CryptoToken Converter Documentation

Privex Inc., Chris (Someguy123)

Jun 08, 2020

Contents:

This documentation is for CryptoToken Converter, an open source project developed by Privex Inc. for simple, anonymous conversion between two cryptocurrencies / tokens.

You can find the full source code for the project on our Github

It allows for both uni-directional and bi-directional conversions between user specified coin pairs, whether that's BTC->LTC, or LTC->LTCP (Litecoin to Pegged Litecoin token).

Using CryptoToken Converter, you can easily operate services such as crypto <-> token gateways, as well as token <-> token and crypto <-> crypto.

Out of the box, CryptoToken Converter comes with two *Coin Handlers*:

- ***Bitcoind Coin Handler*** Handles deposits/sending for coins which have a fork of *bitcoind* without dramatic JSONRPC API changes (e.g. Litecoin, Dogecoin).
- ***SteemEngine Coin Handler*** Handles deposits/issuing/sending for tokens that exist on the Steem Engine platform - a side-chain of the Steem blockchain.

Every "coin pair" has an exchange rate set in the database, which can be either statically set for pegged tokens, or dynamically updated for conversions between two different cryptos/tokens.

CHAPTER 1

Installation and configuration

Attention: This guide is aimed at Ubuntu Bionic 18.04 - if you're not running Ubuntu 18.04, some parts of the guide may not apply to you, or simply won't work.

Tip: If you don't have any machines running Ubuntu 18.04, you can grab a dedicated or virtual server pre-installed with it from Privex - we're the ones who wrote this software! :)

1.1 Requirements and Dependencies

Core Dependencies

- Python 3.7+ (may or may not work on older versions)
- PostgreSQL or MySQL for the database
- Nginx for the production web server
- Linux or macOS (OSX) is recommended (may work on Windows, however we refuse to actively support it)

Additional Requirements

- If you plan to use the *Bitcoind Coin Handler* you'll need one or more coin daemons such as `bitcoind` , `litecoind` or `dogecoind` running in server mode, with an rpcuser and rpcpassword configured.
- If you plan to use the *SteemEngine Coin Handler* you'll need a Steem account - for best operation it's recommended that you use Steem Engine tokens that you've created (can issue them), and you must have the **active private key** of the token owner account.

Knowledge

- You should have basic knowledge of navigating a Linux/Unix system, including running basic commands
- It may help if you have at least a basic understanding of the Python programming language

- If you plan to contribute to the project, or make modifications, you should read the documentation for the Django Framework, and the third-party add-on Django REST Framework

1.2 Install Core Dependencies

For this guide, we'll be using PostgreSQL, but you're free to use MySQL if you're more comfortable with it.

Using your system package manager, install Python 3.7, Postgres server, nginx, git, along with some various important libraries needed for our Python packages.

```
sudo apt update
# Install Python 3.7, Nginx, and Git
sudo apt install -y python3.7 python3.7-dev python3.7-venv nginx git

# Install libssl-dev for the OpenSSL headers (required for the Beem python library)
# and build-essential - various tools required for building and compiling the python␣
↪dependencies
sudo apt install -y build-essential libssl-dev

# The `postgresql` package will install the latest Postgres client and server, we␣
↪also want libpq-dev,
# which is the postgres client dev headers, sometimes needed for Python postgres␣
↪libraries
sudo apt install -y postgresql libpq-dev

# Install MariaDB (cross-compatible with MySQL) and the development headers to avoid␣
↪issues with the Python
# MySQL library
sudo apt install -y mariadb-server libmariadbclient-dev libmariadb-dev
```

Tip: The below step for setting your default `python3` is optional, but it may help prevent issues when python files refer to `python3` and not `python3.7`

To avoid the issue of `python3` referring to an older version of Python 3, you should run the following commands to set up Python 3.7 as the default. On Ubuntu 18.04, Python 3.6 is the default used for `python3`.

```
# Make sure both Python 3.6 (Ubuntu 18.04 default), and 3.7 are registered with␣
↪update-alternatives
sudo update-alternatives --install /usr/bin/python3 python3 /usr/bin/python3.6 1
sudo update-alternatives --install /usr/bin/python3 python3 /usr/bin/python3.7 2
# Set `python3.7` as the default version to use when `python3` is ran
sudo update-alternatives --set python3 /usr/bin/python3.7
```

To check if the above worked, you should see `3.7.x` when running `python3 -V` like below:

```
user@host ~ $ python3 -V
Python 3.7.1
```

1.3 Create Database and DB user

For Postgres, this is very easy.

Simply run the below commands to create a user, a database, and make the user the owner of the DB.

```
# Log in as the postgres user
root@host # su - postgres

# Create a user, you'll be prompted for the password
# S = not a superuser, D = cannot create databases, R = cannot create roles
# l = can login, P = prompt for user's new password
$ createuser -SDRl -P steemengine
    Enter password for new role:
    Enter it again:

# Create the database steemengine_pay with the new user as the owner

$ createdb -O steemengine steemengine_pay

# If you've already created the DB, use psql to manually grant permissions to the user

$ psql
    psql (10.6 (Ubuntu 10.6-0ubuntu0.18.04.1))
    Type "help" for help.

    postgres=# GRANT ALL ON DATABASE steemengine TO steemengine_pay;
```

The above commands create a postgres user called `steemengine` and a database called `steemengine_pay` .

Feel free to adjust the username and database name to your liking.

1.4 Download and install the project

Tip: If you're running this in production, for security you should create a limited account, and install the project using that account.

Clone the repo, and enter the directory.

```
git clone https://github.com/privex/cryptotoken-converter
cd cryptotoken-converter
```

Create and activate a **python virtual environment** to avoid conflicts with any packages installed system-wide, or any upgrades to the python version.

```
# Create the virtual environment in the folder `venv`
python3.7 -m venv venv
# Activate the virtual environment.
source venv/bin/activate
```

You must make sure to activate the virtualenv before you run any python files, or install any python packages.

While the virtualenv is activated, you'll see the text `(venv)` on the side of your shell, like so:

```
(venv) user@host ~/cryptotoken-converter $
```

Now that the virtualenv is created and activated, we can install the python packages required to run this project.

```
# pip3 is the package manager for Python 3, this command will install the packages␣
↪listed in `requirements.txt`
pip3 install -r requirements.txt
```

1.5 Beem Wallet (if using Steem)

If you're using a coin handler that uses the **Steem network**, such as *SteemEngine Coin Handler*, then you must create a Beem wallet, and add the **active private key** for each Steem account you intend to send/issue from.

```
# Create a new Beem wallet, make sure to remember your wallet password, you'll need␣
↪it later.
beempy createwallet
# Import the Private Active Key for each Steem account you plan to send/issue from.
beempy addkey
```

1.6 Basic Configuration

The first step of configuration is creating a `.env` file, this will contain various configuration details needed to run the project.

```
# Creates a file called `.env` if it doesn't already exist
touch .env
# Ensures that `.env` can only be read/written to by your user.
chmod 700 .env
```

Open up `.env` in your favourite text editor (such as `vim` or `nano`).

Paste the following example config:

```
DB_USER=steemengine_pay
DB_PASS=MySuperSecretPassword
DB_NAME=steemengine
DEBUG=false
SECRET_KEY=VeryLongRandomStringUsedToProtectYourUserSessions
UNLOCK=
```

Some of the above options can simply be left out if they're just the default, but it's best to specify them anyway, to avoid the application breaking due to changes to the default values.

Now we'll explain what the above options do, as well as some extras.

Basic Config

`SECRET_KEY` - **Required**

A long (recommended 40+ chars) random string of uppercase letters, lowercase letters, and numbers. It's used for various Django functionality, including encryption of your user sessions/cookies.

`DEBUG` - Optional

If set to `True` Django will output detailed error pages, automatically re-load the app when python files are modified, among other helpful development features. If not specified, it defaults to `False`.

This should always be set to FALSE in production, otherwise the error pages WILL leak a lot of information, including sensitive details such as passwords or API keys.

`EX_FEE` - Optional

This option sets the exchange fee, as a percentage. For example *1* would mean a 1% fee is taken from each exchange from crypto->token and token->crypto.

You may also use decimal numbers, such as *0.5* for 0.5%, or to disable exchange fees, simply set it to *0* or remove the line entirely, **as the default is no fee**.

`COIN_HANDLERS` - Optional.

If you're using any third party *Coin Handlers* or you want to disable some of the default ones, this is a list of comma separated Coin Handler folder names.

Default: `SteemEngine,Bitcoin`

Steem Configuration

If you plan to use *SteemEngine Coin Handler* then you may want to configure these as needed.

`STEEM_RPC_NODES` - Optional

If you want to override the Steem RPC node(s) used for functions such as signing the custom_json transactions from the token issuing account, you can specify them as a comma separated list.

They will be used in the order they are specified.

Default: Automatically use best node determined by Beem

Example: `STEEM_RPC_NODES=https://steemd.privex.io,https://api.steemit.com`

`UNLOCK` - **Required if using Steem**

The wallet password for Beem. This must be specified to allow Steem transactions to be automatically signed. See the section *Beem Wallet (if using Steem)* to create a wallet.

Database Configuration

- `DB_BACKEND` - What type of DB are you using? `mysql` or `postgresql` Default: `postgresql`
- `DB_HOST` - What hostname/ip is the DB on? Default: `localhost`
- `DB_NAME` - What is the name of the database to use? Default: `steemengine_pay`
- `DB_USER` - What username to connect with? Default: `steemengine`
- `DB_PASS` - What password to connect with? Default: no password

1.7 Final Setup

The app is almost ready to go! Just a few last things.

To create the database structure (tables, relations etc.), you'll need to run the Django migrations

```
./manage.py migrate
```

You'll also want to create an admin account (superuser)

```
./manage.py createsuperuser
```

Now, start the Django server

```
./manage.py runserver
```

You should now be able to go to http://127.0.0.1:8000/admin/ in your browser and access the Django admin.

Login using the superuser account you've created.

Using the admin panel, create at least two Coin's (`payments.models.Coin`), and at least one Coin Pair (`payments.models.CoinPair`).

Make sure to set each Coin's "Coin Type" correctly, so that Coin Handlers will detect them (use the types "SteemEngine Token", and "Bitcoind Compatible"). You may have to refresh the "Add Coin" page if some of the types don't show up.

After adding the coins, you should now be able to open one of the API pages in your browser, such as this one: http://127.0.0.1:8000/api/coins/

If you can see your added coins on that page, everything should be working! :)

Now try making some conversions using the API: *REST API Documentation*

1.8 Transaction Scanning and Conversion

To handle incoming deposits, and converting deposits into their destination coin, there are two management commands to run.

`./manage.py load_txs`

The command **load_txs** imports incoming transactions into the Deposits table for any Coin that has a properly configured Coin Handler (*Coin Handlers*).

`./manage.py convert_coins`

The command **convert_coins** scans each deposit in the Deposit table to check if it's valid, and which Coin it should be converted to.

Each valid deposit will then be converted into it's destination coin, and the deposit will be marked as `conv` (Successfully Converted).

If you're running with DEBUG set to true, you'll see a detailed log of what it's doing, so you can diagnose any problems with your coin configuration and fix it.

When running in production, you would normally have these running on a **cron** - a scheduled task.

To find out how to run this in production, please read *Running in Production*

CHAPTER 2

Running in Production

CHAPTER 3

Python Code Documentation

3.1 *steemengine* module

3.1.1 steemengine package

3.1.1.1 Subpackages

3.1.1.1.1 steemengine.settings (Django Settings)

3.1.1.1.1.1 Submodules

3.1.1.1.1.2 steemengine.settings.core module

This file contains the core settings of the application. Settings specified within this file are used directly by the Django framework, or a third-party extension / application for Django.

User specifiable environment variables:

Basic Config

- `DEBUG` - If set to true, enable debugging features, such as **extremely verbose error pages** and automatic code reloading on edit. **DO NOT RUN WITH DEBUG IN PRODUCTION, IT IS NOT SAFE.**

  Default: `False`

- `SECRET_KEY` - **MANDATORY** - A long random string used to encrypt user sessions, among other security features.
- `CORS_ORIGIN_ALLOW_ALL` - If True, allow all cross-origin requests (disable whitelist). **Default:** `True`
- `CORS_ORIGIN_WHITELIST` - Comma separated list of domains and subdomains to allow CORS for. Adding a domain does not automatically include it's subdomains. All subdomains must be added manually. **Default:** Blank

- `ALLOWED_HOSTS` - Comma separated list of the domains you intend to run this on. For security (e.g. preventing cookie theft), Django requires that you specify each hostname that this application should be accessible from. **Default:** `127.0.0.1,localhost` (these are also auto added if DEBUG is True).

Database Settings

- `DB_BACKEND` - What type of DB are you using? `mysql` or `postgresql` **Default:** `postgresql`
- `DB_HOST` - What hostname/ip is the DB on? **Default:** `localhost`
- `DB_NAME` - What is the name of the database to use? **Default:** `steemengine_pay`
- `DB_USER` - What username to connect with? **Default:** `steemengine`
- `DB_PASS` - What password to connect with? **Default:** no password

For more information on this file, see https://docs.djangoproject.com/en/2.1/topics/settings/

For the full list of settings and their values, see https://docs.djangoproject.com/en/2.1/ref/settings/

Copyright:

```
+=====================================================+
|                 © 2019 Privex Inc.                  |
|               https://www.privex.io                 |
+=====================================================+
|                                                     |
|        CryptoToken Converter                        |
|                                                     |
|        Core Developer(s):                           |
|                                                     |
|          (+)  Chris (@someguy123) [Privex]          |
|                                                     |
+=====================================================+
```

3.1.1.1.1.3 steemengine.settings.custom module

This file contains settings that are specific to CryptoToken Converter, and do not affect the core Django framework.

User specifiable environment variables:

- `STEEM_RPC_NODES` - Comma-separated list of one/more Steem RPC nodes. If not set, will use the default beem nodes.
- `BITSHARES_RPC_NODE` - Node to use to connect to Bitshares network if Bitshares coin handler is enabled. If not set, will default to wss://eu.nodes.bitshares.ws
- `EX_FEE` - Conversion fee taken by us, in percentage (i.e. "1" = 1%) **Default:** 0 (no fee)
- `COIN_HANDLERS` - A comma separated list of Coin Handler modules to load. **Default:** SteemEngine,Bitcoin
- `COIN_HANDLERS_BASE` - If your coin handlers are not located in `payments.coin_handlers` then you may change this to point to the base module where your coin handlers are located.
- `LOWFUNDS_NOTIFY` - If you're using the low wallet balance notifications, you can change how often it re-notifies the admin emails `ADMINS` if the balance is still too low to fulfill a conversion. (in hours). **Default:** `12` (hours)

Copyright:

```
+=====================================================+
|                 © 2019 Privex Inc.                  |
|               https://www.privex.io                 |
+=====================================================+
|                                                     |
|        CryptoToken Converter                        |
|                                                     |
|        Core Developer(s):                           |
|                                                     |
|          (+)  Chris (@someguy123) [Privex]          |
|                                                     |
+=====================================================+
```

steemengine.settings.custom.**COIN_HANDLERS = ['SteemEngine', 'HiveEngine', 'Bitcoin', 'Steem**

Specify in the env var COIN_HANDERS a comma separated list of local coin handlers *payments.coin_handlers* to load. If not specified, the default list will be used.

steemengine.settings.custom.**COIN_HANDLERS_BASE = 'payments.coin_handlers'**

Load coin handlers from this absolute module path

steemengine.settings.custom.**COIN_TYPES = (('crypto', 'Generic Cryptocurrency'), ('token',**

This is used for the dropdown "Coin Type" selection in the Django admin panel. Coin handlers may add to this list.

steemengine.settings.custom.**ENCRYPT_KEY = None**

Used for encrypting and decrypting private keys so they cannot be displayed in plain text by the admin panel, or external applications accessing the DB. Generate an encryption key using `./manage.py generate_key`. To print just the key, use `./manage.py generate_key 2> /dev/null`

steemengine.settings.custom.**EX_FEE = Decimal('0')**

Conversion fee taken by us, in percentage (i.e. "1" = 1%)

steemengine.settings.custom.**LOWFUNDS_RENOTIFY = 12**

After the first email to inform admins a wallet is low, how long before we send out a second notification? (in hours) (Default: 12 hrs)

steemengine.settings.custom.**PRIVEX_HANDLERS = ['Golos']**

These handlers are from the **:py:my:'privex.coin_handlers'** package, so they're loaded differently to the handlers listed in COIN_HANDLERS

3.1.1.1.1.4 steemengine.settings.log module

Logging configuration for CryptoToken Converter.

Valid environment log levels (from least to most severe) are:

```
DEBUG, INFO, WARNING
ERROR, FATAL, CRITICAL
```

User customisable environment variables are:

- CONSOLE_LOG_LEVEL - Messages equal to and above this level will be logged to the console (i.e. the output of manage.py commands such as runserver or load_txs) **Default:** INFO in production, DEBUG when DEBUG setting is true
- DBGFILE_LEVEL - Messages equal to and above this level will be logged to the `debug.log` files. **Default:** INFO in production, DEBUG when DEBUG setting is true.
- ERRFILE_LEVEL - Same as DBGFILE_LEVEL but for `error.log` - Default: WARNING

- `LOGGER_NAMES` - A comma separated list of logger instance names to apply the default logging settings onto. **Default:** `privex` (Use same logging for Privex's python packages)
- `BASE_LOGGER_NAME` - The logger instance name to use for the main logger. If this is not specified, or is blank, then the logging API "RootLogger" will be used, which may automatically configure logging for various packages.
- `BASE_LOG_FOLDER` - A relative path from the root of the project (folder with manage.py) to the folder where log files should be stored. **Default:** `logs`
- `BASE_WEB_LOGS` - Relative path from BASE_LOG_FOLDER where logs from the web app should be stored. **Default:** web
- `BASE_CRON_LOGS` - Relative path from BASE_LOG_FOLDER where logs from scheduled commands (load_txs etc.) should be stored. **Default:** crons

`steemengine.settings.log.`**`config_logger`**(**logger_names, log_dir='/home/docs/checkouts/readthedocs.org/user_builds/converter/checkouts/latest/logs'*)

Used to allow isolated parts of this project to easily change the log output folder, e.g. allow Django management commands to change the logs folder to `crons/`

Currently only used by `payments.management.CronLoggerMixin`

Usage:

```
>>> config_logger('someapp', 'otherlogger', 'mylogger', log_dir='/full/path/to/
↪log/folder')
```

Parameters

- **`logger_names`** (*`str`*) – List of logger names to replace logging config for (see LOGGER_NAMES)
- **`log_dir`** (*`str`*) – Fully qualified path. Set each logger's timed_file log directory to this

Returns `logging.Logger` instance of BASE_LOGGER

3.1.1.1.1.5 Module contents

3.1.1.2 Submodules

3.1.1.3 steemengine.helpers module

Various helper functions for use in CryptoToken Converter.

Copyright:

```
+=====================================================+
|                 © 2019 Privex Inc.                  |
|               https://www.privex.io                 |
+=====================================================+
|                                                     |
|        CryptoToken Converter                        |
|                                                     |
|        Core Developer(s):                           |
|                                                     |
|          (+)  Chris (@someguy123) [Privex]          |
|                                                     |
+=====================================================+
```

`steemengine.helpers.`**`decrypt_str`**(*data: Union[str, bytes], key: Union[str, bytes] = None*) → str

Decrypts `data` previously encrypted using *`encrypt_str()`* with the same Fernet compatible `key`, and returns the decrypted version as a string.

The `key` cannot just be a random "password", it must be a 32-byte key encoded with URL Safe base64. Use the management command `./manage.py generate_key` to create a Fernet compatible encryption key.

Under the hood, Fernet uses AES-128 CBC to encrypt the data, with PKCS7 padding and HMAC_SHA256 authentication.

If the `key` parameter isn't passed, or is empty (None / ""), then it will attempt to fall back to `settings.ENCRYPT_KEY` - if that's also empty, EncryptKeyMissing will be raised.

Parameters

- **`data`** (*`str`*) – The base64 encoded data to be decrypted, in the form of either a str or bytes.
- **`key`** (*`str`*) – A Fernet encryption key (base64) for decryption, if blank, will fall back to settings.ENCRYPT_KEY

Raises

- **`EncryptKeyMissing`** – Either no key was passed, or something is wrong with the key.
- **`EncryptionError`** – Something went wrong while attempting to decrypt the data

Return str decrypted_data The decrypted data as a string

`steemengine.helpers.`**`empty`**(*v, zero=False, itr=False*) → bool

Quickly check if a variable is empty or not. By default only '' and None are checked, use *itr* and *zero* to test for empty iterable's and zeroed variables.

Returns True if a variable is None or '', returns False if variable passes the tests

Parameters

- **`v`** – The variable to check if it's empty
- **`zero`** – if zero=True, then return True if the variable is 0
- **`itr`** – if itr=True, then return True if the variable is `[]`, `{}`, or is an iterable and has 0 length

Return bool is_blank True if a variable is blank (`None`, `''`, `0`, `[]` etc.)

Return bool is_blank False if a variable has content (or couldn't be checked properly)

`steemengine.helpers.`**`encrypt_str`**(*data: Union[str, bytes], key: Union[str, bytes] = None*) → str

Encrypts a piece of data `data` passed as a string or bytes using Fernet with the passed 32-bit symmetric encryption key `key`. Outputs the encrypted data as a Base64 string for easy storage.

The `key` cannot just be a random "password", it must be a 32-byte key encoded with URL Safe base64. Use the management command `./manage.py generate_key` to create a Fernet compatible encryption key.

Under the hood, Fernet uses AES-128 CBC to encrypt the data, with PKCS7 padding and HMAC_SHA256 authentication.

If the `key` parameter isn't passed, or is empty (None / ""), then it will attempt to fall back to `settings.ENCRYPT_KEY` - if that's also empty, EncryptKeyMissing will be raised.

Parameters

- **`data`** (*`str`*) – The data to be encrypted, in the form of either a str or bytes.
- **`key`** (*`str`*) – A Fernet encryption key (base64) to be used, if left blank will fall back to settings.ENCRYPT_KEY

Raises

- **EncryptKeyMissing** – Either no key was passed, or something is wrong with the key.
- **EncryptionError** – Something went wrong while attempting to encrypt the data

Return str encrypted_data The encrypted version of the passed `data` as a base64 encoded string.

`steemengine.helpers.`**`get_fernet`**(*key: Union[str, bytes] = None*) → cryptography.fernet.Fernet

Used internally for getting Fernet instance with auto-fallback to settings.ENCRYPT_KEY and exception handling.

Parameters **key** (*str*) – Base64 Fernet symmetric key for en/decrypting data. If empty, will fallback to settings.ENCRYPT_KEY

Raises **EncryptKeyMissing** – Either no key was passed, or something is wrong with the key.

Return Fernet f Instance of Fernet using passed `key` or settings.ENCRYPT_KEY for encryption.

`steemengine.helpers.`**`is_encrypted`**(*data: Union[str, bytes], key: Union[str, bytes] = None*) → bool

Returns True if the passed `data` appears to be encrypted. Can only verify encryption if the same `key` that was used to encrypt the data is passed.

Parameters

- **data** (*str*) – The data to check for encryption, either as a string or bytes
- **key** (*str*) – Base64 encoded Fernet symmetric key for decrypting data. If empty, fallback to settings.ENCRYPT_KEY

Raises **EncryptKeyMissing** – Either no key was passed, or something is wrong with the key.

Return bool is_encrypted True if the data is encrypted, False if it's not encrypted or wrong key used.

`steemengine.helpers.`**`random_str`**(*size=50, chars='abcdefhkmnprstwxyz2345679ACDEFGHJKLMNPRSTWXYZ'*)

3.1.1.4 steemengine.urls module

steemengine URL Configuration

The *urlpatterns* list routes URLs to views. For more information please see: https://docs.djangoproject.com/en/2.1/topics/http/urls/

Copyright:

```
+=====================================================+
|                © 2019 Privex Inc.                   |
|               https://www.privex.io                 |
+=====================================================+
|                                                     |
|        CryptoToken Converter                        |
|                                                     |
|        Core Developer(s):                           |
|                                                     |
|          (+)  Chris (@someguy123) [Privex]          |
|                                                     |
+=====================================================+
```

`steemengine.urls.`**`path`**(*route, view, kwargs=None, name=None, *, Pattern=<class 'django.urls.resolvers.RoutePattern'>*)

`steemengine.urls.`**`re_path`**(*route, view, kwargs=None, name=None, *, Pattern=<class 'django.urls.resolvers.RegexPattern'>*)

3.1.1.5 steemengine.wsgi module

WSGI config for steemengine project.

It exposes the WSGI callable as a module-level variable named `application`.

For more information on this file, see https://docs.djangoproject.com/en/2.1/howto/deployment/wsgi/

3.1.1.6 Module contents

3.2 *payments* module

3.2.1 Coin Handlers

3.2.1.1 Subpackages

3.2.1.1.1 Bitcoind Coin Handler

3.2.1.1.1.1 Module contents

Bitcoind-based Coin Handler

This python module is a **Coin Handler** for Privex's CryptoToken Converter, designed to handle all required functionality for both receiving and sending any cryptocurrency which has a coin daemon that has a JSONRPC API backwards compatible with *bitcoind*.

It will automatically handle any *`payments.models.Coin`* which has it's type set to `bitcoind`

Coin object settings:

For each coin you intend to use with this handler, you should configure it as such:

Coin Key	Description
coin_type	This should be set to `Bitcoind RPC compatible crypto` (db value: bitcoind)
setting_host	The IP or hostname for the daemon. If not specified, defaults to 127.0.0.1 / localhost
setting_port	The RPC port for the daemon. If not specified, defaults to 8332
setting_user	The rpcuser for the daemon. Generally MUST be specified.
setting_pass	The rpcpassword for the daemon. Generally MUST be specified
setting_json	A JSON string for optional extra config (see below)

Extra JSON (Handler Custom) config options:

- `confirms_needed` Default 0; Amount of confirmations needed before loading a TX
- `use_trusted` Default: True; If enabled, TXs returned from the daemon with 'trusted':true will always be accepted at 0 confs regardless of `confirms_needed`
- `string_amt` Default: True; If true, when sending coins, a `Decimal` will be used (as a string). This can cause problems with older coins such as Dogecoin, so for older coins that need floats, set this to False.

Django Settings:

If you'd rather not store the RPC details in the database, you may specify them in Django's settings.py.

If a coin symbol is specified in `settings.COIND_RPC` they will be used exclusively, and any handler settings on the Coin object will be ignored.

If a settings key isn't specified, the default is the same as shown for coin object settings.

Example COIND_RPC Setting:

```
COIND_RPC = {
  "BTC": {
      'user': 'bitcoinrpc',
      'password': 'SuperSecurePass',
      'host':     '127.0.0.1',
      'port':     8332,
      'confirms_needed': 0,
      'string_amt': True,
      'use_trusted': True
  }
}
```

Copyright:

```
+=====================================================+
|                 © 2019 Privex Inc.                  |
|               https://www.privex.io                 |
+=====================================================+
|                                                     |
|        CryptoToken Converter                        |
|                                                     |
|        Core Developer(s):                           |
|                                                     |
|          (+)  Chris (@someguy123) [Privex]          |
|                                                     |
+=====================================================+
```

`payments.coin_handlers.Bitcoin.`**`reload`**`()`

Reload's the `provides` property for the loader and manager from the DB.

By default, as there are many coins that use a direct fork of bitcoind, our classes can provide for any `models.Coin` by scanning for coins with the type `bitcoind`. This saves us from hard coding specific coin symbols.

3.2.1.1.1.2 Submodules

3.2.1.1.1.3 BitcoinLoader module

Copyright:

```
+=====================================================+
|                 © 2019 Privex Inc.                  |
|               https://www.privex.io                 |
+=====================================================+
|                                                     |
|        CryptoToken Converter                        |
|                                                     |
|        Core Developer(s):                           |
|                                                     |
|          (+)  Chris (@someguy123) [Privex]          |
```

(continues on next page)

(continued from previous page)

```
|                                                     |
+=====================================================+
```

class payments.coin_handlers.Bitcoin.BitcoinLoader.**BitcoinLoader**(*symbols*)

Bases: *payments.coin_handlers.base.BatchLoader.BatchLoader*, *payments.coin_handlers.Bitcoin.BitcoinMixin.BitcoinMixin*

BitcoinLoader - Despite the name, loads TXs from any coin that has a bitcoind-compatible JsonRPC API

Known to work with: bitcoind, litecoind, dogecoind

Copyright:

```
+=====================================================+
|                 © 2019 Privex Inc.                  |
|               https://www.privex.io                 |
+=====================================================+
|                                                     |
|        CryptoToken Converter                        |
|                                                     |
|        Core Developer(s):                           |
|                                                     |
|          (+)  Chris (@someguy123) [Privex]          |
|                                                     |
+=====================================================+
```

For the **required Django settings**, please see the module docstring in `coin_handlers.Bitcoin`

clean_txs(*symbol: str, transactions: Iterable[dict], account: str = None*) → Generator[dict, None, None]

Filters a list of transactions *transactions* as required, yields dict's conforming with `models.Deposit`

- Filters out transactions that are not marked as 'receive'
- Filters out mining transactions
- Filters by address if *account* is specified
- Filters out transactions that don't have enough confirms, and are not reported as 'trusted'

Parameters

- **symbol** – Symbol of coin being cleaned
- **transactions** – A `list<dict>` or generator producing dict's
- **account** – If not None, only return TXs sent to this address.

Return Generator<dict> A generator outputting dictionaries formatted as below

Output Format:

```
{
  txid:str, coin:str (symbol), vout:int,
  tx_timestamp:datetime, address:str, amount:Decimal
}
```

load_batch(*symbol, limit=100, offset=0, account=None*)

Loads a batch of transactions for *symbol* in their original format into *self.transactions*

Parameters

- **symbol** (*str*) – The coin symbol to load TXs for
- **limit** (*int*) – The amount of transactions to load
- **offset** (*int*) – The amount of most recent TXs to skip (for pagination)
- **account** (*str*) – NOT USED BY THIS LOADER

provides
Dynamically populated by Bitcoin.__init__

rpcs = {}
For each coin connection specified in *settings.COIND_RPC*, we map it's symbol to an instantiated instance of BitcoinRPC - stored as a static property, ensuring we don't have to constantly re-create them.

settings
To ensure we always get fresh settings from the DB after a reload, self.settings gets _prep_settings()

3.2.1.1.1.4 BitcoinManager module

Copyright:

```
+====================================================+
|                 © 2019 Privex Inc.                 |
|               https://www.privex.io                |
+====================================================+
|                                                    |
|        CryptoToken Converter                       |
|                                                    |
|        Core Developer(s):                          |
|                                                    |
|          (+)  Chris (@someguy123) [Privex]         |
|                                                    |
+====================================================+
```

class payments.coin_handlers.Bitcoin.BitcoinManager.**BitcoinManager**(*symbol: str*)

Bases: *payments.coin_handlers.base.BaseManager.BaseManager*, *payments.coin_handlers.Bitcoin.BitcoinMixin.BitcoinMixin*

BitcoinManager - Despite the name, handles sending, balance, and deposit addresses for any coin that has a bitcoind-compatible JsonRPC API

Known to work with: bitcoind, litecoind, dogecoind

Copyright:

```
+====================================================+
|                 © 2019 Privex Inc.                 |
|               https://www.privex.io                |
+====================================================+
|                                                    |
|        CryptoToken Converter                       |
|                                                    |
|        Core Developer(s):                          |
|                                                    |
|          (+)  Chris (@someguy123) [Privex]         |
|                                                    |
+====================================================+
```

For the **required Django settings**, please see the module docstring in `coin_handlers.Bitcoin`

address_valid(*address*) → bool
If *address* is determined to be valid by the coind RPC, will return True. Otherwise False.

balance(*address: str = None*, *memo: str = None*, *memo_case: bool = False*) → decimal.Decimal
Get the total amount received by an address, or the balance of the wallet if address not specified.

Parameters

- **address** – Crypto address to get balance for, if None, returns whole wallet balance
- **memo** – NOT USED BY THIS MANAGER
- **memo_case** – NOT USED BY THIS MANAGER

Returns Decimal(balance)

get_deposit() → tuple
Returns a deposit address for this symbol :return tuple: A tuple containing ('address', crypto_address)

health() → Tuple[str, tuple, tuple]
Return health data for the passed symbol.

Health data will include: Symbol, Status, Current Block, Node Version, Wallet Balance, and number of p2p connections (all as strings)

Return tuple health_data (manager_name:str, headings:list/tuple, health_data:list/tuple,)

health_test() → bool
Check if the coin daemon is up or not, by requesting basic information such as current block and version.

Return bool True if the coin daemon appears to be working, False if it's not

provides
Dynamically populated by Bitcoin.__init__

rpcs = {}
For each coin connection specified in *settings.COIND_RPC*, we map it's symbol to an instantiated instance of BitcoinRPC - stored as a static property, ensuring we don't have to constantly re-create them.

send(*amount*, *address*, *memo=None*, *from_address=None*, *trigger_data=None*) → dict
Send the amount *amount* of *self.symbol* to a given address.

Example - send 0.1 LTC to LVXXmgcVYBZAuiJM3V99uG48o3yG89h2Ph

```
>>> s = BitcoinManager('LTC')
>>> s.send(address='LVXXmgcVYBZAuiJM3V99uG48o3yG89h2Ph', amount=Decimal('0.1
↪'))
```

Parameters

- **amount** (*Decimal*) – Amount of coins to send, as a Decimal()
- **address** – Address to send the coins to
- **from_address** – NOT USED BY THIS MANAGER
- **memo** – NOT USED BY THIS MANAGER

Raises

- ***AccountNotFound*** – The destination *address* isn't valid
- ***NotEnoughBalance*** – The wallet does not have enough balance to send this amount.

Return dict Result Information

Format:

```
{
    txid:str - Transaction ID - None if not known,
    coin:str - Symbol that was sent,
    amount:Decimal - The amount that was sent (after fees),
    fee:Decimal    - TX Fee that was taken from the amount,
    from:str       - The account/address the coins were sent from,
    send_type:str       - Should be statically set to "send"
}
```

setting
Retrieve only our symbol from self.settings for convenience

settings
To ensure we always get fresh settings from the DB after a reload, self.settings gets _prep_settings()

3.2.1.1.1.5 BitcoinMixin module

Copyright:

```
+=====================================================+
|                 © 2019 Privex Inc.                  |
|               https://www.privex.io                 |
+=====================================================+
|                                                     |
|        CryptoToken Converter                        |
|                                                     |
|        Core Developer(s):                           |
|                                                     |
|          (+)  Chris (@someguy123) [Privex]          |
|                                                     |
+=====================================================+
```

class payments.coin_handlers.Bitcoin.BitcoinMixin.**BitcoinMixin**
Bases: object

BitcoinMixin - shared code used by both `Bitcoin.BitcoinLoader` and `Bitcoin.BitcoinManager`

Copyright:

```
+=====================================================+
|                 © 2019 Privex Inc.                  |
|               https://www.privex.io                 |
+=====================================================+
|                                                     |
|        CryptoToken Converter                        |
|                                                     |
|        Core Developer(s):                           |
|                                                     |
|          (+)  Chris (@someguy123) [Privex]          |
|                                                     |
+=====================================================+
```

all_coins

Since this is a Mixin, it may be self.coin: Coin, or self.coins: List[Coin]. This property detects whether we have a single coin, or multiple, and returns them as a dict.

Return dict coins A dict<str,Coin> of supported coins, mapped by symbol

3.2.1.1.2 Bitshares Coin Handler

3.2.1.1.2.1 Module contents

Bitshares Coin Handler

This python module is a **Coin Handler** for Privex's CryptoToken Converter, designed to handle all required functionality for both receiving and sending tokens on the **Bitshares** network.

It will automatically handle any *payments.models.Coin* which has its type set to `bitshares`

Copyright:

```
+===================================================+
|                 © 2019 Privex Inc.                |
|               https://www.privex.io               |
+===================================================+
|                                                   |
|        CryptoToken Converter                      |
|                                                   |
|        Core Developer(s):                         |
|                                                   |
|          (+)  Chris (@someguy123) [Privex]        |
|                                                   |
+===================================================+
```

`payments.coin_handlers.Bitshares.`**`reload`**`()`

Reload's the `provides` property for the loader and manager from the DB.

By default, since new tokens are constantly being created for Bitshares, our classes can provide for any `models.Coin` by scanning for coins with the type `bitshares`. This saves us from hard coding specific coin symbols.

3.2.1.1.2.2 Submodules

3.2.1.1.2.3 BitsharesLoader module

Copyright:

```
+===================================================+
|                 © 2019 Privex Inc.                |
|               https://www.privex.io               |
+===================================================+
|                                                   |
|        CryptoToken Converter                      |
|                                                   |
|        Core Developer(s):                         |
|                                                   |
|          (+)  Chris (@someguy123) [Privex]        |
```

(continues on next page)

(continued from previous page)

```
|                                                     |
+=====================================================+
```

class payments.coin_handlers.Bitshares.BitsharesLoader.**BitsharesLoader**(*symbols*)

Bases: *payments.coin_handlers.base.BaseLoader.BaseLoader*, *payments.coin_handlers.Bitshares.BitsharesMixin.BitsharesMixin*

This class handles loading transactions for the **Bitshares** network, and can support any token on Bitshares.

Copyright:

```
+=====================================================+
|                 © 2019 Privex Inc.                  |
|               https://www.privex.io                 |
+=====================================================+
|                                                     |
|        CryptoToken Converter                        |
|                                                     |
|        Core Developer(s):                           |
|                                                     |
|          (+)  Chris (@someguy123) [Privex]          |
|                                                     |
+=====================================================+
```

clean_txs(*account: bitshares.account.Account, symbol: str, transactions: Iterable[dict]*) → Generator[dict, None, None]

Filters a list of transactions by the receiving account, yields dict's conforming with *payments.models.Deposit*

Parameters

- **account** (*str*) – The 'to' account to filter by
- **symbol** (*str*) – The symbol of the token being filtered
- **transactions** (*list<dict>*) – A list<dict> of transactions to filter

Returns A generator yielding `dict` s conforming to *payments.models.Deposit*

list_txs(*batch=0*) → Generator[dict, None, None]

Get transactions for all coins in *self.coins* where the 'to' field matches coin.our_account If *load()* hasn't been ran already, it will automatically call self.load() :return: Generator yielding dicts that conform to `models.Deposit`

load(*tx_count=1000*)

The load function should prepare your loader, by either importing all of the data required for filtering, or setting up a generator for the *list_txs()* method to load them paginated.

It does NOT return anything, it simply creates any connections required, sets up generator functions if required for paginating the data, and/or pre-loads the first batch of transaction data.

Parameters **tx_count** – The total amount of transactions that should be loaded PER SYMBOL, most recent first.

Returns None

provides

This attribute is automatically generated by scanning for `models.Coin` s with the type `bitshares`. This saves us from hard coding specific coin symbols. See __init__.py for populating code.

3.2.1.1.2.4 BitsharesManager module

Copyright:

```
+=====================================================+
|                 © 2019 Privex Inc.                  |
|               https://www.privex.io                 |
+=====================================================+
|                                                     |
|        CryptoToken Converter                        |
|                                                     |
|        Core Developer(s):                           |
|                                                     |
|          (+)  Chris (@someguy123) [Privex]          |
|                                                     |
+=====================================================+
```

class payments.coin_handlers.Bitshares.BitsharesManager.**BitsharesManager**(*symbol: str*)

Bases: *payments.coin_handlers.base.BaseManager.BaseManager*, *payments.coin_handlers.Bitshares.BitsharesMixin.BitsharesMixin*

This class handles various operations for the **Bitshares* network, and supports any token on Bitshares.

It handles:

- Validating source/destination accounts
- Checking the balance for a given account, as well as the total amount received with a certain `memo`
- Issuing tokens to users
- Sending tokens to users

Copyright:

```
+=====================================================+
|                 © 2019 Privex Inc.                  |
|               https://www.privex.io                 |
+=====================================================+
|                                                     |
|        CryptoToken Converter                        |
|                                                     |
|        Core Developer(s):                           |
|                                                     |
|          (+)  Chris (@someguy123) [Privex]          |
|                                                     |
+=====================================================+
```

address_valid(*address*) → bool

If an account (`address` param) exists on Bitshares, will return True. Otherwise False.

balance(*address: str = None, memo: str = None, memo_case: bool = False*) → decimal.Decimal

Get token balance for a given Bitshares account, if memo is given - get total symbol amt received with this memo.

Parameters

- **address** – Bitshares account to get balance for, if not set, uses self.coin.our_account
- **memo** – If not None, get total *self.symbol* received with this memo.

- **memo_case** – Case sensitive memo search

Raises ***AccountNotFound*** – The requested account/address doesn't exist

Returns Decimal(balance)

get_deposit() → tuple

Returns the deposit account for this symbol

Return tuple A tuple containing ('account', receiving_account). The memo must be generated by the calling function.

health() → Tuple[str, tuple, tuple]

Return health data for the passed symbol.

Health data will include: symbol, status, API node, symbol issuer, symbol precision, our account for transacting with this symbol, and our account's balance of this symbol

Return tuple health_data (manager_name:str, headings:list/tuple, health_data:list/tuple,)

health_test() → bool

Check if the Bitshares API and node connection works or not, by requesting basic information such as the token metadata, and checking if our sending/receiving account exists.

Return bool True if Bitshares appears to be working, False if broken.

is_amount_above_minimum(*amount: decimal.Decimal*, *precision: int*) → bool

Helper function to test if amount is at least equal to the minimum allowed fractional unit of our token. Returns True if so, otherwise False.

issue(*amount: decimal.Decimal*, *address: str*, *memo: str = None*, *trigger_data=None*) → dict

Issue (create/print) tokens to a given address/account, optionally specifying a memo if desired. The network transaction fee for issuance will be paid by the issuing account in BTS.

Example - Issue 5.10 SGTK to @privex

```
>>> s = BitsharesManager('SGTK')
>>> s.issue(address='privex', amount=Decimal('5.10'))
```

Parameters

- **amount** (*Decimal*) – Amount of tokens to issue, as a Decimal
- **address** – Account to issue the tokens to (which is also the issuer account)
- **memo** – Optional memo for the issuance transaction

Raises

- ***IssuerKeyError*** – Cannot issue because we don't have authority to (missing key etc.)
- ***TokenNotFound*** – When the requested token *symbol* does not exist
- ***AccountNotFound*** – The requested account doesn't exist
- **ArithmeticError** – When the amount is lower than the lowest amount allowed by the token's precision

Return dict Result Information

Format:

```
{
    txid:str - Transaction ID - None if not known,
    coin:str - Symbol that was sent,
    amount:Decimal - The amount that was issued,
    fee:Decimal    - TX Fee that was taken from the amount (will be 0 if fee␣
↪is in BTS rather than the issuing token),
    from:str       - The account/address the coins were issued from,
    send_type:str       - Should be statically set to "issue"
}
```

provides

This attribute is automatically generated by scanning for `models.Coin` s with the type `bitshares`. This saves us from hard coding specific coin symbols. See __init__.py for populating code.

send(*amount*, *address*, *memo=None*, *from_address=None*, *trigger_data=None*) → dict

Send tokens to a given address/account, optionally specifying a memo. The Bitshares network transaction fee will be subtracted from the amount before sending.

There must be a valid `models.CryptoKeyPair` in the database for both 'active' and 'memo' keys for the from_address account, or an AuthorityMissing exception will be thrown.

Example - send 1.23 BUILDTEAM from @someguy123 to @privex with memo 'hello'

```
>>> s = BitsharesManager('BUILDTEAM')
>>> s.send(from_address='someguy123', address='privex', amount=Decimal('1.23
↪'), memo='hello')
```

Parameters

- **amount** (*Decimal*) – Amount of tokens to send, as a Decimal()
- **address** – Account to send the tokens to
- **from_address** – Account to send the tokens from
- **memo** – Memo to send tokens with

Raises

- **AttributeError** – When both *from_address* and *self.coin.our_account* are blank.
- **ArithmeticError** – When the amount is lower than the lowest amount allowed by the token's precision (after subtracting the network transaction fee)
- ***AuthorityMissing*** – Cannot send because we don't have authority to (missing key etc.)
- ***AccountNotFound*** – The requested account/address doesn't exist
- ***TokenNotFound*** – When the requested token *symbol* does not exist
- ***NotEnoughBalance*** – The account *from_address* does not have enough balance to send this amount.

Return dict Result Information

Format:

```
{
    txid:str - Transaction ID - None if not known,
    coin:str - Symbol that was sent,
```

(continues on next page)

(continued from previous page)

```
    amount:Decimal - The amount that was sent (after fees),
    fee:Decimal    - TX Fee that was taken from the amount,
    from:str       - The account/address the coins were sent from,
    send_type:str       - Should be statically set to "send"
}
```

send_or_issue(*amount*, *address*, *memo=None*, *trigger_data=None*) → dict

Send tokens to a given account, optionally specifying a memo. If the balance of the sending account is too low, try to issue new tokens to ourself first. See documentation for the `BitsharesManager.BitsharesManager.send()` and `BitsharesManager.BitsharesManager.issue()` functions for more details.

send_type in the returned dict will be either 'send' or 'issue' depending on the operation performed

3.2.1.1.2.5 BitsharesMixin module

Copyright:

```
+=====================================================+
|                 © 2019 Privex Inc.                  |
|               https://www.privex.io                 |
+=====================================================+
|                                                     |
|        CryptoToken Converter                        |
|                                                     |
|        Core Developer(s):                           |
|                                                     |
|          (+)  Chris (@someguy123) [Privex]          |
|                                                     |
+=====================================================+
```

class `payments.coin_handlers.Bitshares.BitsharesMixin.`**BitsharesMixin**

Bases: `object`

BitsharesMixin - A class that provides shared functionality used by both BitsharesLoader and BitsharesManager.

Main features:

```
- Access the BitShares shared instance via :py:attr:`.bitshares`
- Access the Blockchain shared instance via :py:attr:`.blockchain`
- Safely get Bitshares network data (account data, asset data, and block␣
↪timestamps)
```

Copyright:

```
+=====================================================+
|                 © 2019 Privex Inc.                  |
|               https://www.privex.io                 |
+=====================================================+
|                                                     |
|        CryptoToken Converter                        |
|                                                     |
|        Core Developer(s):                           |
|                                                     |
|          (+)  Chris (@someguy123) [Privex]          |
```

(continues on next page)

(continued from previous page)

```
|                                                      |
+=====================================================+
```

bitshares
Returns an instance of BitShares and caches it in the attribute _bitshares after creation

blockchain
Returns an instance of Blockchain and caches it in the attribute _blockchain after creation

get_account_obj(*account_name*) → bitshares.account.Account
If an account exists on Bitshares, will return a `bitshares.account.Account` object. Otherwise None.

Parameters **account_name** – Bitshares account to get data for

:return Account or None

get_asset_obj(*symbol*) → bitshares.asset.Asset
If a token symbol exists on Bitshares, will return a `bitshares.asset.Asset` object. Otherwise None.

Parameters **symbol** – Bitshares token to get data for (can be symbol name or id)

:return Asset or None

get_block_timestamp(*block_number*) → int
Given a block number, returns the timestamp of that block. If block number is invalid or an error happens, returns 0.

Parameters **block_number** – block number to get data for

:return int

get_decimal_from_amount(*amount_obj: bitshares.amount.Amount*) → decimal.Decimal
Helper function to convert a Bitshares Amount object into a Decimal

get_private_key(*account_name*, *key_type*) → str
Find the Bitshares `models.CryptoKeyPair` in the database for a given account *account_name* and key type 'key_type' (e.g. 'active' or 'memo'), decrypt the private key, then return the plaintext key.

If no matching key could be found, will raise an AuthorityMissing exception.

Parameters

- **account_name** (*str*) – The Bitshares account to find a private key for
- **key_type** (*str*) – Key type to search for. Can be 'active', 'memo', or 'owner'

Raises

- ***AuthorityMissing*** – No key could be found for the given *account_name*
- **EncryptKeyMissing** – CTC admin did not set ENCRYPT_KEY in their *.env*, or it is invalid
- **EncryptionError** – Something went wrong while decrypting the private key (maybe ENCRYPT_KEY is invalid)

Return str key the plaintext key

set_wallet_keys(*account_name, key_types: List[str]*)
Retrieves a `models.CryptoKeyPair` for a given account *account_name* and each key type specified in the key_types list (e.g. 'active' or 'memo'). Each private key is then decrypted and added to the underlying Bitshares wallet for use in transactions.

If no matching key could be found, will raise an AuthorityMissing exception.

Parameters

- **account_name** (*str*) – The Bitshares account to set keys for
- **key_types** (*List*) – Key types to search for. Can include 'active', 'memo', or 'owner'

Raises

- ***AuthorityMissing*** – A key could be found for the given *account_name*
- **EncryptKeyMissing** – CTC admin did not set ENCRYPT_KEY in their *.env*, or it is invalid
- **EncryptionError** – Something went wrong while decrypting the private key (maybe ENCRYPT_KEY is invalid)

3.2.1.1.3 EOS Coin Handler

3.2.1.1.3.1 Module contents

EOS Coin Handler

This python module is a **Coin Handler** for Privex's CryptoToken Converter, designed to handle all required functionality for both receiving and sending tokens on the **EOS** network.

It will automatically handle any *payments.models.Coin* which has it's type set to `eos`

To use this handler, you must first create the base coin with symbol `EOS`:

```
Coin Name:  EOS
Symbol:     EOS
Our Account: (username of account used for sending/receiving native EOS token)
Custom JSON: {"contract": "eosio.token"}
```

To change the RPC node from the admin panel, simply set the host/port/username/password on the EOS Coin:

```
# Below are the defaults used if you don't configure the EOS coin:
Host: eos.greymass.com
Port: 443
User: (leave blank)
Pass: (leave blank)
Custom JSON: {"ssl": True}
```

Coin Settings (Custom JSON settings)

> **Tip:** You can override the defaults for all EOS coins by setting the `settings_json` for a coin with the symbol `EOS`.
>
> All *Coin*'s handled by the EOS handler will inherit the `EOS` coin's custom JSON settings, which can be overrided via the individual coin's `settings_json`.

You can set the following JSON keys inside of a *Coin*'s "settings_json" field to adjust settings such as the "contract account" for the token, whether or not to use SSL with the RPC node, as well as the precision (DP) of the coin, if it's different from the default of `4` decimal places.

Coin Key	Description
endpoint	(str) The base URI to query against, e.g. `/eos_rpc/`
ssl	(bool) Whether or not to use SSL (https). Boolean `true` or `false`
contract	(str) The contract account for this token, e.g. `eosio.token` or `steemenginex`
precision	(int) The precision (decimal places) of this coin (defaults to `4`)
load_method	(str) Either `actions` to use v1/history, or `pvx` to use Privex EOS History
history_url	(str) (if load_method is pvx) Privex history URL, e.g. `https://eos-history.privex.io`

Copyright:

```
+=====================================================+
|                 © 2019 Privex Inc.                  |
|               https://www.privex.io                 |
+=====================================================+
|                                                     |
|        CryptoToken Converter                        |
|                                                     |
|        Core Developer(s):                           |
|                                                     |
|          (+)  Chris (@someguy123) [Privex]          |
|                                                     |
+=====================================================+
```

`payments.coin_handlers.EOS.`**`reload`**`()`

Reload's the `provides` property for the loader and manager from the DB.

By default, since new tokens are constantly being created for EOS, our classes can provide for any `models.Coin` by scanning for coins with the type `eos`. This saves us from hard coding specific coin symbols.

3.2.1.1.3.2 Submodules

3.2.1.1.3.3 EOSLoader module

class `payments.coin_handlers.EOS.EOSLoader.`**`EOSLoader`**`(`*symbols*`)`

Bases: *payments.coin_handlers.base.BaseLoader.BaseLoader*, *payments.coin_handlers.EOS.EOSMixin.EOSMixin*

`clean_txs``(`*account, symbol, contract, transactions: Iterable[dict]*`)` → Generator[dict, None, None]

Filters a given Iterable of dict's containing raw EOS "actions":

- Finds only incoming *transfer* transactions from accounts that are not us (*account*)
- Filters transactions by both *symbol* and verifies they're from the *contract* account
- Outputs valid incoming TXs in the standardised Deposit format.

Parameters

- **`account`** (*str*) – The account which should be receiving *symbol*
- **`symbol`** (*str*) – The coin symbol to search for, e.g. "EOS"
- **`contract`** (*str*) – The EOS contract account to filter by, e.g. "eosio.token"
- **`transactions`** (*Iterable[dict]*) – An iterable list/generator of EOS actions as dict's

Return Generator cleaned_txs A generator yielding valid Deposit TXs as dict's

get_actions(*account: str*, *count=100*) → List[dict]

Loads EOS transactions for a given account, and caches them per account to avoid constant queries.

Parameters

- **account** – The EOS account to load transactions for
- **count** – Amount of transactions to load

Return list transactions A list of EOS transactions as dict's

list_txs(*batch=100*) → Generator[dict, None, None]

Get transactions for all coins in *self.coins* where the 'to' field matches coin.our_account If `load()` hasn't been ran already, it will automatically call self.load()

Parameters **batch** – Amount of transactions to load per batch

Returns Generator yielding dict's that conform to `models.Deposit`

load(*tx_count=1000*)

Prepares the loader by disabling any symbols / coin objects that don't have an *our_account* set, or don't have a *contract* set in either `models.Coin.settings_json` or `default_contracts`

Parameters **tx_count** – Amount of transactions to load per account, most recent first

Returns None

provides

pvx_clean_txs(*account, symbol, contract, transactions: Iterable[dict]*) → Generator[dict, None, None]

Filters a given Iterable of dict's containing EOS "actions" from Privex history API

- Finds only incoming *transfer* transactions from accounts that are not us (*account*)
- Filters transactions by both *symbol* and verifies they're from the *contract* account
- Outputs valid incoming TXs in the standardised Deposit format.

Parameters

- **account** (*str*) – The account which should be receiving *symbol*
- **symbol** (*str*) – The coin symbol to search for, e.g. "EOS"
- **contract** (*str*) – The EOS contract account to filter by, e.g. "eosio.token"
- **transactions** (*Iterable[dict]*) – An iterable list/generator of EOS actions as dict's

Return Generator cleaned_txs A generator yielding valid Deposit TXs as dict's

pvx_get_actions(*account: str*, *count=100*, *symbol=None*, *contract=None*, *history_url=None*) → List[dict]

Loads EOS transactions for a given account, and caches them per account to avoid constant queries.

Parameters

- **account** – The EOS account to load transactions for
- **count** – Amount of transactions to load

Return list transactions A list of EOS transactions as dict's

3.2.1.1.3.4 EOSManager module

class payments.coin_handlers.EOS.EOSManager.**EOSManager**(*symbol: str*)

Bases: *payments.coin_handlers.base.BaseManager.BaseManager*, *payments.coin_handlers.EOS.EOSMixin.EOSMixin*

address_valid(**addresses*) → bool

Check if one or more account usernames exist on the EOS network.

Example:

```
>>> if not self.address_valid('someguy12333', 'steemenginex'):
...     print('The EOS account "someguy12333" and/or "steemenginex" does not␣
↪exist. ')
```

Parameters **addresses** (*str*) – One or more EOS usernames to verify the existence of

Return bool account_exists True if all of the given accounts in *addresses* exist on the EOS network.

Return bool account_exists False if at least one account in *addresses* does not exist on EOS.

address_valid_ex(**addresses*)

Check if one or more account usernames exist on the EOS network. Throws an exception if any do not exist.

A slightly different version of *address_valid()* which raises AccountNotFound with the username that failed the test, instead of simply returning True / False.

Parameters **addresses** (*str*) – One or more EOS usernames to verify the existence of

Raises ***AccountNotFound*** – When one of the accounts in *addresses* does not exist.

balance(*address: str = None*, *memo: str = None*, *memo_case: bool = False*) → decimal.Decimal

Return the balance of *self.symbol* for our "wallet", or a given address/account, optionally filtered by memo

Parameters

- **address** – The address or account to get the balance for. If None, return our total wallet (or default account) balance.
- **memo** – If not None (and coin supports memos), return the total balance of a given memo
- **memo_case** – Whether or not to total memo's case sensitive, or not. False = case-insensitive memo

Raises ***AccountNotFound*** – The requested account/address doesn't exist

Return Decimal Decimal() balance of address/account, optionally balance (total received) of a given memo

build_tx(*tx_type*, *contract*, *sender*, *tx_args: dict*, *key_types=None*, *broadcast: bool = True*) → dict

Crafts an EOS transaction using the various arguments, signs it using the stored private key for *sender*, then broadcasts it (if *broadcast* is True) and returns the result.

Example:

```
>>> args = {"from": "someguy12333", "to": "steemenginex", "quantity": "1.000␣
↪EOS", "memo": ""}
>>> res = self.build_tx('transfer', 'eosio.token', 'someguy12333', args)
>>> print(res['transaction_id'])
dc9ece0dfb8da0b92068e23bdc22c971e0bc713d31ffc1b7552a861197b0d23e
```

Parameters

- **tx_type** (*str*) – The type of transaction, e.g. "transfer" or "issue"
- **contract** (*str*) – The contract username to execute against, e.g. 'eosio.token'
- **sender** (*str*) – The account name that will be signing the transaction, will auto lookup it's private key
- **tx_args** (*dict*) – A dictionary of transaction arguments to add to the payload data
- **key_types** (*list*) – (optional) Which types of key can be used for this TX? e.g. ['owner', 'active']
- **broadcast** (*bool*) – (default: True) If true, broadcasts the TX after signing. Otherwise returns just the signed TX and does not broadcast it to the network.

Return dict tfr The results of the transaction. Includes information about the broadcast if it was sent.

can_issue = True

get_deposit() → tuple

Return tuple If the coin uses addresses, this method should return a tuple of ('address', coin_address)

Return tuple If the coin uses accounts/memos, this method should return a tuple ('account', receiving_account) The memo will automatically be generated by the calling function.

classmethod get_privkey(*from_account: str*, *key_types: list = None*) → Tuple[str, str]

Find the EOS `models.CryptoKeyPair` in the database for a given account *from_account* , decrypt the private key, then returns a tuple containing (key_type:str, priv_key:str,)

If no matching key pair could be found, will raise an AuthorityMissing exception.

Example:

```
>>> key_type, priv_key = EOSManager.get_privkey('steemenginex', key_types=[
↪'active'])
>>> print(key_type)
active
>>> print(priv_key)  # The below private key was randomly generated for this␣
↪pydoc block, is isn't a real key.
5KK4oSvg9n5NxiAK9CXRd7zhbARpx8oxh15miPTXW8htGbYQPKD
```

Parameters

- **from_account** (*str*) – The EOS account to find a private key for
- **key_types** (*list*) – (optional) A list() of key types to search for. Default: ['active', 'owner']

Raises

- ***AuthorityMissing*** – No key pair could be found for the given *from_account*
- **EncryptKeyMissing** – CTC admin did not set ENCRYPT_KEY in their *.env*, or it is invalid
- **EncryptionError** – Something went wrong while decrypting the private key (maybe ENCRYPT_KEY is invalid)

Return tuple k A tuple containing the key type (active/owner etc.) and the private key.

issue (*amount: decimal.Decimal*, *address: str*, *memo: str = None*, *trigger_data=None*)

Issue (create/print) tokens to a given address/account, optionally specifying a memo if supported

Parameters

- **amount** (*Decimal*) – Amount of tokens to issue, as a Decimal()
- **address** – Address or account to issue the tokens to
- **memo** – Memo to issue tokens with (if supported)
- **trigger_data** (*dict*) – Metadata related to this issue transaction (e.g. the deposit that triggered this)

Raises

- ***IssuerKeyError*** – Cannot issue because we don't have authority to (missing key etc.)
- ***IssueNotSupported*** – Class does not support issuing, or requested symbol cannot be issued.
- ***AccountNotFound*** – The requested account/address doesn't exist

Return dict Result Information

Format:

```
dict {
    txid:str - Transaction ID - None if not known,
    coin:str - Symbol that was sent,
    amount:Decimal - The amount that was sent (after fees),
    fee:Decimal    - TX Fee that was taken from the amount,
    from:str       - The account/address the coins were issued from.
                     If it's not possible to determine easily, set this to␣
↪None.
    send_type:str  - Should be statically set to "issue"
}
```

provides

send (*amount*, *address*, *from_address=None*, *memo=None*, *trigger_data=None*) → dict

Send a given `amount` of EOS (or a token on EOS) from `from_address` to `address` with the memo `memo`.

Only `amount` and `address` are mandatory.

Parameters

- **amount** (*Decimal*) – Amount of coins/tokens to send, as a Decimal()
- **address** (*str*) – Destination EOS account to send the coins/tokens to
- **memo** (*str*) – Memo to send coins/tokens with (default: "")
- **from_address** (*str*) – EOS Account to send from (default: uses Coin.our_account)

Raises

- ***AuthorityMissing*** – Cannot send because we don't have authority to (missing key etc.)
- ***AccountNotFound*** – The requested account doesn't exist
- ***NotEnoughBalance*** – Sending account/address does not have enough balance to send

Return dict Result Information

Format:

```
dict {
  txid:str       - Transaction ID - None if not known,
  coin:str       - Symbol that was sent,
  amount:Decimal - The amount that was sent (after fees),
  fee:Decimal    - TX Fee that was taken from the amount (static Decimal(0)␣
↪for EOS)
  from:str       - The account the coins were sent from.
  send_type:str  - Statically set to "send"
}
```

send_or_issue(*amount*, *address*, *memo=None*, *trigger_data=None*) → dict

Attempt to send an amount to an address/account, if not enough balance, attempt to issue it instead. You may override this method if needed.

Parameters

- **amount** (*Decimal*) – Amount of coins/tokens to send/issue, as a Decimal()
- **address** – Address or account to send/issue the coins/tokens to
- **memo** – Memo to send/issue coins/tokens with (if supported)
- **trigger_data** (*dict*) – Metadata related to this issue transaction (e.g. the deposit that triggered this)

Raises

- ***IssuerKeyError*** – Cannot issue because we don't have authority to (missing key etc.)
- ***IssueNotSupported*** – Class does not support issuing, or requested symbol cannot be issued.
- ***AccountNotFound*** – The requested account/address doesn't exist

Return dict Result Information

Format:

```
dict {
  txid:str       - Transaction ID - None if not known,
  coin:str       - Symbol that was sent,
  amount:Decimal - The amount that was sent (after fees),
  fee:Decimal    - TX Fee that was taken from the amount,
  from:str       - The account(s)/address(es) the coins were sent from. if␣
↪more than one, comma separated.
                   If it's not possible to determine easily, set this to None.
  send_type:str  - Should be set to "send" if the coins were sent, or "issue"␣
↪if the coins were issued.
}
```

validate_amount(*amount: Union[decimal.Decimal, float, str], from_account: str = None*) → decimal.Decimal

Validates a user specified EOS token amount by:

- if amount is a float, we round it down to a 4 DP string
- we then pass the amount to Decimal so we can perform more precise calculations
- checks that the amount is at least 0.0001 (minimum amount of EOS that can be sent)
- if *from_account* is specified, will raise NotEnoughBalance if we don't have enough balance to cover the TX.

Example:

```
>>> amount = self.validate_amount(1.23)
>>> amount
Decimal('1.23')
```

Parameters

- **amount** (*Decimal*) – The amount of EOS (or token) to be sent, ideally as Decimal (but works with float/str)
- **from_account** (*str*) – (optional) If specified, check that *from_account* has enough balance for this TX.

Raises

- **ArithmeticError** – When the amount is lower than the lowest amount allowed by the token's precision
- ***NotEnoughBalance*** – The account *from_account* does not have enough balance to send this amount.
- ***TokenNotFound*** – *from_account* does not have a listed balance of *self.symbol*

Return Decimal amount The *amount* after sanitization, converted to a Decimal

3.2.1.1.3.5 EOSMixin module

Copyright:

```
+=====================================================+
|                 © 2019 Privex Inc.                  |
|               https://www.privex.io                 |
+=====================================================+
|                                                     |
|        CryptoToken Converter                        |
|                                                     |
|        Core Developer(s):                           |
|                                                     |
|          (+)  Chris (@someguy123) [Privex]          |
|                                                     |
+=====================================================+
```

class payments.coin_handlers.EOS.EOSMixin.**EOSMixin**

Bases: *payments.coin_handlers.base.SettingsMixin.SettingsMixin*

EOSMixin - A child class of SettingsMixin that is used by both EOSLoader and EOSManager for shared functionality.

Main features:

```
- Access the EOS shared instance via :py:attr:`.eos`
- Get the general ``EOS`` symbol coin settings via :py:attr:`.eos_settings`
- Access individual token settings (e.g. contract) via ``self.settings[symbol]``
- Helper method :py:meth:`.get_contract` - get contract via DB, or fall back to␣
↪:py:attr:`.default_contracts`
- Automatically sets setting defaults, such as the RPC node (using Greymass node␣
↪over SSL)
```

Copyright:

```
+====================================================+
|                 © 2019 Privex Inc.                 |
|               https://www.privex.io                |
+====================================================+
|                                                    |
|        CryptoToken Converter                       |
|                                                    |
|        Core Developer(s):                          |
|                                                    |
|          (+)  Chris (@someguy123) [Privex]         |
|                                                    |
+====================================================+
```

all_coins

Ensures that the coin 'EOS' always has it's settings loaded by `base.SettingsMixin` by overriding this method `all_coins` to inject the coin EOS if it's not our symbol.

Return dict coins A dict<str,Coin> of supported coins, mapped by symbol

chain = 'eos'

This controls the name of the chain and is used for logging, cache keys etc. It may be converted to upper case for logging, and lower case for cache keys. Forks of EOS may override this when sub-classing EOSMixin to adjust logging, cache keys etc.

chain_coin = 'EOS'

Forks of EOS may override this when sub-classing EOSMixin to change the native coin symbol of the network

chain_type = 'eos'

Used for looking up 'coin_type=xxx' Forks of EOS should override this to match the coin_type they use for *provides* generation

current_rpc = None

Contains the current EOS API node as a string

default_contracts = {'EOS': 'eosio.token'}

To make it easier to add common tokens on the EOS network, the loader/manager will fallback to this map between symbols and contracts.

This means that you don't have to set the contract in the custom JSON for popular tokens in this list, such as the native EOS token (which uses the contract account eosio.token).

eos

Returns an instance of Cleos and caches it in the attribute _eos after creation

eos_settings

Since EOS deals with tokens under one network, this is a helper property to quickly get the base EOS settings

Return dict settings A map of setting keys to their values

get_contract(*symbol: str*) → str

Attempt to find the contract account for a given token symbol, searches the database Coin objects first using *settings* - if not found, falls back to *default_contracts*

Example usage:

```
>>> contract_acc = self.get_contract('EOS')
>>> print(contract_acc)
 eosio.token
```

Parameters **symbol** (*str*) – The token symbol to find the contract for, e.g. `EOS`

Raises

- ***TokenNotFound*** – The given `symbol` does not exist in self.settings
- ***MissingTokenMetadata*** – Could not find contract in DB coin settings nor default_contracts

Return str contract_acc The contract username as a string, e.g. `eosio.token`

provides = ['EOS']

This attribute is automatically generated by scanning for `models.Coin` s with the type `eos`. This saves us from hard coding specific coin symbols. See __init__.py for populating code.

replace_eos(***conn*) → eospy.cleos.Cleos

Destroy the EOS `Cleos` instance at `_eos` and re-create it with the modified connection settings `conn`

Also returns the EOS instance for convenience.

Only need to specify settings you want to override.

Example:

```
>>> eos = self.replace_eos(host='example.com', port=80, ssl=False)
>>> eos.get_account('someguy123')
```

Parameters **conn** – Connection settings. Keys: endpoint, ssl, host, port, username, password

Return Cleos eos A `Cleos` instance with the modified connection settings.

setting_defaults = {'endpoint': '/', 'history_url': 'https://eos-history.privex.io'

Default settings to use if any required values are empty, e.g. default to Greymass's RPC node

`load_method` can be either `pvx` for Privex EOS History API, or `actions` to use v1/history from the RPC node.

settings

Get all settings, mapped by coin symbol (each coin symbol dict contains custom json settings merged)

Return dict settings A dictionary mapping coin symbols to settings

url

Creates a URL from the host settings on the EOS coin

3.2.1.1.4 Hive Coin Handler

3.2.1.1.4.1 Module contents

Hive Coin Handler

This python module is a **Coin Handler** for Privex's CryptoToken Converter, designed to handle all required functionality for both receiving and sending tokens on the **Hive** network.

It will automatically handle any *`payments.models.Coin`* which has it's type set to `hivebase`

Coin object settings:

For each *payments.models.Coin* you intend to use with this handler, you should configure it as such:

Coin Key	Description
coin_type	This should be set to `Hive Network (or compatible fork)` (db value: hivebase)
our_account	This should be set to the username of the account you want to use for receiving/sending
setting_json	A JSON string for optional extra config (see below)

Extra JSON (Handler Custom) config options:

- `rpcs` - A JSON list<str> of RPC nodes to use, with a full HTTP/HTTPS URL. If this is not specified, Beem will automatically try to use the best available RPC node for the Hive network.
- `pass_store` - Generally you do not need to touch this. It controls where Beem will look for the wallet password. It defaults to `environment`

Example JSON custom config:

```
{
    "rpcs": [
        "https://steemd.privex.io",
        "https://api.steemit.com",
        "https://api.steem.house"
    ],
    "pass_store": "environment"
}
```

Copyright:

```
+=====================================================+
|                 © 2019 Privex Inc.                  |
|               https://www.privex.io                 |
+=====================================================+
|                                                     |
|        CryptoToken Converter                        |
|                                                     |
|        Core Developer(s):                           |
|                                                     |
|          (+)  Chris (@someguy123) [Privex]          |
|                                                     |
+=====================================================+
```

`payments.coin_handlers.Hive.`**`reload`**`()`

Reload's the `provides` property for the loader and manager from the DB.

By default, since new Hive forks are constantly being created, our classes can provide for any `models.Coin` by scanning for coins with the type `hivebase`. This saves us from hard coding specific coin symbols.

3.2.1.1.4.2 Submodules

3.2.1.1.4.3 HiveLoader module

class payments.coin_handlers.Hive.HiveLoader.**HiveLoader**(*symbols*)

Bases: payments.coin_handlers.base.BaseLoader.BaseLoader, payments.coin_handlers.Hive.HiveMixin.HiveMixin

clean_tx(*tx: dict, symbol: str, account: str, memo: str = None, memo_case: bool = False*) → Optional[dict]

Filters an individual transaction. See clean_txs() for info

clean_txs(*symbol: str, transactions: Iterable[dict], account: str = None*) → Generator[dict, None, None]

Filters a list of transactions *transactions* as required, yields dict's conforming with models.Deposit

- Filters out transactions that are not marked as 'receive'
- Filters out mining transactions
- Filters by address if *account* is specified
- Filters out transactions that don't have enough confirms, and are not reported as 'trusted'

Parameters

- **symbol** – Symbol of coin being cleaned
- **transactions** – A list<dict> or generator producing dict's
- **account** – If not None, only return TXs sent to this address.

Return Generator<dict> A generator outputting dictionaries formatted as below

Output Format:

```
{
  txid:str, coin:str (symbol), vout:int,
  tx_timestamp:datetime, address:str, amount:Decimal
}
```

list_txs(*batch=0*) → Generator[dict, None, None]

The list_txs function processes the transaction data from load(), as well as handling any pagination, if it's required (e.g. only retrieve *batch* transactions at a time from the data source)

It should first check that load() has been ran if it's required, if the data required has not been loaded, it should call self.load()

To prevent memory leaks, this must be a generator function.

Below is an example of a generator function body, it loads *batch* transactions from the full transaction list, pretends to processes them into *txs*, yields them, then loads another batch after the calling function has iterated over the current *txs*

```
>>> t = self.transactions   # All transactions
>>> b = batch
>>> finished = False
>>> offset = 0
>>> # To save memory, process 100 transactions per iteration, and yield them␣
↪(generator)
>>> while not finished:
```

(continues on next page)

(continued from previous page)

```
>>>     txs = []     # Processed transactions
>>>     # If there are less remaining TXs than batch size, get remaining txs␣
↪and finish.
>>>     if (len(t) - offset) < batch:
>>>         finished = True
>>>     # Do some sort-of processing on the tx to make it conform to␣
↪`Deposit`, then append to txs
>>>     for tx in t[offset:offset + batch]:
>>>         txs.append(tx)
>>>     offset += b
>>>     for tx in txs:
>>>         yield tx
>>>     # At this point, the current batch is exhausted. Destroy the tx array␣
↪to save memory.
>>>     del txs
```

Parameters **batch** (*int*) – Amount of transactions to process/load per each batch

Returns Generator A generator returning dictionaries that can be imported into `models.Deposit`

Dict format:

```
{txid:str, coin:str (symbol), vout:int, tx_timestamp:datetime,
 address:str, from_account:str, to_account:str, memo:str, amount:Decimal}
```

`vout` is optional. One of either {from_account, to_account, memo} OR {address} must be included.

load(*tx_count=10000*)

The load function should prepare your loader, by either importing all of the data required for filtering, or setting up a generator for the *`list_txs()`* method to load them paginated.

It does NOT return anything, it simply creates any connections required, sets up generator functions if required for paginating the data, and/or pre-loads the first batch of transaction data.

Parameters **tx_count** – The total amount of transactions that should be loaded PER SYMBOL, most recent first.

Returns None

provides

This attribute is automatically generated by scanning for `models.Coin` s with the type `steembase`. This saves us from hard coding specific coin symbols. See __init__.py for populating code.

settings

To ensure we always get fresh settings from the DB after a reload

3.2.1.1.4.4 HiveManager module

class payments.coin_handlers.Hive.HiveManager.**HiveManager**(*symbol: str*)

Bases: *payments.coin_handlers.base.BaseManager.BaseManager*, *payments.coin_handlers.Hive.HiveMixin.HiveMixin*

address_valid(*address*) → bool

If an account exists on Steem, will return True. Otherwise False.

Parameters **address** – Steem account to check existence of

Return bool True if account exists, False if it doesn't

balance(*address: str = None, memo: str = None, memo_case: bool = False*) → decimal.Decimal

Get token balance for a given Steem account, if memo is given - get total symbol amt received with this memo.

Parameters

- **address** – Steem account to get balance for, if not set, uses self.coin.our_account
- **memo** – If not None, get total *self.symbol* received with this memo.
- **memo_case** – Case sensitive memo search

Returns Decimal(balance)

get_deposit() → tuple

Returns the deposit account for this symbol

Return tuple A tuple containing ('account', receiving_account). The memo must be generated by the calling function.

health() → Tuple[str, tuple, tuple]

Return health data for the passed symbol.

Health data will include: 'Symbol', 'Status', 'Coin Name', 'API Node', 'Head Block', 'Block Time', 'RPC Version', 'Our Account', 'Our Balance' (all strings)

Return tuple health_data (manager_name:str, headings:list/tuple, health_data:list/tuple,)

health_test() → bool

Check if our Steem node works or not, by requesting basic information such as the current block + time, and checking if our sending/receiving account exists on Steem.

Return bool True if Steem appears to be working, False if it seems to be broken.

provides

This attribute is automatically generated by scanning for `models.Coin` s with the type `steembase`. This saves us from hard coding specific coin symbols. See __init__.py for populating code.

send(*amount: decimal.Decimal, address: str, from_address: str = None, memo=None, trigger_data=None*) → dict

Send a supported currency to a given address/account, optionally specifying a memo if supported

Example - send 1.23 STEEM from @someguy123 to @privex with memo 'hello'

```
>>> s = SteemManager('STEEM')
>>> s.send(from_address='someguy123', address='privex', amount=Decimal('1.23
↪'), memo='hello')
```

Parameters

- **amount** (*Decimal*) – Amount of currency to send, as a Decimal()
- **address** – Account to send the currency to
- **from_address** – Account to send the currency from
- **memo** – Memo to send currency with

Raises

- **AttributeError** – When both *from_address* and *self.coin.our_account* are blank.
- **ArithmeticError** – When the amount is lower than the lowest amount allowed by the asset's precision

- ***AuthorityMissing*** – Cannot send because we don't have authority to (missing key etc.)
- ***AccountNotFound*** – The requested account doesn't exist
- ***NotEnoughBalance*** – The account *from_address* does not have enough balance to send this amount.

Return dict Result Information

Format:

```
{
    txid:str - Transaction ID - None if not known,
    coin:str - Symbol that was sent,
    amount:Decimal - The amount that was sent (after fees),
    fee:Decimal    - TX Fee that was taken from the amount,
    from:str       - The account/address the coins were sent from,
    send_type:str       - Should be statically set to "send"
}
```

3.2.1.1.4.5 HiveMixin module

class payments.coin_handlers.Hive.HiveMixin.**HiveMixin**(**args*, ***kwargs*)

Bases: *payments.coin_handlers.base.SettingsMixin.SettingsMixin*

HiveMixin - Shared code between SteemManager and SteemLoader

Designed for the Steem Network with SBD and STEEM support. May or may not work with other Graphene coins.

Copyright:

```
+=====================================================+
|                 © 2019 Privex Inc.                  |
|               https://www.privex.io                 |
+=====================================================+
|                                                     |
|        CryptoToken Converter                        |
|                                                     |
|        Core Developer(s):                           |
|                                                     |
|          (+)  Chris (@someguy123) [Privex]          |
|                                                     |
+=====================================================+
```

For **additional settings**, please see the module docstring in `coin_handlers.Steem`

asset

Easy reference to the BSteem Asset object for our current symbol

find_steem_tx(*tx_data*, *last_blocks=15*) → Optional[dict]

Used internally to get the transaction ID after a transaction has been broadcasted

Parameters

- **tx_data** (*dict*) – Transaction data returned by a beem broadcast operation, must include 'signatures'
- **last_blocks** (*int*) – Amount of previous blocks to search for the transaction

Return dict Transaction data from the blockchain {transaction_id, ref_block_num, ref_block_prefix, expiration, operations, extensions, signatures, block_num, transaction_num}

Return None If the transaction wasn't found, None will be returned.

get_rpc(*symbol: str*) → beem.steem.Steem

Returns a Steem instance for querying data and sending TXs. By default, uses the BSteem shared_steem_instance.

If a custom RPC list is specified in the Coin "custom json" settings, a new instance will be returned with the RPCs specified in the json.

Parameters **symbol** – Coin symbol to get BSteem RPC instance for

Return beem.steem.Steem An instance of `beem.steem.Steem` for querying

precision

rpc

3.2.1.1.5 SteemEngine Coin Handler

3.2.1.1.5.1 Module contents

SteemEngine Coin Handler

This python module is a **Coin Handler** for Privex's CryptoToken Converter, designed to handle all required functionality for both receiving and sending tokens on the **SteemEngine** network.

It will automatically handle any *`payments.models.Coin`* which has it's type set to `steemengine`

Global Settings

The following global settings are used by this handler (set within *`steemengine.settings.custom`*). All of the below settings can be specified with the same key inside of your `.env` file to override the defaults.

Setting	Description
`SENG_RPC_NODE`	The hostname for the contract API server, e.g. `api.steem-engine.com`
`SENG_RPC_URL`	The URL for the contract API e.g. `/rpc/contracts`
`SENG_HISTORY_NODE`	The hostname for the history API server, e.g. `api.steem-engine.com`
`SENG_HISTORY_URL`	The URL for the history API e.g. `accounts/history`
`SENG_NETWORK_ACCOUNT`	The "network account" for SteemEngine, e.g. `ssc-mainnet1`

Coin object settings:

You can set the following JSON keys inside of a *`Coin`*'s "settings_json" field if you want to use an alternative SteemEngine RPC node, or history node just for that coin.

Coin Key	Description
rpc_node	The hostname for the contract API server, e.g. `api.steem-engine.com`
rpc_url	The URL for the contract API e.g. `/rpc/contracts`
history_node	The hostname for the history API server, e.g. `api.steem-engine.com`
history_url	The URL for the history API e.g. `accounts/history`
network_account	The "network account" for SteemEngine, e.g. `ssc-mainnet1`

For example, placing the following JSON inside of `settings_json` for a certain coin, would result in the contract API `https://api.hive-engine.com/contracts` and history API `https://accounts.hive-engine.com/accountHistory` being used only for this particular coin, while coins without any `settings_json` overrides would continue using the global `SENG_RPC_NODE` etc.

```
{
    "rpc_node": "api.hive-engine.com",
    "rpc_url": "/contracts",
    "history_node": "accounts.hive-engine.com",
    "history_url": "accountHistory"
}
```

Copyright:

```
+====================================================+
|                 © 2019 Privex Inc.                 |
|               https://www.privex.io                |
+====================================================+
|                                                    |
|        CryptoToken Converter                       |
|                                                    |
|        Core Developer(s):                          |
|                                                    |
|          (+)  Chris (@someguy123) [Privex]         |
|                                                    |
+====================================================+
```

`payments.coin_handlers.SteemEngine.`**`reload`**`()`

Reload's the `provides` property for the loader and manager from the DB.

By default, since new tokens are constantly being created for SteemEngine, our classes can provide for any `models.Coin` by scanning for coins with the type `steemengine`. This saves us from hard coding specific coin symbols.

3.2.1.1.5.2 Submodules

3.2.1.1.5.3 SteemEngineLoader module

Copyright:

```
+====================================================+
|                 © 2019 Privex Inc.                 |
|               https://www.privex.io                |
+====================================================+
|                                                    |
|        CryptoToken Converter                       |
|                                                    |
|        Core Developer(s):                          |
|                                                    |
|          (+)  Chris (@someguy123) [Privex]         |
|                                                    |
+====================================================+
```

class `payments.coin_handlers.SteemEngine.SteemEngineLoader.`**`SteemEngineLoader`**(*symbols*)

Bases: *payments.coin_handlers.base.BaseLoader.BaseLoader*, *payments.coin_handlers.SteemEngine.SteemEngineMixin.SteemEngineMixin*

This class handles loading transactions for the **SteemEngine** network, and can support almost any token on SteemEngine.

Copyright:

```
+====================================================+
|                 © 2019 Privex Inc.                 |
|               https://www.privex.io                |
+====================================================+
|                                                    |
|        CryptoToken Converter                       |
|                                                    |
|        Core Developer(s):                          |
|                                                    |
|          (+)  Chris (@someguy123) [Privex]         |
|                                                    |
+====================================================+
```

clean_txs(*account: str, symbol: str, transactions: Iterable[privex.steemengine.objects.SETransaction]*) → Generator[dict, None, None]

Filters a list of transactions by the receiving account, yields dict's conforming with `payments.models.Deposit`

Parameters

- **account** (*str*) – The 'to' account to filter by
- **symbol** (*str*) – The symbol of the token being filtered
- **transactions** (*list<dict>*) – A list<dict> of transactions to filter

Returns A generator yielding `dict` s conforming to `payments.models.Deposit`

list_txs(*batch=100*) → Generator[dict, None, None]

Get transactions for all coins in *self.coins* where the 'to' field matches coin.our_account If `load()` hasn't been ran already, it will automatically call self.load() :param batch: Amount of transactions to load per batch :return: Generator yielding dict's that conform to `models.Deposit`

load(*tx_count=1000*)

The load function should prepare your loader, by either importing all of the data required for filtering, or setting up a generator for the `list_txs()` method to load them paginated.

It does NOT return anything, it simply creates any connections required, sets up generator functions if required for paginating the data, and/or pre-loads the first batch of transaction data.

Parameters **tx_count** – The total amount of transactions that should be loaded PER SYMBOL, most recent first.

Returns None

load_batch(*account, symbol, limit=100, offset=0, retry=0*)

Load SteemEngine transactions for account/symbol into self.transactions with automatic retry on error

provides

This attribute is automatically generated by scanning for `models.Coin` s with the type `steemengine`. This saves us from hard coding specific coin symbols. See __init__.py for populating code.

3.2.1.1.5.4 SteemEngineManager module

Copyright:

```
+====================================================+
|                 © 2019 Privex Inc.                 |
|               https://www.privex.io                |
+====================================================+
|                                                    |
|        CryptoToken Converter                       |
|                                                    |
|        Core Developer(s):                          |
|                                                    |
|          (+)  Chris (@someguy123) [Privex]         |
|                                                    |
+====================================================+
```

class payments.coin_handlers.SteemEngine.SteemEngineManager.**SteemEngineManager**(*symbol: str*)

Bases: *payments.coin_handlers.base.BaseManager.BaseManager*, *payments.coin_handlers.SteemEngine.SteemEngineMixin.SteemEngineMixin*

This class handles various operations for the **SteemEngine** network, and supports almost any token on SteemEngine.

It handles:

- Validating source/destination accounts
- Checking the balance for a given account, as well as the total amount received with a certain `memo`
- Issuing tokens to users
- Sending tokens to users

Copyright:

```
+====================================================+
|                 © 2019 Privex Inc.                 |
|               https://www.privex.io                |
+====================================================+
|                                                    |
|        CryptoToken Converter                       |
|                                                    |
|        Core Developer(s):                          |
|                                                    |
|          (+)  Chris (@someguy123) [Privex]         |
|                                                    |
+====================================================+
```

address_valid(*address*) → bool

If an account (`address` param) exists on Steem, will return True. Otherwise False.

balance(*address: str = None*, *memo: str = None*, *memo_case: bool = False*) → decimal.Decimal

Get token balance for a given Steem account, if memo is given - get total symbol amt received with this memo.

Parameters

- **address** – Steem account to get balance for, if not set, uses self.coin.our_account
- **memo** – If not None, get total *self.symbol* received with this memo.
- **memo_case** – Case sensitive memo search

Returns Decimal(balance)

get_deposit() → tuple
Returns the deposit account for this symbol

Return tuple A tuple containing ('account', receiving_account). The memo must be generated by the calling function.

health() → Tuple[str, tuple, tuple]
Return health data for the passed symbol.

Health data will include: Symbol, Status, Current Block, Node Version, Wallet Balance, and number of p2p connections (all as strings)

Return tuple health_data (manager_name:str, headings:list/tuple, health_data:list/tuple,)

health_test() → bool
Check if the SteemEngine API and Steem node works or not, by requesting basic information such as the token metadata, and checking if our sending/receiving account exists on Steem.

Return bool True if SteemEngine and Steem appear to be working, False if either is broken.

issue(*amount: decimal.Decimal*, *address: str*, *memo: str = None*, *trigger_data=None*) → dict
Issue (create/print) tokens to a given address/account, optionally specifying a memo if supported

Example - Issue 5.10 SGTK to @privex

```
>>> s = SteemEngineManager('SGTK')
>>> s.issue(address='privex', amount=Decimal('5.10'))
```

Parameters

- **amount** (*Decimal*) – Amount of tokens to issue, as a Decimal()
- **address** – Address or account to issue the tokens to
- **memo** – (ignored) Cannot issue tokens with a memo on SteemEngine

Raises

- ***IssuerKeyError*** – Cannot issue because we don't have authority to (missing key etc.)
- ***IssueNotSupported*** – Class does not support issuing, or requested symbol cannot be issued.
- ***AccountNotFound*** – The requested account/address doesn't exist

Return dict Result Information

Format:

```
{
    txid:str - Transaction ID - None if not known,
    coin:str - Symbol that was sent,
    amount:Decimal - The amount that was sent (after fees),
    fee:Decimal    - TX Fee that was taken from the amount,
    from:str       - The account/address the coins were issued from,
    send_type:str       - Should be statically set to "issue"
}
```

provides
This attribute is automatically generated by scanning for `models.Coin` s with the type `steemengine`. This saves us from hard coding specific coin symbols. See __init__.py for populating code.

send(*amount*, *address*, *memo=None*, *from_address=None*, *trigger_data=None*) → dict

Send tokens to a given address/account, optionally specifying a memo if supported

Example - send 1.23 SGTK from @someguy123 to @privex with memo 'hello'

```
>>> s = SteemEngineManager('SGTK')
>>> s.send(from_address='someguy123', address='privex', amount=Decimal('1.23
↪'), memo='hello')
```

Parameters

- **amount** (*Decimal*) – Amount of tokens to send, as a Decimal()
- **address** – Account to send the tokens to
- **from_address** – Account to send the tokens from
- **memo** – Memo to send tokens with (if supported)

Raises

- **AttributeError** – When both *from_address* and *self.coin.our_account* are blank.
- **ArithmeticError** – When the amount is lower than the lowest amount allowed by the token's precision
- ***AuthorityMissing*** – Cannot send because we don't have authority to (missing key etc.)
- ***AccountNotFound*** – The requested account/address doesn't exist
- ***TokenNotFound*** – When the requested token *symbol* does not exist
- ***NotEnoughBalance*** – The account *from_address* does not have enough balance to send this amount.

Return dict Result Information

Format:

```
{
    txid:str - Transaction ID - None if not known,
    coin:str - Symbol that was sent,
    amount:Decimal - The amount that was sent (after fees),
    fee:Decimal    - TX Fee that was taken from the amount,
    from:str       - The account/address the coins were sent from,
    send_type:str       - Should be statically set to "send"
}
```

send_or_issue(*amount*, *address*, *memo=None*, *trigger_data=None*) → dict

Attempt to send an amount to an address/account, if not enough balance, attempt to issue it instead. You may override this method if needed.

Parameters

- **amount** (*Decimal*) – Amount of coins/tokens to send/issue, as a Decimal()
- **address** – Address or account to send/issue the coins/tokens to
- **memo** – Memo to send/issue coins/tokens with (if supported)
- **trigger_data** (*dict*) – Metadata related to this issue transaction (e.g. the deposit that triggered this)

Raises

- ***IssuerKeyError*** – Cannot issue because we don't have authority to (missing key etc.)
- ***IssueNotSupported*** – Class does not support issuing, or requested symbol cannot be issued.
- ***AccountNotFound*** – The requested account/address doesn't exist

Return dict Result Information

Format:

```
dict {
  txid:str       - Transaction ID - None if not known,
  coin:str       - Symbol that was sent,
  amount:Decimal - The amount that was sent (after fees),
  fee:Decimal    - TX Fee that was taken from the amount,
  from:str       - The account(s)/address(es) the coins were sent from. if␣
↪more than one, comma separated.
                   If it's not possible to determine easily, set this to None.
  send_type:str  - Should be set to "send" if the coins were sent, or "issue"␣
↪if the coins were issued.
}
```

3.2.1.1.5.5 SteemEngineMixin module

Copyright:

```
+=====================================================+
|                 © 2019 Privex Inc.                  |
|               https://www.privex.io                 |
+=====================================================+
|                                                     |
|        CryptoToken Converter                        |
|                                                     |
|        Core Developer(s):                           |
|                                                     |
|          (+)  Chris (@someguy123) [Privex]          |
|                                                     |
+=====================================================+
```

class payments.coin_handlers.SteemEngine.SteemEngineMixin.**SteemEngineMixin**(**args*, ***kwargs*)

Bases: *payments.coin_handlers.base.SettingsMixin.SettingsMixin*

eng_rpc

get_rpc(*symbol: str*) → privex.steemengine.SteemEngineToken.SteemEngineToken

Returns a SteemEngineToken instance for querying data and sending TXs.

If a custom RPC config is specified in the Coin "custom json" settings, a new instance will be returned with the RPC config specified in the json.

Parameters **symbol** – Coin symbol to get Beem RPC instance for

Return beem.steem.Steem An instance of `beem.steem.Steem` for querying

payments.coin_handlers.SteemEngine.SteemEngineMixin.**mk_seng_rpc**(*rpc_settings: dict = None, **kwargs*) → privex.steemengine.SteemEngineToken

Get a `SteemEngineToken` instance using the default settings:

```
>>> rpc = mk_seng_rpc()
```

Get a `SteemEngineToken` instance using dictionary settings (rpc_settings):

```
>>> rpc2 = mk_seng_rpc({'rpc_node' : 'api.hive-engine.com','network_account' :
↪'ssc-testnet'})
```

Get a `SteemEngineToken` instance using individual kwarg settings:

```
>>> rpc3 = mk_seng_rpc(rpc_node='api.hive-engine.com', network_account='ssc-
↪testnet')
```

Parameters

- **rpc_settings** (*dict*) – Specify the settings as a dictionary (same keys as kwargs below)
- **kwargs** – Alternatively, specify the settings as keyword args
- **rpc_node** (*str*) – The hostname for the contract API server, e.g. `api.steem-engine.com`
- **rpc_url** (*str*) – The URL for the contract API e.g. `/rpc/contracts`
- **history_node** (*str*) – The hostname for the history API server, e.g. `api.steem-engine.com`
- **history_url** (*str*) – The URL for the history API e.g. `accounts/history`
- **network_account** (*str*) – The "network account" for SteemEngine, e.g. `ssc-mainnet1`
- **network** (*str*) – Chain to run on (`steem` or `hive`)

Return SteemEngineToken rpc An instance of `SteemEngineToken`

3.2.1.1.6 Steem Coin Handler

3.2.1.1.6.1 Module contents

Steem Coin Handler

This python module is a **Coin Handler** for Privex's CryptoToken Converter, designed to handle all required functionality for both receiving and sending tokens on the **Steem** network.

It will automatically handle any *`payments.models.Coin`* which has it's type set to `steembase`

Coin object settings:

For each *`payments.models.Coin`* you intend to use with this handler, you should configure it as such:

Coin Key	Description
coin_type	This should be set to `Steem Network (or compatible fork)` (db value: steembase)
our_account	This should be set to the username of the account you want to use for receiving/sending
setting_json	A JSON string for optional extra config (see below)

Extra JSON (Handler Custom) config options:

- `rpcs` - A JSON list<str> of RPC nodes to use, with a full HTTP/HTTPS URL. If this is not specified, Beem will automatically try to use the best available RPC node for the Steem network.
- `pass_store` - Generally you do not need to touch this. It controls where Beem will look for the wallet password. It defaults to `environment`

Example JSON custom config:

```
{
    "rpcs": [
        "https://steemd.privex.io",
        "https://api.steemit.com",
        "https://api.steem.house"
    ],
    "pass_store": "environment"
}
```

Copyright:

```
+===================================================+
|                 © 2019 Privex Inc.                |
|               https://www.privex.io               |
+===================================================+
|                                                   |
|        CryptoToken Converter                      |
|                                                   |
|        Core Developer(s):                         |
|                                                   |
|          (+)  Chris (@someguy123) [Privex]        |
|                                                   |
+===================================================+
```

`payments.coin_handlers.Steem.`**`reload`**`()`

Reload's the `provides` property for the loader and manager from the DB.

By default, since new Steem forks are constantly being created, our classes can provide for any `models.Coin` by scanning for coins with the type `steembase`. This saves us from hard coding specific coin symbols.

3.2.1.1.6.2 Submodules

3.2.1.1.6.3 SteemLoader module

Copyright:

```
+===================================================+
|                 © 2019 Privex Inc.                |
```

(continues on next page)

(continued from previous page)

```
|               https://www.privex.io                |
+===================================================+
|                                                   |
|        CryptoToken Converter                      |
|                                                   |
|        Core Developer(s):                         |
|                                                   |
|          (+)  Chris (@someguy123) [Privex]        |
|                                                   |
+===================================================+
```

class payments.coin_handlers.Steem.SteemLoader.**SteemLoader**(*symbols*)

Bases: *payments.coin_handlers.base.BaseLoader.BaseLoader*, *payments.coin_handlers.Steem.SteemMixin.SteemMixin*

SteemLoader - Loads transactions from the Steem network

Designed for the Steem Network with SBD and STEEM support. May or may not work with other Graphene coins.

Copyright:

```
+===================================================+
|                 © 2019 Privex Inc.                |
|               https://www.privex.io               |
+===================================================+
|                                                   |
|        CryptoToken Converter                      |
|                                                   |
|        Core Developer(s):                         |
|                                                   |
|          (+)  Chris (@someguy123) [Privex]        |
|                                                   |
+===================================================+
```

For **additional settings**, please see the module docstring in `coin_handlers.Steem`

clean_tx(*tx: dict, symbol: str, account: str, memo: str = None, memo_case: bool = False*) → Optional[dict]

Filters an individual transaction. See *clean_txs()* for info

clean_txs(*symbol: str, transactions: Iterable[dict], account: str = None*) → Generator[dict, None, None]

Filters a list of transactions *transactions* as required, yields dict's conforming with `models.Deposit`

- Filters out transactions that are not marked as 'receive'
- Filters out mining transactions
- Filters by address if *account* is specified
- Filters out transactions that don't have enough confirms, and are not reported as 'trusted'

Parameters

- **symbol** – Symbol of coin being cleaned
- **transactions** – A `list<dict>` or generator producing dict's
- **account** – If not None, only return TXs sent to this address.

Return Generator<dict> A generator outputting dictionaries formatted as below

Output Format:

```
{
  txid:str, coin:str (symbol), vout:int,
  tx_timestamp:datetime, address:str, amount:Decimal
}
```

list_txs(*batch=0*) → Generator[dict, None, None]

The list_txs function processes the transaction data from `load()`, as well as handling any pagination, if it's required (e.g. only retrieve *batch* transactions at a time from the data source)

It should first check that `load()` has been ran if it's required, if the data required has not been loaded, it should call self.load()

To prevent memory leaks, this must be a generator function.

Below is an example of a generator function body, it loads *batch* transactions from the full transaction list, pretends to processes them into *txs*, yields them, then loads another batch after the calling function has iterated over the current *txs*

```
>>> t = self.transactions   # All transactions
>>> b = batch
>>> finished = False
>>> offset = 0
>>> # To save memory, process 100 transactions per iteration, and yield them␣
↪(generator)
>>> while not finished:
>>>     txs = []     # Processed transactions
>>>     # If there are less remaining TXs than batch size, get remaining txs␣
↪and finish.
>>>     if (len(t) - offset) < batch:
>>>         finished = True
>>>     # Do some sort-of processing on the tx to make it conform to␣
↪`Deposit`, then append to txs
>>>     for tx in t[offset:offset + batch]:
>>>         txs.append(tx)
>>>     offset += b
>>>     for tx in txs:
>>>         yield tx
>>>     # At this point, the current batch is exhausted. Destroy the tx array␣
↪to save memory.
>>>     del txs
```

Parameters **batch** (*int*) – Amount of transactions to process/load per each batch

Returns Generator A generator returning dictionaries that can be imported into `models.Deposit`

Dict format:

```
{txid:str, coin:str (symbol), vout:int, tx_timestamp:datetime,
 address:str, from_account:str, to_account:str, memo:str, amount:Decimal}
```

`vout` is optional. One of either {from_account, to_account, memo} OR {address} must be included.

load(*tx_count=10000*)

The load function should prepare your loader, by either importing all of the data required for filtering, or setting up a generator for the `list_txs()` method to load them paginated.

It does NOT return anything, it simply creates any connections required, sets up generator functions if required for paginating the data, and/or pre-loads the first batch of transaction data.

Parameters **tx_count** – The total amount of transactions that should be loaded PER SYMBOL, most recent first.

Returns None

provides

This attribute is automatically generated by scanning for `models.Coin` s with the type `steembase`. This saves us from hard coding specific coin symbols. See __init__.py for populating code.

settings

To ensure we always get fresh settings from the DB after a reload

3.2.1.1.6.4 SteemManager module

Copyright:

```
+=====================================================+
|                 © 2019 Privex Inc.                  |
|               https://www.privex.io                 |
+=====================================================+
|                                                     |
|        CryptoToken Converter                        |
|                                                     |
|        Core Developer(s):                           |
|                                                     |
|          (+)  Chris (@someguy123) [Privex]          |
|                                                     |
+=====================================================+
```

class payments.coin_handlers.Steem.SteemManager.**SteemManager**(*symbol: str*)

Bases: *payments.coin_handlers.base.BaseManager.BaseManager*, *payments.coin_handlers.Steem.SteemMixin.SteemMixin*

This class handles various operations for the **Steem** network, and supports both STEEM and SBD.

It may or may not work with other Graphene coins, such as GOLOS / Whaleshares.

It handles:

- Validating source/destination accounts
- Checking the balance for a given account, as well as the total amount received with a certain `memo`
- Health checking
- Sending assets to users

Copyright:

```
+=====================================================+
|                 © 2019 Privex Inc.                  |
|               https://www.privex.io                 |
+=====================================================+
|                                                     |
|        CryptoToken Converter                        |
|                                                     |
|        Core Developer(s):                           |
```

(continues on next page)

(continued from previous page)

```
|                                                    |
|          (+)  Chris (@someguy123) [Privex]         |
|                                                    |
+====================================================+
```

address_valid(*address*) → bool

If an account exists on Steem, will return True. Otherwise False.

Parameters **address** – Steem account to check existence of

Return bool True if account exists, False if it doesn't

balance(*address: str = None, memo: str = None, memo_case: bool = False*) → decimal.Decimal

Get token balance for a given Steem account, if memo is given - get total symbol amt received with this memo.

Parameters

- **address** – Steem account to get balance for, if not set, uses self.coin.our_account
- **memo** – If not None, get total *self.symbol* received with this memo.
- **memo_case** – Case sensitive memo search

Returns Decimal(balance)

get_deposit() → tuple

Returns the deposit account for this symbol

Return tuple A tuple containing ('account', receiving_account). The memo must be generated by the calling function.

health() → Tuple[str, tuple, tuple]

Return health data for the passed symbol.

Health data will include: 'Symbol', 'Status', 'Coin Name', 'API Node', 'Head Block', 'Block Time', 'RPC Version', 'Our Account', 'Our Balance' (all strings)

Return tuple health_data (manager_name:str, headings:list/tuple, health_data:list/tuple,)

health_test() → bool

Check if our Steem node works or not, by requesting basic information such as the current block + time, and checking if our sending/receiving account exists on Steem.

Return bool True if Steem appears to be working, False if it seems to be broken.

provides

This attribute is automatically generated by scanning for `models.Coin` s with the type `steembase`. This saves us from hard coding specific coin symbols. See __init__.py for populating code.

send(*amount: decimal.Decimal, address: str, from_address: str = None, memo=None, trigger_data=None*) → dict

Send a supported currency to a given address/account, optionally specifying a memo if supported

Example - send 1.23 STEEM from @someguy123 to @privex with memo 'hello'

```
>>> s = SteemManager('STEEM')
>>> s.send(from_address='someguy123', address='privex', amount=Decimal('1.23
↪'), memo='hello')
```

Parameters

- **amount** (*Decimal*) – Amount of currency to send, as a Decimal()

- **address** – Account to send the currency to
- **from_address** – Account to send the currency from
- **memo** – Memo to send currency with

Raises

- **AttributeError** – When both *from_address* and *self.coin.our_account* are blank.
- **ArithmeticError** – When the amount is lower than the lowest amount allowed by the asset's precision
- ***AuthorityMissing*** – Cannot send because we don't have authority to (missing key etc.)
- ***AccountNotFound*** – The requested account doesn't exist
- ***NotEnoughBalance*** – The account *from_address* does not have enough balance to send this amount.

Return dict Result Information

Format:

```
{
    txid:str - Transaction ID - None if not known,
    coin:str - Symbol that was sent,
    amount:Decimal - The amount that was sent (after fees),
    fee:Decimal    - TX Fee that was taken from the amount,
    from:str       - The account/address the coins were sent from,
    send_type:str       - Should be statically set to "send"
}
```

3.2.1.1.6.5 SteemMixin module

class payments.coin_handlers.Steem.SteemMixin.**SteemMixin**(**args*, ***kwargs*)

Bases: *payments.coin_handlers.base.SettingsMixin.SettingsMixin*

SteemMixin - Shared code between SteemManager and SteemLoader

Designed for the Steem Network with SBD and STEEM support. May or may not work with other Graphene coins.

Copyright:

```
+=====================================================+
|                 © 2019 Privex Inc.                  |
|               https://www.privex.io                 |
+=====================================================+
|                                                     |
|        CryptoToken Converter                        |
|                                                     |
|        Core Developer(s):                           |
|                                                     |
|          (+)  Chris (@someguy123) [Privex]          |
|                                                     |
+=====================================================+
```

For **additional settings**, please see the module docstring in `coin_handlers.Steem`

asset
Easy reference to the Beem Asset object for our current symbol

find_steem_tx(*tx_data*, *last_blocks=15*) → Optional[dict]
Used internally to get the transaction ID after a transaction has been broadcasted

Parameters

- **tx_data** (*dict*) – Transaction data returned by a beem broadcast operation, must include 'signatures'
- **last_blocks** (*int*) – Amount of previous blocks to search for the transaction

Return dict Transaction data from the blockchain {transaction_id, ref_block_num, ref_block_prefix, expiration, operations, extensions, signatures, block_num, transaction_num}

Return None If the transaction wasn't found, None will be returned.

get_rpc(*symbol: str*) → beem.steem.Steem
Returns a Steem instance for querying data and sending TXs. By default, uses the Beem shared_steem_instance.

If a custom RPC list is specified in the Coin "custom json" settings, a new instance will be returned with the RPCs specified in the json.

Parameters **symbol** – Coin symbol to get Beem RPC instance for

Return beem.steem.Steem An instance of `beem.steem.Steem` for querying

precision

rpc

3.2.1.1.7 Telos Coin Handler

3.2.1.1.7.1 Module contents

Telos Coin Handler

This python module is a **Coin Handler** for Privex's CryptoToken Converter, designed to handle all required functionality for both receiving and sending tokens on the **Telos** network.

It will automatically handle any *`payments.models.Coin`* which has it's type set to `telos`

To use this handler, you must first create the base coin with symbol `Telos`:

```
Coin Name:  TLOS
Symbol:     TLOS
Our Account: (username of account used for sending/receiving native Telos token)
Custom JSON: {"contract": "eosio.token"}
```

To change the RPC node from the admin panel, simply set the host/port/username/password on the Telos Coin:

```
# Below are the defaults used if you don't configure the Telos coin::
Host: telos.caleos.io
Port: 443
User: (leave blank)
Pass: (leave blank)
Custom JSON: {"ssl": True}
```

Coin Settings (Custom JSON settings)

> **Tip:** You can override the defaults for all Telos coins by setting the `settings_json` for a coin with the symbol `TLOS`.
>
> All *`Coin`*'s handled by the Telos handler will inherit the `TLOS` coin's custom JSON settings, which can be overrided via the individual coin's `settings_json`.

You can set the following JSON keys inside of a *`Coin`*'s "settings_json" field to adjust settings such as the "contract account" for the token, whether or not to use SSL with the RPC node, as well as the precision (DP) of the coin, if it's different from the default of `4` decimal places.

Coin Key	Description
endpoint	(str) The base URI to query against, e.g. `/telos_rpc/`
ssl	(bool) Whether or not to use SSL (https). Boolean `true` or `false`
contract	(str) The contract account for this token, e.g. `eosio.token` or `steemenginex`
precision	(int) The precision (decimal places) of this coin (defaults to `4`)
load_method	(str) Either `actions` to use v1/history, or `pvx` to use Privex EOS History
history_url	(str) (if load_method is pvx) Privex history URL, e.g. `https://eos-history.privex.io`

Copyright:

```
+=====================================================+
|                 © 2019 Privex Inc.                  |
|               https://www.privex.io                 |
+=====================================================+
|                                                     |
|        CryptoToken Converter                        |
|                                                     |
|        Core Developer(s):                           |
|                                                     |
|          (+)  Chris (@someguy123) [Privex]          |
|                                                     |
+=====================================================+
```

`payments.coin_handlers.Telos.`**`reload`**`()`

Reload's the `provides` property for the loader and manager from the DB.

By default, since new tokens are constantly being created for Telos, our classes can provide for any `models.Coin` by scanning for coins with the type `telos`. This saves us from hard coding specific coin symbols.

3.2.1.1.7.2 Submodules

3.2.1.1.7.3 TelosLoader module

class `payments.coin_handlers.Telos.TelosLoader.`**`TelosLoader`**`(`*symbols*`)`

Bases: *`payments.coin_handlers.EOS.EOSLoader.EOSLoader`*, *`payments.coin_handlers.Telos.TelosMixin.TelosMixin`*

This is a stub class which simply glues *`TelosMixin`* onto *`EOSLoader`*

Once the mixin is applied, it adjusts the chain settings and overrides any required methods, then the standard EOSLoader code should just work.

provides

3.2.1.1.7.4 TelosManager module

class payments.coin_handlers.Telos.TelosManager.**TelosManager**(*symbol: str*)

Bases: *payments.coin_handlers.EOS.EOSManager.EOSManager*, *payments.coin_handlers.Telos.TelosMixin.TelosMixin*

This is a stub class which simply glues *TelosMixin* onto *EOSManager*

Once the mixin is applied, it adjusts the chain settings and overrides any required methods, then the standard EOSManager code should just work.

provides

3.2.1.1.7.5 TelosMixin module

class payments.coin_handlers.Telos.TelosMixin.**TelosMixin**

Bases: *payments.coin_handlers.EOS.EOSMixin.EOSMixin*

chain = 'telos'

chain_coin = 'TLOS'

chain_type = 'telos'

default_contracts = {'TLOS': 'eosio.token'}

eos

Returns an instance of Cleos and caches it in the attribute `_telos` after creation

provides = ['TLOS']

replace_eos(***conn*) → eospy.cleos.Cleos

Destroy the EOS `Cleos` instance at `_eos` and re-create it with the modified connection settings `conn`

Also returns the EOS instance for convenience.

Only need to specify settings you want to override.

Example:

```
>>> eos = self.replace_eos(host='example.com', port=80, ssl=False)
>>> eos.get_account('someguy123')
```

Parameters **conn** – Connection settings. Keys: endpoint, ssl, host, port, username, password

Return Cleos eos A `Cleos` instance with the modified connection settings.

setting_defaults = {'endpoint': '/', 'history_url': '', 'host': 'telos.caleos.io'

3.2.1.1.8 Coin Handler Base Classes

3.2.1.1.8.1 Submodules

3.2.1.1.8.2 BaseLoader

class payments.coin_handlers.base.BaseLoader.**BaseLoader**(*symbols: list = None*)

Bases: abc.ABC

BaseLoader - Base class for Transaction loaders

A transaction loader loads incoming transactions from one or more cryptocurrencies or tokens, whether through a block explorer, or through a direct connection to a local RPC node such as steemd or bitcoind using connection settings set by the user in their Django settings.

Transaction loaders must be able to initialise themselves using the following data:

- The coin symbols `self.symbols` passed to the constructor
- The setting_xxx fields on `self.coin` *payments.models.Coin*
- The Django settings *from django.conf import settings*
- They should also use the logging instance `settings.LOGGER_NAME`

If your class requires anything to be added to the Coin object settings, or the Django `settings` file, you should write a comment listing which settings are required, which are optional, and their format/type.

e.g. (Optional) settings.STEEM_NODE - list of steem RPC nodes, or string of individual node, URL format

They must implement all of the methods in this class, as well as configure the *provides* list to display the tokens/coins that this loader handles.

list_txs(*batch=100*) → Generator[dict, None, None]

The list_txs function processes the transaction data from *load()*, as well as handling any pagination, if it's required (e.g. only retrieve *batch* transactions at a time from the data source)

It should first check that *load()* has been ran if it's required, if the data required has not been loaded, it should call self.load()

To prevent memory leaks, this must be a generator function.

Below is an example of a generator function body, it loads *batch* transactions from the full transaction list, pretends to processes them into *txs*, yields them, then loads another batch after the calling function has iterated over the current *txs*

```
>>> t = self.transactions   # All transactions
>>> b = batch
>>> finished = False
>>> offset = 0
>>> # To save memory, process 100 transactions per iteration, and yield them␣
↪(generator)
>>> while not finished:
>>>     txs = []     # Processed transactions
>>>     # If there are less remaining TXs than batch size, get remaining txs␣
↪and finish.
>>>     if (len(t) - offset) < batch:
>>>         finished = True
>>>     # Do some sort-of processing on the tx to make it conform to␣
↪`Deposit`, then append to txs
>>>     for tx in t[offset:offset + batch]:
```

(continues on next page)

(continued from previous page)

```
>>>         txs.append(tx)
>>>     offset += b
>>>     for tx in txs:
>>>         yield tx
>>>     # At this point, the current batch is exhausted. Destroy the tx array␣
↪to save memory.
>>>     del txs
```

Parameters **batch** (*int*) – Amount of transactions to process/load per each batch

Returns Generator A generator returning dictionaries that can be imported into `models.Deposit`

Dict format:

```
{txid:str, coin:str (symbol), vout:int, tx_timestamp:datetime,
 address:str, from_account:str, to_account:str, memo:str, amount:Decimal}
```

`vout` is optional. One of either {from_account, to_account, memo} OR {address} must be included.

load(*tx_count=1000*)

The load function should prepare your loader, by either importing all of the data required for filtering, or setting up a generator for the *`list_txs()`* method to load them paginated.

It does NOT return anything, it simply creates any connections required, sets up generator functions if required for paginating the data, and/or pre-loads the first batch of transaction data.

Parameters **tx_count** – The total amount of transactions that should be loaded PER SYMBOL, most recent first.

Returns None

provides = []

3.2.1.1.8.3 BaseManager

class payments.coin_handlers.base.BaseManager.**BaseManager**(*symbol: str*)

Bases: `abc.ABC`

BaseManager - Base class for coin/token management

A coin manager handles balance checking, sending, and issuing of one or more cryptocurrencies or tokens, generally through a direct connection to a local/remote RPC node such as steemd or bitcoind using connection settings set by the user in their Django settings.

Coin managers must be able to initialise themselves using the following data:

- The coin symbol `self.symbol` passed to the constructor
- The setting_xxx fields on `self.coin` *`payments.models.Coin`*
- The Django settings *from django.conf import settings*

If your class requires anything to be added to the Coin object settings, or the Django `settings` file, you should write a comment listing which settings are required, which are optional, and their format/type.

e.g. (Optional) settings.STEEM_NODE - list of steem RPC nodes, or string of individual node, URL format

They must implement all of the methods in this class, set the *can_issue* boolean for detecting if this manager can be used to issue (create/print) tokens/coins, as well as configure the *provides* list to display the tokens/coins that this manager handles.

address_valid(*address*) → bool

A simple boolean method, allowing API requests to validate the destination address/account prior to giving the user deposit details.

Parameters **address** – An address or account to send to

Return bool Is the *address* valid? True if it is, False if it isn't

balance(*address: str = None*, *memo: str = None*, *memo_case: bool = False*) → decimal.Decimal

Return the balance of *self.symbol* for our "wallet", or a given address/account, optionally filtered by memo

Parameters

- **address** – The address or account to get the balance for. If None, return our total wallet (or default account) balance.
- **memo** – If not None (and coin supports memos), return the total balance of a given memo
- **memo_case** – Whether or not to total memo's case sensitive, or not. False = case-insensitive memo

Raises ***AccountNotFound*** – The requested account/address doesn't exist

Return Decimal Decimal() balance of address/account, optionally balance (total received) of a given memo

can_issue = False

If this manager supports issuing (creating/printing) tokens/coins, set this to True

get_deposit() → tuple

Return tuple If the coin uses addresses, this method should return a tuple of ('address', coin_address)

Return tuple If the coin uses accounts/memos, this method should return a tuple ('account', receiving_account) The memo will automatically be generated by the calling function.

health() → Tuple[str, tuple, tuple]

Return health data for the passed symbol, e.g. current block height, block time, wallet balance whether the daemon / API is accessible, etc.

It should return a tuple containing the manager name, the headings for a health table, and the health data for the passed symbol (Should include a `symbol` or coin name column)

You may use basic HTML tags in the health data result list, such as `<b>` `<em>` `<u>` and `<span style=""></span>`

Return tuple health_data (manager_name:str, headings:list/tuple, health_data:list/tuple,)

health_test() → bool

To reduce the risk of unhandled exceptions by sending code, this method should do some basic checks against the API to test whether the coin daemon / API is responding correctly.

This allows code which calls your send() or issue() method to detect the daemon / API is not working, and then delay sending/issuing until later, instead of marking a convert / withdrawal status to an error.

The method body should be wrapped in a try/except, ensuring there's a non-targeted except which returns False

Return bool True if the coin daemon / API appears to be working, False if it's not

issue(*amount: decimal.Decimal, address: str, memo: str = None, trigger_data: Union[dict, list] = None*) → dict

Issue (create/print) tokens to a given address/account, optionally specifying a memo if supported

Parameters

- **amount** (*Decimal*) – Amount of tokens to issue, as a Decimal()
- **address** – Address or account to issue the tokens to
- **memo** – Memo to issue tokens with (if supported)
- **trigger_data** (*dict*) – Metadata related to this issue transaction (e.g. the deposit that triggered this)

Raises

- ***IssuerKeyError*** – Cannot issue because we don't have authority to (missing key etc.)
- ***IssueNotSupported*** – Class does not support issuing, or requested symbol cannot be issued.
- ***AccountNotFound*** – The requested account/address doesn't exist

Return dict Result Information

Format:

```
dict {
    txid:str - Transaction ID - None if not known,
    coin:str - Symbol that was sent,
    amount:Decimal - The amount that was sent (after fees),
    fee:Decimal    - TX Fee that was taken from the amount,
    from:str       - The account/address the coins were issued from.
                     If it's not possible to determine easily, set this to␣
↪None.
    send_type:str  - Should be statically set to "issue"
}
```

orig_symbol = None

The original unique database symbol ID

provides = []

A list of token/coin symbols in uppercase that this loader supports e.g:

```
provides = ["LTC", "BTC", "BCH"]
```

send(*amount: decimal.Decimal, address: str, from_address: str = None, memo: str = None, trigger_data: Union[dict, list] = None*) → dict

Send tokens to a given address/account, optionally specifying a memo and sender address/account if supported

Your send method should automatically subtract any blockchain transaction fees from the amount sent.

Parameters

- **amount** (*Decimal*) – Amount of coins/tokens to send, as a Decimal()
- **address** – Address or account to send the coins/tokens to
- **memo** – Memo to send coins/tokens with (if supported)
- **from_address** – Address or account to send from (if required)

- **trigger_data** (*dict*) – Metadata related to this send transaction (e.g. the deposit that triggered this)

Raises

- ***AuthorityMissing*** – Cannot send because we don't have authority to (missing key etc.)
- ***AccountNotFound*** – The requested account/address doesn't exist
- ***NotEnoughBalance*** – Sending account/address does not have enough balance to send

Return dict Result Information

Format:

```
dict {
  txid:str - Transaction ID - None if not known,
  coin:str - Symbol that was sent,
  amount:Decimal - The amount that was sent (after fees),
  fee:Decimal    - TX Fee that was taken from the amount,
  from:str       - The account(s)/address(es) the coins were sent from. if␣
↪more than one, comma separated.
                   If it's not possible to determine easily, set this to None.
  send_type:str  - Should be statically set to "send"
}
```

send_or_issue (*amount*, *address*, *memo=None*, *trigger_data: Union[dict, list] = None*) → dict

Attempt to send an amount to an address/account, if not enough balance, attempt to issue it instead. You may override this method if needed.

Parameters

- **amount** (*Decimal*) – Amount of coins/tokens to send/issue, as a Decimal()
- **address** – Address or account to send/issue the coins/tokens to
- **memo** – Memo to send/issue coins/tokens with (if supported)
- **trigger_data** (*dict*) – Metadata related to this issue transaction (e.g. the deposit that triggered this)

Raises

- ***IssuerKeyError*** – Cannot issue because we don't have authority to (missing key etc.)
- ***IssueNotSupported*** – Class does not support issuing, or requested symbol cannot be issued.
- ***AccountNotFound*** – The requested account/address doesn't exist

Return dict Result Information

Format:

```
dict {
  txid:str       - Transaction ID - None if not known,
  coin:str       - Symbol that was sent,
  amount:Decimal - The amount that was sent (after fees),
  fee:Decimal    - TX Fee that was taken from the amount,
  from:str       - The account(s)/address(es) the coins were sent from. if␣
↪more than one, comma separated.
                   If it's not possible to determine easily, set this to None.
```

(continues on next page)

(continued from previous page)

```
  send_type:str  - Should be set to "send" if the coins were sent, or "issue"␣
↪if the coins were issued.
}
```

symbol = None

The native coin symbol, e.g. BTC, LTC, etc. (non-unique)

3.2.1.1.8.4 BatchLoader

class payments.coin_handlers.base.BatchLoader.**BatchLoader**(*symbols: list = None*)

Bases: *payments.coin_handlers.base.BaseLoader.BaseLoader*, abc.ABC

BatchLoader - An abstract sub-class of BaseLoader which comes with some pre-written batching/chunking functions

Copyright:

```
+=====================================================+
|                  © 2019 Privex Inc.                 |
|                https://www.privex.io                |
+=====================================================+
|                                                     |
|        CryptoToken Converter                        |
|                                                     |
|        Core Developer(s):                           |
|                                                     |
|          (+)  Chris (@someguy123) [Privex]          |
|                                                     |
+=====================================================+
```

This class is designed to save you time from re-writing your own "batching" / "chunking" functions.

Batching / chunking is a memory efficiency technique to prevent RAM leaks causing poor performance or crashes. Instead of loading all 1K - 10K transactions into memory, you load only a small amount of transactions, such as 100 transactions, then you use a Python generator (the yield keyword) to return individual transactions, quietly loading the next "batch" / "chunk" of 100 TXs after the first set has been processed, without interrupting the caller's *for* loop or other iteration.

This allows other functions to iterate over the transactions and process them on the fly, instead of having to load the entire 1-10K transaction list into memory first.

The use of generators throughout this class helps to prevent the problem of RAM leaks due to constant duplication of the transaction list (e.g. self.transactions, self.filtered_txs, self.cleaned_txs), especially when the transaction lists contains thousands of transactions.

To use this class, simply extend it (instead of BaseLoader), and make sure to implement the two abstract methods:

- *load_batch* - Loads and stores a small batch of raw (original format) transactions for a given coin
- ***clean_txs* - Filters the loaded TXs, yielding TXs (conformed to be compatible with `models.Deposit`)** that were received by us (not sent), and various sanity checks depending on the type of coin.

If your Loader is for a coin which uses an account/memo system, set *self.need_account* = *True* before calling BatchLoader's constructor, and it will remove coins in self.symbols/coins that do not have a non-empty/null *our_account* column.

You're free to override any methods if you need to, just make sure to call this class's constructor __init__ before/after your own constructor, otherwise some methods may break.

Flow of this class:

```
Transaction loading cron
   |
   V--> __init__(symbols:list)
   |--> load(tx_count:int)
   |--> list_txs(batch:int) -> _list_txs(coin:Coin, batch:int)
   V                              |--> load_batch(account, symbol, offset)
                                  V--> clean_txs(account, symbol, txs)
```

clean_txs(*symbol: str, transactions: Iterable[dict], account: str = None*) → Generator[dict, None, None]

Filters a list of transactions `transactions` as required, yields dict's conforming with `models.Deposit`

Important things when implementing this function:

- Make sure to filter out transactions that were sent from our own wallet/account - otherwise internal transfers will cause problems.
- Make sure each transaction is destined to us
  - If your loader is account-based, make sure to only yield transactions where tx["to_account"] == account.
  - If your loader is address-based, make sure that you only return transactions that are being received by our wallet, not being sent from it.
    - If `account` isn't None, assume that you must yield TXs sent to the given crypto address `account`
- If your loader deals with smart contract networks e.g. ETH, EOS, make sure that you only return transactions valid on the matching smart contract, don't blindly trust the symbol!
- Make sure that every dict that you `yield` conforms with the return standard shown for `BaseLoader.list_txs()`
- While transactions is normally a list<dict> you should assume that it could potentially be a Generator, writing the code Generator-friendly will ensure it can handle both lists and Generator's.

Example:

```
>>> def clean_txs(self, symbol: str, transactions: Iterable[dict],
>>>               account: str = None) -> Generator[dict, None, None]:
>>>     for tx in transactions:
>>>         try:
>>>             if tx['from'].lower() == 'tokens': continue       # Ignore␣
↪token issues
>>>             if tx['from'].lower() == account: continue        # Ignore␣
↪transfers from ourselves.
>>>             if tx['to'].lower() != account.lower(): continue  # If we aren
↪'t the receiver, we don't need it.
>>>             clean_tx = dict(
>>>                 txid=tx['txid'], coin=symbol, tx_timestamp=parse(tx[
↪'timestamp']),
>>>                 from_account=tx['from'], to_account=tx['to'], memo=tx[
↪'memo'],
>>>                 amount=Decimal(tx['quantity'])
```

(continues on next page)

(continued from previous page)

```
>>>             )
>>>             yield clean_tx
>>>         except:
>>>             log.exception('Error parsing transaction data. Skipping this␣
↪TX. tx = %s', tx)
>>>             continue
```

Parameters

- **symbol** – The symbol of the token being filtered
- **transactions** – A list<dict> of transactions to filter
- **account** – The 'to' account or crypto address to filter by (only required for account-based loaders)

Returns A generator yielding dict's conforming to `models.Deposit`, check the PyDoc return info for `coin_handlers.base.BaseLoader.list_txs()` for current format.

list_txs(*batch=100*) → Generator[dict, None, None]

Yield transactions for all coins in *self.coins* as a generator, loads transactions in batches of *batch* and returns them seamlessly using a generator.

If *`load()`* hasn't been ran already, it will automatically call self.load()

Parameters **batch** – Amount of transactions to load per batch

Return Generator[dict, None, None] Generator yielding dict's that conform to `models.Deposit`

load(*tx_count=1000*)

Simply imports *tx_count* into an instance variable, and then sets *self.loaded* to True.

If *self.need_account* is set to True by a child/parent class, this method will remove any coins from *self.coins* and *self.symbols* which have a blank/null *our_account* in the DB, ensuring that you can trust that all coins listed in symbols/coins have an *our_account* which isn't empty or None.

Parameters **tx_count** (*int*) – The amount of transactions to load per symbol specified in constructor

load_batch(*symbol*, *limit=100*, *offset=0*, *account=None*)

This function should load *limit* transactions in their raw format from your data source, skipping the *offset* newest TXs efficiently, and store them in the instance var *self.transactions*

If you use the included decorator `decorators.retry_on_err()`, if any exceptions are thrown by your method, it will simply re-run it with the same arguments up to 3 tries by default.

Basic implementation:

```
>>> @retry_on_err()
>>> def load_batch(self, symbol, limit=100, offset=0, account=None):
>>>     self.transactions = self.my_rpc.get_tx_list(limit, offset)
```

Parameters

- **symbol** – The symbol to load a batch of transactions for
- **limit** – The amount of transactions to load
- **offset** – Skip this many transactions (most recent first)

- **account** – An account name, or coin address to filter transactions using

3.2.1.1.8.5 SettingsMixin

Copyright:

```
+=====================================================+
|                 © 2019 Privex Inc.                  |
|               https://www.privex.io                 |
+=====================================================+
|                                                     |
|        CryptoToken Converter                        |
|                                                     |
|        Core Developer(s):                           |
|                                                     |
|          (+)  Chris (@someguy123) [Privex]          |
|                                                     |
+=====================================================+
```

class payments.coin_handlers.base.SettingsMixin.**SettingsMixin**

Bases: object

SettingsMixin - A mixin that can be used by coin loaders/managers for easy access to database/file settings, with handling of default settings

Copyright:

```
+=====================================================+
|                 © 2019 Privex Inc.                  |
|               https://www.privex.io                 |
+=====================================================+
|                                                     |
|        CryptoToken Converter                        |
|                                                     |
|        Core Developer(s):                           |
|                                                     |
|          (+)  Chris (@someguy123) [Privex]          |
|                                                     |
+=====================================================+
```

all_coins

Since this is a Mixin, it may be self.coin: Coin, or self.coins: List[Coin]. This property detects whether we have a single coin, or multiple, and returns them as a dict.

Return dict coins A dict<str,Coin> of supported coins, mapped by symbol

setting_defaults = {'host': '127.0.0.1', 'password': None, 'user': None}

If a setting isn't specified, use this dict for defaults, include both RPC defaults and custom json defaults

settings

Get all settings, mapped by coin symbol (each coin symbol dict contains custom json settings merged)

Return dict settings A dictionary mapping coin symbols to settings

use_coind_settings = True

If True, merges symbol settings from settings.COIND_RPC with precedence over database Coin settings

Override this to False in child classes to disable loading from the settings file

3.2.1.1.8.6 Base Decorators

payments.coin_handlers.base.decorators.**retry_on_err**(*max_retries: int = 3, delay: int = 3, **retry_conf*)

Decorates a function or class method, wraps the function/method with a try/catch block, and will automatically re-run the function with the same arguments up to *max_retries* time after any exception is raised, with a `delay` second delay between re-tries.

If it still throws an exception after `max_retries` retries, it will log the exception details with `fail_msg`, and then re-raise it.

Usage (retry up to 5 times, 1 second between retries, stop immediately if IOError is detected):

```
>>> @retry_on_err(5, 1, fail_on=[IOError])
... def my_func(self, some=None, args=None):
...     if some == 'io': raise IOError()
...      raise FileExistsError()
```

This will be re-ran 5 times, 1 second apart after each exception is raised, before giving up:

```
>>> my_func()
```

Where-as this one will immediately re-raise the caught IOError on the first attempt, as it's passed in `fail_on`:

```
>>> my_func('io')
```

Parameters

- **max_retries** (*int*) – Maximum total retry attempts before giving up
- **delay** (*int*) – Amount of time in seconds to sleep before re-trying the wrapped function
- **retry_conf** – Less frequently used arguments, pass in as keyword args:

- (list) fail_on: A list() of Exception types that should result in immediate failure (don't retry, raise)
- (str) retry_msg: Override the log message used for retry attempts. First message param %s is func name, second message param %d is retry attempts remaining
- (str) fail_msg: Override the log message used after all retry attempts are exhausted. First message param %s is func name, and second param %d is amount of times retried.

3.2.1.1.8.7 Base Exceptions

exception payments.coin_handlers.base.exceptions.**AccountNotFound**

Bases: *payments.coin_handlers.base.exceptions.CoinHandlerException*

The sending or receiving account requested doesn't exist

exception payments.coin_handlers.base.exceptions.**AuthorityMissing**

Bases: *payments.coin_handlers.base.exceptions.CoinHandlerException*

Missing private key or other authorization for this operation

exception payments.coin_handlers.base.exceptions.**CoinHandlerException**

Bases: `Exception`

Base exception for all Coin handler exceptions to inherit

exception payments.coin_handlers.base.exceptions.**DeadAPIError**
Bases: *payments.coin_handlers.base.exceptions.CoinHandlerException*

A main API, e.g. a coin daemon or public node used by this coin handler is offline.

exception payments.coin_handlers.base.exceptions.**IssueNotSupported**
Bases: *payments.coin_handlers.base.exceptions.CoinHandlerException*

This class does not support issuing, the token name cannot be issued, or other issue problems.

exception payments.coin_handlers.base.exceptions.**IssuerKeyError**
Bases: *payments.coin_handlers.base.exceptions.AuthorityMissing*

Attempted to issue tokens you don't have the issuer key for

exception payments.coin_handlers.base.exceptions.**MissingTokenMetadata**
Bases: *payments.coin_handlers.base.exceptions.CoinHandlerException*

Could not process a transaction or run the requested Loader/Manager method as required coin metadata is missing, such as *payments.models.Coin.our_account* or a required key in the custom JSON settings.

exception payments.coin_handlers.base.exceptions.**NotEnoughBalance**
Bases: *payments.coin_handlers.base.exceptions.CoinHandlerException*

The sending account does not have enough balance for this operation

exception payments.coin_handlers.base.exceptions.**TokenNotFound**
Bases: *payments.coin_handlers.base.exceptions.CoinHandlerException*

The token/coin requested doesn't exist

3.2.1.1.8.8 Module contents

3.2.1.2 Module contents

This module init file is responsible for loading the Coin Handler modules, and offering methods for accessing loaders and managers.

A **Coin Handler** is a Python module (folder containing classes and init file) designed to handle sending/receiving cryptocurrency/tokens for a certain network, or certain family of networks sharing similar code.

They may handle just one single coin, several coins, or they may even allow users to dynamically add coins by querying for a specific `coin_type` from the model *payments.models.Coin*

A coin handler must contain:

- An `__init__.py` with a dictionary named `exports`, containing the keys 'loader' and/or 'manager' pointing to the un-instantiated loader/manager class.
  - If your init file needs to do some sort-of initialisation, such as dynamically generating `provides` for your classes, or adding a new coin type to settings.COIN_TYPES, it's best to place it in a function named "reload" with a global boolean `loaded` so that you only initialise the module the first time it's loaded.

    See the example __init__.py near the bottom of this module docstring.

    This is optional, but it will allow reload_handlers() to properly re-trigger your initialisation code only when changes occur, such as Coin's being created/updated in the database.
- Two classes, a **Loader** and a **Manager**. Each class can either in it's own file, or in a single file containing other classes / functions.
  - A **Loader** is a class which extends *base.BaseLoader*, and is responsible for retrieving transactions that occur on that coin to detect incoming transactions.

- A **Manager** is a class which extends `base.BaseManager`, and is responsible for handling sending/issuing of coins/tokens, as well as other small functions such as validating addresses, and checking balances.

Your **Loader** class may choose to extend the helper class `base.BatchLoader`, allowing your loader to use batches/chunking for memory efficiency, without having to write much code.

Your Coin Handler classes should ONLY use the exceptions in `base.exceptions`, along with any exceptions listed in the *:raises:* pydoc statements of the overridden method.

For handling automatic retry when something goes wrong, you can use the decorator `base.decorators.retry_on_err()`

Example *__init__.py*:

```
>>> from django.conf import settings
>>> from payments.coin_handlers.SteemEngine.SteemEngineLoader import SteemEngineLoader
>>> from payments.coin_handlers.SteemEngine.SteemEngineManager import␣
↪SteemEngineManager
>>>
>>> loaded = False
>>>
>>> def reload():
>>>     global loaded
>>>     if 'steemengine' not in dict(settings.COIN_TYPES):
>>>         settings.COIN_TYPES += (('steemengine', 'SteemEngine Token',),)
>>>     loaded = True
>>>
>>> if not loaded:
>>>    reload()
>>>
>>> exports = {
>>>     "loader": SteemEngineLoader,
>>>     "manager": SteemEngineManager
>>> }
```

For an example of how to layout your coin handler module, check out the pre-included Coin Handlers:

- `SteemEngine`
- `Bitcoin`

`payments.coin_handlers.`**`add_handler`**(*handler*, *handler_type*)

`payments.coin_handlers.`**`ch_base = 'payments.coin_handlers'`**

Base module path to where the coin handler modules are located. E.g. payments.coin_handlers

`payments.coin_handlers.`**`get_loader`**(*symbol: str*) → payments.coin_handlers.base.BaseLoader.BaseLoader

For some use-cases, you may want to just grab the first loader that supports this coin.

```
>>> m = get_loader('ENG')
>>> m.send(amount=Decimal(1), from_address='someguy123', address='privex')
```

Parameters **`symbol`** – The coin symbol to get the loader for (uppercase)

Return BaseLoader An instance implementing `base.BaseLoader`

`payments.coin_handlers.`**`get_loaders`**(*symbol: str = None*) → list

Get all loader's, or all loader's for a certain coin

Parameters **symbol** – The coin symbol to get all loaders for (uppercase)

Return list If symbol not specified, a list of tuples (symbol, list<BaseLoader>,)

Return list If symbol IS specified, a list of instantiated *base.BaseLoader*'s

payments.coin_handlers.**get_manager**(*symbol: str*) → payments.coin_handlers.base.BaseManager.BaseManager

For some use-cases, you may want to just grab the first manager that supports this coin.

```
>>> m = get_manager('ENG')
>>> m.send(amount=Decimal(1), from_address='someguy123', address='privex')
```

Parameters **symbol** – The coin symbol to get the manager for (uppercase)

Return BaseManager An instance implementing *base.BaseManager*

payments.coin_handlers.**get_managers**(*symbol: str = None*) → list

Get all manager's, or all manager's for a certain coin

Parameters **symbol** – The coin symbol to get all managers for (uppercase)

Return list If symbol not specified, a list of tuples (symbol, list<BaseManager>,)

Return list If symbol IS specified, a list of instantiated *base.BaseManager*'s

payments.coin_handlers.**handlers = {}**

A dictionary of coin symbols, containing instantiated managers (BaseManager) and loaders (BaseLoader)

Example layout:

```
handlers = {
    'ENG': {
        'loaders':  [ SteemEngineLoader, ],
        'managers': [ SteemEngineLoader, ],
    },
    'SGTK': {
        'loaders':  [ SteemEngineLoader, ],
        'managers': [ SteemEngineLoader, ],
    },
}
```

payments.coin_handlers.**handlers_loaded = False**

Used to track whether the Coin Handlers have been initialized, so reload_handlers can be auto-called.

payments.coin_handlers.**has_loader**(*symbol: str*) → bool

Helper function - does this symbol have a loader class?

payments.coin_handlers.**has_manager**(*symbol: str*) → bool

Helper function - does this symbol have a manager class?

payments.coin_handlers.**init_privex_handler**(*name: str*)

Attempt to import a `privex.coin_handlers` handler module, adapting it for SteemEngine's older Coin Handler system.

- Extracts the handler type and description for injection into `settings.COIN_TYPES` (since privex handlers are framework independent and cannot auto-inject into `settings.COIN_TYPE`)
- Configures the Privex coin_handlers coin object based on a database *Coin* row
- Detects coins which are mapped to a Privex coin handler and registers the handler's manager/loader into the global handlers dictionary

Parameters **name** (*str*) – The name of a `privex.coin_handlers` handler module, e.g. `Golos`

`payments.coin_handlers.`**`is_database_synchronized`**(*database: str*) → bool

Check if all migrations have been ran. Useful for preventing auto-running code accessing models before the tables even exist, thus preventing you from migrating…

```
>>> from django.db import DEFAULT_DB_ALIAS
>>> if not is_database_synchronized(DEFAULT_DB_ALIAS):
>>>     log.warning('Cannot run reload_handlers because there are unapplied␣
↪migrations!')
>>>     return
```

Parameters **database** (*str*) – Which Django database config is being used? Generally just pass django.db.DEFAULT_DB_ALIAS

Return **bool** True if all migrations have been ran, False if not.

`payments.coin_handlers.`**`reload_handlers`**()

Resets *handlers* to an empty dict, then loads all *settings.COIN_HANDLER* classes into the dictionary *handlers* using *settings.COIN_HANDLERS_BASE* as the base module path to load from

3.2.2 payments package

3.2.2.1 Subpackages

3.2.2.2 Submodules

3.2.2.3 payments.admin module

class `payments.admin.`**`AddCoinPairView`**(***kwargs*)

Bases: `django.views.generic.base.TemplateView`

Admin view for easily adding two coins + two pairs in each direction

`coin_types`()

View function to be called from template, for getting list of coin handler errors

`get`(*request*, **args*, ***kwargs*)

`post`(*request*, **args*, ***kwargs*)

`template_name = 'admin/add_pair.html'`

class `payments.admin.`**`AddressAccountMapAdmin`**(*model*, *admin_site*)

Bases: `django.contrib.admin.options.ModelAdmin`

`list_display = ('deposit_coin', 'deposit_address', 'destination_coin', 'destination_ad`

`list_filter = ('deposit_coin', 'destination_coin')`

`media`

`search_fields = ('deposit_address', 'destination_address')`

class `payments.admin.`**`CoinAdmin`**(*model*, *admin_site*)

Bases: `django.contrib.admin.options.ModelAdmin`

`fieldsets = (('Unique Coin Symbol for refrencing from the API', {'fields': ('symbol',`

get_fieldsets(*request*, *obj=None*)
Hook for specifying fieldsets.

list_display = ('__str__', 'symbol', 'coin_type', 'enabled', 'our_account', 'can_issue

list_filter = ('coin_type',)

media

ordering = ('symbol',)

class payments.admin.**CoinHealthView**(***kwargs*)
Bases: `django.views.generic.base.TemplateView`

Admin view for viewing health/status information of all coins in the system.

Loads the coin handler manager for each coin, and uses the health() function to grab status info for the coin.

Uses caching API to avoid constant RPC queries, and displays results as a standard admin view.

get(*request*, **args*, ***kwargs*)

get_fails()
View function to be called from template, for getting list of coin handler errors

handler_dic()
View function to be called from template. Loads and queries coin handlers for health, with caching.

template_name = 'admin/coin_health.html'

class payments.admin.**CoinPairAdmin**(*model*, *admin_site*)
Bases: `django.contrib.admin.options.ModelAdmin`

list_display = ('__str__', 'from_coin', 'to_coin', 'exchange_rate')

media

ordering = ('from_coin', 'to_coin')

class payments.admin.**ConversionAdmin**(*model*, *admin_site*)
Bases: `django.contrib.admin.options.ModelAdmin`

list_display = ('from_coin', 'from_address', 'from_amount', 'to_coin', 'to_address', '

list_filter = ('from_coin', 'to_coin')

media

ordering = ('-created_at',)

search_fields = ('id', 'from_address', 'to_address', 'to_memo', 'to_txid')

class payments.admin.**CustomAdmin**(*name='admin'*)
Bases: `django.contrib.admin.sites.AdminSite`

To allow for custom admin views, we override AdminSite, so we can add custom URLs, among other things.

get_urls()

class payments.admin.**DepositAdmin**(*model*, *admin_site*)
Bases: `django.contrib.admin.options.ModelAdmin`

actions = [<function confirm_refund_deposit>]

list_display = ('txid', 'status', 'coin', 'amount', 'address', 'from_account', 'to_acc

list_filter = ('status', 'coin')

media

ordering = ('-tx_timestamp',)

search_fields = ('id', 'txid', 'address', 'from_account', 'to_account', 'memo', 'refun

class payments.admin.**KeyPairAdmin**(*model*, *admin_site*)
Bases: `django.contrib.admin.options.ModelAdmin`

list_display = ('network', 'public_key', 'account', 'key_type')

media

ordering = ('network', 'account')

payments.admin.**clear_cache**(*request*)
Allow admins to clear the Django cache system

payments.admin.**confirm_refund_deposit**(*modeladmin*, *request*, *queryset: django.db.models.query.QuerySet*)
Confirmation page for the "Refund Deposits to Sender" page. :param modeladmin: :param request: :param queryset: :return:

payments.admin.**path**(*route*, *view*, *kwargs=None*, *name=None*, *, *Pattern=<class 'django.urls.resolvers.RoutePattern'>*)

payments.admin.**refund_deposits**(*request*)

3.2.2.4 payments.apps module

class payments.apps.**PaymentsConfig**(*app_name*, *app_module*)
Bases: `django.apps.config.AppConfig`

name = 'payments'

3.2.2.5 payments.models module

This file contains Models, classes which define database tables, and how they relate to each other.

Models are used for both querying the database, as well as inserting new rows and updating existing ones.

Models may also contain **properties** and **functions** to help make them easier to use.

Note: The `coin_type` choices tuple, `COIN_TYPES` is located in settings.py, and may be dynamically altered by Coin Handlers. It does not enforce an enum on columns using it for `choices` , it's simply used for a dropdown list in the admin panel.

Copyright:

```
+=====================================================+
|                 © 2019 Privex Inc.                  |
|               https://www.privex.io                 |
+=====================================================+
|                                                     |
|        CryptoToken Converter                        |
|                                                     |
|        Core Developer(s):                           |
|                                                     |
|          (+)  Chris (@someguy123) [Privex]          |
|                                                     |
+=====================================================+
```

class payments.models.**AddressAccountMap**(**args*, ***kwargs*)

Bases: `django.db.models.base.Model`

This database model maps normal Bitcoin-like addresses to a destination token, and their token account/address.

This is because deposits of coins such as Bitcoin/Litecoin do not contain any form of "memo", so they must be manually mapped onto a destination.

This model may be used for handling deposits for both memo-based (Bitshares-like) and address-based (Bitcoin-like) deposits, as there is both a memo and address (or account) field for deposits + destination coin

exception DoesNotExist

Bases: `django.core.exceptions.ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `django.core.exceptions.MultipleObjectsReturned`

conversions

deposit_address

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

deposit_coin

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

deposit_coin_id

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

deposit_memo

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

destination_address

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

destination_coin

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

destination_coin_id

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

destination_memo
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

id
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

objects = <django.db.models.manager.Manager object>

class payments.models.**Coin**(**args*, ***kwargs*)
Bases: `django.db.models.base.Model`

The operator of the service should define all coins and tokens they would like to support using the Django Admin. The symbol is used as the primary key, so it must be unique. It will automatically be made uppercase. Native Coin Symbol (e.g. BTC)

exception DoesNotExist
Bases: `django.core.exceptions.ObjectDoesNotExist`

exception MultipleObjectsReturned
Bases: `django.core.exceptions.MultipleObjectsReturned`

can_issue
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

coin_type
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

conversions_from
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

conversions_to
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

deposit_converts
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

deposit_maps

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

deposits

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

dest_maps

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

display_name

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

enabled

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

funds_low

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

last_notified

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

notify_low_funds
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

objects = <django.db.models.manager.Manager object>

our_account
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

pairs

pairs_from
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

pairs_to
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

save(**args*, ***kwargs*)
To avoid inconsistency, the symbol is automatically made uppercase

setting_host
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

setting_json
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

setting_pass
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

setting_port
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

setting_user
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

settings

Small helper property for quickly accessing the setting_xxxx fields, while also decoding the custom json field into a dictionary/list

Returns dict(host:str, port:str, user:str, password:str, json:dict/list)

should_notify_low

Should we notify the admins that this coin's wallet balance is too low?

Used to rate limit "???coin wallet balance is too low" emails sent to admins.

Usage:

```
>>> from django.core.mail import mail_admins
>>> c = Coin.objects.get(symbol='BTC')
>>> if c.should_notify_low:
>>>     mail_admins('BTC hot wallet is low!', 'The hot wallet is low. Please␣
↪refill.')
```

Return bool `True` if we should notify the admins

Return bool `False` if we should skip this email notification for now, or notifications are disabled.

symbol

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

symbol_id

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

class payments.models.**CoinPair**(**args*, ***kwargs*)

Bases: `django.db.models.base.Model`

A coin pair defines an allowed conversion direction between two coins For example LTC (Litecoin) -> LTCP (Pegged Litecoin)

exception DoesNotExist

Bases: `django.core.exceptions.ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `django.core.exceptions.MultipleObjectsReturned`

exchange_rate

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

from_coin

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

from_coin_id
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

from_coin_symbol

id
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

objects = <django.db.models.manager.Manager object>

to_coin
Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

to_coin_id
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

to_coin_symbol

class payments.models.**Conversion**(**args*, ***kwargs*)
Bases: `django.db.models.base.Model`

Once a `models.Deposit` has been scanned, assuming it has a valid address or account/memo, the destination cryptocurrency/token will be sent to the user.

Successful conversion attempts are logged here, allowing for reference of where the coins came from, where they went, and what fees were taken.

exception DoesNotExist
Bases: `django.core.exceptions.ObjectDoesNotExist`

exception MultipleObjectsReturned
Bases: `django.core.exceptions.MultipleObjectsReturned`

created_at
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

deposit
Accessor to the related object on the forward side of a one-to-one relation.

In the example:

```
class Restaurant(Model):
    place = OneToOneField(Place, related_name='restaurant')
```

`Restaurant.place` is a `ForwardOneToOneDescriptor` instance.

deposit_id
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

ex_fee
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

from_address
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

from_amount

from_coin
The coin that we were sent

from_coin_id
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

from_coin_symbol

get_next_by_created_at(*, *field=<django.db.models.fields.DateTimeField: created_at>*, *is_next=True*, ***kwargs*)

get_next_by_updated_at(*, *field=<django.db.models.fields.DateTimeField: updated_at>*, *is_next=True*, ***kwargs*)

get_previous_by_created_at(*, *field=<django.db.models.fields.DateTimeField: created_at>*, *is_next=False*, ***kwargs*)

get_previous_by_updated_at(*, *field=<django.db.models.fields.DateTimeField: updated_at>*, *is_next=False*, ***kwargs*)

id
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

objects = <django.db.models.manager.Manager object>

to_address
Where was it sent to?

to_amount
The amount of `to_coin` that was sent, stored as a high precision Decimal

to_coin
The destination token/crypto this token will be converted to

to_coin_id
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

to_coin_symbol

to_memo
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

to_txid
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

tx_fee
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

updated_at
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

class payments.models.**CryptoKeyPair**(**args*, ***kwargs*)
Bases: `django.db.models.base.Model`

This model allows for storing key pairs (generally for cryptocurrency addresses/accounts) safely in the database.

The private key is automatically encrypted with AES-128 upon saving, ensuring it cannot be read from the admin panel, any API leaks, or third party applications reading from the database.

For this model to function correctly, you must set ENCRYPT_KEY in `.env` by generating an encryption key using `./manage.py generate_key`

exception DoesNotExist
Bases: `django.core.exceptions.ObjectDoesNotExist`

exception MultipleObjectsReturned
Bases: `django.core.exceptions.MultipleObjectsReturned`

account
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

balance
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

id
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

key_type
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

network
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

objects = <django.db.models.manager.Manager object>

private_key
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

public_key
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

save(**args*, ***kwargs*)
To ensure that private keys can only be entered / updated from the admin panel and not viewed, we encrypt them with AES-128 when saving.

To avoid encrypting an already encrypted key, we only encrypt the key if we're sure it's not encrypted already.

Raises

- **EncryptionError** – Something went wrong while encrypting the key

- **EncryptKeyMissing** – The key `settings.ENCRYPT_KEY` is not set or is not a valid encryption key.

used
For disposable addresses, e.g. Bitcoin addresses, this field tracks whether it has been used for a deposit.

class `payments.models.`**Deposit**(**args*, ***kwargs*)
Bases: `django.db.models.base.Model`

A log of incoming token/crypto deposits, which will later be converted into crypto.

The primary key of a *Deposit* is the auto-generated *id* field - an auto incrementing integer.

There is a composite unique constraint on (txid, coin, vout), ensuring duplicate transactions do not get stored.

Deposits start out in state `new` , as they are processed by the conversion system they progress into either:

'err' - An error occurred while converting / importing During the import/conversion there was a serious error that could not be recovered from This should be investigated by a developer.

'inv' - Invalid source/destination, user did not follow instructions correctly The coins were sent to a non-registered address, or a memo we don't know how to process. An admin should attempt to refund these coins to the sender.

'refund' - The coins sent in this Deppsit were refunded Info about the refund should be in the refund_* fields

'mapped' - Deposit passed initial sanity checks, and we know the destination coin, address/account and memo. Most deposits should only stay in this state for a few seconds, before they're converted. If a deposit stays in this state for more than a few minutes, it generally means something is wrong with the Coin Handler, preventing it from sending the coins, e.g. low balance.

'conv' - Successfully Converted The deposited coins were successfully converted into their destination coin, and there should be a related `models.Conversion` containing the conversion details.

exception DoesNotExist
Bases: `django.core.exceptions.ObjectDoesNotExist`

exception MultipleObjectsReturned
Bases: `django.core.exceptions.MultipleObjectsReturned`

STATUSES = (('err', 'Error Processing Transaction'), ('inv', 'Transaction is invalid')

address
If the deposit is from a classic Bitcoin-like cryptocurrency with addresses, then you should enter the address where the coins were deposited into, in this field.

amount
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

coin
The symbol of the cryptocurrency or token that was deposited, in uppercase. e.g. LTC, LTCP, BTCP, STEEMP

coin_id
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

coin_symbol

conversion
Accessor to the related object on the reverse side of a one-to-one relation.

In the example:

```
class Restaurant(Model):
    place = OneToOneField(Place, related_name='restaurant')
```

`Place.restaurant` is a `ReverseOneToOneDescriptor` instance.

convert_dest_address
The destination address. Set after a deposit has been analyzed, and we know what coin it will be converted to.

convert_dest_memo
The destination memo. Set after a deposit has been analyzed, and we know what coin it will be converted to.

convert_to
The destination coin. Set after a deposit has been analyzed, and we know what coin it will be converted to

convert_to_id
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

created_at
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

error_reason
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

from_account
If account-based coin, contains the name of the account that sent the coins

get_next_by_created_at(*, *field=<django.db.models.fields.DateTimeField: created_at>*, *is_next=True*, ***kwargs*)

get_next_by_updated_at(*, *field=<django.db.models.fields.DateTimeField: updated_at>*, *is_next=True*, ***kwargs*)

get_previous_by_created_at(*, *field=<django.db.models.fields.DateTimeField: created_at>*, *is_next=False*, ***kwargs*)

get_previous_by_updated_at(*, *field=<django.db.models.fields.DateTimeField: updated_at>*, *is_next=False*, ***kwargs*)

get_status_display(*, *field=<django.db.models.fields.CharField: status>*)

id
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

last_convert_attempt
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

memo
If the coin supports memos, and they're required to identify a deposit, use this field.

objects = <django.db.models.manager.Manager object>

processed_at
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

refund_address
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

refund_amount
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

refund_coin
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

refund_memo
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

refund_txid
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

refunded_at
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

status
The current status of this deposit, see `STATUSES`

to_account
If account-based coin, contains the name of the account that the coins were deposited into

tx_timestamp
The date/time the transaction actually occurred on the chain

txid
The transaction ID where the coins were received.

updated_at
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

vout
If a transaction contains multiple deposits, for example, a Bitcoin transaction that contains several outputs (vout's) for our addresses, then each vout must have an consistent output number, i.e. one that will not change each time the blockchain transaction is compared against the database.

3.2.2.6 payments.serializers module

class payments.serializers.**CoinPairSerializer**(*instance=None, data=<class 'rest_framework.fields.empty'>, **kwargs*)

Bases: `rest_framework.serializers.HyperlinkedModelSerializer`

class Meta

Bases: `object`

fields = ('id', 'from_coin', 'from_coin_symbol', 'to_coin', 'to_coin_symbol', 'exch

model
alias of *`payments.models.CoinPair`*

class payments.serializers.**CoinSerializer**(*instance=None, data=<class 'rest_framework.fields.empty'>, **kwargs*)

Bases: rest_framework.serializers.HyperlinkedModelSerializer

class Meta

Bases: object

fields = ('symbol', 'display_name', 'our_account', 'can_issue', 'coin_type', 'symbo

model

alias of *payments.models.Coin*

class payments.serializers.**ConversionSerializer**(*instance=None, data=<class 'rest_framework.fields.empty'>, **kwargs*)

Bases: rest_framework.serializers.HyperlinkedModelSerializer

class Meta

Bases: object

exclude = ()

model

alias of *payments.models.Conversion*

class payments.serializers.**DepositSerializer**(*instance=None, data=<class 'rest_framework.fields.empty'>, **kwargs*)

Bases: rest_framework.serializers.HyperlinkedModelSerializer

class Meta

Bases: object

fields = ('id', 'txid', 'coin', 'coin_symbol', 'vout', 'status', 'tx_timestamp', 'a

model

alias of *payments.models.Deposit*

3.2.2.7 payments.tests module

3.2.2.8 payments.views module

class payments.views.**CoinAPI**(***kwargs*)

Bases: rest_framework.viewsets.ReadOnlyModelViewSet

filterset_fields = ('symbol', 'symbol_id', 'our_account', 'coin_type', 'can_issue')

lookup_value_regex = '[^/]+'

queryset

serializer_class

alias of *payments.serializers.CoinSerializer*

class payments.views.**CoinPairAPI**(***kwargs*)

Bases: rest_framework.viewsets.ReadOnlyModelViewSet

filterset_fields = ('from_coin', 'to_coin')

lookup_value_regex = '[^/]+'

queryset

serializer_class
alias of *payments.serializers.CoinPairSerializer*

class payments.views.**ConversionAPI**(***kwargs*)
Bases: rest_framework.viewsets.ReadOnlyModelViewSet

filterset_fields = ('from_coin', 'to_coin', 'from_address', 'to_address', 'deposit__fr

pagination_class
alias of *CustomPaginator*

queryset

serializer_class
alias of *payments.serializers.ConversionSerializer*

class payments.views.**ConvertAPI**(***kwargs*)
Bases: rest_framework.views.APIView

Required form / JSON fields:

- from_coin - The API coin symbol to convert from (send this coin)
- to_coin - The API coin symbol to convert into (we send you this coin)
- destination - The account / address to send to

Optional:

- dest_memo - For coins that support memos, you can specify a custom memo to use when sending.

Example (application/json)

```
{"from_coin": "BTC", "to_coin": "BTCP", "destination": "someguy123"}
```

Example (application/x-www-form-urlencoded):

```
from_coin=BTC&to_coin=BTCP&destination=someguy123
```

authentication_classes = (<class 'payments.views.DRFNoCSRF'>,)

post(*request: rest_framework.request.Request*)

class payments.views.**ConvertView**(***kwargs*)
Bases: django.views.generic.base.TemplateView

template_name = 'convert.html'

class payments.views.**CustomPaginator**
Bases: rest_framework.pagination.LimitOffsetPagination

default_limit = 100

max_limit = 1000

class payments.views.**DRFNoCSRF**
Bases: rest_framework.authentication.SessionAuthentication

enforce_csrf(*request*)
Enforce CSRF validation for session based authentication.

class payments.views.**DepositAPI**(***kwargs*)
Bases: rest_framework.viewsets.ReadOnlyModelViewSet

filterset_fields = ('address', 'from_account', 'to_account', 'txid', 'memo', 'conversi

order_by = 'created'

pagination_class
alias of *CustomPaginator*

queryset

serializer_class
alias of *payments.serializers.DepositSerializer*

class payments.views.**IndexView**(***kwargs*)
Bases: `django.views.generic.base.TemplateView`

template_name = 'base.html'

payments.views.**api_root**(*self*, *request*, **args*, ***kwargs*)

payments.views.**r_err**(*msg*, *status=500*)

3.2.2.9 Module contents

CHAPTER 4

REST API Documentation

CryptoToken Converter exposes a REST API under the URL `/api` to allow any application to easily interact with the system.

It uses Django REST Framework which automatically generates a lot of the code running behind the API endpoints.

4.1 Endpoints

For **GET** requests, any request parameters must either be sent as either:

Standard GET parameters - e.g. `/api/deposits/?from_address=someguy123`

Directly in the URL - e.g. `/api/coins/LTC`

For **POST** requests, you may send your request data/params as a normal URL encoded form, or you may choose to send it as JSON.

application/json - JSON Encoded Body

```
{
    "my_param": "somevalue",
    "other.param": "other value"
}
```

application/x-www-form-urlencoded - Standard POST body

```
my_param=somevalue&other.param=other%20value
```

4.2 /api/convert/

Starts the conversion process between two coins.

Returns the deposit details for you to send the coins to.

Methods: POST (URL Encoded Form, or JSON)

POST Parameters:

Parameter	Type	Description
from_coin	String	Symbol of the coin to convert from
to_coin	String	Symbol of the destination coin
destination	String	The address or account on `to_coin` for receiving your converted coins

All parameters are required.

Errors:

If the JSON response `error` key is present and set to `true`, the error message will be placed in `message`, and a non-200 status code will be returned, related to the error reason.

Potential errors and their status codes:

- "An unknown error has occurred... please contact support", 500
- "You must specify 'from_coin', 'to_coin', and 'destination'", 400
- "There is no such coin pair {} -> {}", 404
- "The destination {} address/account '{}' is not valid", 400

Example error response:

```
POST /api/convert/

HTTP 400 Bad Request
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "error": true,
    "message": "You must specify 'from_coin', 'to_coin', and 'destination'"
}
```

Return Data:

All successful requests will include `ex_rate` (the amount of `to_coin` per `from_coin`), `pair` (details about the coin pair that you have chosen), and `destination` (where the `from_coin` will be sent to).

Depending on whether the `from_coin` is an **address based** coin, or an **account/memo based** coin, the actual deposit details will be returned differently. Address based coins will return `address`, while account based coins will return `account` and `memo`.

Below are two examples to help explain this. `SGTK` is "Sometoken", a SteemEngine token, meaning it's account+memo based. `LTC` is Litecoin, a classic address based cryptocurrency.

Example 1 (address based -> account based):

```
POST /api/convert/
from_coin=LTC&to_coin=SGTK&destination=someguy123

HTTP 200 OK
Content-Type: application/json
```

(continues on next page)

(continued from previous page)

```
{
    "ex_rate": 100000.0,
    "destination": "someguy123",
    "pair": "LTC -> SGTK (100000.0000 SGTK per LTC)",
    "address": "MJL1E5oSqFLpdL9BswKmYonxU1Cq1WKWGL"
}
```

Example 2 (account based -> address based):

```
POST /api/convert/
from_coin=SGTK&to_coin=LTC&destination=MVYBriQcasb6zvtGjPfLKbbWcRoKWh4sAf

HTTP 200 OK
Content-Type: application/json
{
    "ex_rate": 0.01,
    "destination": "MVYBriQcasb6zvtGjPfLKbbWcRoKWh4sAf",
    "pair": "SGTK -> LTC (0.0100 LTC per SGTK)",
    "memo": "LTC MVYBriQcasb6zvtGjPfLKbbWcRoKWh4sAf",
    "account": "someguy123"
}
```

4.3 /api/deposits/

`/api/deposits/` Returns all deposit attempts received by the system. Can be filtered using the **GET Parameters** listed below.

`/api/deposits/<id>` Returns a single deposit attempt by it's ID

Methods: GET

GET Parameters:

These parameters can be used with the plain `/api/deposits/` URL, to filter deposits based on various columns.

Note: Results from `/api/deposits/` will always be returned as a list, even if there's only one.

Parameter	Type	Description
address	String	Return deposits that were sent to this address (only for address-based coins)
txid	String	Return deposits with a matching transaction ID
from_account	String	Return deposits that were sent from this account (only for account-based coins)
to_account	String	Return deposits that were sent to this account (only for account-based coins)
memo	String	Return deposits that were sent using this memo (normally only for account-based coins)

Return Data:

Example 1 (Plain GET request):

```
GET /api/deposits/

HTTP 200 OK
Content-Type: application/json

[
```

(continues on next page)

(continued from previous page)

```
    {
        "id": 4,
        "txid": "635dd656b3bd8c61699e6066c9b3c6e74696e195",
        "coin": "http://127.0.0.1:8000/api/coins/SGTK/",
        "vout": 0,
        "status": "conv",
        "tx_timestamp": "2019-03-20T03:46:30Z",
        "address": null,
        "from_account": "privex",
        "to_account": "someguy123",
        "amount": "1.00000000000000000000",
        "memo": "LTC LKjpPtgMbcFgbJJYwzfe1ZtR8x4bbs2V3o",
        "processed_at": "2019-03-20T04:31:30.643406Z",
        "convert_to": "http://127.0.0.1:8000/api/coins/LTC/"
    },
    {
        "id": 5,
        "txid":
↪"b881d1ae8cf280184960c9c2d74bc1bd230f18f5adcd7fe695239dbf46b06c45",
        "coin": "http://127.0.0.1:8000/api/coins/LTC/",
        "vout": 0,
        "status": "conv",
        "tx_timestamp": "2019-03-20T01:34:20Z",
        "address": "MFht1FmYhsRaSChGdqomxQpjtGtsjFHDQX",
        "from_account": null,
        "to_account": null,
        "amount": "0.10000000000000000000",
        "memo": null,
        "processed_at": "2019-03-20T04:46:53.602857Z",
        "convert_to": "http://127.0.0.1:8000/api/coins/SGTK/"
    }
]
```

Example 2 (Filtering results):

```
GET /api/deposits/?txid=635dd656b3bd8c61699e6066c9b3c6e74696e195

HTTP 200 OK
Content-Type: application/json

[
    {
        "id": 4,
        "txid": "635dd656b3bd8c61699e6066c9b3c6e74696e195",
        "coin": "http://127.0.0.1:8000/api/coins/SGTK/",
        "vout": 0,
        "status": "conv",
        "tx_timestamp": "2019-03-20T03:46:30Z",
        "address": null,
        "from_account": "privex",
        "to_account": "someguy123",
        "amount": "1.00000000000000000000",
        "memo": "LTC LKjpPtgMbcFgbJJYwzfe1ZtR8x4bbs2V3o",
        "processed_at": "2019-03-20T04:31:30.643406Z",
        "convert_to": "http://127.0.0.1:8000/api/coins/LTC/"
    }
]
```

Example 3 (ID Lookup):

```
GET /api/deposits/4/

HTTP 200 OK
Content-Type: application/json

{
    "id": 4,
    "txid": "635dd656b3bd8c61699e6066c9b3c6e74696e195",
    "coin": "http://127.0.0.1:8000/api/coins/SGTK/",
    "vout": 0,
    "status": "conv",
    "tx_timestamp": "2019-03-20T03:46:30Z",
    "address": null,
    "from_account": "privex",
    "to_account": "someguy123",
    "amount": "1.00000000000000000000",
    "memo": "LTC LKjpPtgMbcFgbJJYwzfe1ZtR8x4bbs2V3o",
    "processed_at": "2019-03-20T04:31:30.643406Z",
    "convert_to": "http://127.0.0.1:8000/api/coins/LTC/"
}
```

4.4 /api/conversions/

`/api/conversions/` Returns all successful conversions sent by the system. Can be filtered using the **GET Parameters** listed below.

`/api/conversions/<id>` Returns a single conversion by it's ID

Methods: GET

GET Parameters:

These parameters can be used with the plain `/api/conversions/` URL, to filter conversions based on various columns.

Note: Results from `/api/conversions/` will always be returned as a list, even if there's only one.

Parameter	Type	Description
to_address	String	Return conversions that were sent to this address or account (it's used for both)
to_txid	String	Return conversions with this outgoing TXID
to_coin	String	Return conversions into this coin symbol
from_coin	String	Return conversions from this coin symbol
from_address	String	Return conversions that were sent from this address or account (it's used for both)

Return Data:

Note: The `to_amount` is the final amount that the user should have received AFTER `ex_fee` and `tx_fee` were removed.

Example 1 (Plain GET request):

```
GET /api/conversions/

HTTP 200 OK
```

(continues on next page)

(continued from previous page)

```
Content-Type: application/json
[
    {
        "url": "http://127.0.0.1:8000/api/conversions/6/",
        "from_address": "LKjpPtgMbcFgbJJYwzfe1ZtR8x4bbs2V3o",
        "to_address": "LKjpPtgMbcFgbJJYwzfe1ZtR8x4bbs2V3o",
        "to_memo": "Token Conversion from SGTK account privex",
        "to_amount": "0.00883200000000000000",
        "to_txid":
↪"e4a5cb3ccc5524e20a39b1a076cef16a85efc68bf929e7a3ec4a834c30711e55",
        "tx_fee": "0.00016800000000000000",
        "ex_fee": "0.00100000000000000000",
        "created_at": "2019-03-21T10:14:20.021360Z",
        "updated_at": "2019-03-21T10:14:20.021373Z",
        "deposit": "http://127.0.0.1:8000/api/deposits/10/",
        "from_coin": "http://127.0.0.1:8000/api/coins/SGTK/",
        "to_coin": "http://127.0.0.1:8000/api/coins/LTC/"
    },
    {
        "url": "http://127.0.0.1:8000/api/conversions/7/",
        "from_address": "someguy123",
        "to_address": "privex",
        "to_memo": "Token Conversion via LTC deposit address␣
↪MTcPHSipXBzwhTWT8wXMtNf6vwAxovjpx9",
        "to_amount": "900.00000000000000000000",
        "to_txid": "55c30e43088c8aa6d7a74da1e29d3843cd7157e7",
        "tx_fee": "0.00000000000000000000",
        "ex_fee": "100.00000000000000000000",
        "created_at": "2019-03-21T10:15:47.071323Z",
        "updated_at": "2019-03-21T10:15:47.071340Z",
        "deposit": "http://127.0.0.1:8000/api/deposits/9/",
        "from_coin": "http://127.0.0.1:8000/api/coins/LTC/",
        "to_coin": "http://127.0.0.1:8000/api/coins/SGTK/"
    }
]
```

Example 2 (Filtering results):

```
GET /api/conversions/?from_coin=SGTK&to_coin=LTC

HTTP 200 OK
Content-Type: application/json
[
    {
        "url": "http://127.0.0.1:8000/api/conversions/6/",
        "from_address": "LKjpPtgMbcFgbJJYwzfe1ZtR8x4bbs2V3o",
        "to_address": "LKjpPtgMbcFgbJJYwzfe1ZtR8x4bbs2V3o",
        "to_memo": "Token Conversion from SGTK account privex",
        "to_amount": "0.00883200000000000000",
        "to_txid":
↪"e4a5cb3ccc5524e20a39b1a076cef16a85efc68bf929e7a3ec4a834c30711e55",
        "tx_fee": "0.00016800000000000000",
        "ex_fee": "0.00100000000000000000",
        "created_at": "2019-03-21T10:14:20.021360Z",
        "updated_at": "2019-03-21T10:14:20.021373Z",
        "deposit": "http://127.0.0.1:8000/api/deposits/10/",
        "from_coin": "http://127.0.0.1:8000/api/coins/SGTK/",
```

(continues on next page)

(continued from previous page)

```
        "to_coin": "http://127.0.0.1:8000/api/coins/LTC/"
    },
    {
        "url": "http://127.0.0.1:8000/api/conversions/5/",
        "from_address": "LKjpPtgMbcFgbJJYwzfe1ZtR8x4bbs2V3o",
        "to_address": "LKjpPtgMbcFgbJJYwzfe1ZtR8x4bbs2V3o",
        "to_memo": "Token Conversion from SGTK account privex",
        "to_amount": "0.00433200000000000000",
        "to_txid": null,
        "tx_fee": "0.00016800000000000000",
        "ex_fee": "0.00050000000000000000",
        "created_at": "2019-03-20T04:56:53.859675Z",
        "updated_at": "2019-03-20T04:56:53.859691Z",
        "deposit": "http://127.0.0.1:8000/api/deposits/7/",
        "from_coin": "http://127.0.0.1:8000/api/coins/SGTK/",
        "to_coin": "http://127.0.0.1:8000/api/coins/LTC/"
    }
]
```

Example 3 (ID Lookup):

```
GET /api/conversions/5/

HTTP 200 OK
Content-Type: application/json

{
    "url": "http://127.0.0.1:8000/api/conversions/5/",
    "from_address": "LKjpPtgMbcFgbJJYwzfe1ZtR8x4bbs2V3o",
    "to_address": "LKjpPtgMbcFgbJJYwzfe1ZtR8x4bbs2V3o",
    "to_memo": "Token Conversion from SGTK account privex",
    "to_amount": "0.00433200000000000000",
    "to_txid": null,
    "tx_fee": "0.00016800000000000000",
    "ex_fee": "0.00050000000000000000",
    "created_at": "2019-03-20T04:56:53.859675Z",
    "updated_at": "2019-03-20T04:56:53.859691Z",
    "deposit": "http://127.0.0.1:8000/api/deposits/7/",
    "from_coin": "http://127.0.0.1:8000/api/coins/SGTK/",
    "to_coin": "http://127.0.0.1:8000/api/coins/LTC/"
}
```

4.5 /api/pairs/

`/api/pairs/` Returns all coin pairs supported by the system Can be filtered using the **GET Parameters** listed below.

`/api/pairs/<id>` Returns a single coin pair by it's ID

Methods: GET

GET Parameters:

These parameters can be used with the plain `/api/pairs/` URL, to filter coin pairs based on from/to symbol.

Note: Results from `/api/pairs/` will always be returned as a list, even if there's only one.

Parameter	Type	Description
to_coin	String	Return pairs with this destination coin symbol
from_coin	String	Return pairs with this deposit coin symbol

Example 1 (Plain GET request):

```
GET /api/pairs/

HTTP 200 OK
Content-Type: application/json

[
    {
        "id": 1,
        "from_coin": "http://127.0.0.1:8000/api/coins/LTC/",
        "from_coin_symbol": "LTC",
        "to_coin": "http://127.0.0.1:8000/api/coins/SGTK/",
        "to_coin_symbol": "SGTK",
        "exchange_rate": "100000.00000000000000000000",
        "__str__": "LTC -> SGTK (100000.0000 SGTK per LTC)"
    },
    {
        "id": 2,
        "from_coin": "http://127.0.0.1:8000/api/coins/SGTK/",
        "from_coin_symbol": "SGTK",
        "to_coin": "http://127.0.0.1:8000/api/coins/LTC/",
        "to_coin_symbol": "LTC",
        "exchange_rate": "0.01000000000000000000",
        "__str__": "SGTK -> LTC (0.0100 LTC per SGTK)"
    }
]
```

Example 2 (Filtering results):

```
GET /api/pairs/?from_coin=LTC

HTTP 200 OK
Content-Type: application/json

[
    {
        "id": 1,
        "from_coin": "http://127.0.0.1:8000/api/coins/LTC/",
        "from_coin_symbol": "LTC",
        "to_coin": "http://127.0.0.1:8000/api/coins/SGTK/",
        "to_coin_symbol": "SGTK",
        "exchange_rate": "100000.00000000000000000000",
        "__str__": "LTC -> SGTK (100000.0000 SGTK per LTC)"
    }
]
```

Example 3 (ID Lookup):

```
GET /api/pairs/1/

HTTP 200 OK
```

(continues on next page)

(continued from previous page)

```
Content-Type: application/json

{
    "id": 1,
    "from_coin": "http://127.0.0.1:8000/api/coins/LTC/",
    "from_coin_symbol": "LTC",
    "to_coin": "http://127.0.0.1:8000/api/coins/SGTK/",
    "to_coin_symbol": "SGTK",
    "exchange_rate": "100000.00000000000000000000",
    "__str__": "LTC -> SGTK (100000.0000 SGTK per LTC)"
}
```

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

s

Index

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V