aioeos

Maciej Janiszewski

Apr 03, 2020

CONTENTS

CHAPTER

ONE

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 Getting Started

This guide step-by-step explains how to use aioeos library to submit your first transaction. Complete example is available at the end of this chapter. Before we begin, please make sure you have `cleos` utility installed in your system (part of `eosio` package) and that `aioeos` is installed.

On macOS:

```
$ brew install eosio
$ pip install aioeos
```

1.3.1 Running your testnet

Along with the library, we provide an EOS testnet Docker image. Due to this issue we recommend cloning the eos-testnet repository and running `ensure_eosio.sh` script.

```
$ git clone https://github.com/ulamlabs/eos-testnet.git
$ cd eos-testnet
$ ./ensure_eosio.sh
# You can check that server is running
$ cleos get info
```

Image by default comes with a hardcoded test account:

- Account name: `eostest12345`
- Private key: `5JeaxignXEg3mGwvgmwxG6w6wHcRp9ooPw81KjrP2ah6TWSECDN`
- Public key: `EOS8VhvYTcUMwp9jFD8UWRMPgWsGQoqBfpBvrjjfMCouqRH9JF5qW`

You can parametrize this through env variables, please refer to the Docker image README.

Let's create another account to send funds to.

```
# If you don't have a wallet yet, otherwise open it and unlock
$ cleos wallet create -f ~/.eosio-wallet-pass

# Import keys for eostest12345 account
$ cleos wallet import --private-key␣
↪5JeaxignXEg3mGwvgmwxG6w6wHcRp9ooPw81KjrP2ah6TWSECDN

# Create your second account, for example mysecondacc1
$ cleos system newaccount eostest12345 --transfer mysecondacc1␣
↪EOS8VhvYTcUMwp9jFD8UWRMPgWsGQoqBfpBvrjjfMCouqRH9JF5qW --stake-net "1.0000 EOS" --
↪stake-cpu "1.0000 EOS" --buy-ram-kbytes 8192
```

1.3.2 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 *eostest12345* to *mysecondacc1* account.

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

action = eosio_token.transfer(
    from_addr='eostest12345',
    to_addr='mysecondacc1',
    quantity='1.0000 EOS',
    authorization=[
        EosAuthorization(actor='eostest12345', 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(url='http://127.0.0.1:8888')
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='5JeaxignXEg3mGwvgmwxG6w6wHcRp9ooPw81KjrP2ah6TWSECDN')
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()
)
```

1.3.3 Example code

Complete example code:

```
import asyncio
import binascii
from datetime import datetime, timedelta
import hashlib

import pytz

from aioeos.serializer import serialize
from aioeos.contracts import eosio_token
from aioeos.keys import EOSKey
from aioeos.rpc import EosJsonRpc
from aioeos.types import EosAuthorization, EosTransaction
```

(continues on next page)

(continued from previous page)

```
async def example():
    action = eosio_token.transfer(
        from_addr='eostest12345',
        to_addr='mysecondacc1',
        quantity='1.0000 EOS',
        authorization=[
            EosAuthorization(actor='eostest12345', permission='active')
        ]
    )

    rpc = EosJsonRpc(url='http://127.0.0.1:8888')
    abi_bin = await rpc.abi_json_to_bin(
        action.account, action.name, action.data
    )
    action.data = binascii.unhexlify(abi_bin['binargs'])

    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]
    )

    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()
    )

    key = EOSKey(
        private_key='5JeaxignXEg3mGwvgmwxG6w6wHcRp9ooPw81KjrP2ah6TWSECDN'
    )
    signature = key.sign(digest)

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


asyncio.get_event_loop().run_until_complete(example())
```

CHAPTER

TWO

API

2.1 Contracts

2.1.1 eosio

Helpers for creating actions on eosio contract

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

Return type *EosAction*

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

Return type *EosAction*

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

Return type *EosAction*

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

Return type *EosAction*

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

Return type *EosAction*

2.1.2 eosio_token

Helpers for creating actions on eosio.token contract

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

Return type *EosAction*

aioeos.contracts.eosio_token.**transfer**(*from_addr*, *to_addr*, *quantity*, *memo=''*, *authorization=[]*)

Return type *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.**EosSerializerAbiNameInvalidCharactersException**

exception aioeos.exceptions.**EosSerializerAbiNameTooLongException**

exception aioeos.exceptions.**EosSerializerException**
Base exception class for serializer errors

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

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

2.5 Serializer

class aioeos.serializer.**AbiBytesSerializer**
Serializer for ABI bytes type. Serialized value consists of raw bytes prefixed with payload size encoded as VarUInt.

deserialize(*value*)
Returns a tuple containing length of original data and deserialized value

Return type Tuple[int, bytes]

serialize(*value*)
Returns byte-encoded value

Return type bytes

class aioeos.serializer.**AbiListSerializer**(*list_type*)
Serializer for ABI List type. In binary format, it basically looks like this: [count][item 1][item 2]...

deserialize(*value*)
Returns a tuple containing length of original data and deserialized value

Return type Tuple[int, List[Any]]

serialize(*value*)
Returns byte-encoded value

Return type bytes

class aioeos.serializer.**AbiNameSerializer**
Serializer for ABI names. ABI names can only contain these characters: .12345abcdefghijklmnopqrstuvwxyz. Maximum length is 13 chars.

deserialize(*value*)
Returns a tuple containing length of original data and deserialized value

Return type Tuple[int, str]

serialize(*value*)
Returns byte-encoded value

Return type bytes

class aioeos.serializer.**AbiObjectSerializer**(*abi_class*)

deserialize(*value*)
Returns a tuple containing length of original data and deserialized value

Return type Tuple[int, *BaseAbiObject*]

serialize(*value*)
Returns byte-encoded value

Return type bytes

class aioeos.serializer.**AbiStringSerializer**
Serializer for ABI String type. String format is similar to bytes as it's prefixed with length but it's comprised of ASCII codes for each character packed in binary format.

deserialize(*value*)
Returns a tuple containing length of original data and deserialized value

Return type Tuple[int, str]

serialize(*value*)
Returns byte-encoded value

Return type bytes

class aioeos.serializer.**AbiTimePointSecSerializer**
Serializer for ABI TimePointSec type. It's essentially a timestamp.

deserialize(*value*)
Returns a tuple containing length of original data and deserialized value

Return type Tuple[int, datetime]

serialize(*value*)
Returns byte-encoded value

Return type bytes

class aioeos.serializer.**AbiTimePointSerializer**
Serializer for ABI TimePoint type. Encodes timestamp with milisecond precision.

deserialize(*value*)
Returns a tuple containing length of original data and deserialized value

Return type Tuple[int, datetime]

serialize(*value*)
Returns byte-encoded value

Return type bytes

class aioeos.serializer.**BaseSerializer**

abstract deserialize(*value*)
Returns a tuple containing length of original data and deserialized value

Return type Tuple[int, Any]

abstract serialize(*value*)
Returns byte-encoded value

Return type bytes

class aioeos.serializer.**BasicTypeSerializer**(*fmt=''*)
Serializes basic types such as integers and floats using `struct` module

Params fmt format string, please refer to documentation for struct module

deserialize(*value*)
Returns a tuple containing length of original data and deserialized value

Return type Tuple[int, Any]

serialize(*value*)
Returns byte-encoded value

Return type bytes

class aioeos.serializer.**VarUIntSerializer**
Serializer for ABI VarUInt type. This type has different length based on how many bytes are required to encode given integer.

deserialize(*value*)
Returns a tuple containing length of original data and deserialized value

Return type Tuple[int, int]

serialize(*value*)
Returns byte-encoded value

Return type bytes

aioeos.serializer.**deserialize**(*value*, *abi_class*)
Deserializes ABI values from binary format

Return type Tuple[int, Any]

aioeos.serializer.**get_abi_type_serializer**(*abi_type*)

Return type *BaseSerializer*

aioeos.serializer.**serialize**(*value*, *abi_type=None*)
Serializes ABI values to binary format

Return type bytes

2.6 Types

class aioeos.types.**BaseAbiObject**

class aioeos.types.**EosAction**(*account*, *name*, *authorization*, *data*)

account: Name = None

authorization: List[EosAuthorization] = None

data: AbiActionPayload = None

name: Name = None

class aioeos.types.**EosAuthorization**(*actor*, *permission*)

actor: Name = None

permission: Name = None

class aioeos.types.**EosExtension**(*extension_type*, *data*)

data: AbiBytes = None

extension_type: UInt16 = None

class aioeos.types.**EosTransaction**(*expiration=<factory>*, *ref_block_num=0*, *ref_block_prefix=0*, *max_net_usage_words=0*, *max_cpu_usage_ms=0*, *delay_sec=0*, *context_free_actions=<factory>*, *actions=<factory>*, *transaction_extensions=<factory>*)

actions: List[EosAction] = None

context_free_actions: List[EosAction] = None

delay_sec: int = 0

expiration: TimePointSec = None

max_cpu_usage_ms: int = 0

max_net_usage_words: int = 0

ref_block_num: int = 0

ref_block_prefix: int = 0

transaction_extensions: List[EosExtension] = None

CHAPTER

THREE

CHANGELOG

3.1 1.0.1 (03.04.2020)

- Improved docs and “Getting Started” guide,
- Added .coveragerc, fixed abstract class coverage,
- Added mypy,
- Fixed typechecking errors,
- Added docstrings,
- Added missing validation,
- Dropped custom String type,
- Cleaned up serializer.py,
- Bumped base58 to 2.0.0,
- Added tests for RPC client,
- Fixed typehints in docs,
- Added check for building docs

3.2 1.0.0 (01.04.2020)

- Initial release

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

INDEX