Exonum Python light client

Release 1.0.0

The Exonum team

Apr 02, 2020

CONTENTS:

CHAPTER

ONE

OVERVIEW

Exonum Light Client is a Python library for working with Exonum blockchain from the client side. It can be easily integrated to an existing application. Also, Exonum Light Client provides access to common utils toolkit which contains some helpful functions for hashing, cryptography, serialization, etc.

CHAPTER

TWO

CAPABILITIES

By using the client you are able to perform the following operations:

- Submit transactions to the node
- Receive information on transactions
- Receive information on blockchain blocks
- Receive information on the node system
- Receive information on the node status

CHAPTER

THREE

SYSTEM DEPENDENCIES

- Python 3.5 or above.
- Package installer for Python3 (pip3)

CHAPTER

FOUR

EXAMPLES

The following example shows how to create an instance of the Exonum client which will be able to work with an Exonum node with the Cryptocurrency Advanced service mounted on it, at http://localhost:8080 address:

4.1 Installing Python Light Client

First of all, we need to install our client library:

```
git clone git@github.com:exonum/exonum-python-client.git
pip3 install -e exonum-python-client
```

4.2 Exonum Client Initialization

```
>>> from exonum_client import ExonumClient, ModuleManager, MessageGenerator
>>> from exonum_client.crypto import KeyPair
>>> client = ExonumClient(hostname="localhost", public_api_port=8080, private_api_
↪port=8081, ssl=False)
```

4.3 Compiling Proto Files

To compile proto files into the Python analogues we need a Protobuf loader:

```
>>> with client.protobuf_loader() as loader:
>>>     #  Your code goes here.
```

Since loader acquires resources on initialization, it is recommended that you create the loader via the context manager. Otherwise you should initialize and deinitialize the client manually:

```
>>> loader = client.protobuf_loader()
>>> loader.initialize()
>>> # ... Some usage
>>> loader.deinitialize()
```

Then we need to run the following code:

```
>>> loader.load_main_proto_files()  # Loads and compiles main proto files, such as␣
↪`runtime.proto`, `consensus.proto`, etc.
>>> loader.load_service_proto_files(runtime_id=0, service_name='exonum-supervisor:0.
↪12.0')  # Same for a specific service.
```

- runtime_id=0 here means, that service works in Rust runtime.

4.4 Creating Transaction Messages

The following example shows how to create a transaction message:

```
>>> alice_keys = KeyPair.generate()
>>>
>>> cryptocurrency_service_name = 'exonum-cryptocurrency-advanced:0.12.0'
>>> loader.load_service_proto_files(runtime_id=0, service_name=cryptocurrency_service_
↪name)
>>>
>>> cryptocurrency_module = ModuleManager.import_service_module(cryptocurrency_
↪service_name, 'service')
>>> cryptocurrency_message_generator = MessageGenerator(service_id=1024, artifact_
↪name=cryptocurrency_service_name)
>>>
>>> create_wallet_alice = cryptocurrency_module.CreateWallet()
>>> create_wallet_alice.name = 'Alice'
>>> create_wallet_alice_tx = cryptocurrency_message_generator.create_message(create_
↪wallet_alice)
>>> create_wallet_alice_tx.sign(alice_keys)
```

- 1024 - service instance ID.
- alice_keys - public and private keys of the ed25519 public-key signature system.

After invoking the sign method we get a signed transaction. This transaction is ready for sending to an Exonum node.

4.5 Sending Transaction to the Exonum Node

After successfully sending the message, we'll get a response which will contain a hash of the transaction:

```
>>> response = client.public_api.send_transaction(create_wallet_alice_tx)
{
    "tx_hash": "3541201bb7f367b802d089d8765cc7de3b7dfc253b12330b8974268572c54c01"
}
```

4.6 Subscribing to events

If you want to subscribe to events (subscription_type: "transactions" or "blocks"), use the following code:

```
>>> with client.create_subscriber(subscription_type="blocks") as subscriber:
>>>     subscriber.wait_for_new_event()
>>>     subscriber.wait_for_new_event()
```

Context manager will automatically create a connection and will disconnect after use. Or you can manually do the same:

```
>>> subscriber = client.create_subscriber(subscription_type="blocks")
>>> subscriber.connect()
>>> # ... Your code
>>> subscriber.stop()
```

Keep in mind that if you forget to stop the subscriber, you may discover HTTP errors when you try to use Exonum API.

4.7 Getting Data on the Available Services

The code below will show a list of the artifacts available for the start and a list of working services:

```
>>> client.public_api.available_services().json()
{
  "artifacts": [
    {
      "runtime_id": 0,
      "name": "exonum-supervisor",
      "version": "1.0.0"
    },
    {
      "runtime_id": 0,
      "name": "exonum-explorer-service",
      "version": "1.0.0"
    }
  ],
  "services": [
    {
      "spec": {
        "id": 2,
        "name": "explorer",
        "artifact": {
          "runtime_id": 0,
          "name": "exonum-explorer-service",
          "version": "1.0.0"
        }
      },
      "status": "Active",
      "pending_status": null
    },
    {
      "spec": {
        "id": 0,
        "name": "supervisor",
        "artifact": {
          "runtime_id": 0,
          "name": "exonum-supervisor",
          "version": "1.0.0"
        }
      },
      "status": "Active",
      "pending_status": null
    }
```

(continues on next page)

(continued from previous page)

```
    ]
}
```

4.8 More Examples

You can find the sample scripts in the GitHub repository examples section:

CHAPTER

FIVE

MODULES DOCUMENTATION

Documentation for the modules in the Exonum Python Light Client:

5.1 exonum

5.1.1 exonum package

Module Contents

Subpackages

exonum_client.proofs package

Module Contents

Subpackages

exonum_client.proofs.list_proof package

Module Contents

Submodules

exonum_client.proofs.list_proof.list_proof module

exonum_client.proofs.list_proof.errors module

exonum_client.proofs.map_proof package

Module Contents

Submodules

exonum_client.proofs.map_proof.map_proof_builder module

exonum_client.proofs.map_proof.map_proof module

exonum_client.proofs.map_proof.errors module

Submodules

exonum_client.proofs.encoder module

Submodules

exonum_client.client module

exonum_client.crypto module

Module with the Common Cryptography-assotiated Utils.

This module uses libsodium as a backend.

class Hash(*hash_bytes: bytes*)
Representation of a SHA-256 hash.

classmethod hash_data(*data: Optional[bytes]*) → exonum_client.crypto.Hash
Calculates a hash of the provided bytes sequence and returns a Hash object.

If *None* is provided, a hash of the empty sequence will be returned.

class KeyPair(*public_key: exonum_client.crypto.PublicKey*, *secret_key: exonum_client.crypto.SecretKey*)
Representation of a Ed25519 keypair.

classmethod generate() → exonum_client.crypto.KeyPair
Generates a new random keypair

class PublicKey(*key: bytes*)
Representation of a Ed25519 Public Key.

class SecretKey(*key: bytes*)
Representation of a Ed25519 Secret Key.

class Signature(*signature: bytes*)
Representation of a Ed25519 signature

classmethod sign(*data: bytes*, *key: exonum_client.crypto.SecretKey*) → exonum_client.crypto.Signature
Signs the provided bytes sequence with the provided secret key.

verify(*data: bytes*, *key: exonum_client.crypto.PublicKey*) → bool
Verifies the signature against the provided data and the public key.

exonum_client.message module

This module is capable of creating and signing Exonum transactions.

class ExonumMessage(*instance_id: int*, *message_id: int*, *msg: google.protobuf.message.Message*, *prebuilt: Optional[bytes] = None*)
Generic Exonum transaction class.

Exonum message can be created:

- by using MessageGenerator (if you want to send a transaction to the Exonum blockchain)
- by using ExonumMessage.from_hex (if you want to parse a retrieved transaction).

Example workflow:

Sending a message:

```
>>> instance_id = ... # Get the ID of the desired service instance.
>>> artifact_name = ... # Get the name of the artifact (not the instance).
>>> cryptocurrency_message_generator = MessageGenerator(instance_id, artifact_
↪name) # Create a message generator.
>>> create_wallet_alice = cryptocurrency_module.CreateWallet() # Create a␣
↪Protobuf message.
>>> create_wallet_alice.name = "Alice1" # Fill the Protobuf message manually.
>>> create_wallet_alice_tx = cryptocurrency_message_generator.create_
↪message(create_wallet_alice)
>>> create_wallet_alice_tx.sign(keypair) # You should sign the message before␣
↪sending.
>>> client.public_api.send_transaction(create_wallet_alice_tx)
```

Parsing a message:

```
>>> message_hex = ... # Retrieve the message as a hexadecimal string.
>>> artifact_name = ... # Get the name of the artifact (not the instance).
>>> transaction_name = "CreateWallet" # Get the name of the transaction.
>>> parsed_message = ExonumMessage.from_hex(message_hex, artifact_name,␣
↪transaction_name)
>>> assert parsed_message.validate() # Verify the signature of the retrieved␣
↪message.
```

Other methods: >>> message = ExonumMessage.from_hex(...) >>> author = message.author() # Get the author's public key. >>> tx_hash = message.hash() # Get the transaction hash. >>> signature = message.signature() # Get the transaction signature. >>> any_tx_raw = message.any_tx_raw() # Get AnyTx of the message serialized to bytes. >>> signed_tx_raw = message.signed_tx_raw() # Get SignedMessage of the message serialized to bytes. >>> tx_json = message.pack_into_json() # Create a JSON with the transaction in the format expected by Exonum.

__init__(*instance_id: int*, *message_id: int*, *msg: google.protobuf.message.Message*, *prebuilt: Optional[bytes] = None*)

Exonum message constructor. It is not intended to be used directly, see *MessageGenerator.create_message* and *ExonumMessage.from_hex* instead.

any_tx_raw() → bytes

Returns a serialized AnyTx message as bytes.

author() → Optional[exonum_client.crypto.PublicKey]

Returns an author's public key. If the author is not set, returns None.

classmethod from_hex(*message_hex: str*, *artifact_name: str*, *artifact_version: str*, *tx_name: str*) → Optional[exonum_client.message.ExonumMessage]

Attempts to parse an Exonum message from a serialized hexadecimal string.

Parameters

- **message_hex** (*str*) – Serialized message as a hexadecimal string.
- **artifact_name** (*str*) – The name of the service artifact you want to communicate with, e.g. 'exonum-cryptocurrency-advanced'.
- **artifact_version** (*str*) – Version of artifact as string, e.g. '1.0.0'.

- **tx_name** (*str*) – The name of the transaction to be parsed, e.g. 'CreateWallet'.

Returns **parsed_message** – If parsing is successfull, an ExonumMessage object is returned. Otherwise the returned value is None.

Return type Optional[*ExonumMessage*]

hash() → exonum_client.crypto.Hash
Returns a hash of the message. If the message is not signed, a hash of an empty message will be returned.

pack_into_json() → str
Packs a serialized signed message into the JSON format expected by Exonum.

Please note that this method does not serialize the message to JSON.

Returns **json_message** – String with a JSON representation of the serialized message.

Return type str

Raises **RuntimeError** – An error will be raised on attempt to call *pack_into_json* with an unsigned message.

sign(*keys: exonum_client.crypto.KeyPair*) → None
Signs the message with the provided pair of keys.

Please note that signing is required before sending a message to the Exonum blockchain.

Parameters **keys** (*exonum.crypto.KeyPair*) – A pair of public_key and secret_key as bytes.

signature() → Optional[exonum_client.crypto.Signature]
Returns a signature. If the message is not signed, returns None.

signed_raw() → Optional[bytes]
Returns a serialized SignedMessage as bytes. If the message is not signed, returns None.

validate() → bool
Validates the message. Checks if the transaction signature is correct. :return: bool

class MessageGenerator(*instance_id: int*, *artifact_name: str*, *artifact_version: str*)
MessageGenerator is a class which helps you create transactions. It is capable of transforming a Protobuf message object into an Exonum transaction with a set of the required metadata.

Example of usage: >>> instance_id = ... # Get the ID of the desired service instance. >>> artifact_name = ... # Get the name of the artifact (not the instance). >>> artifact_version = ... # Get the version of the artifact (not the instance). >>> cryptocurrency_message_generator = MessageGenerator(instance_id, artifact_name, artifact_version) >>> create_wallet_alice = cryptocurrency_message_generator.CreateWallet() # Create a Protobuf message. >>> create_wallet_alice.name = "Alice1" # Fill the Protobuf message manually.

Then you can transform the Protobuf message into an Exonum transaction.

```
>>> create_wallet_alice_tx = cryptocurrency_message_generator.create_
↪message(create_wallet_alice)
>>> create_wallet_alice_tx.sign(keypair) # You should sign the message before␣
↪sending.
>>> client.public_api.send_transaction(create_wallet_alice_tx)
```

__init__(*instance_id: int*, *artifact_name: str*, *artifact_version: str*)
MessageGenerator constructor.

Parameters

- **instance_id** (*int*) – ID of the desired Exonum service instance.

- **artifact_name** (*str*) – The name of the service artifact you want to communicate with. The name should be in the format provided by Exonum, like 'exonum-cryptocurrency-advanced:1.0.0'.

create_message(*message: google.protobuf.message.Message*) → exonum_client.message.ExonumMessage

Method to convert a Protobuf message into an Exonum message.

Parameters **message** (*google.protobuf.message.Message*) – A Protobuf message.

Returns **exonum_message** – Exonum message object.

Return type *ExonumMessage*

static pk_to_hash_address(*public_key: exonum_client.crypto.PublicKey*) → Optional[exonum_client.crypto.Hash]

Converts *PublicKey* into a *Hash*, which is a uniform presentation of any transaction authorization supported by Exonum.

exonum_client.module_manager module

Module capable of loading the Protobuf-generated modules.

class ModuleManager

ModuleManager class provides an interface for importing modules generated from the previously downloaded Protobuf sources.

It is supposed that you call those methods only after downloading the corresponding module via ProtobufLoader. Otherwise an error will be raised.

Example usage:

```
>>> with client.protobuf_loader() as loader:
>>>     loader.load_main_proto_files()
>>>     loader.load_service_proto_files(0, "exonum-supervisor", "1.0.0")
>>>     blockchanin_module = ModuleManager.import_main_module("exonum.blockchain")
>>>     auth_module = ModuleManager.import_main_module("exonum.runtime.auth")
>>>     service_module = ModuleManager.import_service_module("exonum-supervisor",
↪"1.0.0", "service")
```

static import_main_module(*module_name: str*) → Any

Imports the main (used by the Exonum core) module, e.g. "consensus", "runtime", etc.

static import_service_module(*artifact_name: str*, *artifact_version: str*, *module_name: str*) → Any

Imports the service (corresponding to some artifact) module.

exonum_client.protobuf_loader module

Module Containing the ProtobufLoader Class.

ProtobufLoader is capable of downloading Protobuf sources from Exonum.

class ProtoFile

Structure that represents a proto file.

property content

Alias for field number 1

property name
Alias for field number 0

class ProtobufLoader(*client: Optional[exonum_client.protobuf_loader.ProtobufProviderInterface] = None*)
ProtobufLoader is a class capable of loading and compiling Protobuf sources from Exonum.

This class is a Singleton, meaning that only one entity of that class is created at a time.

Example workflow:

```
>>> with client.protobuf_loader() as loader:
>>>     loader.load_main_proto_files()
>>>     loader.load_service_proto_files(0, "exonum-supervisor", "1.0.0")
```

Code above will initialize loader, download core Exonum proto files and proto files for the Supervisor service. The code will compile the files into the Python modules. After that you will be able to load those modules via ModuleManager.

Please note that it is recommended to create a ProtobufLoader object via the context manager. Otherwise you will have to call *initialize* and *deinitialize* methods manually:

```
>>> loader = client.protobuf_loader()
>>> loader.initialize()
>>> ... # Some code
>>> loader.deinitialize()
```

If you forget to call *deinitialize* (or if the code exits earlier, for example because of unhandled exception), the recourses created in the temp folder (which may differ depending on your OS) will not be removed.

Creating more than one entity at a time will result in retrieving the same object:

```
>>> with client.protobuf_loader() as loader_1:
>>>     with client.protobuf_loader() as loader_2:
>>>         assert loader_1 == loader_2
```

This may be useful if you have several modules that should work with ProtobufLoader:

```
>>> # main.py
>>> loader = ProtobufLoader(client)
>>> loader.initialize()
>>> loader.load_main_proto_files()
>>> ...
>>> loader.deinitialize()
```

```
>>> # module_a.py
>>> loader = ProtobufLoader() # Since loader is already initialized with the␣
↪client, you do not have to provide it.
>>> loader.load_service_proto_files(runtime_a, service_a, service_a_version)
```

```
>>> # module_b.py
>>> loader = ProtobufLoader()
>>> loader.load_service_proto_files(runtime_b, service_b, service_b_version)
```

However, if you try to create the second loader, different from the first one, from the client, ValueError will be raised.

deinitialize() → None
Performs a deinitialization process.

initialize() → None
Performs an initialization process.

load_main_proto_files() → None
Loads and compiles the main Exonum proto files.

load_service_proto_files(*runtime_id: int*, *artifact_name: str*, *artifact_version: str*) → None
Loads and compiles proto files for a service.

class ProtobufProviderInterface
Interface for Protobuf sources provider.

get_main_proto_sources() → List[exonum_client.protobuf_loader.ProtoFile]
Gets the Exonum core proto sources.

get_proto_sources_for_artifact(*runtime_id: int*, *artifact_name: str*, *artifact_version: str*) → List[exonum_client.protobuf_loader.ProtoFile]
Gets the Exonum core proto sources.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

e

INDEX

Symbols

A

C

D

E

F

G

H

I

K

L

M

N

P

S

V