litewax Documentation

Release 0.1.8.dev5

abuztrade

Aug 11, 2023

CONTENTS

litewax - library for iteracting with WAX and other EOS-type blockchains. It is an addition to the official library eospy.

CHAPTER

ONE

CONTENTS

1.1 Installation Guide

1.1.1 Using PIP

```
$ pip install -U litewax
```

1.1.2 Using Pipenv

```
$ pipenv install litewax
```

1.1.3 From sources

Development versions:

```
$ git clone https://github.com/makarworld/litewax.git
$ cd litewax
$ python setup.py install
```

Or if you want to install stable version (The same with version from PyPi):

```
$ git clone https://github.com/makarworld/litewax.git
$ cd litewax
$ git checkout master
$ python setup.py install
```

1.2 Quick start

1.2.1 Simple template

At first you have to import *Client* from litewax

```
from litewax import Client
```

Then you have to initialize *Client* with private key or session_token from Wax Cloud Wallet.

Note: how-to-get-session-token

Initialize *Client* with private key:

```
# Create a client with a private key
client = Client(private_key="5K...")
```

Initialize *Client* with session_token:

```
# Create a client with a session token from https://wallet.wax.io/. Guide: https://
↪litewax.readthedocs.io/en/latest/other/get_token_session.html
client = Client(cookie="V5cS...vhF3")
```

Next step: Create you first *Transaction*. For example, transfer tokens.

```
# to - the account send the tokens to
to = "abuztradewax"

# Create a eosio.token contract object
contract = client.Contract("eosio.token")

# Create a transaction object
trx = client.Transaction(
    contract.transfer(
        _from=client.name,
        to=to,
        quantity="1.00000000 WAX",
        memo="Test send"
    )
)
```

Last step: Push *Transaction* to the blockchain.

```
# Push the transaction
r = trx.push()

print(r)
```

See also:

Different ways to set a payer for CPU

1.2.2 Summary

```python
from litewax import Client

# Create a client with a private key
client = Client(private_key="5K...")

# to - the account send the tokens to
to = "abuztradewax"

# Create a eosio.token contract object
contract = client.Contract("eosio.token")

# Create a transaction object
trx = client.Transaction(
    contract.transfer(
        _from=client.name,
        to=to,
        quantity="1.00000000 WAX",
        memo="Test send"
    )
)

# Push the transaction
r = trx.push()

print(r)
```

1.3 Different ways to set a payer for CPU

1.3.1 Simple template

All transactions in EOS-based blockchains use the CPU to process transaction by node. See CPU bandwidth for understanding how CPU works.

See also:

- How to set a payer
- CPU bandwidth
- Delegate CPU
- Undelegate CPU

In litewax you can set the payer of the CPU for a transaction. Before, import and initialize 2 clients, one for the payer and one for the sender.

```python
from litewax import Client, WAXPayer

# create sender Client instance
sender = Client(private_key='5K1...')
```

(continues on next page)

(continued from previous page)

```
# create payer Client instance
payer = Client(private_key='5K2...')
```

Next: Create a *Transaction*.

```
# create transaction
trx = sender.Transaction(
    sender.Contract('eosio.token').transfer(
        _from=sender.name,
        to='receiver',
        quantity='1.00000000 WAX',
        memo='memo'
    )
)
```

Now, set the payer of the CPU for the *Transaction*. Payer can be set in different ways:

- by other *Client*.

```
trx = trx.payer(payer)
```

- by litewax.payers.AtomicHub. If you don't have enough CPU, and actions in whitelist by atomichub like: atomicassets.transfer(). (Only WAX mainnet)

```
trx = trx.payer(WAXPayer.ATOMICHUB)
```

- by litewax.payers.NeftyBlocks. If actions in whitelist by neftyblocks like: neftyblocksd.claim_drop(). (Only WAX mainnet and testnet)

```
trx = trx.payer(WAXPayer.NEFTYBLOCKS)
```

- or if you use a *Multi Client*, you can set the payer when creating a *MultiTransaction*.

```
from litewax import MultiClient

# create MultiClient instance
client = MultiClient(private_keys=['5K1...', '5K2...', '5K3...'])

# create transaction.
# 1st client send 1 WAX to 2nd client,
# 2nd client send 1 WAX to 1st client,
# 3rd client pay CPU.
trx = sender.Transaction(
    # some 1st action
    client[0].Contract('eosio.token').transfer(
        _from=client[0].name,
        to=client[1].name,
        quantity='1.00000000 WAX',
        memo='memo'
    ),

    # some 2nd action
    client[1].Contract('eosio.token').transfer(
```

(continues on next page)

(continued from previous page)

```
        _from=client[1].name,
        to=client[0].name,
        quantity='1.00000000 WAX',
        memo='memo'
    ),

    # add last action for pay CPU. You can use any contract and action. litewax owner␣
↪created a custom empty contract, which has only one .noop() action in mainnet and␣
↪testnet.
    client[2].Contract('litewaxpayer').noop()
)
```

Last step: Push transaction.

```
# send transaction
trx.push()
```

Note: If you set the payer of the CPU for the transaction, you must have enough CPU for the payer. If you don't have enough CPU, you can delegate CPU to the payer.

See Delegate CPU for more information.

See also:

Contract usage examples

1.3.2 Summary

```
from litewax import Client, MultiClient, WAXPayer

# 1. create sender and payer Client instances
sender = Client(private_key='5K1...')
payer = Client(private_key='5K2...')

# or create MultiClient instance
# client = MultiClient(private_keys=['5K1...', '5K2...', '5K3...'])

# 2. create transaction with sender
trx = sender.Transaction(
    sender.Contract('eosio.token').transfer(
        _from=sender.name,
        to='receiver',
        quantity='1.00000000 WAX',
        memo='memo'
    )
)

# or create payed transaction with MultiClient
# create transaction.
# 1st client send 1 WAX to 2nd client,
```

(continues on next page)

(continued from previous page)

```
# 2nd client send 1 WAX to 1st client,
# 3rd client pay CPU.
#
# trx = sender.Transaction(
#     # some 1st action
#     client[0].Contract('eosio.token').transfer(
#         _from=client[0].name,
#         to=client[1].name,
#         quantity='1.00000000 WAX',
#         memo='memo'
#     ),
#
#     # some 2nd action
#     client[1].Contract('eosio.token').transfer(
#         _from=client[1].name,
#         to=client[0].name,
#         quantity='1.00000000 WAX',
#         memo='memo'
#     ),
#
#     # add last action for pay CPU. You can use any contract and action. litewax owner␣
↪created a custom empty contract, which has only one .noop() action in mainnet and␣
↪testnet.
#     client[2].Contract('litewaxpayer').noop()
# )

# 3. set payer of the CPU for the transaction
trx = trx.payer(payer)

# or set atomichub as a payer
# trx = trx.payer(WAXPayer.ATOMICHUB)

# or set neftyblocks as a payer
# trx = trx.payer(WAXPayer.NEFTYBLOCKS)

# 4. send transaction
resp = trx.push()
print(resp)
# {"transaction_id": "b0e...", ...}
```

1.4 Contract usage examples

All transaction in EOS-based blockchain are executed by smart contracts. litewax have *abigen* for generate .py files from contract abi which will got from node.

For example, we can generate *eosio_token.py* contract:

```
$ python -c "from litewax import Contract; Contract('eosio.token')"
```

Then we can use *eosio_token.py* to call *eosio.token* contract:

```
from contracts.eosio_token import eosio_token
contract = eosio_token(actor='alice', permission='active')
single_action = contract.transfer('alice', 'bob', '1.00000000 WAX', 'memo')
```

You also can send transaction without any litewax *Client*. Only with generated contract file:

```
from contracts.eosio_token import eosio_token
contract = eosio_token(actor='alice', permission='active')
contract.push_actions(
    private_keys=['5K...'],
    contract.transfer('alice', 'bob', '1.00000000 WAX', 'memo')
)
```

For easy iteracting with any contract, litewax have *Contract* function, which create a .py contract file, dynamicly import it and return initialized contract object:

```
from litewax import Contract
contract = Contract('eosio.token', actor='alice')
single_action = contract.transfer('alice', 'bob', '1.00000000 WAX', 'memo')
```

Also you can use *Contract* function in *Client* object:

Note: If you use *Contract* function in *Client* object, you don't need to specify *actor* in contract constructor, but may specify *permission* if need.

```
from litewax import Client
client = Client(private_key='5K...', 'https://wax.greymass.com')
contract = client.Contract('eosio.token')
single_action = contract.transfer('alice', 'bob', '1.00000000 WAX', 'memo')
```

1.5 Client

1.5.1 Client object

Client

Main Client for all interactions with blockchain.

class litewax.clients.**Client**(*private_key: Optional[str] = ''*, *cookie: Optional[str] = ''*, *node: Optional[str] = 'https://wax.greymass.com'*)

Client for interacting with the blockchain

Parameters

- **private_key** (*str*) – Private key string (if cookie is not provided)
- **cookie** (*str*) – WCW session token (if private_key is not provided)
- **node** (*str*) – Node URL

Example

```
>>> from litewax import Client
>>> # init client with private key
>>> client = Client(private_key=private_key)
>>> # or init client with WCW session token
>>> client = Client(cookie=cookie)
>>> client.Transaction(
>>>     client.Contract("eosio.token").transfer(
>>>         "from", "to", "1.00000000 WAX", "memo"
>>>     )
>>> ).push()
```

property root: Union[*AnchorClient*, *WCWClient*]

Root client object

property node: str

Node URL

property wax: Cleos

Eospy cleos object

property name: str

Account Name

property change_node: Callable[[str], None]

Change node URL.

Inherited from *AnchorClient object* or *WCWClient object*

property sign: Callable[[Transaction], Transaction]

Sign transaction.

Inherited from *AnchorClient object* or *WCWClient object*

Contract(*name: str*, *actor: Optional[str] = None*, *force_recreate: Optional[bool] = False*, *node: Optional[str] = None*) → *ExampleContract*

Create a *litewax.contract.ExampleContract* object

Parameters

- **name** (*str*) – contract name
- **actor** (*str*) – actor name
- **force_recreate** (*bool*) – force recreate contract object
- **node** (*str*) – node url

Returns

litewax.contract.ExampleContract object

Return type

litewax.contract.ExampleContract

Example

```
>>> from litewax import Client
>>> # init client with private key
>>> client = Client(private_key=private_key)
```

(continues on next page)

(continued from previous page)

```
>>> # create contract object
>>> contract = client.Contract("eosio.token")
>>> # create action object
>>> action = contract.transfer("from", "to", "1.00000000 WAX", "memo")
>>> # create transaction object
>>> trx = client.Transaction(action)
>>> # push transaction
>>> trx.push()
```

Transaction(**actions: tuple[*Action*, ...]*) → *Transaction*

Create a *litewax.clients.Transaction* object

Parameters

actions (*tuple*) – actions of contracts

Returns

litewax.clients.Transaction object

Return type

litewax.clients.Transaction

Example

```
>>> from litewax import Client
>>> # init client with private key
>>> client = Client(private_key=private_key)
>>> # create transaction object
>>> trx = client.Transaction(
>>>     client.Contract("eosio.token").transfer(
>>>         "from", "to", "1.00000000 WAX", "memo"
>>>     )
>>> )
>>> # push transaction
>>> trx.push()
```

Multi Client

This class based on *Client* It allows you to work with multiple clients at the same time.

class litewax.clients.**MultiClient**(*private_keys: Optional[List[str]] = []*, *cookies: Optional[List[str]] = []*, *clients: Optional[List[*Client*]] = []*, *node: Optional[str] = 'https://wax.greymass.com'*)

Bases: `list`

MultiClient class for interacting with blockchain using many clients.

Parameters

- **private_keys** (*list*) – list of private keys (optional)
- **cookies** (*list*) – list of cookies (optional)
- **clients** (*list*) – list of *litewax.clients.Client* objects (optional)
- **node** (*str*) – node url (optional): default https://wax.greymass.com

Raises
litewax.exceptions.AuthNotFound – if you not provide a private key, a cookie or a clients

Example

```
>>> from litewax import MultiClient
>>> client = MultiClient(
>>>     private_keys = [
>>>         "EOS7...1",
>>>         "EOS7...2",
>>>         "EOS7...3"
>>>     ],
>>>     node = "https://wax.greymass.com"
>>> )
>>> # Change node
>>> client.change_node("https://wax.eosn.io")
>>> # Append client
>>> client.append(Client(private_key="EOS7...4"))
>>> # Create transaction
>>> trx = client.Transaction(
>>>     Contract("eosio.token").transfer(
>>>         "account1", "account2", "1.0000 WAX", "memo"
>>>     )
>>> )
>>> # Add payer
>>> trx = trx.payer(client[2])
>>> # Push transaction
>>> trx.push()
```

property clients: List[*Client*]

Clients list

change_node(*node: str*)

Change node url for all clients

Parameters
node (*str*) – Node URL

Returns

Return type
None

append(*client:* Client) → None

Append client to clients list

Parameters
client (litewax.clients.Client) – *litewax.clients.Client* object

Returns

Return type
None

sign(*trx: bytearray*, *whitelist: Optional[List[str]] = []*, *chain_id: Optional[str] = None*) → List[str]

Sign a transaction with all whitelisted clients

Parameters

- **trx** (*bytearray*) – bytearray of transaction
- **whitelist** (*list*) – list of clients to sign with (optional)
- **chain_id** (*str*) – chain id of the network (optional)

Returns
list of signatures

Return type
list

Transaction(**actions: tuple[*Action, *...]*)

Create a *litewax.clients.MultiTransaction* object

Parameters
actions (*tuple*) – list of actions

Returns
litewax.clients.MultiTransaction object

Return type
litewax.clients.MultiTransaction

Example

```
>>> from litewax import Client, MultiClient
>>> # init client with private key
>>> client1 = Client(private_key=private_key1)
>>> client2 = Client(private_key=private_key2)
>>> multi_client = MultiClient(clients=[client1, client2])
>>> # create transaction object
>>> trx = multi_client.Transaction(
>>>     multi_client[1].Contract("eosio.token").transfer(
>>>         "from", "to", "1.00000000 WAX", "memo"
>>>     ),
>>>     multi_client[0].Contract("litewaxpayer").noop()
>>> )
>>> # push transaction
>>> trx.push()
```

1.5.2 Transaction object

Transaction

Used by *Client* class.

This transaction may be signed with only one client.

May transform to *MultiTransaction* object if you use *litewax.clients.Transaction.payer()* method with *Client* as payer.

class litewax.clients.**Transaction**(*client:* Client, **actions: tuple[*Action, *...]*)

litewax.clients.Transaction object Create a transaction object for pushing to the blockchain

Parameters

- **client** (litewax.clients.Client) – *litewax.clients.Client* object
- **actions** (*tuple[Action, ...]*) – actions of contracts

Example

```
>>> from litewax import Client
>>> # init client with private key
>>> client = Client(private_key=private_key)
>>> # create transaction object
>>> trx = client.Transaction(
>>>     client.Contract("eosio.token").transfer(
>>>         "account1", "account2", "1.00000000 WAX", "memo"
>>>     )
>>> )
>>> print(trx)
litewax.Client.Transaction(
    node=https://wax.greymass.com,
    sender=account1,
    actions=[
        [active] account1 > eosio.token::transfer({"from": "account1", "to":
↪"account2", "quantity": "1.00000000 WAX", "memo": "memo"})
    ]
)
>>> # Add payer for CPU
>>> # init payer client with private key
>>> payer = Client(private_key=private_key2)
>>> # add payer to transaction
>>> trx = trx.payer(payer)
>>> print(trx)
litewax.MultiClient.MultiTransaction(
    node=ttps://wax.greymass.com,
    accounts=[account1, account2],
    actions=[
        [active] account1 > eosio.token::transfer({"from": "account1", "to":
↪"account2", "quantity": "1.00000000 WAX", "memo": "memo"}),
        [active] account2 > litewaxpayer::noop({})
    ]
)
>>> # push transaction
>>> push_resp = trx.push()
>>> print(push_resp)
{'transaction_id': '928802d253bffc29d6178e634052ec5f044b2fcce0c4c8bc5b44d978e22ec5d4
↪', ...}
```

property client: *Client*

Client object

property actions: list[*Action*]

List of actions

payer(*payer: Union[*Client*, atomichub, neftyblocks, str]*, *permission: Optional[str] = 'active'*) → Union[*MultiTransaction*, *AtomicHub*, *NeftyBlocks*]

Set payer for all actions

Parameters

- **payer** (`litewax.clients.Client` *or* `str`) – payer name or *litewax.clients.Client* object

- **permission** (*str*) – payer permission (optional): default *active*

Raises
NotImplementedError – if payer is not *litewax.clients.Client*, *litewax.payers.AtomicHub* or *litewax.payers.NeftyBlocks*.

Returns
litewax.clients.MultiTransaction object or *litewax.payers.AtomicHub* object or *litewax.payers.NeftyBlocks* object

Return type
litewax.clients.MultiTransaction or *litewax.payers.AtomicHub* or *litewax.payers.NeftyBlocks*

pack(*chain_info: Optional[dict] = {}*, *lib_info: Optional[dict] = {}*, *expiration: Optional[int] = 180*)

Pack transaction with client and return *litewax.types.TransactionInfo*.

Parameters

- **chain_info** (*dict*) – chain info. Provide it if you not want to get it from blockchain (optional)
- **lib_info** (*dict*) – lib info. Provide it if you not want to get it from blockchain (optional)
- **expiration** (*int*) – transaction expiration time in seconds (optional): default 180

Returns
litewax.types.TransactionInfo

Return type
litewax.types.TransactionInfo

prepare_trx(*chain_info: Optional[dict] = {}*, *lib_info: Optional[dict] = {}*, *expiration: Optional[int] = 180*) → *TransactionInfo*

Sign transaction with client and return *litewax.types.TransactionInfo*.

Parameters

- **chain_info** (*dict*) – chain info. Provide it if you not want to get it from blockchain (optional)
- **lib_info** (*dict*) – lib info. Provide it if you not want to get it from blockchain (optional)
- **expiration** (*int*) – transaction expiration time in seconds (optional): default 180

Returns
litewax.types.TransactionInfo

Return type
litewax.types.TransactionInfo

push(*data: Optional[*TransactionInfo*] = {}*, *expiration: Optional[int] = 180*) → dict

Push transaction to blockchain

Parameters

- **data** (*litewax.types.TransactionInfo*) – *litewax.types.TransactionInfo* object (optional)
- **expiration** (*int*) – transaction expiration time in seconds (optional): default 180

Raises

- ***litewax.exceptions.CPUlimit*** – if transaction exceeded the current CPU usage limit imposed on the transaction

- *litewax.exceptions.ExpiredTransaction* – if transaction is expired
- *litewax.exceptions.UnknownError* – if unknown error

Returns
transaction information

Return type
dict

MultiTransaction

Used by *Multi Client* class.

This transaction may be signed with many clients.

class litewax.clients.**MultiTransaction**(*client:* MultiClient, **actions: tuple[*Action, *...]*)

MultiTransaction class for creating and pushing transactions using many signatures

Parameters

- **client** (litewax.clients.MultiClient) – *litewax.clients.MultiClient* object
- **actions** (*tuple*) – list of actions

Example

```
>>> from litewax import MultiClient
>>> # init client with private keys
>>> client = MultiClient(
>>>     private_keys = [
>>>         "EOS7...1",
>>>         "EOS7...2"
>>>     ],
>>>     node = "https://wax.greymass.com"
>>> )
>>> # create transaction object
>>> trx = client.Transaction(
>>>     client[0].Contract("eosio.token").transfer(
>>>         "from", "to", "1.00000000 WAX", "memo"
>>>     )
>>> )
>>> # add payer
>>> trx = trx.payer(client[1])
>>> # push transaction
>>> trx.push()
```

property wax: Cleos

eospy.cleos.Cleos

property client: *MultiClient*

client.MultiClient object

property actions: List[*Action*]

Actions list

payer(*payer: Union[*Client, *atomichub, neftyblocks, str]*, *permission: Optional[str] = 'active'*) → Union[*MultiTransaction*, *AtomicHub*, *NeftyBlocks*]

Set payer

Parameters

- **payer** (*str or* litewax.clients.Client) – payer account name or *litewax.clients.Client* object
- **permission** (*str*) – payer permission (optional): default *active*

Raises

NotImplementedError – if payer is not *litewax.clients.Client*, *litewax.payers.AtomicHub* or *litewax.payers.NeftyBlocks*

Returns

litewax.clients.MultiTransaction object or *litewax.payers.AtomicHub* or *litewax.payers.NeftyBlocks* object

Return type

litewax.clients.MultiTransaction or *litewax.payers.AtomicHub* or *litewax.payers.NeftyBlocks*

pack(*chain_info: Optional[dict] = {}*, *lib_info: Optional[dict] = {}*, *expiration: Optional[int] = 180*)

Pack transaction with client and return *litewax.types.TransactionInfo*.

Parameters

- **chain_info** (*dict*) – chain info. Provide it if you not want to get it from blockchain (optional)
- **lib_info** (*dict*) – lib info. Provide it if you not want to get it from blockchain (optional)
- **expiration** (*int*) – transaction expiration time in seconds (optional): default 180

Returns

litewax.types.TransactionInfo

Return type

litewax.types.TransactionInfo

prepare_trx(*chain_info: Optional[dict] = {}*, *lib_info: Optional[dict] = {}*, *expiration: Optional[int] = 180*) → *TransactionInfo*

Sign transaction with clients and return signatures, packed and serialized transaction

Parameters

- **chain_info** (*dict*) – chain info. Provide it if you not want to get it from blockchain (optional)
- **lib_info** (*dict*) – lib info. Provide it if you not want to get it from blockchain (optional)
- **expiration** (*int*) – transaction expiration time in seconds (optional): default 180

Returns

litewax.types.TransactionInfo object

Return type

litewax.types.TransactionInfo

push(*data: Optional[*TransactionInfo*] = {}*, *expiration: Optional[int] = 180*) → dict

Push transaction to blockchain

Parameters

- **data** (litewax.types.TransactionInfo) – *litewax.types.TransactionInfo* object (optional)

- **expiration** (*int*) – transaction expiration time in seconds (optional): default 180

Raises

- *litewax.exceptions.CPUlimit* – if transaction exceeded the current CPU usage limit imposed on the transaction
- *litewax.exceptions.ExpiredTransaction* – if transaction is expired
- *litewax.exceptions.UnknownError* – if unknown error

Returns
transaction information

Return type
dict

1.5.3 BaseClients

BaseClient object

The BaseClient object is the base class for all clients. It provides the basic functionality for all clients. It is not meant to be used directly. Instead, use one of the subclasses. BaseClient supports iteraction with node url and cleos.

class litewax.baseclients.**BaseClient**(*node: str*)

BaseClient is a base client for interacting with the blockchain

Parameters
node (*str*) – Node URL

Returns

__node

__wax

property node: str

Node URL

property wax: Cleos

Cleos instance

change_node(*node: str*) → None

Change node for client by redeffining dynamic_url in *Cleos* instance

Parameters
node (*str*) – Node URL

Returns

AnchorClient object

Class for iteracting with private, public keys, signing transactions. Based on eospy.cleos.

class litewax.baseclients.**AnchorClient**(*private_key: str*, *node: str*)

AnchorClient is a client for interacting with the blockchain using a private key

Parameters

- **private_key** (*str*) – Private key string
- **node** (*str*) – Node URL

Returns

property private_key: EOSKey

Private key

property public_key: EOSKey

Public key

property name: str

Wallet name

get_name() → str

Get wallet name by public key

Raises
KeyError if account not found

Returns
wallet name

Return type
str

sign(*trx: bytearray*) → List[str]

Sign transaction with private key

Parameters
trx – transaction to sign

Returns
signatures (length: 1)

Return type
list

WCWClient object

Class for iteracting with blockchain via Wax Cloud Wallet Client by cookies. Use Wax Cloud Wallet for sing transaction. Provide token session for authorization.

Note: how-to-get-session-token

class litewax.baseclients.**WCWClient**(*cookie: str*, *node: str*)

WCWClient is a client for interacting with the blockchain using a WCW session token

Parameters

- **cookie** (*str*) – WCW session token
- **node** (*str*) – Node URL

Returns

property cookie: str

WCW session token

property session: CloudScraper

CloudScraper instance

property name: str

Wallet name

get_name() → str

Get wallet name by session_token

Raises
litewax.exceptions.CookiesExpired if session token is expired or invalid

Returns
wallet name

Return type
str

sign(*trx: bytearray*) → List[str]

Sign transaction with WCW session token

Parameters
trx (*bytearray*) – transaction to sign

Returns
signatures (length: 2)

Return type
list

1.6 Payers

class litewax.payers.**AtomicHub**(*client*, *trx*, *network='mainnet'*)

Pay for transaction with AtomicHub

Allowed actions: - atomicassets - atomicmarket

Parameters

- **client** (*litewax.clients.MultiClient*) – MultiClient instance
- **trx** (*litewax.clients.MultiTransaction*) – MultiTransaction instance
- **network** (*str*) – network name (default: mainnet)

Raises
NotImplementedError – if network is not mainnet

property trx

MultiTransaction instance

property client

Multi Client instance

property scraper

CloudScraper instance

property sign_link

Link to sign transaction

property push_link

Link to push transaction

push(*signed={}*, *expiration=180*) → dict

Push transaction to blockchain with AtomicHub

Parameters

- **signed** (*dict*) – signed transaction (default: {})
- **expiration** (*int*) – expiration time in seconds (default: 180)

Raises

AtomicHubPushError – if transaction is not signed

Returns

dict with transaction data

Return type

dict

class litewax.payers.**NeftyBlocks**(*client*, *trx*, *network='mainnet'*)

Pay for transaction with NeftyBlocks

Parameters

- **client** (litewax.clients.MultiClient) – *litewax.clients.MultiClient* instance
- **trx** (litewax.clients.MultiTransaction) – *litewax.clients.MultiTransaction* instance
- **network** (*str*) – network name (default: mainnet)

Raises

ValueError – if network is not mainnet or testnet

property client

Multi Client instance

property trx

MultiTransaction instance

property scraper

Cloudscraper instance

property sign_link

Sign link

property push_link

Push link

push(*signed={}*, *expiration=180*) → dict

Push transaction to blockchain with NeftyBlocks

Parameters

- **signed** (*dict*) – signed transaction (default: {})
- **expiration** (*int*) – expiration time in seconds (default: 180)

Raises

NeftyBlocksPushError – if transaction is not signed

Returns

`dict` with transaction data

Return type

dict

1.7 Types

class litewax.types.**WAXPayer**

WAXPayer is a class for storing supported payers

Key NEFTYBLOCKS

neftyblocks payer

Key ATOMICHUB

atomichub payer

class litewax.types.**TransactionInfo**(*signatures: List[str]*, *packed: str*, *serealized: List[int]*)

TransactionInfo is a dataclass for storing transaction info

Key signatures

signatures

Key packed

packed transaction

Key serealized

serealized transaction

1.8 Nodes

class litewax.nodes.**Nodes**

Get nodes from antelope and ping them

static get_nodes(*network='mainnet'*) → list

Get producers nodes from antelope

Parameters

network (*str*) – mainnet or testnet

Returns

list of nodes

Return type

list

static ping_nodes(*network='mainnet'*) → dict

Ping nodes and return dict. key - URL, value - ping (ms)

Parameters
network (*str*) – mainnet or testnet

Returns
dict. key - URL, value - ping (ms)

Return type
dict

static best_nodes(*network='mainnet'*) → dict

Search best nodes for you. It may take a 20-50 sec. to get the result.

Parameters
network (*str*) – mainnet or testnet

Returns
sorted dict. key - URL, value - ping (ms)

Return type
dict

static best_node(*network='mainnet'*) → str

Returns the best node with the lowest ping. It may take a 20-50 sec. to get the result.

Parameters
network (*str*) – mainnet or testnet

Returns
URL

Return type
str

1.9 Contract

litewax.contract.**Contract**(*name: str*, *client: Optional[Any] = None*, *actor: Optional[str] = None*, *permission: Optional[str] = 'active'*, *force_recreate: Optional[bool] = False*, *node: Optional[str] = None*) → object

Function for creating a contract object using wax abigen. Contract objects will be saved in the contracts folder.

Note: If you will pack your application to executable, generate the contracts before packing.

Parameters

- **name** (*str*) – The name of the contract (ex: res.pink)
- **client** (litewax.clients.Client) – A *litewax.clients.Client* object (if actor is not provided)
- **actor** (*str*) – The actor name (if client is not provided)
- **permission** (*str*) – The permission to use (default: active)
- **force_recreate** (*bool*) – Force the contract to be recreated (default: False)

- **node** (*str*) – The node to use (default: https://wax.greymass.com)

Returns
Contract object

Return type
object

class litewax.contract.**Action**(*contract: object*, *action: str*, *args: dict*)

Example Action object for calling actions on a contract

class litewax.contract.**ExampleContract**(*actor: str = ''*, *permission: str = 'active'*, *node: str = 'https://wax.greymass.com'*)

Example contract object

set_actor(*actor: str*)

generatePayload(*account: str*, *name: str*) → dict

return_payload(*payload*, *args*) → dict

call(*action: str*, *args: dict*) → dict

action(*arg1: str*) → dict

push_actions(*private_keys: list*, **actions*) → tuple

create_trx(*private_key: str*, ***actions*) → tuple

1.10 abigen

litewax.abigen.**check_ban**(*text*)

class litewax.abigen.**abigen**(*node: str = 'https://wax.greymass.com'*)

abigen class for generating python classes from abi to interact with contracts

Parameters
node – wax node url

Returns
abigen class

property node: str

Node URL

gen(*name: str*) → str

Generate python class from abi

Parameters
name (*str*) – contract name

Returns
content of generated file

Return type
str

get_abi(*account_name: str*) → dict

Get contract abi from node

Parameters
account_name (*str*) – contract name

Returns
abi

Return type
dict

get_tx_info(*tx: str*) → dict

Depricated Get transaction info from node

Parameters
tx (*str*) – transaction id

Returns
transaction info

Return type
dict

1.11 Exceptions

exception litewax.exceptions.**AuthNotFound**
Raised when no auth is found

exception litewax.exceptions.**SessionExpired**
Raised when session is expired or token is invalid

exception litewax.exceptions.**SignError**
Raised when signing fails

exception litewax.exceptions.**UnknownError**
Raised when unknown error occurs

exception litewax.exceptions.**CPUlimit**
Raised when CPU limit is reached

exception litewax.exceptions.**ExpiredTransaction**
Raised when transaction is expired

exception litewax.exceptions.**NotImplementedError**
Raised when method is not implemented

exception litewax.exceptions.**CookiesExpired**
Raised when cookies are expired or invalid

exception litewax.exceptions.**AtomicHubPushError**
Raised when transaction is not signed by AtomicHub

exception litewax.exceptions.**NeftyBlocksPushError**
Raised when transaction is not signed by NeftyBlocks

1.12 Examples

1.12.1 Claim drop AtomicHub

Listing 1: mainnet_claim_drop_atomichub.py

```python
from litewax import Client
from litewax import WAXPayer


# try to get free cpu from atomichub
client = Client(
    private_key="5K...",
    node="https://wax.pink.gg"
)

trx = client.Transaction(
    client.Contract("atomicdropsx").assertdrop(
        assets_to_mint_to_assert=[{"template_id":172121,"tokens_to_back":[]}],
        drop_id="63075",
        listing_price_to_assert="0.01 USD",
        settlement_symbol_to_assert="8,WAX"
    ),
    client.Contract("atomicdropsx").claimdrop(
        claim_amount="1",
        claimer="atonicmaiket",
        country="RU",
        drop_id="63075",
        intended_delphi_median="830",
        referrer="atomichub"
    ),
    client.Contract("eosio.token").transfer(
        _from="atonicmaiket",
        to="atomicdropsx",
        quantity="0.12048192 WAX",
        memo="deposit"
    )
)


trx = trx.payer(WAXPayer.ATOMICHUB) # atomichub pay your trx cpu only if you haven't␣
↪enough wax staked in cpu
print(trx)
print(trx.push())
```

1.12.2 Claim drop NeftyBlocks

Listing 2: testnet_claim_drop_neftyblocks.py

```python
from litewax import Client, WAXPayer


# try to get free cpu from neftyblocks
client = Client(private_key="5K...", node="https://testnet.waxsweden.org")

neftyblocksd = client.Contract("neftyblocksd")

trx = client.Transaction(
    neftyblocksd.claimdrop(
        claimer=client.name,
        drop_id=2020,
        amount=1,
        intended_delphi_median=0,
        referrer="NeftyBlocks",
        country="GB",
        currency="0,NULL"
    ),
    neftyblocksd.assertprice(
        drop_id=2020,
        listing_price="0 NULL",
        settlement_symbol="0,NULL"
    )
)
# add neftyblocks as payer
trx = trx.payer(WAXPayer.NEFTYBLOCKS, network="testnet")

# push transaction
push_resp = trx.push()

print(push_resp)
# {'transaction_id': '928802d253bffc29d6178e634052ec5f044b2fcce0c4c8bc5b44d978e22ec5d4', ...
↪}
```

1.12.3 Pay CPU

Listing 3: pay_for_cpu_multiclient.py

```python
from litewax import MultiClient, Contract

# Create a client with a private keys
client = MultiClient(private_keys=["5K1...", "5K2...", "5K3..."])

# Create the transaction
trx = client.Transaction(

    # Use a some contract action 1
    Contract("eosio.token", client[0]).transfer(
```

(continues on next page)

(continued from previous page)

```
        _from=client[0].name,
        to=client[1].name,
        quantity="1.00000000 WAX",
        memo="Test send"
    ),

    # Use a some contract action 2
    Contract("eosio.token", client[1]).transfer(
        _from=client[1].name,
        to=client[0].name,
        quantity="1.00000000 WAX",
        memo="Test send"
    ),

    # Sign last empty action for pay CPU. client[2] will pay for all transcation CPU
    Contract("res.pink", client[2]).noop()
)

# Push the transaction
r = trx.push()

print(r)
# {'transaction_id': '928802d253bffc29d6178e634052ec5f044b2fcce0c4c8bc5b44d978e22ec5d4', ...
↪}
```

1.12.4 Sell NFT

Listing 4: sell_nft.py

```
from litewax import Client

# Create a client with a private key
client = Client(private_key="5K...")

# to - the account to send the tokens to
to = "abuztradewax"

# Create a atomicassets contract object
atomicassets = client.Contract("atomicassets")

# Create a atomicmarket contract object
atomicmarket = client.Contract("atomicmarket")

# Create a transaction object (https://wax.bloks.io/transaction/
↪e6b2708b291bc2af06d95bfdad6fb65b71835c611b5c4228777d1ee602f4b9b4)
trx = client.Transaction(
    atomicassets.createoffer(
        memo="sale",
        recipient="atomicmarket",
        recipient_asset_ids=[],
```

(continues on next page)

(continued from previous page)

```
        sender=client.name,
        sender_asset_ids= ["1099608856151"]
    ),
    atomicmarket.announcesale(
        asset_ids=["1099608856151"],
        listing_price="100.00000000 WAX",
        maker_marketplace="",
        seller=client.name,
        settlement_symbol="8,WAX"
    )
)

# Push the transaction
r = trx.push()

print(r)
# {'transaction_id': '928802d253bffc29d6178e634052ec5f044b2fcce0c4c8bc5b44d978e22ec5d4', ...
↪}

# Edit nft listing price (https://wax.bloks.io/transaction/
↪37efdd4da70f97807fbf56efae5b438ba1a22ac7cce6224a4e33a020200cac00)
trx = client.Transaction(
    atomicmarket.cancelsale(
        sale_id="94472677"
    ),
    atomicassets.createoffer(
        memo="sale",
        recipient="atomicmarket",
        recipient_asset_ids=[],
        sender=client.name,
        sender_asset_ids= ["1099608856151"]
    ),
    atomicmarket.announcesale(
        asset_ids=["1099608856151"],
        listing_price="99.00000000 WAX",
        maker_marketplace="",
        seller=client.name,
        settlement_symbol="8,WAX"
    )
)

# Push the transaction
r = trx.push()

print(r)
# {'transaction_id': '928802d253bffc29d6178e634052ec5f044b2fcce0c4c8bc5b44d978e22ec5d4', ...
↪}

# Cancel nft listing (https://wax.bloks.io/transaction/
↪fec3677e2df0abc516d552d7fedfd9d9f1a0d702752843287904ec3c5dd59f3c)
trx = client.Transaction(
    atomicmarket.cancelsale(
```

(continues on next page)

(continued from previous page)

```
        sale_id="97921174"
    )
)
```

1.12.5 Transfer NFT

Listing 5: transfer_nft.py

```python
from litewax import Client

# Create a client with a private key
client = Client(private_key="5K...")

# to - the account to send the tokens to
to = "abuztradewax"

# Create a atomicassets contract object
contract = client.Contract("atomicassets")

# Create a transaction object
trx = client.Transaction(
    contract.transfer(
        _from=client.name,
        to=to,
        asset_ids=["1099608856151"], # https://wax.atomichub.io/explorer/asset/1099608856151
        memo="Test send"
    )
)

# Push the transaction
r = trx.push()

print(r)
# {'transaction_id': '928802d253bffc29d6178e634052ec5f044b2fcce0c4c8bc5b44d978e22ec5d4', ...}
```

1.12.6 Transfer Token

Listing 6: transfer_tokens.py

```python
from litewax import Client

# Create a client with a private key
client = Client(private_key="5K...")

# to - the account send the tokens to
to = "abuztradewax"

```

(continues on next page)

(continued from previous page)

```
# Create a eosio.token contract object
contract = client.Contract("eosio.token")

# Create a transaction object
trx = client.Transaction(
    contract.transfer(
        _from=client.name,
        to=to,
        quantity="1.00000000 WAX",
        memo="Test send"
    )
)

# Push the transaction
r = trx.push()

print(r)
```

1.12.7 Use Contract directly

Listing 7: use_contract_directly.py

```
# import library
from litewax import Client

# Create a client with a private key
client = Client(private_key="5K...")

# import contract object (for ex. eosio.token). Before you can use this contract, you must create a contract .py file via abigen.
# For example, you can use this command: `python -c "from litewax import Contract; Contract('eosio.token')"`
from contracts.eosio_token import eosio_token

# init contract object
contract = eosio_token(actor=client.name)

# Create a transaction object
trx = client.Transaction(
    contract.transfer(
        _from=client.name,
        to="abuztradewax",
        quantity="1.00000000 WAX",
        memo="Test send"
    )
)

# Push the transaction
r = trx.push()

```

(continues on next page)

(continued from previous page)

```
print(r)
# {'transaction_id': '928802d253bffc29d6178e634052ec5f044b2fcce0c4c8bc5b44d978e22ec5d4', ...
↪}
```

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

E

G

L

M

N

P

R

S

T

U

W