aioeos

Apr 02, 2020

Contents

CHAPTER 1

Introduction

aioeos is an ssync Python library for interacting with EOS.io blockchain. Library consists of an async wrapper for Nodeos RPC API, a serializer for basic ABI types like transactions and actions and private key management. Helpers for generating actions such as creating new accounts, buying and selling RAM etc. can be imported from *aioeos.contracts* namespace.

Please bear in mind that the serializer is not complete. Action payloads need to be converted to binary format using */abi_json_to_bin* endpoint on the RPC node. Use only nodes you trust.

1.1 Features

1. Async JSON-RPC client.
2. Signing and verifying transactions using private and public keys.
3. Serializer for basic EOS.io blockchain ABI types.
4. Helpers which provide an easy way to generate common actions, such as token transfer.

1.2 Missing features

1. Serializer and deserializer for action payloads.
2. Support for types:
    - bool,
    - uint128,
    - int128,
    - float128,
    - block_timestamp_type,

- symbol,
- symbol_code,
- asset,
- checksum160,
- checksum256,
- checksum512,
- public_key,
- private_key,
- signature,
- extended_asset

1.3 Submitting your first transaction

Let's serialize and submit a basic transaction to EOS.io blockchain. We can think about a transaction as a set of contract calls that we want to execute. These are called actions. Along with the action itself, we provide a list of authorizations. These are defined per action. It basically tells the blockchain which keys will be used to sign this transaction.

Let's say we want to transfer 1.0000 EOS from *myaddress* to *ulamlabscoin* account.

```
from aioeos.contracts import eosio_token
from aioeos.types import EosAuthorization, EosTransaction

action = eosio_token.transfer(
    from_addr='myaddress',
    to_addr='ulamlabscoin',
    quantity='1.0000 EOS',
    authorization=[
        EosAuthorization(actor='myaddress', permission='active')
    ]
)
```

Because aioeos doesn't currently support serialization of action payloads, for this transaction to be ready to be submitted to the blockchain, we need to ask our RPC node to convert it for us. Remember to always **USE ONLY NODES THAT YOU TRUST.**

```
import binascii
from aioeos.rpc import EosJsonRpc

rpc = EosJsonRpc()
abi_bin = await rpc.abi_json_to_bin(
    action.account, action.name, action.data
)
action.data = binascii.unhexlify(abi_bin['binargs'])
```

Now, let's create a transaction containing this action. Each transaction needs to contain TAPOS fields. These tell the EOS.io blockchain when the transaction is considered valid, such as the first block in which it can be included, as well as an expiration date. While we can provide those parameters manually if we want to, we can also use the RPC to find out the right block number and prefix. Let's assume that we want these transaction to be valid since current block, for 2 minutes after it was mined.

```
from datetime import datetime, timedelta
import pytz

info = await rpc.get_info()
block = await rpc.get_block(info['head_block_num'])

expiration = datetime.fromisoformat(block['timestamp']).replace(tzinfo=pytz.UTC)
expiration += timedelta(seconds=120)

transaction = EosTransaction(
    expiration=expiration,
    ref_block_num=block['block_num'] & 65535,
    ref_block_prefix=block['ref_block_prefix'],
    actions=[action]
)
```

Transaction is now ready to be submitted to the blockchain. It's time to serialize, sign and push it. An EOS transaction signature is a digest of the following data:

- Chain ID,
- Transaction,
- 32 context-free bytes

While we can hardcode the first one, let's use the data we already got from RPC. Context-free bytes can be left empty.

```
import hashlib
from aioeos.serializer import serialize

chain_id = info.get('chain_id')
serialized_transaction = serialize(transaction)
context_free_bytes = bytes(32)

digest = (
    hashlib.sha256(
        b''.join(
            binascii.unhexlify(chain_id),
            serialized_transaction,
            context_free_bytes
        )
    ).digest()
)
```

For signing, we're going to use EOSKey class. You can initialize it with your private key, public key (if you want to simply verify a signature) or just leave it empty. By default, a new signing key will be generated.

```
from aioeos.keys import EOSKey

key = EOSKey(private_key=my_private_key)
signature = key.sign(digest)
```

A signed and serialized transaction can be now submitted to the blockchain:

```
response = await rpc.push_transaction(
    signatures=[signature],
    serialized_transaction=binascii.hexlify(serialized_transaction).decode()
)
```

CHAPTER 2

API

2.1 Contracts

2.1.1 eosio

Helpers for creating actions on eosio contract

aioeos.contracts.eosio.**buyrambytes**(*payer*, *receiver*, *amount*, *authorization=[]*) → aioeos.types.EosAction

aioeos.contracts.eosio.**delegatebw**(*from_account*, *receiver*, *stake_net_quantity*, *stake_cpu_quantity*, *transfer=False*, *authorization=[]*) → aioeos.types.EosAction

aioeos.contracts.eosio.**newaccount**(*creator*, *account_name*, *owner_keys*, *active_keys=None*, *authorization=[]*) → aioeos.types.EosAction

aioeos.contracts.eosio.**sellram**(*account*, *amount*, *authorization=[]*) → aioeos.types.EosAction

aioeos.contracts.eosio.**undelegatebw**(*from_account*, *receiver*, *unstake_net_quantity*, *unstake_cpu_quantity*, *authorization=[]*) → aioeos.types.EosAction

2.1.2 eosio_token

Helpers for creating actions on eosio.token contract

aioeos.contracts.eosio_token.**close**(*owner*, *symbol*, *authorization=[]*) → aioeos.types.EosAction

aioeos.contracts.eosio_token.**transfer**(*from_addr: str*, *to_addr: str*, *quantity: str*, *memo: str = ''*, *authorization=[]*) → aioeos.types.EosAction

2.2 Exceptions

exception aioeos.exceptions.**EosAccountDoesntExistException**
Thrown by get_account where account doesn't exist

exception aioeos.exceptions.**EosAccountExistsException**
Thrown by create_wallet where account with given name already exists

exception aioeos.exceptions.**EosActionValidateException**
Raised when action payload is invalid

exception aioeos.exceptions.**EosAssertMessageException**
Generic assertion error from smart contract, can mean literally anything, need to parse C++ traceback to figure out what went wrong.

exception aioeos.exceptions.**EosDeadlineException**
Transaction timed out

exception aioeos.exceptions.**EosMissingTaposFieldsException**
TAPOS fields are missing from Transaction object

exception aioeos.exceptions.**EosRamUsageExceededException**
Transaction requires more RAM than what's available on the account

exception aioeos.exceptions.**EosRpcException**
Base EOS exception

exception aioeos.exceptions.**EosSerializerUnsupportedTypeException**
Our serializer doesn't support provided object type, shouldn't happen

exception aioeos.exceptions.**EosTxCpuUsageExceededException**
Not enough EOS were staked for CPU

exception aioeos.exceptions.**EosTxNetUsageExceededException**
Not enough EOS were staked for NET

2.3 Keys

class aioeos.keys.**EOSKey**(*private_key=None*, *public_key=None*)
EOSKey instance.

Depends on which kwargs are given, this works in a different way: - No kwargs - generates a new private key - Only private_key - public key is being derived from private key - Only public_key - EOSKey instance has no private key

sign(*digest*)
Signs sha256 hash with private key. Returns signature in format: *SIG_K1_{digest}*

to_public()
Returns compressed, base58 encoded public key prefixed with EOS

to_pvt(*key_type='K1'*)
Converts private key to PVT format

to_wif()
Converts private key to legacy WIF format

verify(*encoded_sig*, *digest*)
Verifies signature with private key

2.4 RPC

class aioeos.rpc.**EosJsonRpc**(*url*)

abi_json_to_bin(*code*, *action*, *args*)

get_abi(*account_name: str*)

get_account(*account_name: str*)

get_actions(*account_name: str*, *pos=None*, *offset=None*)

get_block(*block_num_or_id*)

get_block_header_state(*block_num_or_id*)

get_code(*account_name: str*)

get_controlled_accounts(*account_name: str*)

get_currency_balance(*code: str*, *account: str*, *symbol: str*)

get_currency_stats(*code: str*, *symbol: str*)

get_db_size()

get_info()

get_key_accounts(*public_key*)

get_producer_schedule()

get_producers(*json=True*, *lower_bound=''*, *limit=50*)

get_raw_abi(*account_name: str*)

get_raw_code_and_abi(*account_name: str*)

get_required_keys(*transaction*, *available_keys*)

get_table_by_scope(*code*, *table*, *lower_bound=''*, *upper_bound=''*, *limit=10*)

get_table_rows(*code*, *scope*, *table*, *table_key=''*, *lower_bound=''*, *upper_bound=''*, *index_position=1*, *key_type=''*, *limit=10*, *reverse=False*, *show_payer=False*, *json=True*)

get_transaction(*tx_id: str*, *block_num_hint=None*)

post(*endpoint*, *json={}*)

push_transaction(*signatures*, *serialized_transaction*)

2.5 Serializer

class aioeos.serializer.**AbiBytesSerializer**

deserialize(*value*)

serialize(*value*)

class aioeos.serializer.**AbiListSerializer**(*list_type*)

deserialize(*value*)

serialize(*value*)

class aioeos.serializer.**AbiNameSerializer**

deserialize(*value*)

serialize(*value*)

class aioeos.serializer.**AbiStringSerializer**

EOS String format is similar to bytes as it's prefixed with length and is comprised of ASCII codes for each character packed in binary format.

deserialize(*value*)

serialize(*value*)

class aioeos.serializer.**AbiTimePointSecSerializer**

deserialize(*value*)

serialize(*value*)

class aioeos.serializer.**AbiTimePointSerializer**

deserialize(*value*)

serialize(*value*)

class aioeos.serializer.**BaseSerializer**

deserialize(*value*)

serialize(*value*)

class aioeos.serializer.**BasicTypeSerializer**(*fmt=None*)

deserialize(*value*)

serialize(*value*)

class aioeos.serializer.**VarUIntSerializer**

deserialize(*value*)

serialize(*value*)

aioeos.serializer.**decode**(*abi_class*, *field_name*, *binary*)

aioeos.serializer.**decode_eos_type**(*eos_type*, *binary*)

aioeos.serializer.**deserialize**(*binary*, *abi_class*)

aioeos.serializer.**encode**(*obj*, *field_name*)

aioeos.serializer.**encode_eos_type**(*eos_type*, *value*)

aioeos.serializer.**serialize**(*obj*)

2.6 Types

class aioeos.types.**BaseAbiObject**

class aioeos.types.**EosAction**(*account: <function NewType.<locals>.new_type at 0x7fc89a5c36a8>, name: <function NewType.<locals>.new_type at 0x7fc89a5c36a8>, authorization: List[aioeos.types.EosAuthorization], data: <function NewType.<locals>.new_type at 0x7fc89a5c38c8>*)

class aioeos.types.**EosAuthorization**(*actor: <function NewType.<locals>.new_type at 0x7fc89a5c36a8>, permission: <function NewType.<locals>.new_type at 0x7fc89a5c36a8>*)

class aioeos.types.**EosExtension**(*extension_type: <function NewType.<locals>.new_type at 0x7fc89a5c31e0>, data: <function NewType.<locals>.new_type at 0x7fc89a5c38c8>*)

class aioeos.types.**EosTransaction**(*expiration: <function NewType.<locals>.new_type at 0x7fc89a5c37b8> = None, ref_block_num: <function NewType.<locals>.new_type at 0x7fc89a5c31e0> = 0, ref_block_prefix: <function NewType.<locals>.new_type at 0x7fc89a5c3268> = 0, max_net_usage_words: <function NewType.<locals>.new_type at 0x7fc89a5c3950> = 0, max_cpu_usage_ms: <function NewType.<locals>.new_type at 0x7fc89a64c9d8> = 0, delay_sec: <function NewType.<locals>.new_type at 0x7fc89a5c3950> = 0, context_free_actions: List[aioeos.types.EosAction] = <factory>, actions: List[aioeos.types.EosAction] = <factory>, transaction_extensions: List[aioeos.types.EosExtension] = <factory>*)

delay_sec = 0

expiration = None

max_cpu_usage_ms = 0

max_net_usage_words = 0

ref_block_num = 0

ref_block_prefix = 0

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

G

M

N

P

R

S

T

U

V