptoassets.core.backend.bitcoind.Bitcoind",
            "url": "http://foo:bar@127.0.0.1:8332/"
            "max_tracked_incoming_confirmations": 20,
            "walletnotify":
                "class": "cryptoassets.core.backend.pipewalletnotify.PipedWalletNotifyHandler",
                "fname": "/tmp/cryptoassets-unittest-walletnotify"
        }
    }
},
```

models

Optional.

You can set up your own altcoin or override default SQLAlchemy model configuration for an existing cryptoasset.

The value of this variable is the Python module containing `coin_description` variable. For more information how to describe your cryptoasset models, see *`cryptoassets.core.coin.registry`*.

Example:

```
"jesuscoin": {
    "backend": {
        "class": "cryptoassets.core.backend.bitcoind.Bitcoind",
        "url": "http://x:y@127.0.0.1:8607/",
        "walletnotify": {
            "class": "cryptoassets.core.backend.httpwalletnotify.HTTPWalletNotifyHandler",
            "ip": "127.0.0.1",
            "port": 28882
        },
    },
    "models": "mycoin.models"
},
```

testnet

Set to `true` (Python `True`) if the coin backend is connected to a testnet.

This may affect address validation in the future. Currently this information is not utilized.

Example:

```
"jesuscoin": {
    "backend": {
        "class": "cryptoassets.core.backend.bitcoind.Bitcoind",
        "url": "http://x:y@127.0.0.1:8607/",
    },
    "testnet": True
},
```

max_confirmation_count

This is how many confirmations `tools.confirmationupdate` tracks for each network transactions, both incoming and outgoing, until we consider it "closed" and stop polling backend for updates. The default value is `15`.

For more information see *`cryptoassets.core.tools.confirmationupdate`*.

backend

Installed backends for one cryptoasset in `coins` section.

For the available backends see backends list.

Each backend contains the following options

> **param class** tells which backend we are going to use
>
> **param walletnofiy** tells what kind of incoming transaction notifications we have from the backend
>
> **param max_tracked_incoming_confirmations** This applications for mined coins and backends which do not actively post confirmations updates. It tells up to how many confirmations we poll the backend for confirmation updates. For details see `cryptoassets.core.tools.opentransactions`.

Other options: All backends take connection details (url, IPs) and credentials (passwords, API keys, etc.) These options are backend specific, so see the details from the backend documentation.

Example:

```
"coins" : {
    "btc": {
        "backend": {
            "class": "cryptoassets.core.backend.bitcoind.Bitcoind",
            "url": "http://x:y@127.0.0.1:8607/",
            "walletnotify": {
                "class": "cryptoassets.core.backend.httpwalletnotify.HTTPWalletNotifyHandler",
                "ip": "127.0.0.1",
                "port": 28882
            },
        },
    },
}
```

walletnotify

Wallet notify configuration tells how cryptoassets helper service receives cryptoasset transaction updates from the cryptoassets backend (bitcoind, API service). Unless this is configured, cryptoassets service or your application won't know about incoming transactions.

`walletnotify` section must be given in under backend configuration. It's content depends on the chosen wallet notification method. For more information see qallet notification documentation.

Example:

```
"jesuscoin": {
    "backend": {
        "class": "cryptoassets.core.backend.bitcoind.Bitcoind",
        "url": "http://x:y@127.0.0.1:8607/",
        "walletnotify": {
            "class": "cryptoassets.core.backend.httpwalletnotify.HTTPWalletNotifyHandler",
            "ip": "127.0.0.1",
            "port": 28882
        },
    },
},
```

5.3.3 events

Event handling is configured in the `events` section of the configuration file.

Event handling configuration tells how cryptoassets helper service notifies your application about occured events (transaction updates, etc.). There exist various means to communicate between your application and *cryptoassets helper service*.

For more information and examples read event API documentation.

Event section consists name and configuration data pairs. Currently event handler name is only used for logging purposes. You can configure multiple event handlers

Each handler gets **class** parameters and event handler specific setup parameters.

Example configuration

```
# List of cryptoassets notification handlers.
# Use this special handler to convert cryptoassets notifications to Django signals.
"events": {
    "django": {
```

```
        "class": "cryptoassets.core.event.python.InProcessNotifier",
        "callback": "cryptoassets.django.incoming.handle_tx_update"
    }
},
```

5.3.4 status_server

Configure mini status server which you can use to check *cryptoassets helper service* status.

Example:

```
"database": {
    "url": "sqlite:////tmp/payments.sqlite",
},

...

"status-server": {
    "ip": "127.0.0.1",
    "port": 9000
}
```

ip

IP address the status server will be listening to. Default 127.0.0.1.

port

Port the status server is listening to.s

5.3.5 service

Cryptoassets helper service process specific configuration. These sections only concern *cryptoassets helper service* process itself, not any framework calls you make inside your own Python application.

Example:

```
service:
    broadcast_period: 60
    logging:
        formatters:
            verbose:
                format: '%(levelname)s %(asctime)s %(module)s %(process)d %(thread)d %(message)s'

        handlers:
            file:
                level: DEBUG
                class: logging.FileHandler
                filename: /tmp/cryptoassets-startstop-test.log
                formatter: verbose
            console:
                level: DEBUG
                class: logging.StreamHandler
                formatter: verbose
```

```
                stream: ext://sys.stdout

        root:
            handlers:
                - file
                - console
            level: DEBUG
```

broadcast_period

How often (seconds) the helper service will check for outgoing transactions to broadcast.

Default is 30 seconds.

logging

Configure logging for *cryptoassets helper service*. Loggers are not configured if you import and call *cryptoassets.core* framework from your application.

cryptoassets.core uses standard Python logging mechanism.

For logging within your application when calling model methods configure logging with Python logging configuration.

CHAPTER 6

Application and API service support

6.1 Introduction

cryptoassets.core can operate on raw cryptocurrency server daemon. Alternative you can choose one of the API services in the case you do not have the budget to run the full cryptocurrency node.

One instance of *cryptoassets.core* supports multiple backends. E.g. you can run application doing both Bitcoin and Dogecoin at the same time. However one backend can be enabled for one cryptoasset at a time.

After the backend has been set up you rarely interact it with directly. Instead, you use model APIs and cryptoassets helper service takes care of cryptoassets operations.

6.1.1 Running your cryptocurrency daemon (bitcoind)

Pros

- You have 100% control of assets
- You are not dependend of any vendor (downtime, asset seizure)

Cons

- Running *bitcoind* requires root server access, 2 GB of RAM and 25 GB of disk space minimum and cannot be done on low budget web hosting accounts
- You need to have sysadmin skills and be aware of server security

6.1.2 Using API service

Pros

- Easy to set up
- Works with even low budget hosting

Cons

- Increases attack surface
- If the service provider goes out of business you might lose your hot wallet assets

6.2 Configuration

The backend is configured separate for each cryptocurrency (BTC, Doge) and registered in `cryptoassets.backend.registry`. Each backend takes different initialization arguments like API keys and passwords. You usually set up these in cryptoassets.core config.

The active backend configuration can be read through *`cryptoassets.core.coin.registry.CoinRegistry`*. Bindings between the backends and the cryptocurrenct are described by *`cryptoassets.core.coin.registry.Coin`* class.

6.3 Backends

6.3.1 bitcoind

bitcoind and bitcoind-derivate backend. Interact directly with *bitcoind* service running on your own server.

Because most bitcoin forks have the same JSON-RPC API as the original *bitcoind*, you can use this backend for having service for most bitcoind-derived altcoins.

You must configure *bitcoind* on your server to work with *cryptoassets.core*. This happens by editing bitcoin.conf.

Example `bitcoin.conf`:

```
# We use bitcoin testnet, not real bitcoins
testnet=1

# Enable JSON-RPC
server=1
```

```
# Username and password
rpcuser=foo
rpcpassword=bar

rpctimeout=5
rpcport=8332

# This must be enabled for gettransaction() to work
txindex=1

# Send notifications to cryptoassetshelper service over HTTP
walletnotify=curl --data "txid=%s" http://localhost:28882
```

Note: You need to install curl on your server too (sudo apt install curl)

The backend configuration takes following parameters.

param class Always `cryptoassets.core.backend.bitcoind.Bitcoind`

param url Bitcoind connection URL with username and password (rpcuser and rpcassword in bitcoin config) for AuthServiceProxy. Usually something like `http://foo:bar@127.0.0.1:8332/`

param walletnotify Dictionary of parameters to set up walletnotify handler.

param timeout Timeout for JSON-RPC call. Default is 15 seconds. If the timeout occurs, the API operation can be considered as failed and the bitcoind as dead.

Wallet notifications

Wallet notifications (or, short, `walletnotify`) is the term used by *cryptoasset.core* to describe how backend communicates back to cryptoassets helper service. It's named after *bitcoind* walletnotify option.

- You can setup different wallet notify method depending if you run daemon application locally, on a remote server or you use some API service
- In theory, you could mix and match backends and wallet notifications methods. But just stick to what is recommended for the backend recommends.
- Each cryptoasset require its own notification channel (named pipe, HTTP server port)

HTTP webhook for bitcoind

Handle walletnofify notificatians from bitcoind through curl / wget HTTP request.

This is useful in cases where *bitcoind* or alike is running on a remote server and you wish to receive walletnotifications from there. In this case, you can set up SSH tunnel and forward the locally started HTTP wallet notify listener to the bitcoind server.

Creates a HTTP server running in port 28882 (default). To receive a new transaction notification do a HTTP POST to this server:

```
curl --data "txid=%s" http://localhost:28882
```

E.g. in `bitcoind.conf`:

```
walletnotify=curl --data "txid=%s" http://localhost:28882
```

Options

param class Always `cryptoassets.core.backend.httpwalletnotify.HTTPWalletNotifyHandler`

param ip Bound IP address. Default 127.0.0.1 (localhost).

parma port Bound port. Default 28882.

Testing

To test that the wallet notifications are coming through

1. Make sure `cryptoassetshelper` service is running
2. Do `curl --data "txid=foobar" http://localhost:28882` on the server where *bitcoind* is running
3. You should see in the logs of `cryptoassetshelper`: *Error communicating with bitcoind API call gettransaction: Invalid or non-wallet transaction id*

Named UNIX pipe for bitcoind

Named pipes can be used to send incoming transactions notifications from locally installed cryptoassets daemons, namely *bitcoind* likes.

Cryptoassets helper service creates a name unix pipe.

Bitcoind or similar writes the transaction id to this pipe when updates are available for the transaction.

Named pipes are little bit more flexible than running a shell command, as you can do in-process handling of incoming transactions in a background thread, making it suitable for unit testing and such.

Example configuration:

```
# Locally running bitcoind in testnet
coins:
    btc:
        backend:
            class: cryptoassets.core.backend.bitcoind.Bitcoind
            url: http://foo:bar@127.0.0.1:8332/
            walletnotify:
                class: cryptoassets.core.backend.pipewalletnotify.PipedWalletNotifyHandler
                fname: /tmp/cryptoassets-walletnotify-pipe
```

And corresponding `bitcoind.conf`:

```
walletnotify=echo $1 >> /tmp/cryptoassets--walletnotify-pipe
```

Please note that you might want to use `timeout` (`gtimeout` on OSX) to prevent bad behavior in the case cryptoassets helper service is unable to read from the time. Thus, incoming transaction notifications are discarded after certain timeout:

```
walletnotify=gtimeout --kill-after=10 5 /bin/bash -c "echo $1 >> /tmp/cryptoassets-unittest-walletnot
```

Options

param class Always `cryptoassets.core.backend.pipewalletnotify.PipedWalletNotifyHandler`

param fname Filename where the pipe is opened. Please note that any existing filename which same name is removed.

param mode Unix file mode for the created pipe.

Redis pubsub for bitcoind

Use Redis pubsub for walletnotify notifications.

Redis offers mechanism called pubsub for channeled communication which can be used e.g. for interprocess communications.

1. Connects to a Redis database over authenticated conneciton
2. Opens a pubsub connection to a specific channel
3. bitcoind walletnofify writes notifies to this channel using `redis-cli` command line tool
4. This thread reads pubsub channel, triggers the service logic on upcoming notify

Example *walletnotify* line:

```
walletnotify=redis-cli publish bitcoind_walletnotify_pubsub %self
```

To install Redis on bitcoind server:

```
apt-get install redis-server redis-tools
```

Warning: The Redis authenticated connection is not encrypted. Use VPN or SSH tunnel to connect Redis over Internet.

Options

param class Always `cryptoassets.core.backend.rediswalletnotify.RedisWalletNotifyHandler`

param host IP/domain where Redis is running, default is `localhost`.

param port TCP/IP port Redis is listetning to

param db Redis database number

param username optional username

param password optional password

param channel Name of Redis pubsub channel where we write transaction txids, default `bitcoind_walletnotify_pubsub`

6.3.2 block.io

Block.Io API backend.

Supports Bitcoin, Dogecoin and Litecoin on block.io API.

The backend configuration takes following parameters.

param class Always `cryptoassets.core.backend.blockio.BlockIo`

param api_key block.io API key

param password block.io password

param network one of `btc`, `btctest`, `doge`, `dogetest`, see chain.so for full list

param walletnotify Configuration of wallet notify service set up for incoming transactions. You must use `cryptoassets.core.backend.blockiowebhook.BlockIoWebhookNotifyHandler` or `cryptoassets.core.backend.blockiowebocket.BlockIoWebsocketNotifyHandler` as `walletnotify` for incoming transactions for now. See below for more details.

Example configuration for block.io backend using websockets.

```
---
# Cryptoassets.core configuration for running block.io unit tests

database:
  url: sqlite:////tmp/cryptoassts-unittest-blockio.sqlite

coins:
    doge:
        backend:
            class: cryptoassets.core.backend.blockio.BlockIo
            api_key: yyy
            pin: xxxx
            network: dogetest
            walletnotify:
                class: cryptoassets.core.backend.blockiowebsocket.BlockIoWebsocketNotifyHandler
```

Wallet notifications over websockets

Handle notifications for incoming transactions using block.io websockets API.

https://block.io/docs/notifications

This will spin off a thread opening a websocket connection to block.io and listening for incoming events.

We use websocket-client library for websockets communications from Python.

Options

param class Always `cryptoassets.core.backend.blockiowebsocket.BlockIoWebsocketNotifyHandler`

Wallet notifications over web hooks (HTTP)

Handle notifications for incoming transactions using block.io HTTPO POST webhook API.

https://block.io/docs/notifications

This will spin off a HTTP server in a separate IP and port to listen to HTTP requests made by the block.io. You need to specify an external URL how block.io can reach the public IP address of your server.

Options

param class Always `cryptoassets.core.backend.blockiowebhook.BlockIoWebhookNotifyHandler`

param url To which public URL your webhook handler is mapped. The URL must not be guessable and must cointain random string, so that malicious actors cannot spoof incoming transaction requests.

param ip Bound IP address. Default 127.0.0.1 (localhost).

parma port Bound port. Default 33233.

Securing the webhooks

Do not expose webhook service port directly to the internet. Instead, use your web server to create a reverse proxy behind a hidden URL, so you can safely receive notifications from block.io.

HTTPS support through Nginx

Here is an example Nginx web server configuration how decode HTTPS and then forward block.io requets to the upstream server running in the cryptoassets helper service process:

```
# Secret block.io webhook endpoint
location /blockio-account-nofity/xyz {
    proxy_set_header   Host $host;
    proxy_set_header   X-Real-IP $remote_addr;
    proxy_set_header   X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header   X-Forwarded-Host $server_name;
    proxy_set_header   X-Forwarded-Proto $scheme;
    proxy_pass http://localhost:33233;
}
```

6.3.3 blockchain.info

blockchain.info backend.

The support is disabled in this version

6.3.4 null

Non-functional cryptocurrency backend doing nothing. Use for testing purposes.

The backend configuration takes following parameters.

param class Always `cryptoassets.core.backend.null.DummyCoinBackend`

CHAPTER 7

Cryptocurrency and asset support

7.1 Introduction

`cryptoassets.core` supports different cryptocurrencies. We use SQLAlchemy to generate a set of database tables and model classes for each coin. Then this coin can be send and received using a backend which takes care of communicating with the underlying cryptocurrency network.

See how to add support for more cryptocurrencies.

7.2 Bitcoin

Supported backends:

- bitcoind
- block.io
- blockchain.info

More info about Bitcoin. Bitcoin database implementation.

All amounts are stored in satoshis in integer fields.

Modify `BitcoinTransaction.confirmation_count` global to set the threshold when transcations are considered confirmed.

7.3 Dogecoin

More info about Dogecoin.

Supported backends:

- dogecoind (bitcoind-like)

- block.io

Dogecoin database support.

7.4 AppleByte

More info about AppleByte.

Supported backends:

- applebyted (bitcoind-like)

AppleByte database support.

CHAPTER 8

API documentation

Warning: readthedocs.org has issues building Python API documentation at the moment. This issue is being investigated. Meanwhile you can checkout cryptoassets.core source, install Sphinx package and run "make html" to build docs yourself.

Contents:

8.1 Base models

Base models describe how *cryptoassets.core* handles any cryptocurrency on the database level. SQLAlchemy library is used for modeling.

Models are abstract and when you instiate a new cryptocurrency, you inherit from the base classes and set the cryptocurrency specific properties.

Models also specify the core API how to interact with *cryptoassets.core*

See how to get started interacting with models.

For more information, see coin documentation and how to extend the framework with your own altcoins.

8.1.1 Cryptoasset registry

All running cryptoassets are maintained in a coin registry.

Each cryptoasset provides its own Wallet SQLAlchemy model and backend instance which is used to communicate with the network of the cryptoasset.

class cryptoassets.core.coin.registry.**CoinModelDescription**(*coin_name*, *wallet_model_name*, *address_model_name*, *account_model_name*, *transaction_model_name*, *network_transaction_model_name*, *address_validator*)

Describe one cryptocurrency data structures: what SQLAlchemy models and database tables it uses.

The instance of this class is used by `cryptoassets.core.models.CoinDescriptionModel` to build the model relatinoships and foreign keys between the tables of one cryptoasset.

Create the description with fully dotted paths to Python classes.

Parameters **coin_name** – Name of this coin, lowercase acronym

Wallet
Get wallet model class.

Address
Get address model class.

Account
Get account model class.

NetworkTransaction
Get network transaction model class.

Transaction
Get transaction model class.

class cryptoassets.core.coin.registry.**Coin**(*coin_description*, *backend=None*, *max_confirmation_count=15*, *testnet=False*)

Describe one cryptocurrency setup.

Binds cryptocurrency to its backend and database models.

We also carry a flag if we are running in testnet or not. This affects address validation.

Create a binding between asset models and backend.

Parameters

- **coin_description** – *cryptoassets.core.coin.registry.CoinModelDescription*
- **testnet** – Are we running a testnet node or real node.
- **backend** – *cryptoassets.core.backend.base.CoinBackend*

backend = None
Subclass of *cryptoassets.core.backend.base.CoinBackend*.

name = None
Lowercase acronym name of this asset

max_confirmation_count = None
This is how many confirmations `tools.confirmationupdate` tracks for each network transactions, both incoming and outgoing, until we consider it "closed" and stop polling backend for updates.

address_model
Property to get SQLAlchemy model for address of this cryptoasset.

Subclass of *cryptoassets.core.models.GenericAddress*.

transaction_model
Property to get SQLAlchemy model for transaction of this cryptoasset.

Subclass of *cryptoassets.core.models.GenericTransaction*.

account_model
Property to get SQLAlchemy model for account of this cryptoasset.

Subclass of *cryptoassets.core.models.GenericAccount*.

wallet_model
Property to get SQLAlchemy model for account of this cryptoasset.

Subclass of *cryptoassets.core.models.GenericWallet*.

network_transaction_model
Property to get SQLAlchemy model for account of this cryptoasset.

Subclass of *cryptoassets.core.models.GenericWallet*.

validate_address(*address*)
Check the address validy against current network.

Returns True if given address is valid.

class cryptoassets.core.coin.registry.**CoinRegistry**
Holds data of set up cryptocurrencies.

Usually you access this through `cryptoasssets.core.app.CryptoassetsApp.coins` instance.

Example:

```
cryptoassets_app = CryptoassetsApp()
# ... setup ...

bitcoin = cryptoassets_app.coins.get("btc)

print("We are running bitcoin with backend {}".format(bitcoin.backend))
```

all()
Get all registered coin models.

Returns List of tuples(coin name, Coin)

get(*name*)
Return coin setup data by its acronym name.

Parameters **name** – All lowercase, e.g. `btc`.

8.1.2 Default models

Default cryptocurrency names and their models.

cryptoassets.core.coin.defaults.**COIN_MODEL_DEFAULTS = {'ltc': 'cryptoassets.core.coin.litecoin.models', 'btc':**
This is the default mapping between the three-letter coin acronyms and their SQLAlchemy model presentations.
If you want to use your own database models you can override any of these in your configuration.

8.1.3 Model API conventions

The following conventions are followed in the model API

Model discovery

- Abstract base classes are called *GenericXxx* like *GenericWallet*.
- Actual class implementation is in `coin` module, e.g. `cryptoassets.core.coin.bitcoin.models.BitcoinWallet`
- You do not access the model classes directly, but through configured assets registry. E.g. to get a hold of `BitcoinWallet` class you do `Wallet = cryptoassets_app.coins.get("btc").coin_model`.
- The usual starting point for the calls is to get or create *cryptoassets.core.models.GenericWallet* instance. Check out *cryptoassets.core.models.GenericWallet.get_or_create_by_name()*.

Session lifecycle

- API tries to use the SQLAlchemy database session of the object if possible: `Session.object_session(self)`. If not, session must be explicitly given and you get your session inside a helper closure function decorated by *cryptoassets.core.utils.conflictresolver.ConflictResolver.managed_transaction()*. This way we guarantee graceful handling of transaction conflicts.
- API never does `session.flush()` or `session.commit()`
- API will do `session.add()` for newly created objects

8.1.4 Model classes

Below are the base classes for models. All cryptoassets have the same API as described these models.

Account

class `cryptoassets.core.models.`**`GenericAccount`**

An account within the wallet.

We associate addresses and transactions to one account.

The accountn can be owned by some user (user's wallet), or it can be escrow account or some other form of automatic transaction account.

The transaction between the accounts of the same wallet are internal and happen off-blockhain.

A special account is reserved for network fees caused by outgoing transactions.

`NETWORK_FEE_ACCOUNT` = 'Network fees'

Special label for an account where wallet will put all network fees charged by the backend

`id` = Column(None, Integer(), table=None, primary_key=True, nullable=False)

Running counter used in foreign key references

`name` = Column(None, String(length=255), table=None)

Human-readable name for this account

created_at = Column(None, DateTime(), table=None, default=ColumnDefault(<function ColumnDefault._maybe_wra
When this account was created

updated_at = Column(None, DateTime(), table=None, onupdate=ColumnDefault(<function ColumnDefault._maybe_w
Then the balance was updated, or new address generated

balance = Column(None, Numeric(precision=21, scale=8), table=None, nullable=False, default=ColumnDefault(0))
Available internal balance on this account NOTE: Accuracy checked for bitcoin only

pick_next_receiving_address_label()
Generates a new receiving address label which is not taken yet.

Some services, like block.io, requires all receiving addresses to have an unique label. We use this helper function in the situations where it is not meaningful to hand-generate labels every time.

Generated labels are not user-readable, they are only useful for admin and accounting purposes.

get_unconfirmed_balance()
Get the balance of this incoming transactions balance.

TODO: Move to its own subclass

TODO: Denormalize unconfirmed balances for faster look up?

Returns Decimal

Address

class cryptoassets.core.models.**GenericAddress**(***kwargs*)
The base class for cryptocurrency addresses.

The address can represent a

- Receiving address in our system. In this case we have **account** set to non-NULL.
- External address outside our system. In this **account** is set to NULL. This address has been referred in outgoing broadcast (XXX: subject to change)

We can know about receiving addresses which are addresses without our system where somebody can deposit cryptocurrency. We also know about outgoing addresses where somebody has sent cryptocurrency from our system. For outgoing addresses `wallet` reference is null.

> **Warning:** Some backends (block.io) enforce that receiving address labels must be unique across the system. Other's don't. Just bear this in mind when creating address labels. E.g. suffix them with a timetamp to make them more unique.

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

id = Column(None, Integer(), table=None, primary_key=True, nullable=False)
Running counter used in foreign key references

address = Column(None, String(length=127), table=None, nullable=False)
The string presenting the address label in the network

label = Column(None, String(length=255), table=None)
Human-readable label for this address. User for the transaction history listing of the user.

balance = Column(None, Numeric(precision=21, scale=8), table=None, nullable=False, default=ColumnDefault(0))
Received balance of this address. Only *confirmed* deposits count, filtered by GenericConfirmationTransaction.confirmations. For getting other balances, check `get_balance_by_confirmations()`. NOTE: Numeric Accuracy checked for Bitcoin only ATM

archived_at = Column(None, DateTime(), table=None)
Archived addresses are no longer in active incoming transaction polling and may not appear in the user wallet list

get_received_transactions(*external=True*, *internal=True*)
Get all transactions this address have received, both internal and external deposits.

get_balance_by_confirmations(*confirmations=0*, *include_internal=True*)
Calculates address's received balance of all arrived incoming transactions where confirmation count threshold is met.

By default confirmations is zero, so we get unconfirmed balance.

Note: This is all time received balance, not balance left after spending.

TODO: Move to its own subclass

Parameters **confirmations** – Confirmation count as threshold

Transaction

class cryptoassets.core.models.**GenericTransaction**(***kwargs*)
A transaction between accounts, incoming transaction or outgoing transaction.

Transactions can be classified as following:

- Deposit: Incoming, external, transaction from cryptocurrency network.
  - Has `network_transaction` set.
  - Has `receiving_account` set.
  - No `sending_account`
- Broadcast: Outgoign, external, transaction to cryptocurrency network.
  - Has `network_transaction` set.
  - Has `receiving_account` set.
  - No `receiving_account`
- Internal transactions
  - Which are not visible outside our system.
  - have both `sending_account` and `receiving_account` set.
  - `network_transaction` is null
  - Internal transactions can be further classified as: `internal` (normal between accounts), `balance_import` (initial wallet import to system) and `network_fee` (fees accounted to the network fee account when transaction was broadcasted)

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

id = Column(None, Integer(), table=None, primary_key=True, nullable=False)
Running counter used in foreign key references

created_at = Column(None, DateTime(), table=None, default=ColumnDefault(<function ColumnDefault._maybe_wra
When this transaction become visible in our database

credited_at = Column(None, DateTime(), table=None)
When the incoming transaction was credited on the account. For internal transactions it is instantly. For external transactions this is when the confirmation threshold is exceeded.

processed_at = Column(None, DateTime(), table=None)
When this transaction was processed by the application. For outgoing transactions this is the broadcasting time. For incoming transactions, your application may call `mark_as_processed` to mark it has handled the transaction.

amount = Column(None, Numeric(precision=21, scale=8), table=None)
Amount in the cryptocurrency minimum unit Note: Accuracy checked for Bitcoin only

state = Column(None, Enum('pending', 'broadcasted', 'incoming', 'processed', 'internal', 'network_fee', 'balance_imp
Different states this transaction can be

pending: outgoing transaction waiting for the broadcast

broadcasted: outgoing transaction has been sent to the network

incoming: we see the transaction incoming to our system, but the confirmation threshold is not exceeded yet **processed**: the application marked this transaction as handled and cryptoassets.core stops trying to notify your application about the transaction

internal: This transaction was between the accounts within one of our wallets

network_fee: When the transaction has been broadcasted, we create an internal transaction to account the occured network fees

label = Column(None, String(length=255), table=None)
Human readable label what this transaction is all about. Must be unique for each account

can_be_confirmed()
Return if the transaction can be considered as final.

txid
Return txid of associated network transaction (if any).

Shortcut for `self.network_transaction.txid`.

NetworkTransaction

class cryptoassets.core.models.**GenericNetworkTransaction**(***kwargs*)
A transaction in cryptocurrency networkwhich is concern of our system.

External transactions can be classified as

- Deposits: incoming transactions to our receiving addresses
- Broadcasts: we are sending out currency to the network

If our intenal transaction (`cryptoassets.core.models.Transaction`) has associated network transaction, it's `transaction.network_transaction` reference is set. Otherwise transactions are internal transactions and not visible in blockchain.

Note: NetworkTransaction does not have reference to wallet. One network transaction may contain transfers to many wallets.

Handling incoming deposit transactions

For more information see *cryptoassets.core.backend.transactionupdater* and *cryptoassets.core.tools.confirmationupdate*.

Broadcasting outgoing transactions

Broadcast constructs an network transaction and bundles any number of outgoing pending transactions to it. During the broadcast, one can freely bundle transactions together to lower the network fees, or mix transactions for additional privacy.

Broadcasts are constructed by Cryptoassets helper service which will periodically scan for outgoing transactions and construct broadcasts of them. After constructing, broadcasting is attempted. If the backend, for a reason or another, fails to make a broadcast then this broadcast is marked as open and must be manually vetted to succeeded or failed.

For more information see *cryptoassets.core.tools.broadcast*.

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

id = Column(None, Integer(), table=None, primary_key=True, nullable=False)
Running counter used in foreign key references

created_at = Column(None, DateTime(), table=None, default=ColumnDefault(<function ColumnDefault._maybe_wra
When this transaction become visible in our database

txid = Column(None, String(length=255), table=None)
Network transaction has associated with this transaction. E.g. Bitcoin transaction hash.

transaction_type = Column(None, Enum('deposit', 'broadcast', name='network_transaction_type'), table=None, nu
Is this transaction incoming or outgoing from our system

opened_at = Column(None, DateTime(), table=None)
When broadcast was marked as outgoing

closed_at = Column(None, DateTime(), table=None)
When broadcast was marked as sent

classmethod get_or_create_deposit(*session*, *txid*)
Get a hold of incoming transaction.

Returns tuple(Instance of *cryptoassets.core.models.GenericNetworkTransaction*., bool created)

class cryptoassets.core.models.**GenericConfirmationNetworkTransaction**(***kwargs*)
Mined transaction which receives "confirmations" from miners in blockchain.

This is a subtype of `GenericNetworkTransaction` with confirmation counting abilities.

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

confirmations = Column(None, Integer(), table=None, nullable=False, default=ColumnDefault(-1))
How many miner confirmations this tx has received. The value is `-1` until the transaction is succesfully broadcasted, after which is it `0`

confirmation_count = 3
How many confirmations to wait until the transaction is set as confirmed. TODO: Make this configurable.

can_be_confirmed()
Does this transaction have enough confirmations it could be confirmed by our standards.

Wallet

class cryptoassets.core.models.**GenericWallet**
A generic wallet implemetation.

Inside the wallet there is a number of accounts.

We support internal transaction between the accounts of the same wallet as off-chain transactions. If you call `send()``for the address which is managed by the same wallet, an internal transaction is created by ``send_internal()`.

id = Column(None, Integer(), table=None, primary_key=True, nullable=False)
Running counter used in foreign key references

name = Column(None, String(length=255), table=None)
The human-readable name for this wallet. Only used for debugging purposes.

created_at = Column(None, Date(), table=None, default=ColumnDefault(<function ColumnDefault._maybe_wrap_cal
When this wallet was created

updated_at = Column(None, Date(), table=None, onupdate=ColumnDefault(<function ColumnDefault._maybe_wrap_c
Last time when the balance was updated or new receiving address created.

balance = Column(None, Numeric(precision=21, scale=8), table=None)
The total balance of this wallet in the minimum unit of cryptocurrency NOTE: accuracy checked for Bitcoin only

classmethod get_by_id(*session*, *wallet_id*)
Returns an existing wallet instance by its id.

Returns Wallet instance

classmethod get_or_create_by_name(*name*, *session*)
Returns a new or existing instance of a named wallet.

Returns Wallet instance

create_account(*name*)
Create a new account inside this wallet.

Returns GenericAccout object

get_or_create_network_fee_account()
Lazily create the special account where we account all network fees.

This is for internal bookkeeping only. These fees MAY be charged from the users doing the actual transaction, but it must be solved on the application level.

create_receiving_address(*account*, *label=None*, *automatic_label=False*)
Creates a new receiving address.

All incoming transactions on this address are put on the given account.

The notifications for transctions to the address might not be immediately available after the address creation depending on the backend. For example, with block.io you need to wait some seconds before it is safe to send anything to the address if you wish to receive the wallet notification.

Parameters

- **account** – GenericAccount object
- **label** – Label for this address - must be human-readable

Returns GenericAddress object

get_or_create_external_address(*address*)

Create an accounting entry for an address which is outside our system.

When we send out external transactions, they go to these address entries. These addresses do not have wallet or account connected to our system.

Parameters **address** – Address as a string

send(*from_account*, *receiving_address*, *amount*, *label*, *force_external=False*, *testnet=False*)

Send the amount of cryptocurrency to the target address.

If the address is hosted in the same wallet do the internal send with *cryptoassets.core.models.GenericWallet.send_internal()*, otherwise go through the public blockchain with *cryptoassets.core.models.GenericWallet.send_external()*.

Parameters

- **from_account** – The account owner from whose balance we
- **receiving_address** – Receiving address as a string
- **amount** – Instance of *Decimal*
- **label** – Recorded text to the sending wallet
- **testnet** – Assume the address is testnet address. Currently not used, but might affect address validation in the future.
- **force_external** – Set to true to force the transaction go through the network even if the target address is in our system.

Returns Transaction object

add_address(*account*, *label*, *address*)

Adds an external address under this wallet, under this account.

There shouldn't be reason to call this directly, unless it is for testing purposes.

Parameters

- **account** – Account instance
- **address** – Address instance

get_account_by_address(*address*)

Check if a particular address belongs to receiving address of this wallet and return its account.

This does not consider bitcoin change addresses and such.

Returns Account instance or None if the wallet doesn't know about the address

get_pending_outgoing_transactions()

Get the list of outgoing transactions which have not been associated with any broadcast yet.

get_receiving_addresses(*archived=False*)
Get all receiving addresses for this wallet.

This is mostly used by the backend to get the list of receiving addresses to monitor for incoming transactions on the startup.

Parameters **expired** – Include expired addresses

get_deposit_transactions()
Get all deposit transactions to this wallet.

These are external incoming transactions, both unconfirmed and confirmed.

Returns SQLAlchemy query of Transaction model

get_active_external_received_transcations()
Return unconfirmed transactions which are still pending the network confirmations to be credited.

Returns SQLAlchemy query of Transaction model

refresh_account_balance(*account*)
Refresh the balance for one account.

If you have imported any addresses, this will recalculate balances from the backend.

TODO: This method will be replaced with wallet import.

TODO: This screws ups bookkeeping, so DON'T call this on production. It doesn't write fixing entries yet.

Parameters **account** – GenericAccount instance

send_internal(*from_account*, *to_account*, *amount*, *label*, *allow_negative_balance=False*)
Tranfer currency internally between the accounts of this wallet.

Parameters

- **from_account** – GenericAccount
- **to_account** – GenericAccount
- **amount** – The amount to transfer in wallet book keeping unit

send_external(*from_account*, *to_address*, *amount*, *label*, *testnet=False*)
Create a new external transaction and put it to the transaction queue.

When you send cryptocurrency out from the wallet, the transaction is put to the outgoing queue. Only after you broadcast has been performed (*cryptoassets.core.tools.broadcast*) the transaction is send out to the network. This is to guarantee the system responsiveness and fault-tolerance, so that outgoing transactions are created even if we have temporarily lost the connection with the cryptocurrency network. Broadcasting is usually handled by *cryptoassets helper service*.

Parameters

- **from_account** – Instance of *cryptoassets.core.models.GenericAccount*
- **to_address** – Address as a string
- **amount** – Instance of *Decimal*
- **label** – Recorded to the sending wallet history
- **testnet** – to_address is a testnet address

Returns Instance of *cryptoassets.core.models.GenericTransaction*

charge_network_fees(*broadcast*, *fee*)
Account network fees due to transaction broadcast.

By default this creates a new accounting entry on a special account (*GenericAccount.NETWORK_FEE_ACCOUNT*) where the network fees are put.

Parameters

- **txs** – Internal transactions participating in send
- **txid** – External transaction id
- **fee** – Fee as the integer

refresh_total_balance()
Make the balance to match with the actual backend.

This is only useful for send_external() balance checks. Actual address balances will be out of sync after calling this (if the balance is incorrect).

deposit(*ntx*, *address*, *amount*, *extra=None*)
Informs the wallet updates regarding external incoming transction.

This method should be called by the coin backend only.

Write the transaction to the database. Notify the application of the new transaction status. Wait for the application to mark the transaction as processed.

Note that we may receive the transaction many times with different confirmation counts.

Parameters

- **ntx** – Associated `cryptoassets.core.models.NetworkTransaction`
- **address** – Address as a string
- **amount** – Int, as the basic currency unit
- **extra** – Extra variables to set on the transaction object as a dictionary. (Currently not used)

Returns tuple (Account instance, new or existing Transaction object, credited boolean)

mark_transaction_processed(*transaction_id*)
Mark that the transaction was processed by the client application.

This will stop retrying to post the transaction to the application.

8.1.5 Validation

Coin models support pluggable address validators.

We provide some validators just to make sure we don't write bad outgoing transactions to our database.

class `cryptoassets.core.coin.validate.`**AddressValidator**
Define address validation interface.

You should not call this directly, instead use *`cryptoassets.core.coin.registry.Coin.validate_address()`*.

validate_address(*address*, *testnet*)

Parameters

- **address** – Address as a string

- **testnet** – We are in testnet

Returns True if the address is valid

class cryptoassets.core.coin.validate.**HashAddresValidator**
Check that hash in the address is good.

Does not do extensive checks like address type, etc. one could do with pycoin.

http://rosettacode.org/wiki/Bitcoin/address_validation

class cryptoassets.core.coin.validate.**NetworkCodeAddressValidator**(*netcode*, *testnetcode*)
Check if Bitcoin style address is valid using pycoin library.

XXX: Issues, could not get working.

8.2 Configuration and setup

8.2.1 Introduction

APIs for configuring *cryptoassets.core*.

8.2.2 Config coding conventions

- All config variables use underscore notitation (e.g. `status_server` instead of `status-server`) to be consistent with Python code

8.2.3 Cryptoassets Application

Cryptoassets application manager.

class cryptoassets.core.app.**Subsystem**
Enumerator for available cryptoassets library subsystems.

Depending on your application entry point and user case, you might not want to initialize all features of cryptoassets framework within your Python application. For example, multiple web server processes cannot initialize status server each, but this functinonality is purposed for the daemon applications.

database = None
Initialize database connections

status_server = None
Open HTTP status server running

backend = None
Try to connect to backend APIs

broadcast = None
Start processes and threads for broadcasting outgoing transactions

incoming_transactions = None
Start processes and threads for walletnotify hooks

event_handler_registry = None
Post notifications

class cryptoassets.core.app.**CryptoAssetsApp**(*subsystems=[<Subsystem.database: 1>, <Subsystem.backend: 3>]*)
This class ties all strings together to make a runnable cryptoassets app.

Initialize a cryptoassets framework.

Parameters **subsystems** – Give the list of subsystems you want to initialize. Because the same configuration file can be used by e.g. both web server and command line application, and config file parts like status server are only relevant in the context of the command line application, this can tell the cryptoassets framework how to set up itself. By default it initializes all the subsystems.

engine = None
SQLAlchemy database used engine

coins = None
cryptoassets.core.coin.registry.CoinRegistry instance

event_handler_registry = None
Dict of notify handlers

status_server = None
Configured status server See notes in cryptoassets.core.service.main.Service

transaction_retries = None
The number of attempts we try to replay conflicted transactions. Set by configuration.

conflict_resolver = None
cryptoassets.core.utils.conflictresolver.ConflictResolver instance we use to resolve database conflicts

is_enabled(*subsystem*)
Are we running with a specific subsystem enabled.

setup_session(*transaction_retries=3*)
Configure SQLAlchemy models and transaction conflict resolutoin.

Also, bind created cryptocurrency models to their configured backends.

open_session()
Get new read-write session for the database.

open_readonly_session()
Get new read-only access to database.

This session can never write to db, so db can ignore transactions and optimize for speed.

TODO

create_tables()
Create database tables.

Usually call only once when settings up the production database, or every time unit test case runs.

clear_tables()
Delete all data in the database, but leaving tables intact.

Useful to get clean state in unit testing.

> **Warning:** No questions asked. Don't dare to call outside testing or your data is really gone.

8.2.4 Configuration API

Configuring cryptoassets.core for your project.

Setup SQLAlchemy, backends, etc. based on individual dictionaries or YAML syntax configuration file.

`cryptoassets.core.configure.`**`logger`** = None
XXX: logger cannot be used in this module due to order of logger initialization?

exception `cryptoassets.core.configure.`**`ConfigurationError`**
ConfigurationError is thrown when the Configurator thinks somethink cannot make sense with the config data.

class `cryptoassets.core.configure.`**`Configurator`**(*app*, *service=None*)
Read configuration data and set up Cryptoassets library.

Reads Python or YAML format config data and then setss `cryptoassets.core.app.CryptoassetsApp` up and running accordingly.

Parameters

- **`app`** – `cryptoassets.core.app.CryptoassetsApp` instance
- **`service`** – *`cryptoassets.core.service.main.Service`* instance (optional)

`config` = None
Store full parsed configuration as Python dict for later consumption

`setup_engine`(*configuration*)
Setup database engine.

See `sqlalchemy.engine_from_config` for details.

TODO: Move engine to its own module?

Parameters **`configuration`** (*dict*) – `engine` configuration section

`setup_backend`(*coin*, *data*)
Setup backends.

Parameters **`data`** – dictionary of backend configuration entries

`setup_model`(*module*)
Setup SQLAlchemy models.

Parameters **`module`** – Python module defining SQLAlchemy models for a cryptocurrency

Returns *`cryptoassets.core.coin.registry.CoinModelDescription`* instance

`setup_event_handlers`(*event_handler_registry*)
Read notification settings.

Example notifier format:

```
{
    "shell": {
        "class": "cryptoassets.core.event_handler_registry.shell.ShellNotifier",
        "script": "/usr/bin/local/new-payment.sh"
```

```
            }
        }
```

setup_status_server(*config*)
Prepare status server instance for the cryptoassets helper service.

setup_service(*config*)
Configure cryptoassets service helper process.

load_from_dict(*config*)
Load configuration from Python dictionary.

Populates `app` with instances required to run `cryptocurrency.core` framework.

classmethod setup_service_logging(*config*)
Setup Python loggers for the helper service process.

Parameters **config** – service -> logging configure section.

classmethod setup_startup(*config*)
Service helper process specific setup when launched from command line.

Reads configuration `service` section, ATM only interested in `logging` subsection.

This is run before the actual Cryptoassets application initialization. We need logging initialized beforehand so that we can print out nice `$VERSIONNUMBER is starting` message.

static prepare_yaml_file(*fname*)
Extract config dictionary from a YAML file.

load_yaml_file(*fname*, *overrides={}*)
Load config from a YAML file.

Parameters

- **fname** – Path to the YAML file
- **overrides** – Python nested dicts for specific setting overrides

8.3 Events

8.3.1 Introduction

cryptoassets.core fires events which your application may listen. Most interesting once are:

- New incoming transaction
- New confirmations for incoming transactions

- New confirmations for outgoing transactions

cryptoassets.core will also post more complex events in the future (cold wallet top ups, etc.).

Also see event handling configuration.

8.3.2 Events types

cryptoassets.core currently sends the following events.

Note: In the future the scope of the events will be expanded: cold wallet top ups, network issues, etc.

`cryptoassets.core.event.events.`**`txupdate`**(*coin_name*, *transaction*, *network_transaction*, *transaction_type*, *txid*, *account*, *address*, *amount*, *credited*, ***extra*)

txupdate event reports the confirmation changes of incoming transaction (deposit) or outgoing transaction (broadcasted).

This event is fired for each transaction, when its `confirmations` changes. One network transaction may contain several deposit or broadcast transactions and they all trigger the event.

When the incoming transaction is first seen in the network, but it is not yet confirmed, confirmations is 0. Evaluate the risk of double spending for these kind of transactions in your application context.

Parameters

- **`coin_name`** – Lowercase acronym name for this asset
- **`transaction`** – Id of `cryptoasset.core.models.GenericTransaction` instance
- **`network_transaction`** – Id of `cryptoasset.core.models.GenericNetworkTransaction` instance
- **`transaction_type`** – String `deposit` (incoming) or `broadcast` (outgoing)
- **`txid`** – Network transaction id (transaction hash) as a string
- **`account`** – Database account id as int, either receiving account (deposit) or sending account (broadcast)
- **`amount`** – How much the transaction is worth of, as Decimal
- **`credited`** – Has this transaction reaches system-set confirmation threshold
- **`extra`** – Any cryptoasset specific data as dict, e.g. `dict(confirmations=0)` in the case of mined coins

Returns Event data as dict()

8.3.3 Event handlers

Event handlers tell how *cryptoassets.core* will post the event to your application.

cryptoassets.core offers a framework how you can flexbile queue notifications for your application, regardless of API service or cryptocurrency you work on.

- If you want to your web server process handle events, configure HTTP webhook
- If you want to run event handling code inside *cryptoasset helper service*, use in-process Python notifications

HTTP webhook

Send events to your application as HTTP POST request.

The HTTP POST contains two fields, `event_name` (string) and `data` (JSON).

Decimals are converted to strings for serialization.

Configuration options

param class Always `cryptoassets.core.event.http.HTTPEventHandler`.

param url Do a HTTP POST to this URL on a new event. Example: `http://localhost:30000`.

In-process Python

In-process Python event handling.

Run Python function each time event occures. This assumes you have your Python application code in the same virtualenv as *cryptoassets.core* is. The code is executed directly within cryptoassets helper service process.

Configuration options

param class Always `cryptoassets.core.event.python.InProcessEventHandler`.

param callback A dotted name to Python callback function fn(event_name, data) which will be called upon a notification. `event_name` is a string, `data` is a dict.

Shell script

Run a script on a notification.

Execute an UNIX command on a new event.

Blocks the execution until the executed command returns.

The following environment variables are set for the script:

```
CRYPTOASSETS_EVENT_NAME="event name as a string"
CRYPTOASSETS_EVENT_DATA="JSON encoded data"
```

If the executed command returns non-zero status, this notification handler raises `ShellNotificationFailed`.

Configuration options

param class Always `cryptoassets.core.event.script.ScriptEventHandler`.

param script Executed shell command

param log_output If true send the output from the executed command to cryptoassets logs on INFO log level

exception `cryptoassets.core.event.script.`**`ScriptNotificationFailed`**
Script executed for the notification returned non-zero exit code.

8.3.4 Incoming transaction confirmation updates

Handling incoming cryptoasset transactions is as not as straightforward as one would hope, especially with limited APIs provided with *bitcoind* and its derivates. Incoming transaction event chain for *bitcoind* goes as following:

For 0 confirmations and 1 confirmations

 # Receive raw cryptocurrency protocol packet

- Read transaction from the network

 # API service notification / bitcoind walletnotify shell hook

- Push out notification about the updated transaction status

 # Cryptoassets helper service (`cryptoassets-helper`)

- Catch the low level transaction update notification (via named pipe, HTTP hook)
- Write updated transaction information to the database
- Update account balances, etc.
- Call all generic cryptoasets notification handlers with `txupdate` event

 # Your application

- Listen for `txupdate` event
- Process updated transaction

For 2 and more confirmations

 # Cryptoassets helper service (`cryptoassets-helper`)

- Run periodical open transaction update task - `cryptoassets.core.tools.opentransactions`
- Poll the *bitcond* for transactions where the confirmation count in the database has not reached the maximum threshold yet. This is 15 confirmations by default.
- **If the transaction confirmation count has changed in the backend.**
  - Update account balances, etc.
  - Call all generic cryptoasets notification handlers

For 15 and more confirmations

- These transactions are not polled anymore in the backend and are considered final.
- The threshold can be adjusted in backend settings.

8.4 Tools and asset management

8.4.1 Transaction confirmation count updates

Network transactions are considered open as long as the confirmation threshold has not been reached.

Because backends do not actively report the progress of confirmation status, we poll the backend for all network transactions (deposits, broadcasts) until the confirmation threshold has been reached. For example, *bitcoind* gives you a walletnotify only for 0 and 1 confirmations. *block.io* does not have any confirmation hooks, but you can subcribe to *chain.so* real-time API to receive 0 confirmations notificatoin to an address.

cryptoassets.core.tools.confirmationupdate.update_confirmations() polls the backend. It will scan all transactions where confirmation threshold has not been reached and then ask the backend of more transaction details. Eventually all open incoming transactions exceed the confirmation threshold and we can stop polling them.

The poller is set up in *cryptoassets.core.service.main.Service*.

More information about walletnotify behavior

- http://bitcoin.stackexchange.com/a/24483/5464

cryptoassets.core.tools.confirmationupdate.**get_open_network_transactions**(*session*, *Network-Transaction*, *confirmation_threshold*)

Get list of transaction_type, txid of transactions we need to check.

cryptoassets.core.tools.confirmationupdate.**update_confirmations**(*transaction_updater*, *confirmation_threshold*)

Periodically rescan all open transactions for one particular cryptocurrency.

We try to keep transaction conflicts in minimum by not batching too many backend operations per each database session.

Parameters

- **confirmation_treshold** – Rescan the transaction if it has less confirmations than this
- **transaction_updater** – *cryptoassets.core.backend.transactionupdater.Transaction* instance

Returns Number of txupdate events fired

8.4.2 Broadcasting outgoing transactions

Broadcast outgoing transactions.

Broadcaster is responsible for the following things

- Check that there hasn't been any interrupted broadcats before
- Make sure there can be one and only one attempt to broadcast at any moment - so we don't have double broadcast problems
- Scan database for outgoing external transactions
- Merge and allocate these transactions to outgoing broadcasts
- If there are any unbroadcasted broadcasts, mark them scheduled for broadcast and attempt to broadcast them

class cryptoassets.core.tools.broadcast.**Broadcaster**(*wallet*, *conflict_resolver*, *backend*)

Create and send transactions to the cryptoasset networks.

get_wallet(*session*)

Get a wallet instance within db transaction.

get_broadcast(*session*, *broadcast_id*)
Get a wallet instance within db transaction.

collect_for_broadcast()

Returns Number of outgoing transactions collected for a broadcast

check_interrupted_broadcasts()
Check that there aren't any broadcasts which where opened, but never closed.

Returns List Open broadcast ids or empty list if all good

send_broadcasts()
Pick up any unbroadcasted broadcasts and attempt to send them.

Carefully do broadcasts within managed transactions, so that if something goes wrong we have a clear audit trail where it failed. Then one can manually check the blockchain if our transaction got there and close the broadcast by hand.

Returns tuple (broadcasted network transaction count, total charged network fees)

do_broadcasts()
Collect new outgoing transactions for a broadcast and send out all existing and new outgoing transactions.

8.4.3 Rescan receiving addresses

Scan all receiving addresses to see if we have missed any incoming transactions.

8.4.4 Importing existing wallet

Import existing wallet balance to the accounting.

If you have a wallet in some service and you wish to use it with *cryptoassets.core*, you need to tell *cryptoassets.core* what to do with the existing balance in the wallet, from the time before the wallet was managed by *cryptoassets.core*.

This is especially useful for testing. To run unit tests you need to have some cryptocurrency balance somewhere. You have known backends which you configure the unit tests to connect to. These backends have default wallets and there is some balance on these wallets, so unit tests can perform withdraw tests.

cryptoassets.core.tools.walletimport.**create_import_transaction**(*amount*, *account*)
Put wallet extra coins, for which we do not know the owner, on a specific account.

Execute inside transaction manager.

Parameters

- **amount** (*Decimal*) – How many extra coins to account
- **account** – Account instance where to put coins

cryptoassets.core.tools.walletimport.**import_unaccounted_balance**(*backend*, *wallet*, *account*)
Creates a new transaction which will put all assets in the wallet on a new account.

8.5 Cryptoassets helper service

8.5.1 Service main

Cryptoassets helper service is a standalone process managing cryptoasset backend connections and transaction updates.

Manages asynchronous tasks for sending and receiving cryptocurrency over various APIs. This includes

- Broadcasting transactions to the cryptocurrency network asynchronously
- Handle incoming transactions and write them to the database, calls your application via event handlers
- Updates confirmation counts of open transactions

`cryptoassets.core.service.main.`**`logger`** **= None**
Must be instiated after the logging configure is passed in

`cryptoassets.core.service.main.`**`splash_version`**`()`
Log out cryptoassets.core package version.

class `cryptoassets.core.service.main.`**`Service`**(*config, subsystems=[<Subsystem.database: 1>, <Subsystem.backend: 3>], daemon=False, logging=True*)
Main cryptoassets helper service.

This class runs *cryptoassets helper service* process itself and various command line utilities (*initialize-database*, etc.)

We uses Advanced Python Scheduler to run timed jobs (broadcasts, confirmatino updates).

Status server (`cryptoassets.core.service.status`) can be started for inspecting our backend connections are running well.

Parameters

- **`config`** – cryptoassets configuration dictionary
- **`subsystems`** – List of subsystems needed to initialize for this process
- **`daemon`** – Run as a service

`status_server` **= None**
Status server instance

`incoming_transaction_runnables` **= None**
coin name -> IncomingTransactionRunnable

`broadcast_period` **= None**
How often we check out for outgoing transactions

`config`(*config*, *logging_*)
Load configuration from Python dict.

Initialize logging system if necessary.

`setup``()`
Start background threads and such.

`setup_session``()`
Setup database sessions and conflict resolution.

initialize_db()

start_status_server()
Start the status server on HTTP.

The server is previously set up by `configure` module.We need just to pass the status report generator of this service to it before starting it up.

setup_incoming_notifications()
Start incoming transaction handlers.

setup_sigterm()
Capture SIGTERM and shutdown on it.

poll_broadcast()
"A scheduled task to broadcast any new transactions to the bitcoin network.

Each wallet is broadcasted in its own transaction.

poll_network_transaction_confirmations()
Scan incoming open transactions.

Returns Number of rescans attempted

scan_received()
Scan through all received transactions, see if we missed some through walletnotify.

start()
Start cryptoassets helper service.

Keep running until we get SIGTERM or CTRL+C.

Returns Process exit code

run_thread_monitor()
Run thread monitor until terminated by SIGTERM.

check_threads()
Check all the critical threads are running and do shutdown if any of the threads has died unexpectly.

Returns True if all threads stil alive

shutdown(*unclean=False*)
Shutdown the service process.

Parameters **unclean** – True if we terminate due to exception

8.6 Backends

8.6.1 Introduction

Backends are responsible to interact with cryptoassets network themselves. *cryptoassets.core* provides abstraction layer, so that you can separate your own application database and code from the underlying cryptoasset network API.

8.6.2 Backend base class

Base classes for cryptocurrency backend.

class `cryptoassets.core.backend.base.`**`CoinBackend`**

Cryptocurrency management backend.

Provide necessecities for low-level cryptocurrency usage, like creating wallets, addresses, sending and receiving the currency.

Manage communications with the cryptocurrency network. The commucications can be done either by API service (block.io, blockchain.info) or raw protocol daemon (bitcoind).

The accounting amounts are in the integer amounts defined by the datbase models, e.g. satoshis for Bitcoin. If the backend supplies amounts in different unit, they most be converted forth and back by the backend. For the example, see *cryptoassets.core.backend.blockio*.

`max_tracked_incoming_confirmations` = None

If `track_incoming_confirmations` is set to true, this is how many confirmations we track for each incoming transactions until we consider it "closed". Please note that this is API will most likely be changed in the future and this variable move to somewhere else. The variable is set by `Configurator.setup_backend`.

`require_tracking_incoming_confirmations`()

Does this backend need to have some help to get incoming transaction confirmations tracked.

Some daemons and walletnotify methods, namely bitcoind, only notify us back the first occurence of an incoming transactions. If we want to receive further confirmations from the transaction, we need to manually poll the transactions where our confirmation threshold is not yet met.

Set this to true and the cryptoassets helper service will start a background job (*cryptoassets.core.tools.confirmationupdate* to keep receiving updates about the confirmations).

Returns True or False

`create_address`(*label*)

Create a new receiving address.

`get_balances`(*addresses*)

Get balances on multiple addresses.

Return the address balance in the native format (backend converts to satoshis, etc.)

Yield (address, balance) tuples

`send`(*recipients*)

Broadcast outgoing transaction.

This is called by send/receive process.

Parameters **`recipients`** – Dict of (address, internal amount)

`get_backend_balance`()

Get full available hot wallet balance on the backend.

May take backend-specific optional kwargs like `confirmations`.

This is used for *cryptoassets.core.tools.walletimport*.

Returns Decimal

`list_received_transactions`(*extra*)

List all received transactions the backend is aware off.

Parameters **extra** – Dict of backend-specific optional arguments like `dict(confirmations=0)`.

Returns Instance of *cryptoassets.core.backend.base.ListTransactionsIterator*.

create_transaction_updater(*conflict_resolver*, *event_handler_registry*)
Create transaction updater to handle database writes with this backend.

Creates *cryptoassets.core.backend.transactionupdater.TransactionUpdater* instance. This TransactionUpdater is bound to this backend and provides safe APIs for doing broadcast and deposit updates.

setup_incoming_transactions(*conflict_resolver*, *event_handler_registry*)
Configure the incoming transaction notifies from backend.

The configuration for wallet notifies have been given to the backend earlier in the backend constructor. Now we read this configure, resolve the walletnotify handler class and instiate it.

We'll hook into backend by creating `cryptoassets.core.backend.transactionupdater.TransactionUpd` instance, which gets the list of event_handler_registry it needs to call on upcoming transaction.

Parameters

- **conflict_resolver** – cryptoassets.core.utils.conflictresolver.ConflictResolver instance which is used to manage transactions
- **event_handler_registry** –

  param event_handler_registry :py:class‘cryptoassets.core.event.registry.EventHandlerRegistry‘ instance or None if we don't want to notify of new transactions and just update the database

Returns Instance of *cryptoassets.core.backend.base.IncomingTransactionRunnable*

class cryptoassets.core.backend.base.**ListTransactionsIterator**(*backend*)
Helper to iterate all transactions in the backend.

Because different backends iterate to different directions, we abstract this away.

Note: bitcoind iterates from index 0 with different batch sizes. block.io iterates from the latest transcation with fixed batch size of 100 and needs before txid parameter for the next batch.

---

fetch_next_txids()
Get next batch of transactions.

txdata must be dict bitcoind-like format:

```
{
    confirmations: 0,
    txid: "xxx",
    "details": {
        "category": "received",
        "amount": Decimal(1),
        "address": "foobar"
    }
}
```

Returns List of next (txid, txdata) paits to iterate or empty list if iterating is done.

class cryptoassets.core.backend.base.**IncomingTransactionRunnable**
Backend specific thread/process taking care of accepting incoming transaction notifications from the network.

register_new_addresses()
Backend has created new addresses and the incoming transcation monitor must know about them.

Some monitoring systems need to refresh after new addresses have been added to the pool.

8.6.3 Transaction updater

class cryptoassets.core.backend.transactionupdater.**TransactionUpdater**(*conflict_resolver*, *backend*, *coin*, *event_handler_registry*)
TransactionUpdater write transactions updates from API/backend to the database.

TransactionUpdater uses *cryptoassets.core.utils.conflictresolver.ConflictResolver* database transaction helper when updating transactions. This gives us guarantees that updates having possible db transaction conflicts are gracefully handled.

The backend has hooked up some kind of wallet notify handler. The wallet notify handler uses TransactionUpdater to write updates of incoming transactoins to the database.

TransactionUpdater is also responsible to fire any notification handlers to signal the cryptoassets client application to handle new transactions.

TransactionUpdater is generally run inside cryptoassets.core.service.main.Server process, as this process is responsible for all incoming transaction updates. No web or other front end should try to make their own updates.

Parameters

- **conflict_resolver** – *cryptoassets.core.utils.conflictresolver.ConflictResolver*
- **backend** – cryptoasets.core.backend.base.CoinBackend instance.
  TODO: To be removed - redundant with coin.
- **coin** – cryptoasets.core.coin.registry.Coin instance
- **event_handler_registry** – :py:class‘cryptoassets.core.event.registry.EventHandlerRegistry‘ instance

last_wallet_notify = None
UTC timestamp when we got the last transaction notification

event_handler_registry = None
event_handler_registry registry we are going to inform about transaction status updates

stats = None
Diagnostics and bookkeeping statistics

verify_amount(*transaction_type*, *txdata*, *address*, *amount*)
Check that transaction amounts have not somehow changed between confirmations.

It gets tricky here because bitcoind reports its internal stuff and has negative amounts for send transactions, versus what you see in blockchain and other services is only receiving outputs. We place some temporary workaround we hope to get rid of later.

update_network_transaction_confirmations(*transaction_type*, *txid*, *txdata*)
Create or update NetworkTransaction in the database.

Ask the backend about updates for a network transaction. Any action is taken only if the confirmation count has changed since the last call.

For desposits, updates the confirmation count of inbound network deposit transaction. For all associated receiving addresses and transactions, confirmation and crediting check if performed, account balances updated and `txupdate` event fired.

For broadcasts, updates the confirmation count of outbound transactions.

Relevant event handlers are fired (`cryptoassets.core.transactionupdater.TransactionUpdater.even`

Parameters

- **`transaction_type`** – "deposit" or "broadcast". Note that we might have two ntx's for one real network transaction, as we are sending bitcoins to ourselves.
- **`txid`** – Network transaction hash
- **`txdata`** – Transaction details, as given by the backend, translated to *bitcoind* format

Returns Tuple (new or existing network transaction id, fired txupdate events as a list)

`handle_wallet_notify`(*txid*)

Handle incoming wallet notifications.

Fetch transaction info from the backend and update all receiving addresses we are managing within that transaction.

Parameters **`txid`** – Network transaction hash

8.7 Utilities

8.7.1 Introduction

Here is collection of helper classes.

8.7.2 Transaction conflict resolver

ConflictResolver is a helper class to provide serialized transaction conflict resolution mechanism in your SQLAlchemy application.

Preface

Transaction conflict resolution is a way to deal with concurrency and race condition issues within multiuser application. It is a way to resolve race conditions when two users, or two threads, are performing an action affecting the same data set simultaneously.

There are two basic ways of concurrency control

- Up-front locking: You use interprocess / interserver locks to signal you are about to access and modify resources. If there are concurrent access the actors accessing the resource wait for the lock before taking action. This is pessimistic concurrency control mechanism.
- Transaction serialization: Database detects concurrent access from different clients (a.k.a serialization anomaly) and do not let concurrent modifications to take place. Instead, only one transaction is let through and other conflicting transactions are rolled back. The strongest level of transaction isolation is achieved using SQL Serializable isolation level. This is optimistic concurrency control mechanism.

For complex systems, locking may pose scalability and complexity issues. More fine grained locking is required, placing cognitive load on the software developer to carefully think and manage all locks upfront to prevent race conditions and deadlocks. Thus, locking may be error prone approach in real world application development (TBD needs better sources).

Relying on database transaction serialization is easier from the development perspective. If you use serialized transactions you know there will never be database race conditions. In the worst case there is an user error saying there was concurrency error. But transaction serialization creates another problem: your application must be aware of potential transaction conflicts and in the case of transaction conflict it must be able to recover from them.

Please note that when system is under high load and having high concurrent issue rate, both approaches will lead to degraded performance. In pessimistic approach, clients are waiting for locks, never getting them and eventually timing out. In optimistic approach high transaction conflict rate may exceed the rate the system can successfully replay transactions. Long running transaction are also an issue in both approaches, thus batch processing is encouraged to use limited batch size for each transaction if possible.

Benefits and design goals

`cryptoassets.core.utils.conflictresolver.ConflictResolver` is a helper class to manage serialized transaction conflicts in your code and resolve them in idiomatic Python manner. The design goals include

- Race condition free codebase because there is no need for application level locking
- Easy, Pythonic, to use
- Simple
- Have fine-grained control over transaction life cycle
- Works with SQLAlchemy

These all should contribute toward cleaner, more robust and bug free, application codebase.

The work was inspired by ZODB transaction package which provides abstract two-phase commit protocol for Python. *transaction* package contains more features, works across databases, but also has more complex codebase and lacks decorator approach provided by *ConflictResolver*. Whereas ConflictResolver works directly with SQLAlchemy sessions, making it more straightforward to use in SQLAlchemy-only applications.

Transaction retries

In the core of transaction serialization approach is recovery from the transaction conflict. If you do not have any recovery mechanism, when two users edit the same item on a website and press save simultaneously, leading to a transaction conflict in the database, one of the user gets save succeed the other gets an internal error page. The core principle here is that we consider transaction conflict a rare event under normal system load conditions i.e. it is rare users press the save simultaneously. But it still very bad user experience to serve an error page for one of the users, especially if the system itself knows how it could recovery from the situation - without needing intervention from the user.

ConflictResolver approach to recovery is to

- Run a transaction sensitive code within a marked Python code block
- If the code block raises an exception which we identify to be a transaction conflict error from the database, just reset the situation and replay the code block
- Repeat this X times and give up if it seems like our transaction is never going through (because of too high system load or misdesigned long running transaction blocking all writes)

Marked Python code blocks are created using Python function decorators. This is not optimal approach in the sense of code cleanness and Python `with` block would be preferred. However, Python `with` lacks ability to run loops which is prerequisite for transaction retries. However combined with Python closures, the boilerplate is quite minimal.

Example

Here is a simple example how to use ConflictResolver:

```
from cryptoassets.core.utils.conflictresolver import ConflictResolver
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

engine = create_engine('postgresql:///unittest-conflict-resolution',
    isolation_level='SERIALIZABLE')

# Create new session for SQLAlchemy engine
def create_session():
    Session = sessionmaker()
    Session.configure(bind=engine)
    return Session()

conflict_resolver = ConflictResolver(create_session, retries=3)

# Create a decorated function which can try to re-run itself in the case of conflict
@conflict_resolver.managed_transaction
def top_up_balance(session, amount):

    # Many threads could modify this account simultanously,
    # as incrementing the value in application code is
    # not atomic
    acc = session.query(Account).get(1)
    acc.balance += amount

# Execute the conflict sensitive code inside a transaction aware code block
top_up_balance(100)
```

Rules and limitations

The rules:

- You must not blindly swallow all exceptions (generic Python `Exception`) within `managed_transactions`. Example how to handle exceptions if generic exception catching is needed:

```
# Create a decorated function which can try to re-run itself in the case of conflict
@conflict_resolver.managed_transaction
def myfunc(session):

    try:
        my_code()
    except Exception as e:
        if ConflictResolver.is_retryable_exception(e):
            # This must be passed to the function decorator, so it can attempt retry
            raise
        # Otherwise the exception is all yours
```

- Use special read-only database sessions if you know you do not need to modify the database and you need weaker transaction guarantees e.g. for displaying the total balance.
- Never do external actions, like sending emails, inside `managed_transaction`. If the database transaction is replayed, the code is run twice and you end up sending the same email twice.
- Managed transaction code block should be as small and fast as possible to avoid transaction conflict congestion. Avoid long-running transactions by splitting up big transaction to smaller worker batches.

Compatibility

ConflictResolver should be compatible with all SQL databases providing Serializable isolation level. However, because Python SQL drivers and SQLAlchemy do not standardize the way how SQL execution communicates the transaction conflict back to the application, the exception mapping code might need to be updated to handle your database driver.

API documentation

See *ConflictResolver* API documentation below.

cryptoassets.core.utils.conflictresolver.**DATABASE_COFLICT_ERRORS = [(<class 'sqlalchemy.orm.exc.Stalel**
Tuples of (Exception class, test function). Behavior copied from _retryable_errors definitions copied from zope.sqlalchemy

class cryptoassets.core.utils.conflictresolver.**ConflictResolver**(*session_factory*, *retries*)
Helper class to resolve transaction conflicts in graceful manner.

Parameters

- **session_factory** – *callback()* which will give us a new SQLAlchemy session object for each transaction and retry
- **retries** – The number of attempst we try to re-run the transaction in the case of transaction conflict.

classmethod is_retryable_exception(*e*)
Does the exception look like a database conflict error?

Check for database driver specific cases.

Parameters **e** – Python Exception instance

managed_transaction (*func*)
Function decorator for SQL Serialized transaction conflict resolution through retries.

`managed_transaction` decorator will retry to run the decorator function. Retries are attempted until `ConflictResolver.retries` is exceeded, in the case the original SQL exception is let to fall through.

Please obey the rules and limitations of transaction retries in the decorated functions.

managed_non_retryable_transaction (*func*)
Provide `managed_transactions` decorator API compatibility without retrying.

Decorate your transaction handling functions with this method if you absolute must not run the code twice for transaction retry and the user error is desirable outcome.

transaction()
Get a transaction contextmanager instance using the conflict resolver session.

This approach **does not** support conflict resolution, because Python context managers don't support looping. Instead, it will let any exception fall through. `ConflictResolver.transaction` is only useful to access the configured SQLAlchemy session in easy manner.

- Useful for unit testing
- Useful for shell sessions

Transaction handling

- Transaction is committed if the context manager exists succesfully
- Transaction is rolled back on an exception

Example:

```
conflict_resolver = ConflictResolver(create_session, retries=3)
with conflict_resolver.transaction() as session:
    account = session.query(Account).get(1)
    account.balance += 1
```

exception cryptoassets.core.utils.conflictresolver.**CannotResolveDatabaseConflict**
The managed_transaction decorator has given up trying to resolve the conflict.

We have exceeded the threshold for database conflicts. Probably long-running transactions or overload are blocking our rows in the database, so that this transaction would never succeed in error free manner. Thus, we need to tell our service user that unfortunately this time you cannot do your thing.

Conflict resolver unit tests provide tests for different transaction conflict resolution outcomes and their resolution. If you are unsure Python database driver can handle transaction conflicts, this is a good smoke test to find out.

8.7.3 Automatic enumeration classes

class cryptoassets.core.utils.enum.**AutoNumber**
Enum pattern with automatic numbering of values.

https://docs.python.org/3/library/enum.html#autonumber

8.7.4 Python dictionary deep merge

cryptoassets.core.utils.dictutil.**merge_dict**(*a*, *b*)
merges b into a and return merged result.

NOTE: tuples and arbitrary objects are not handled as it is totally ambiguous what should happen

Courtesy of http://stackoverflow.com/a/15836901/315168

8.7.5 HTTP event listener decorator

Convenience decorator to open HTTP event listever for configured cryptoassets service.

Opens a new HTTP server running a background thread. Whenever cryptoassets helper service posts a new event, it will be received by this HTTP server which then executes the event in your application context.

This can be used only once per application, so you need to dispatch listened events to your own event handling funcions in one singleton handler.

The callback receives two arguments, `event_name` (string) and `data` (dict). Data payload depends on the event type.

Example:

```
app = CryptoAssetsApp()

# This will load the configuration file for the cryptoassets framework
configurer = Configurator(app)
configurer.load_yaml_file("cryptoassets-settings.yaml")

@simple_http_event_listener(configurer.config)
def my_event_callback(event_name, data):
    if event_name == "txupdate":
        print("Got transaction update {}".format(data))
```

cryptoassets.core.utils.httpeventlistener.**simple_http_event_listener**(*config*, *daemon=True*)
Function decorator to make the target function to retrieve events from cryptoassets helper service over HTTP event callback.

You can also call this manually from command line from testing:

```
    curl --data 'event_name=txupdate&data={"transaction_type":"broadcast","address":"x","confirmatic
```

Parameters

- **config** – *cryptoassets.core* app configuration as Python dict. We'll extract the information which port and IP to listen to on HTTP server from there.
- **func** – The event handling callback function, `callback(event_name, data_dict)`.
- **daemon** – Should the server be started as a daemon thread (does not prevent Python application quitting unless explictly stopped)

8.7.6 Ngrok automatic HTTP endpoint tunneling

Expose local HTTP ports to the world using ngrok service.

Today many API services provide webhooks calling back your website or system over HTTP. This enables simple third party interprocess communications for websites. However unless you are running in production, you often find yourself in a situation where it is not possible to get an Internet exposed HTTP endpoint over publicly accessible IP address. These situations may include your home desktop, public WI-FI access point or continuous integration services. Thus, developing or testing against webhook APIs become painful for contemporary nomad developers.

ngrok (*source <https://github.com/inconshreveable/ngrok>_*) is a pay-what-you-want service to create HTTP tunnels through third party relays. What makes ngrok attractice is that the registration is dead simple with Github credentials and upfront payments are not required. ngrok is also open source, so you can run your own relay for sensitive traffic.

In this blog post, I present a Python solution how to programmatically create ngrok tunnels on-demand. This is especially useful for webhook unit tests, as you have zero configuration tunnels available anywhere where you run your code. ngrok is spawned as a controlled subprocess for a given URL. Then, you can tell your webhook service provider to use this URL to make calls back to your unit tests.

One could use ngrok completely login free. In this case you lose the ability to name your HTTP endpoints. I have found it practical to have control over the endpoint URLs, as this makes debugging much more easier.

For real-life usage, you can check cryptoassets.core project where I came up with ngrok method. ngrok succesfully tunneled me out from drone.io CI service and my laptop.

Installation

Installing ngrok on OSX from Homebrew:

```
brew install ngrok
```

Installing ngrok for Ubuntu:

```
apt-get install -y unzip
cd /tmp
wget -O ngrok.zip "https://api.equinox.io/1/Applications/ap_pJSFC5wQYkAyI0FIVwKYs9h1hW/Updates/Asset/
unzip ngrok
mv ngrok /usr/local/bin
```

Official ngrok download, self-contained zips.

Sign up for the ngrok service and grab your auth token.

Export auth token as an environment variable in your shell, don't store it in version control system:

```
export NGROK_AUTH_TOKEN=xxx
```

Ngrok tunnel code

Below is Python 3 code for `NgrokTunnel` class. See the full source code here.

Example code

Here is a short pseudo example from cryptoassets.core block.io webhook handler unit tests. See the full unit test code here.:

```
class BlockWebhookTestCase(CoinTestRoot, unittest.TestCase):

    def setUp(self):

        self.ngrok = None

        self.backend.walletnotify_config["class"] = "cryptoassets.core.backend.blockiowebhook.BlockI

        # We need ngrok tunnel for webhook notifications
        auth_token = os.environ["NGROK_AUTH_TOKEN"]
        self.ngrok = NgrokTunnel(21211, auth_token)

        # Pass dynamically generated tunnel URL to backend config
        tunnel_url = self.ngrok.start()
        self.backend.walletnotify_config["url"] = tunnel_url
        self.backend.walletnotify_config["port"] = 21211

        # Start the web server
        self.incoming_transactions_runnable = self.backend.setup_incoming_transactions(self.app.confi

        self.incoming_transactions_runnable.start()

    def teardown(self):

        # Stop webserver
        incoming_transactions_runnable = getattr(self, "incoming_transactions_runnable", None)
        if incoming_transactions_runnable:
            incoming_transactions_runnable.stop()

        # Stop tunnelling
        if self.ngrok:
            self.ngrok.stop()
            self.ngrok = None
```

Other

Please see the unit tests for `NgrokTunnel` class itself.

8.8 Cryptocurrency models

8.8.1 Introduction

List of available cryptocurrency models out of the box.

8.8.2 Bitcoin

Bitcoin database implementation.

All amounts are stored in satoshis in integer fields.

Modify `BitcoinTransaction.confirmation_count` global to set the threshold when transcations are considered confirmed.

8.8.3 Dogecoin

Dogecoin database support.

8.8.4 Litecoin

Litecoin database support.

8.8.5 AppleByte

AppleByte database support.

8.9 Unit tests

8.9.1 Introduction

Some unit tests used to verify functionality.

8.9.2 Backend tests

`cryptoassets.core.tests.base.`**`has_inet`**`()`
py.test condition for checking if we are online.

`cryptoassets.core.tests.base.`**`has_local_bitcoind`**`()`
Use this to disable some tests in CI enviroment where 15 minute deadline applies.

`cryptoassets.core.tests.base.`**`is_slow_test_hostile`**`()`
Use this to disable some tests in CI enviroment where 15 minute deadline applies.

class `cryptoassets.core.tests.base.`**`CoinTestRoot`**
Have only initialization methods for the tests.

create_engine()
Create SQLAclhemy database engine for the tests.

wait_address(*address*)
block.io needs subscription refresh every time we create a new address.

Because we do not have IPC mechanism to tell when block.io refresh is ready, we just wait few seconds for now. block.io poller should recheck the database for new addresses every second.

setup_receiving(*wallet*)
Necerssary setup to monitor incoming transactions for the backend.

teardown_receiving()
Teardown incoming transaction monitoring.

setup_coin()
Setup coin backend for this test case.

setup_balance()
Create an a wallet and an account with balance.

class cryptoassets.core.tests.base.**CoinTestCase**
Abstract base class for all cryptocurrency backend tests.

This verifies that a cryptocurrency backend works against cryptoassets.core models API.

Inherit from this test case, implement backend abstract methods and run the test case. If all test passes, the backend is compatible with *cryptoassets.core*.

test_create_address()
Creates a new wallet and fresh bitcoin address there.

test_get_receiving_addresses()
Creates a new wallet and fresh bitcoin address there.

test_create_account()
Creates a new wallet and fresh bitcoin address there.

test_send_internal()
Creates a new wallet and fresh bitcoin address there.

test_send_internal_low_balance()
Does internal transaction where balance requirement is not met.

test_send_internal_same_account()
Does internal transaction where balance requirement is not met.

test_cannot_import_existing_address()
Do not allow importing an address which already exists.

test_refresh_account_balance()
Read the external balance to an account.

test_send_external()
Send Bitcoins from external address

test_charge_network_fee()
Do an external transaction and see we account network fees correctly.

test_broadcast_no_transactions()
Broadcast must not fail even we don't have any transactions.

test_receive_external_spoofed()
Test receiving external transaction.

Don't actually receive anything, spoof the incoming transaction.

CHAPTER 9

Extending

9.1 Introduction

cryptoassets.core has extensible architecture

- You can easily include new crytocurrencies and assets
- You can choose to use any protocol backend instead of *bitcoind*
- You can override almost any part of the system with your own class or subclass

Note: Currently the architecture is heavily geared towards mined coins. This will change in the future and class hiearchy is abstracted so that traits like mining (block confirmations) go into their own class tree. Alternatively, consensus based coins (Ripple, Stellar) get their own corresponding base classes.

9.2 Adding new cryptocurrency model

Adding support for *bitcoind* derived altcoin is as easy as creating models file (for example see `cryptoassets.core.coin.applebyte.mdoels`) and givin the models module in the config file. You can use the stock *`cryptoassets.core.backend.bitcoind`* if altcoin is JSON-RPC compatible with *bitcoind* (they are).

Adding support for non-bitcoin like cryptoassets includes subclassing API classes and having corresponding backend. You can still use services like database transaction conflict resolution.

9.3 Adding new cryptocurrecy backend

Subclass *`cryptoassets.core.backend.base.CoinBackend`*.

Create a backend specific unit test which subclasses *cryptoassets.core.tests.base.CoinTestCase*. If all *CoinTestCase* tests passed, your backend is more or less feature complete and complete with *cryptoassets.core*.

9.4 Overriding parts of the framework

You can switch and replace any part of the framework. For example, you might want to optimize certain components, like *bitcoind* connections for scalability.

- Database models can be overridden with models configuration and thus you can replace any stock API method with your own.
- You can plug in your own backend.
- You can subclass `cryptoassets.core.app.CryptoassetsApp` and override initialization methods to plug-in your own code.

CHAPTER 10

Security and data integrity

10.1 Introduction

cryptoassets.core is built following defensive programming principles to mitigate developer human error, data integrity and security issues.

When dealing with financial transactions, especially ones which cannot be reversed, it is imperative that one gets its transaction handling correctly. *cryptoassets.core* provides tools and methods, so that even inexperienced developers do not shoot themselves into a foot when writing cryptoassets code.

This includes mitigation against

- Human-errors by the developers
- External attackers trying to exploit issues in the financial code

Potential issues and threads for cryptoassets services include

- Race conditions allowing over-balance withdrawals or account balance mix up (data integrity issues)
- Double transaction broadcasts doing double withdrawals from hot wallet
- Partially lost data on unclean service shutdown
- Partially lost data when having Internet connectivity issues
- Database damage with bad migration
- Improper cold wallet handling increases the risk of losing customer assets

Below is how cryptoassets.core addresses these issues.

10.2 Eliminating race conditions

The production *cryptoassets.core* always runs its database transactions on serializable transactino isolation level. Note that this is not the default for most database setups. Serializable transactions isolation level means that each transaction would happen in a complete isolation, one after each another, and thus there cannot be race conditions. If the database detects transactions touching the same data, only one of conflicting transactions may pass through and the others are aborted with application-level exception.

Serializable transaction isolation simply prevents all kind of race conditions. Alternative would be writing application level locking code, which is prone to errors, as it incumbers more cognitive overhead for the developers themselves. Better let the database developers take care of the locking, as they have liven their life by solving concurrency issues and they are expert on it.

- PostgreSQL transaction isolation levels

10.2.1 Transaction conflict handling

cryptoassets.core provides tools to handle serialized transaction rollbacks in Pythonic way.

`cryptoassets.core.utils.conflictresolver` is an utility class extensively used through *cryptoassets.core*. It's `cryptoassets.core.utils.conflictresolver.ConflictResolver.managed_transaction()` function decorator allows one easily write transaction sensitive code blocks.

10.3 Data separation

Each cryptoasset gets it own set of database tables. This sets some static-typing like limits making it less likely for a developer to accidentally mix and match wrong currencies.

Having own set of tables is future-proof path: when cryptocurrencies themselves develop and get new features, you can migrate the cryptocurrency specific tables to support these features.

10.4 Data integrity on failed broadcasts

One possible error condition is not knowing if the outgoing transaction was broadcasted. For example, you send out a transaction and the network connection to *bitcoind* dies, or server goes down, just about when *bitcoind* is about to write JSON-RPC API reply "transaction broadcasted".

When *cryptoassets.core* does outgoing transaction broadcasts, it separately commits when broadcast was started (`opened_at`) when broadcast was ended (`closed_at`). Broadcasts which never receives ending mark is considered "broken" and cryptoassets helper service never managed to write to the database whether this broadcast got out to the network or not.

For broken transactions one needs to manually check from blockchain, by matching `opened_at` timestamp and transaction amount, whether the broadcast made to blockchain before the broadcasting process failed.

- Set `txid` and `closed_at` for the transactions if they got out to blockchain
- Remove `opened_at` timestamp if the transaction never made to the blockchain and should be rebroadcasted

10.5 Missed incoming transactions

For a reason or another, *cryptoassets.core* may miss the initial wallet notification from the network for new deposit transaction arriving to the address in your application wallet. Particularly, *cryptoassets helper service* could be down when the incoming transaction was broadcasted.

cryptoassets helper service rescans all receiving addresses on start up. Thus, restarting *cryptoassets helper service* fixes the problem. Alternatively, you can manually run rescan command.

10.6 Missed transactions confirmations

For a reason or another, your application may fail to process transaction update events.

E.g.

- Event hook calling your application failed
- *Cryptoassets helper service* was down when wallet notification arrived

Cryptoassets helper service will poll all transactions where the transaction confirmation count is below a threshold value. If you miss confirmation notification *cryptoassets.core* keeps polling the transaction and resend the transaction update message to your application. When your application is satisfied with the confirmation count it can mark the transaction processed.

10.7 Choosing your database

MySQL InnoDB engine is known for various prone-to-human-error issues, sacrifing predictability and data integrity for legacy compatibility and performance. It is recommended you use *cryptoassets.core* on PostgreSQL or other alternative database unless you have considerable MySQL experience.

CHAPTER 11

Developing cryptoassets.core

11.1 Running tests

Unit tests are PyTest based.

11.1.1 Testing prerequisites

To run all tests several components must be in-place

- *pip install test-extra-requirements.txt‘*
- Check block.io credentials in tests/config.yaml files (hardcoded accounts, testnet coins)
- *bitcoind* running testnet on localhost, configured for named UNIX pipe wallet notifications (see config snipped below). bitcoind must have account *cryptoassets* with enough balance to do withdraw tests.
- PostgreSQL database `unittest-conflict-resolution` where you can connect on localhost without username and password
- Redis installed, with preferable empty database 0
- ngrok account is required for running block.io webhook tests. You need to create export `NGROK_AUTH_TOKEN` environment variable in order run block.io tests:

```
    export NGROK_AUTH_TOKEN=xxx
```

11.1.2 Examples for running tests

Running all tests:

```
py.test cryptoassets
```

Running a single test case:

```
py.test cryptoassets/core/tests/test_conflictresolver.py
```

Running a single test:

```
py.test -k "BitcoindTestCase.test_send_internal" cryptoassets
```

Running a single test with verbose Python logging output to stdout (useful for pinning down *why* the test fails):

```
VERBOSE_TEST=1 py.test -k "BitcoindTestCase.test_send_internal" cryptoassets
```

Running tests for continuous integration service (15 minute timeout) and skipping slow tests where transactions are routed through cryptocurrency network (full BTC send/receive test, etc.):

```
CI=true py.test cryptoassets
```

Running unittests using vanilla Python 3 unittest:

```
python -m unittest discover
```

(This ignores all skipping hints)

More info

- http://pytest.org/latest/usage.html

11.2 Bitcoind testnet

11.2.1 Setting up TESTNET bitcoind on OSX

Edit `/Users/mikko/Library/Application Support/Bitcoin/bitcoin.conf`:

```
testnet=1
server=1
rpcuser=foo
rpcpassword=bar
rpctimeout=5
rpcport=8332
txindex=1
rpcthreads=64
walletnotify=gtimeout --kill-after=10 5 /bin/bash -c "echo %s >> /tmp/cryptoassets-unittest-walletnot
```

Restart **Bitcoin-Qt**. Now it should give green icon instead of standard orange.

Test the JSON-RPC server connection:

```
curl --user foo:bar --data-binary '{"id":"t0", "method": "getinfo", "params": [] }' http://127.0.0.1
```

http://suffix.be/blog/getting-started-bitcoin-testnet

Starting bitcoind in debug mode:

```
/Applications/Bitcoin-Qt.app/Contents/MacOS/Bitcoin-Qt -printtoconsole -debug
```

11.2.2 Building bitcoind on Ubuntu

- http://bitzuma.com/posts/compile-bitcoin-core-from-source-on-ubuntu/

11.2.3 Topping up bitcoind

First create a receiving address under `bitcoind` accounting account `cryptoassets`:

```
curl --user foo:bar --data-binary '{"id":"t0", "method": "getnewaddress", "params": ["cryptoassets"]
```

Write down the *result*.

11.2.4 Testnet faucet

Get Testnet coins from here:

http://tpfaucet.appspot.com/

(Alternative testnet faucets.)

Send them to the receiving address you created.

Then list `bitcoind` accounts and balances, to see you have the new receiving address and the balance arrives there:

> curl –user foo:bar –data-binary '{"id":"t0", "method": "listaccounts", "params": [] }' http://127.0.0.1:8332/

11.2.5 Dumping your TESTNET private address for importing in tests

Example using public address `mk2o9anFwtHFGFKeD89Qxh5YBhNMQk7NrS`:

```
curl --user foo:bar --data-binary '{"id":"t0", "method": "dumpprivkey", "params": ["mk2o9anFwtHFGFKeD
```

11.2.6 Using bitcoind with multiple backends

If you are using same bitcoind testnet instance to drive several cryptoassets backends, you can multiplex incoming transactions to several wallet notify pipes with a shell script like:

```
#!/bin/bash
echo "Got txid $1" >> /tmp/txlist.txt
# Timeout is needed to work around for hanging named pipe cases where Bitcoin-QT process starts to wr
gtimeout --kill-after=10 5 /bin/bash -c "echo $1 >> /tmp/cryptoassets-unittest-walletnotify-pipe"
gtimeout --kill-after=10 5 /bin/bash -c "echo $1 >> /tmp/tatianastore-cryptoassets-helper-walletnotif
exit 0
```

Also needs coreutils on OSX:

```
brew install coreutils
```

11.2.7 Conflicted transactions

If Bitcoin-QT starts to display transactions sent via RPC as **conflicted** status

1. Your walletnotifty script might be broken, CTRL+C abort Bitcoin-QT in terminal, check error messages:

```
    /Users/mikko/code/notify.sh: line 3: timeout: command not found
    runCommand error: system(/Users/mikko/code/notify.sh 94506c797452745b87e734caf35ec4b62c0ef61f6c7
```

2. rescan blockchain (unclean shutdown?):

```
    /Applications/Bitcoin-Qt.app/Contents/MacOS/Bitcoin-Qt -printtoconsole -debug -rescan
```

3. Make sure "Spend unconfirmed outputs" is toggled off in Bitcoin-QT preferences
4. Make sure you are displaying correct transactions and not old ones (Bitcoin QT pops old conflicting transactions at the top of the history list). Choose "Today" from Bitcoin QT transaction list filters.

11.3 Continuous integration

Continuous integration is running on drone.io <https://drone.io/bitbucket.org/miohtama/cryptoassets/>`_.

See `tests/setup-testing-droneio.sh` how tests are executed.

11.3.1 Full CI test suite

Because some tests may take more than 15 minutes to execute, full test suite cannot be run on CI environment. There is script `full-run-tests.sh` which can be used to run tests on Linux VM + bitcoind testnet instance.

Run this script on a server having running Bitcoind instance.

11.4 Releases

CHAPTER 12

Indices and tables

- genindex
- modindex
- search

C

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W