NeoJsonRPC Documentation

Release 0.1.0.dev

Morgan Aubert

Jan 07, 2018

Contents

CHAPTER 1

Getting started

1.1 Requirements

- Python 3.4+
- Requests 2.0+

1.2 Installation

To install NeoJsonRPC, please use pip (or pipenv) as follows:

```
$ pip install neojsonrpc
```

1.3 Basic usage

In order to interact with the NEO JSON-RPC interface, all you have to do is to initialize a `neojsonrpc.Client` instance and to call one of the JSON-RPC methods provided by the NEO nodes. For example you can get the block count of the NEO TestNet using:

```
>>> from neojsonrpc import Client
>>> testnet_client = Client.for_testnet()
>>> client.get_block_count()
973369
```

CHAPTER 2

Usage

Here are simple guidelines regarding how to use the NeoJsonRPC client.

2.1 Client initialization

The first step to interact with the NEO JSON-RPC interface is to initialize a `neojsonrpc.Client` instance. The following examples respectively show how to get clients for the TestNet, the MainNet and an hypothetical local PrivNet:

```
>>> from neojsonrpc import Client
>>> testnet_client = Client.for_testnet()
>>> mainnet_client = Client.for_mainnet()
>>> privnet_client = Client(host='localhost', port='30333')
```

It should be noted that you can configure you client to interact with a JSON-RPC server over TLS using the `tls` keyword argument:

```
>>> from neojsonrpc import Client
>>> client = Client(host='seed3.neo.org', port=20331, tls=True)
```

2.2 Interacting with the blockchain

You can easily call some of the JSON-RPC methods once your client is initialized. These methods allow you to get various blockchain data from NEO nodes. Here are some examples:

```
>>> from neojsonrpc import Client
>>> client = Client.for_testnet()
>>>
>>> # Retrieve the block count.
>>> client.get_block_count()
977981
```

```
>>>
>>> # Retrieve the information associated with a specific block index.
>>> client.get_block(977981)
{'confirmations': 3,
 'hash': '0x155d6f65eb79730fcb98c8da07d216a150eab432803692c9c9aa04fc895c830d',
  'index': 977981,
  'merkleroot': '0x0e6abef01f17d2ce984fd215266db005b67f0f324df6bcf2b42b1b692226941f',
  'nextblockhash': '0xd7127e3b204b3c5e2ac71914f1a0bec5107e83203571b4e551364df60496d205
↪',
  'nextconsensus': 'AdyQbbn6ENjqWDa5JNYMwN3ikNcA4JeZdk',
  'nonce': '42bfbbe8b95c8796',
  'previousblockhash':
↪'0x3f1438acc3c949fa9f97181cefb4b4f0e376d996c25398bc70159ae75e52be00',
  'script': {'invocation':
↪'402432acc12f362031be5d4da6cbeda3a9b7aa61482e234b979f96e30d051278c4ad5711e86e136cf9a6a8450626a88ec8
↪',
  'verification':
↪'55210209e7fd41dfb5c2f8dc72eb30358ac100ea8c72da18847befe06eade68cebfcb9210327da12b5c40200e9f655694
↪'},
  'size': 987,
  'time': 1515362895,
  'tx': [{'attributes': [],
          'net_fee': '0',
          'nonce': 3109848982,
          'scripts': [],
          'size': 10,
          'sys_fee': '0',
          'txid': '0xd0884f26dc433e40f45ac5a5c310979e8a9c14925ba8523582933e2905cf51ed
↪',
          'type': 'MinerTransaction',
          'version': 0,
          'vin': [],
          'vout': []},
         {'attributes': [],
          'gas': '0',
          'net_fee': '0',
          'script':
↪'4cf67b227265636f72644964223a227265636f72645f746573745f39383335222c22616464726573746726f6d223a223
↪',
          'scripts': [],
          'size': 301,
          'sys_fee': '0',
          'txid': '0x5c9cdc113a7adb58320786f2be8009b1fb723bd164b6d9e6025f019203beeeb8
↪',
          'type': 'InvocationTransaction',
          'version': 1,
          'vin': [],
          'vout': []}],
 'version': 0}
```

Note: Please refer to *Client reference* for a full list of the available methods.

2.3 Invoking & testing smart contracts

NeoJsonRPC implements the `invoke`, `invokefunction` and `invokescript` methods provided by NEO nodes through the JSON-RPC interface. It should be noted that these methods are to test VM scripts as if they wre ran on the blockchain. The underlying RPC calls don't affect the blockchain in any way (no transaction is generated, nothing is stored on the contract's storage). Here is a simple example using the `invoke_function` method:

```
>>> from neojsonrpc import Client
>>> client = Client.for_testnet()
>>> result = client.invoke_function('34af1b6634fcd7cfcff0158965b18601d3837e32',
↪'symbol', [])
{'gas_consumed': '0.217',
 'stack': [{'type': 'ByteArray', 'value': bytearray(b'TKN')}],
 'state': 'HALT, BREAK'}
```

It should be noted that NeoJsonRPC provides a more high-level interface for interacting with contract fonctions as if they were Python class instance methods:

```
>>> contract = client.contract('34af1b6634fcd7cfcff0158965b18601d3837e32')
>>> contract.symbol()
{'gas_consumed': '0.217',
 'stack': [{'type': 'ByteArray', 'value': bytearray(b'TKN')}],
 'state': 'HALT, BREAK'}
```

CHAPTER 3

Client reference

class neojsonrpc.client.**Client**(*host=None*, *port=None*, *tls=False*, *http_max_retries=None*)

The NEO JSON-RPC client class.

contract(*script_hash*)

Returns a `ContractWrapper` instance allowing to easily invoke contract functions.

This method allows to invoke smart contract functions as if they were Python class instance methods. For example:

```
>>> contract = client.contract('34af1b6634fcd7cfcff0158965b18601d3837e32')
>>> contract.symbol()
{...}
>>> contract.getBalance('<address>')
{...}
```

Parameters **script_hash** (*str*) – contract script hash

Returns `ContractWrapper` object

Return type neojsonrpc.client.ContractWrapper

classmethod for_mainnet()

Creates a `Client` instance for use with the NEO Main Net.

classmethod for_testnet()

Creates a `Client` instance for use with the NEO Test Net.

get_account_state(*address*, ***kwargs*)

Returns the account state information associated with a specific address.

Parameters **address** (*str*) – a 34-bit length address (eg. AJBENSwajTzQtwyJFkiJSv7MAaaMc7DsRz)

Returns dictionary containing the account state information

Return type dict

get_asset_state(*asset_id*, ***kwargs*)
Returns the asset information associated with a specific asset ID.

Parameters **asset_id** (*str*) – an asset identifier (the transaction ID of the RegistTransaction when the asset is registered)

Returns dictionary containing the asset state information

Return type dict

get_best_block_hash(***kwargs*)
Returns the hash of the tallest block in the main chain.

Returns hash of the tallest block in the chain

Return type str

get_block(*block_hash*, *verbose=True*, ***kwargs*)
Returns the block information associated with a specific hash value or block index.

Parameters

- **block_hash** (*str or int*) – a block hash value or a block index (block height)
- **verbose** (*bool*) – a boolean indicating whether the detailed block information should be returned in JSON format (otherwise the block information is returned as an hexadecimal string by the JSON-RPC endpoint)

Returns dictionary containing the block information (or an hexadecimal string if verbose is set to False)

Return type dict or str

get_block_count(***kwargs*)
Returns the number of blocks in the chain.

Returns number of blocks in the chain

Return type int

get_block_hash(*block_index*, ***kwargs*)
Returns the hash value associated with a specific block index.

Parameters **block_index** (*int*) – a block index (block height)

Returns hash of the block associated with the considered index

Return type str

get_block_sys_fee(*block_index*, ***kwargs*)
Returns the system fees associated with a specific block index.

Parameters **block_index** (*int*) – a block index (block height)

Returns system fees of the block, expressed in NeoGas units

Return type str

get_connection_count(***kwargs*)
Returns the current number of connections for the considered node.

Returns number of connections for the node

Return type int

get_contract_state(*script_hash*, ***kwargs*)
Returns the contract information associated with a specific script hash.

Parameters **script_hash** (*str*) – contract script hash

Returns dictionary containing the contract information

Return type dict

get_peers(***kwargs*)
Returns a list of nodes that the node is currently connected/disconnected from.

Returns dictionary containing the nodes the current node is connected/disconnected from

Return type dict

get_raw_mem_pool(***kwargs*)
Returns a list of unconfirmed transactions in memory associated with the node.

Returns list of unconfirmed transaction hashes

Return type list

get_raw_transaction(*tx_hash*, *verbose=True*, ***kwargs*)
Returns detailed information associated with a specific transaction hash.

Parameters

- **tx_hash** (*str*) – transaction hash
- **verbose** (*bool*) – a boolean indicating whether the detailed transaction information should be returned in JSON format (otherwise the transaction information is returned as an hexadecimal string by the JSON-RPC endpoint)

Returns dictionary containing the transaction information (or an hexadecimal string if verbose is set to False)

Return type dict or str

get_storage(*script_hash*, *key*, ***kwargs*)
Returns the value stored in the storage of a contract script hash for a given key.

Parameters

- **script_hash** (*str*) – contract script hash
- **key** (*str*) – key to look up in the storage

Returns value associated with the storage key

Return type bytearray

get_tx_out(*tx_hash*, *index*, ***kwargs*)
Returns the transaction output information corresponding to a hash and index.

Parameters

- **tx_hash** (*str*) – transaction hash
- **index** (*int*) – index of the transaction output to be obtained in the transaction (starts from 0)

Returns dictionary containing the transaction output

Return type dict

get_version(***kwargs*)
Returns version information about the current node.

Returns dictionary containing version information about the current node

Return type dict

invoke(*script_hash*, *params*, ***kwargs*)
Invokes a contract with given parameters and returns the result.

It should be noted that the name of the function invoked in the contract should be part of paramaters.

Parameters

- **script_hash** (*str*) – contract script hash
- **params** (*list*) – list of paramaters to be passed in to the smart contract

Returns result of the invocation

Return type dictionary

invoke_function(*script_hash*, *operation*, *params*, ***kwargs*)
Invokes a contract's function with given parameters and returns the result.

Parameters

- **script_hash** (*str*) – contract script hash
- **operation** (*str*) – name of the operation to invoke
- **params** (*list*) – list of paramaters to be passed in to the smart contract

Returns result of the invocation

Return type dictionary

invoke_script(*script*, ***kwargs*)
Invokes a script on the VM and returns the result.

Parameters **script** (*str*) – script runnable by the VM

Returns result of the invocation

Return type dictionary

send_raw_transaction(*hextx*, ***kwargs*)
Broadcasts a transaction over the NEO network and returns the result.

Parameters **hextx** (*str*) – hexadecimal string that has been serialized

Returns result of the transaction

Return type bool

validate_address(*addr*, ***kwargs*)
Validates if the considered string is a valid NEO address.

Parameters **hex** (*str*) – string containing a potential NEO address

Returns dictionary containing the result of the verification

Return type dictionary

CHAPTER 4

Contributing to NeoJsonRPC

Here are some simple rules & tips to help you contribute to NeoJsonRPC. You can contribute in many ways!

4.1 Contributing code

The preferred way to contribute to NeoJsonRPC is to submit pull requests to the project's Github repository. Here are some general tips regarding pull requests.

4.1.1 Development environment

Note: The following steps assumes you have Pipenv installed on your system.

You should first fork the NeoJsonRpc's repository. Then you can get a working copy of the project using the following commands:

```
$ git clone git@github.com:<username>/neojsonrpc.git
$ cd neojsonrpc
$ make
```

Coding style

Please make sure that your code is compliant with the PEP8 style guide. You can ignore the "Maximum Line Length" requirement but the length of your lines should not exceed 100 characters. Remember that your code will be checked using flake8 and isort. You can use the following command to trigger such quality assurance checks:

```
$ make qa
```

Tests

You should not submit pull requests without providing tests. NeoJsonRPC relies on pytest: py.test is used instead of unittest for its test runner but also for its syntax. So you should write your tests using pytest instead of unittest and you should not use the built-in `TestCase`.

You can run the whole test suite using the following command:

```
$ make tests
```

Code coverage should not decrease with pull requests! You can easily get the code coverage of the project using the following command:

```
$ make coverage
```

4.1.2 Using the issue tracker

You should use the project's issue tracker if you've found a bug or if you want to propose a new feature. Don't forget to include as many details as possible in your tickets (eg. tracebacks if this is appropriate).

CHAPTER 5

Release notes

Here are listed the release notes for each version of NeoJsonRPC.

5.1 NeoJsonRPC 0.1

5.1.1 NeoJsonRPC 0.1 release notes (UNDER DEVELOPMENT)

Requirements and compatibility

Python 3.4, 3.5 and 3.6. Requests 2.0+.

New features

This is the initial release of NeoJsonRPC!

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Index