aepp-sdk Documentation

Andrea Giacobino, Shubhendu Shekhar

Jan 31, 2020

CONTENTS

Everything you need to know about the Aeternity Python SDK.

CHAPTER

ONE

GETTING STARTED

Are you new to Aeternity blockchain? Well, you came to the right place: read this material to quickly get up and running.

1.1 Quick install guide

Before you can use the Aeternity SDK, you'll need to get it installed. This guide will guide you to a minimal installation that'll work while you walk through the introduction. For more installation options check the *installation guide*.

1.1.1 Install Python

Get the latest version of Python at https://www.python.org/downloads/ or with your operating system's package manager.

You can verify that Python is installed by typing `python` from your shell; you should see something like:

```
Python 3.7.y
[GCC 4.x] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Hint: The mimimum required python version is 3.7

1.1.2 Install the SDK

The Aeternity Python SDK package name is `aepp-sdk` and it is available via the pypi.org repository.

To install or upgrade run the command

```
pip install -U aepp-sdk
```

1.1.3 Verifying

To verify that the Aeternity SDK can be seen by Python, type `python` from your shell. Then at the Python prompt, try to import aeternity:

```
>>> import aeternity
>>> print(aeternity._version())
snapshot
```

To verify that the CLI is available in your `PATH` run the command

```
$ aecli --version
aecli, version snapshot
```

1.1.4 That's it!

That's it – you can now *move onto the tutorial*.

1.2 Transfer funds

Let's learn by example.

Throughout this tutorial, we'll walk you through the creation of a basic script to transfer funds from an account to another.

It'll consist of three parts:

- Generate 2 accounts via command line client
- Get some funds for the first account
- Transfer the funds from an account to the other

We'll assume you have *the SDK installed* already. You can tell the SDK is installed and which version by running the following command in a shell prompt (indicated by the $ prefix):

```
$ aecli --version
```

First step will be to generate two new accounts using the command line client:

First import the required libraries

```
from aeternity.node import NodeClient, Config
from aeternity.signing import Account
from aeternity import utils
import os
```

Then instantiate the node client and generate 2 accounts

```
NODE_URL = os.environ.get('TEST_URL', 'https://testnet.aeternity.io')

node_cli = NodeClient(Config(
    external_url=NODE_URL,
    blocking_mode=True,
```

(continues on next page)

(continued from previous page)

```
))

# generate ALICE account
alice = Account.generate()

# generate BOB account
bob = Account.generate()

# retrieve the balances for the accounts
bob_balance = node_cli.get_balance(bob)
alice_balance = node_cli.get_balance(alice)

# print the balance of the two accounts
print(f"Alice address is {alice.get_address()}")
print(f"with balance {utils.format_amount(alice_balance)}")
print(f"Bob address is {bob.get_address()}")
print(f"with balance {utils.format_amount(bob_balance)}")
```

Now copy the Alice address and paste it into the Aeternity Faucet to top up the account; once the Alice account has some positive balance we can execute the spend transaction:

```
#TODO pause the execution while using the faucet
# execute the spend transaction
tx = node_cli.spend(alice, bob.get_address(), "3AE")
print(f"transaction hash: {tx.hash}")
print(f"inspect transaction at {NODE_URL}/v2/transactions/{tx.hash}")
```

And finally verify the new balance:

```
# retrieve the balances for the accounts
bob_balance = node_cli.get_balance(bob)
alice_balance = node_cli.get_balance(alice)

print(f"Alice balance is {utils.format_amount(alice_balance)}")
print(f"Bob balance is {utils.format_amount(bob_balance)}")
```

Thats it! You have successfully executed your transaction in the Aeternity Blockchain testnet network. For the mainnet network the procedure is the same except you will have to get some tokens via an exchange or via other means.

1.3 Deploying a contract

This guide describes how you can leverage aepp-sdk to compile and deploy an aeternity smart contracts.

See also:

Sophia: An Æternity Blockchain Language

The Sophia is a language in the ML family. It is strongly typed and has restricted mutable state. Check out more about sophia here.

1.3.1 Prerequisites

1. An account with some initial AE
2. A smart contract written in sophia
3. An aeternity node
4. A sophia aehttp compiler

Assumptions

We're going to assume that you have working knowledge of the SDK and know how to create an Account instance, a NodeClient, and a CompilerClient.

1.3.2 Sample Sophia Contract

Below is the sample sophia contract that we'll use in this guide.

```
@compiler >= 4

contract CryptoHamster =

    record state = {
        index : int,
        map_hamsters : map(string, hamster),
        testvalue: int}

    record hamster = {
        id : int,
        name : string,
        dna : int}

    stateful entrypoint init() =
        { index = 1,
            map_hamsters = {},
            testvalue = 42}

    public entrypoint read_test_value() : int =
        state.testvalue

    public entrypoint return_caller() : address =
        Call.caller

    public entrypoint cause_error() : unit =
        require(2 == 1, "require failed")

    public stateful entrypoint add_test_value(one: int, two: int) : int =
        put(state{testvalue = one + two})
        one + two

    public entrypoint locally_add_two(one: int, two: int) : int =
        one + two

    public stateful entrypoint statefully_add_two(one: int, two: int) =
        put(state{testvalue = one + two})
```

(continues on next page)

(continued from previous page)

```
    stateful entrypoint create_hamster(hamster_name: string) =
        require(!name_exists(hamster_name), "Name is already taken")
        let dna : int = generate_random_dna(hamster_name)
        create_hamster_by_name_dna(hamster_name, dna)

    entrypoint name_exists(name: string) : bool =
        Map.member(name, state.map_hamsters)

    entrypoint get_hamster_dna(name: string, test: option(int)) : int =
        require(name_exists(name), "There is no hamster with that name!")

        let needed_hamster : hamster = state.map_hamsters[name]

        needed_hamster.dna

    private stateful function create_hamster_by_name_dna(name: string, dna: int) =
        let new_hamster : hamster = {
            id = state.index,
            name = name,
            dna = dna}

        put(state{map_hamsters[name] = new_hamster})
        put(state{index = (state.index + 1)})

    private function generate_random_dna(name: string) : int =
        get_block_hash_bytes_as_int() - Chain.timestamp + state.index

    private function get_block_hash_bytes_as_int() : int =
        switch(Chain.block_hash(Chain.block_height - 1))
            None => abort("blockhash not found")
            Some(bytes) => Bytes.to_int(bytes)

    entrypoint test(name: string) : hash =
        String.sha3(name)
```

1.3.3 Importing required classes and methods

We need to import the following classes to use contracts.

```
from aeternity.node import NodeClient, Config
from aeternity.compiler import CompilerClient
from aeternity.contract_native import ContractNative
from aeternity.signing import Account
```

1.3.4 Initializing NodeClient and Compiler

Below are the steps required to initialize the *NodeClient* and *Compiler*. As you can see below, during the initialization of *NodeClient* we're also providing the *internal_url*.

internal_url provides the debug endpoint to *dry_run* a contract method which can also be used to do static calls on deployed contracts and this is what exactly we're going to use this for.

You can also not provide the *internal_url* but then you'll have to disable the use of *dry-run* endpoint. We'll see how to do that when we initialize our contract.

```
NODE_URL = os.environ.get('TEST_URL', 'http://127.0.0.1:3013')
NODE_INTERNAL_URL = os.environ.get('TEST_DEBUG_URL', 'http://127.0.0.1:3113')
COMPILER_URL = os.environ.get('TEST_COMPILER__URL', 'https://compiler.aepps.com')

node_cli = NodeClient(Config(
    external_url=NODE_URL,
    internal_url=NODE_INTERNAL_URL,
    blocking_mode=True,
))

compiler = CompilerClient(compiler_url=COMPILER_URL)
```

1.3.5 Generate an Account

You'll need an account (using the *Account* class) to deploy the contract and also for stateful contract calls.

```
# genrate ALICE account (and transfer AE to alice account)
alice = Account.generate()
```

1.3.6 Read the Contract from file and initialize

You can read the contract from the stored *.aes* file and use it to initialize the contract instance. If you have not provided the *internal_endpoint* or simple do not want to use the *dry-run* functionality you can disable it by passing *use-dry-run=False* to the *ContractNative* constructor.

> **Warning:** If you DO NOT provide the *internal_url* during NodeClient initialization and also DID NOT disable the *dry-run* then the contract method calls for *un-stateful* methods WILL FAIL.

```
CONTRACT_FILE = os.path.join(os.path.dirname(__file__), "testdata/CryptoHamster.aes")

# read contract file
with open(CONTRACT_FILE, 'r') as file:
    crypto_hamster_contract = file.read()

"""
Initialize the contract instance
Note: To disable use of dry-run endpoint add the parameter
use_dry_run=False
"""
crypto_hamster = ContractNative(client=node_cli,
                                compiler=compiler,
```

(continues on next page)

(continued from previous page)

```
                                account=alice,
                                source=crypto_hamster_contract)
```

1.3.7 Compile and deploy the contract

You can compile the contract and deploy it using the *deploy* method. If your *init* method accepts any arguments then please provide them inside the *deploy* method. Once the contract is compiled and deployed, the signed transaction is returned.

```
# deploy the contract
# (also pass the args of thr init function - if any)
tx = crypto_hamster.deploy()
print(f"contract address: {crypto_hamster.address}")
```

Thats all, you have successfully deployed a smart contract in Aeternity, the next tutorial will guide you to call the contract

1.4 Interacting with a contract

This guide describes how you can leverage aepp-sdk interact with a deployed aeternity smart contract.

See also:

Sophia: An Æternity Blockchain Language

The Sophia is a language in the ML family. It is strongly typed and has restricted mutable state. Check out more about sophia here.

1.4.1 Prerequisites

1. An account with some initial AE
2. An address/contract_id of a deployed contract
3. An aeternity node
4. A sophia aehttp compiler

Assumptions

We're going to assume that you have working knowledge of the SDK and know how to create an Account instance, a NodeClient, and a CompilerClient.

1.4.2 Sample Sophia Contract

Below is the sample sophia contract that we'll use in this guide.

```
@compiler >= 4

contract CryptoHamster =

    record state = {
        index : int,
        map_hamsters : map(string, hamster),
        testvalue: int}

    record hamster = {
        id : int,
        name : string,
        dna : int}

    stateful entrypoint init() =
        { index = 1,
            map_hamsters = {},
            testvalue = 42}

    public entrypoint read_test_value() : int =
        state.testvalue

    public entrypoint return_caller() : address =
        Call.caller

    public entrypoint cause_error() : unit =
        require(2 == 1, "require failed")

    public stateful entrypoint add_test_value(one: int, two: int) : int =
        put(state{testvalue = one + two})
        one + two

    public entrypoint locally_add_two(one: int, two: int) : int =
        one + two

    public stateful entrypoint statefully_add_two(one: int, two: int) =
        put(state{testvalue = one + two})

    stateful entrypoint create_hamster(hamster_name: string) =
        require(!name_exists(hamster_name), "Name is already taken")
        let dna : int = generate_random_dna(hamster_name)
        create_hamster_by_name_dna(hamster_name, dna)

    entrypoint name_exists(name: string) : bool =
        Map.member(name, state.map_hamsters)

    entrypoint get_hamster_dna(name: string, test: option(int)) : int =
        require(name_exists(name), "There is no hamster with that name!")

        let needed_hamster : hamster = state.map_hamsters[name]

        needed_hamster.dna

    private stateful function create_hamster_by_name_dna(name: string, dna: int) =
```

(continues on next page)

(continued from previous page)

```
        let new_hamster : hamster = {
            id = state.index,
            name = name,
            dna = dna}

        put(state{map_hamsters[name] = new_hamster})
        put(state{index = (state.index + 1)})

    private function generate_random_dna(name: string) : int =
        get_block_hash_bytes_as_int() - Chain.timestamp + state.index

    private function get_block_hash_bytes_as_int() : int =
        switch(Chain.block_hash(Chain.block_height - 1))
            None => abort("blockhash not found")
            Some(bytes) => Bytes.to_int(bytes)

    entrypoint test(name: string) : hash =
        String.sha3(name)
```

1.4.3 Importing required classes and methods

We need to import the following classes to use contracts.

```
from aeternity.node import NodeClient, Config
from aeternity.compiler import CompilerClient
from aeternity.contract_native import ContractNative
from aeternity.signing import Account
```

1.4.4 Initializing NodeClient and Compiler

Below are the steps required to initialize the *NodeClient* and *Compiler*. As you can see below, during the initialization of *NodeClient* we're also providing the *internal_url*.

internal_url provides the debug endpoint to *dry_run* a contract method which can also be used to do static calls on deployed contracts and this is what exactly we're going to use this for.

You can also not provide the *internal_url* but then you'll have to disable the use of *dry-run* endpoint. We'll see how to do that when we initialize our contract.

```
NODE_URL = os.environ.get('TEST_URL', 'http://127.0.0.1:3013')
NODE_INTERNAL_URL = os.environ.get('TEST_DEBUG_URL', 'http://127.0.0.1:3113')
COMPILER_URL = os.environ.get('TEST_COMPILER__URL', 'https://compiler.aepps.com')

node_cli = NodeClient(Config(
    external_url=NODE_URL,
    internal_url=NODE_INTERNAL_URL,
    blocking_mode=True,
))

compiler = CompilerClient(compiler_url=COMPILER_URL)
```

1.4.5 Generate an Account

You'll need an account (using the *Account* class) for stateful contract calls.

```
# genrate ALICE account (and transfer AE to alice account)
alice = Account.generate()
```

1.4.6 Read the Contract from file and initialize

You can read the contract from the stored *.aes* file and use it to initialize the contract instance. If you have not provided the *internal_endpoint* or simple do not want to use the *dry-run* functionality you can disable it by passing *use-dry-run=False* to the *ContractNative* constructor.

Warning: If you DO NOT provide the *internal_url* during NodeClient initialization and also DID NOT disable the *dry-run* then the contract method calls for *un-stateful* methods WILL FAIL.

```
CONTRACT_FILE = os.path.join(os.path.dirname(__file__), "testdata/CryptoHamster.aes")

# read contract file
with open(CONTRACT_FILE, 'r') as file:
    crypto_hamster_contract = file.read()

"""
Initialize the contract instance
Note: To disable use of dry-run endpoint add the parameter
use_dry_run=False
"""
crypto_hamster = ContractNative(client=node_cli,
                                compiler=compiler,
                                account=alice,
                                source=crypto_hamster_contract)
```

Now pass the address of the deployed contract

Warning: If the contract is not found at the provided address or the on-chain bytecode does not match, for the given network, the method will fail.

```
# CONTRACT_ID is the address of the deployed contract
crypto_hamster.at(CONTRACT_ID)
```

1.4.7 Call the contract methods

All the methods inside the contract are also available (with same signature) to use from the contract instance.

Note: All the methods that are NOT *stateful*, by default are processed using the *dry-run* endpoint to save gas. And therefore, a transaction hash will also not be provided for them. This functionality can be either diabled for the contract instance or per method by using *use_dry_run* argument.

```
# call the contract method (stateful)
tx_info, tx_result = crypto_hamster.create_hamster("SuperCryptoHamster")

print(f"Transaction Hash: {tx_info.tx_hash}")
print(f"Transaction Result/Return Data: {tx_result}")
```

And in a similar way a not stateful call can be invoked

```
# call contract method (not stateful)
tx_info, tx_result = crypto_hamster.get_hamster_dna("SuperCryptoHamster", None)

print(f"Transaction Result/Return Data: {tx_result}")
```

1.5 Claiming a name

This guide describes how you can leverage aepp-sdk to claim a name on the aternity blockchain.

1.5.1 Prerequisites

Assumptions

We're going to assume that you have working knowledge of the SDK and know how to create an Account instance, a NodeClient, and a CompilerClient.

TODO

1.6 Using Generalized Accounts

This page describe how to use the Python SDK to convert a basic account to a generalized one. For additional information about genrealized accounts visit the **`protocol documentation`_**

1.6.1 Example

The following example describe how to transform a basic account to a generalized one and how to execute a spend transaction from a generalized account.

```
from aeternity.node import NodeClient, Config
from aeternity.signing import Account
from aeternity.transactions import TxBuilder
from aeternity.compiler import CompilerClient
from aeternity import defaults, hashing, utils

import requests



def top_up_account(account_address):

    print()
    print(f"top up account {account_address} using the testnet.faucet.aepps.com app")
    r = requests.post(f"https://testnet.faucet.aepps.com/account/{account_address}").
↪json()
    tx_hash = r.get("tx_hash")
    balance = utils.format_amount(r.get("balance"))
    print(f"account {account_address} has now a balance of {balance}")
    print(f"faucet transaction hash {tx_hash}")
    print()



def main():

    """
    This example is divided in 2 parts
    - the first one is about transforming a Basic account to Generalized
    - the second shows how to post transactions from GA accounts
    """

    #
    # PART 1: fro Basic to GA
    #
    print("Part #1 - transform a Basic account to a Generalized one")
    print()

    # first we create a new account
    # this will be the ga account
    print(f"generate a new account")
    ga_account = Account.generate()
    print(f"your basic account address is {ga_account.get_address()}")

    # use the faucet to top up your ga account
    top_up_account(ga_account.get_address())

    # Instantiate a node client
    node_url = "http://sdk-testnet.aepps.com"
    print(f"using node at {node_url}")
    # get the node client
    n = NodeClient(Config(
        external_url=node_url,
```

(continues on next page)

(continued from previous page)

```
        blocking_mode=True # we want the client to verify the transactions are␣
↪included in the chain
    ))


    # we are going to use a contract that will not be performing
    # any real authorization, just to provide a proof of concept
    # for the ga interaction
    #
    # DO NOT USE THIS CONTRACT IN A REAL SCENARIO
    # DO NOT USE THIS CONTRACT IN A REAL SCENARIO
    # DO NOT USE THIS CONTRACT IN A REAL SCENARIO
    #
    # this contract authorizes anybody to perform transactions
    # using the funds of the account!
    #
    blind_auth_contract = """contract BlindAuth =
    record state = { owner : address }

    entrypoint init(owner' : address) = { owner = owner' }

    stateful entrypoint authorize(r: int) : bool =
        // r is a random number only used to make tx hashes unique
        switch(Auth.tx_hash)
            None          => abort("Not in Auth context")
            Some(tx_hash) => true
    """

    # Instantiate a compiler client
    compiler_url = "https://compiler.aepps.com"
    print(f"using compiler at {compiler_url}")
    # get the node client
    c = CompilerClient(compiler_url=compiler_url)

    print()

    # compile the contract for the ga an retrieve the bytecode
    print("compile ga contract")
    bytecode = c.compile(blind_auth_contract).bytecode

    # prepare the calldata for the init function
    print("encode the init function calldata")
    init_calldata = c.encode_calldata(blind_auth_contract, "init", [ga_account.get_
↪address()]).calldata

    # now we execute the first step, we'll be transforming the account into a ga
    print("execute the GaAttach transaction")
    ga_attach_tx = n.account_basic_to_ga(
        ga_account, # the ga account
        bytecode, # the bytecode of the ga contract
        init_calldata=init_calldata, # the encoded parameters of the init function
        auth_fun="authorize", # the name of the authentication function to use from␣
↪the contract
        gas=1000
    )
    print(f"GaAttachTx hash is {ga_attach_tx.hash}")
    print(f"the account {ga_account.get_address()} is now generalized")
```

(continues on next page)

(continued from previous page)

```
    #
    # END of PART 1
    #

    print()

    #
    # PART 2: posting a transaction from a GA account
    #
    # In this part we will be creating a spend transaction and we'll transfer 4AE
    # from the generalized account to a newly created account
    #
    print("Part #2 - Create a spend transaction from a GA account")
    print()

    # we will be using the compiler client and the node client from the PART 1

    # first we create a new account
    # this will be the recipient account
    rc_account_address = "ak_2iBPH7HUz3cSDVEUWiHg76MZJ6tZooVNBmmxcgVK6VV8KAE688"
    print(f"the recipient account address is {rc_account_address}")

    # then we prepare the parameters for a spend transaction
    sender_id = ga_account.get_address() # the ga sendder account
    amount = 4000000000000000000 # we will be sending 4.9AE
    payload = "" # we'll be sending an empty payload
    fee = defaults.FEE # we'll use the default fee (the client will generate the␣
↪right fee for us)
    ttl = defaults.TX_TTL # we'll use the default ttl for the transaction
    nonce = defaults.GA_ACCOUNTS_NONCE # we'll use 0 as nonce since is is a special␣
↪case

    # now we'll use the builder to prepare the spend transaction
    print(f"prepare a spend transaction from {sender_id} to {rc_account_address} of
↪{utils.format_amount(amount)}")
    builder = TxBuilder()
    spend_tx = builder.tx_spend(sender_id, rc_account_address, amount, payload, fee,␣
↪ttl, nonce)

    # now that we have the transaction we need to prepare the authentication data for␣
↪the ga transaction
    print("encode the authorize function calldata")
    auth_calldata = c.encode_calldata(blind_auth_contract, "authorize", [hashing.
↪randint()]).calldata

    # and use the sign_transaction with the auth_calldata to automatically
    # prepare the ga transaction for us
    print("execute the GaMeta transaction")
    ga_meta_tx = n.sign_transaction(ga_account, spend_tx, auth_data=auth_calldata)

    # and finally we can broadcast the transaction
    ga_meta_tx_hash = n.broadcast_transaction(ga_meta_tx)
    print(f"GaMetaTx hash is {ga_meta_tx_hash}")
    print(f"the account spend transaction has been executed")

    # note that you can verify all the steps above using the command line client
```

(continues on next page)

(continued from previous page)

```
    print()
    print("Verify the steps using the command line client")
    print("1. Check the ga account:")
    print(f"aecli inspect {ga_account.get_address()}")
    print("2. Check the GaAttachTx:")
    print(f"aecli inspect {ga_attach_tx.hash}")
    print("3. Check the GaMetaTx:")
    print(f"aecli inspect {ga_meta_tx_hash}")


if __name__ == "__main__":
    main()
```

1.7 Command Line Interface (CLI)

We'll assume you have *the SDK installed* already. You can tell the SDK is installed and which version by running the following command in a shell prompt (indicated by the $ prefix):

1.7.1 CLI Usage

See below for programmatic usage

You can launch the command line tool using

```
$ aecli
```

Available commands:

```
Usage: aecli [OPTIONS] COMMAND [ARGS]...

  Welcome to the aecli client.

  The client is to interact with an node node.

Options:
  --version            Show the version and exit.
  -u, --url URL        Node node url
  -d, --debug-url URL
  --force              Ignore node version compatibility check
  --wait               Wait for transactions to be included
  --json               Print output in JSON format
  --version            Show the version and exit.
  --help               Show this message and exit.

Commands:
  account  Handle account operations
  chain    Interact with the blockchain
  config   Print the client configuration
  inspect  Get information on transactions, blocks,...
  name     Handle name lifecycle
  oracle   Interact with oracles
tx       Handle transactions creation
```

Environment variables

Use the environment variables

- NODE_URL
- NODE_URL_DEBUG

1.7.2 Example usage

The following is a walkthrough to execute a spend transaction on the testnet network

1. Set the environment variables

```
$ export NODE_URL=https://testnet.aeternity.io
```

When not set the command line client will connect to mainnet

1. Retrieve the top block

```
$ aecli chain top
<top for node at https://testnet.aeternity.io >
  Hash ____________________________________________________ mh_
↪2GKNC4Nft3mTtiUizsgmNySM6bmrtEk47AFd7QYxGLni6dHuSH
  Height __________________________________________________ 171074
  Pof hash ________________________________________________ no_fraud
  Prev hash _______________________________________________ mh_
↪2C1qtupSRTGhfgXStkFgPgTU7EtRpbj3R6JeqY94mKDnUM5erw
  Prev key hash ___________________________________________ kh_
↪2mMnrh68xvs25qqnY6Q6BCYPPGEfW2AshvThNDugut12KGCjTp
  Signature _______________________________________________ sg_
↪UGs78hj7QJ96TZR4vLmrUPmXW2STFPQCR49q1mEvyEHWotC1k8DB6qfqwJGFjRtZL27rSEitLRee5wCqbs4XtGVZAFiCk
  State hash ______________________________________________ bs_
↪2ZvCLwHsVPkEn43qvviRpr1vQ1snpGd1izex6kScbU2Zd99s9S
  Time ____________________________________________________ 2019-11-20T11:11:22.934000+00:00
  Txs hash ________________________________________________ bx_
↪D4Y9x2RjobeCEETZqbVgYFsNu6UKhdSTAJXdXZBb5JbAuW7QA
  Version _________________________________________________ 4
</top for node at https://testnet.aeternity.io >
```

1. Create a new account

```
$ aecli account create Bob.json
Enter the account password []:
<account>
  Address _________________________________________________ ak_
↪BobY97QUVR4iDLg4k3RKmy6shZYx9FR75nLaN33GsVmSnhWxn
  Path ____________________________________________________ /.../Bob.json
</account>
```

Make sure that you use a long and difficult-to-guess password for an account that you plan to use on mainnet

1. Go to testnet.faucet.aepps.com and top up your account
2. Inspect the transaction reported by the faucet app

```
$ aecli inspect th_2CV4a7xxDYj5ysaDjXNoCSLxnkowGM5bbyAvtdoPvHZwTSYykX
<transaction>
  <data>
```

(continues on next page)

(continued from previous page)

```
    Block height ____________________________________ 12472
    Block hash ______________________________________ mh_
↪2vjFffExUZPVGo3q6CHRSzxVUhzLcUnQQUWpijFtSvKfoHwQWe
    Hash ____________________________________________ th_
↪2CV4a7xxDYj5ysaDjXNoCSLxnkowGM5bbyAvtdoPvHZwTSYykX
    <signatures 1>
      Signature #1 __________________________________ sg_
↪WtPeyKWN4zmcnZZXpAxCT8EvjF3qSjiUidc9cdxQooxe1JCLADTVbKDFm9S5bNwv3yq57PQKTG4XuUP4eTzD5jymPHpNu
    </signatures>
    <tx>
      <data>
        Amount ________________________________________ 5AE
        Fee ___________________________________________ 2.0000E-14AE
        Nonce _________________________________________ 146
        Payload _______________________________________ ba_RmF1Y2V0IFR4tYtyuw==
        Recipient id __________________________________ ak_
↪2ioQbdSViNKjknaLUWphdRjpbTNVpMHpXf9X5ZkoVrhrCZGuyW
        Sender id _____________________________________ ak_
↪2iBPH7HUz3cSDVEUWiHg76MZJ6tZooVNBmmxcgVK6VV8KAE688
        Ttl ___________________________________________ 12522
        Type __________________________________________ SpendTx
        Version _______________________________________ 1
        Tag ___________________________________________ 12
      </data>
      <metadata>
        Min fee _______________________________________ 0.00001706AE
      </metadata>
      Tx _____________________________________________ tx_+GEMAaEB4TK48d23oE5jt/
↪qWR5pUu8UlpTGn8bwM5JISGQMGf7ChAeKbvpV6jbxy/
↪le3tbquBoRjk0ehgOsRdSzc09bKfw3uiEVjkYJE9AAAgk4ggjDqgZKJRmF1Y2V0IFR47rafGA==
    </tx>
    Tag ________________________________________________ 11
    Type _______________________________________________ SignedTx
    Version ____________________________________________ 1
  </data>
  Tx ____________________________________________________ tx_+KsLAfhCuEDkb9XQq/
↪ihtZ+DNbDBI/9ntVGwcJJLCV0qZE8c9wMxAeTyyG3hVqthIco/
↪NLuaVQ0N3XRYhYb6PsYOjAf8hzMJuGP4YQwBoQHhMrjx3begTmO3+pZHmlS7xSWlMafxvAzkkhIZAwZ/
↪sKEB4pu+lXqNvHL+V7e1uq4GhGOTR6GA6xF1LNzT1sp/
↪De6IRWORgkT0AACCTiCCMOqBkolGYXVjZXQgVHgQZGwg
  Hash _________________________________________________ th_
↪2CV4a7xxDYj5ysaDjXNoCSLxnkowGM5bbyAvtdoPvHZwTSYykX
</transaction>
```

1. Create another account

```
$ aecli account create Alice.json
Enter the account password []:
<account>
  Address ______________________________________________ ak_
↪9j8akv2PE2Mnt5khFeDvS9BGc3TBBrJkfcgaJHgBXcLLagX8M
  Path _________________________________________________ /.../Alice.json
</account>
```

1. Transfer some tokens to an account to the other

```
$ aecli account spend Bob.json ak_9j8akv2PE2Mnt5khFeDvS9BGc3TBBrJkfcgaJHgBXcLLagX8M␣
↪1AE
Enter the account password []:
<spend transaction>
  <data>
    Tag ____________________________________________ 12
    Vsn ____________________________________________ 1
    Sender id ______________________________________ ak_
↪BobY97QUVR4iDLg4k3RKmy6shZYx9FR75nLaN33GsVmSnhWxn
    Recipient id ___________________________________ ak_
↪9j8akv2PE2Mnt5khFeDvS9BGc3TBBrJkfcgaJHgBXcLLagX8M
    Amount _________________________________________ 1AE
    Fee ____________________________________________ 0.00001686AE
    Ttl ____________________________________________ 0
    Nonce __________________________________________ 4
    Payload ________________________________________
  </data>
  Metadata
  Tx _____________________________________________ tx_
↪+KMLAfhCuEAKN05UwTV0fSgO5woziVNnAMBcDrh46XlNFTZTJQlI05fz/8pVSyrb1guCLcw8n7++O887k/
↪JEu6/XHcCSHOMMuFv4WQwBoQEYh8aMDs7saMDBvys+lbKds3Omnzm4crYNbs9xGolBm6EBE9B4l/BeyxMO//
↪3ANxwyT+ZHL52j9nAZosRe/YFuK4eIDeC2s6dkAACGD1WGT5gAAASAN24JGA==
  Hash ___________________________________________ th_
↪2gAL72dtnaeDcZoZA9MbfSL1JrWzNErMJuikmTRvBY8zhkGh91
  Signature ______________________________________ sg_
↪2LX9hnJRiYGSspzpS34QeN3PLT9bGSkFRbad9LXvLj5QUFoV5eHRf9SueDgLiiquCGbeFEBPBe7xMJidf8NMSuF16dngr
  Network id _____________________________________ ae_uat
</spend transaction>
```

1. Verify the balance of the new account

```
$ aecli inspect ak_9j8akv2PE2Mnt5khFeDvS9BGc3TBBrJkfcgaJHgBXcLLagX8M
<account>
  Balance ________________________________________ 1AE
  Id ______________________________________________ ak_
↪9j8akv2PE2Mnt5khFeDvS9BGc3TBBrJkfcgaJHgBXcLLagX8M
  Kind ___________________________________________ basic
  Nonce __________________________________________ 0
  Payable ________________________________________ True
</account>
```

1.8 Generating and using a HD Wallet

This tutorial covers the generation, use and management of a HD Wallet using aepp-sdk.

Note: Do not know what HD Wallets are? You can read about it here.

1.8.1 Importing the HD Wallet class

```
from aeternity.hdwallet import HDWallet
```

1.8.2 Generating a Mnemonic

The HdWallet class exposes a static method to generate a BIP39 compatible mnemonic seed phrase.

```
# Generating a mnemonic
mnemonic = HDWallet.generate_mnemonic()
```

1.8.3 Generating a HD Wallet

You can instantiate a HDWallet instance by using the *HDWallet* class. The constructor accepts a mnemonic seed phrase and if no seed phrase is provided then one is auto generated.

```
# Initializing the HDWallet
# (if mnemonic is not provided, a one will be autogenerated during initialization)
hdwallet = HDWallet(mnemonic)
```

1.8.4 Deriving a child private key

Once the HD wallet is instantiated, you can access the provided class method to derive child private/secret keys.

```
# Derive child accounts
# if you want to derive a child key with an specific account and/or address index
# then you can pass them to the method
# By default, every time you generate a child key, the account index is auto generated
key_path, account = hdwallet.derive_child()
```

1.9 Offline transactions

Throughout this tutorial, we'll walk you through the creation of a basic script to generate a list of spend transactions and to sign them.

It'll consist of three parts:

- Build a list of spend transactions (offline)
- Sign each transactions (offline)
- Broadcast the transactions obtained in the above steps

We'll assume you have *the SDK installed*.

First import the required libraries

```
from aeternity.node import NodeClient, Config
from aeternity.signing import Account
from aeternity.transactions import TxBuilder, TxSigner
from aeternity import utils, defaults, identifiers
import os
```

For the next steps we need to have an account available.

```
# Accounts addresses
account = Account.generate()
```

Hint: To successfully complete the tutorial, you will have to have a positive balance on the account just createad. How to get some funds is *explained in the first tutorial* using the Faucet app.

Once we have an account available we will use the *TxBuilder* to prepare the transactions.

```
# instantiate the transactions builder
build = TxBuilder()

# we will be creating 5 transactions for later broadcast TODO: warn about the nonce␣
↪limit
txs = []

# each transaction is going to be a spend
amount = utils.amount_to_aettos("0.05AE")
payload = b''

for i in range(5):
    # increase the account nonce
    account.nonce = account.nonce + 1
    # build the transaction
    tx = build.tx_spend(
        account.get_address(),  # sender
        Account.generate().get_address(),  # random generated recipient
        amount,
        payload,
        defaults.FEE,
        defaults.TX_TTL,
        account.nonce
    )
    # save the transaction
    txs.append(tx)
```

Once the transactions are ready they need to be signed and encapsulated in signed transaction:

Hint: the Network ID is required to compute a valid signature for transactions; when using the online node client the Network ID is automatically retrieved from the node.

```
# define the network_id
network_id = identifiers.NETWORK_ID_TESTNET
```

The *TxSigner* object provides the functionality to compute the signature for transactions.

```
# instantiate a transaction signer
signer = TxSigner(account, network_id)

# collect the signed tx for broadcast
signed_txs = []
# sign all transactions
```

(continues on next page)

(continued from previous page)

```
for tx in txs:
    signature = signer.sign_transaction(tx)
    signed_tx = build.tx_signed([signature], tx)
    signed_txs.append(signed_tx)
```

Finally we can instantiate a node client and broadcast the transactions:

```
# Broadcast the transactions
NODE_URL = os.environ.get('TEST_URL', 'https://testnet.aeternity.io')

node_cli = NodeClient(Config(
    external_url=NODE_URL,
    blocking_mode=False,
))

# broadcast all transactions
for stx in signed_txs:
    node_cli.broadcast_transaction(stx)

# verify that all transactions have been posted
for stx in signed_txs:
    height = node_cli.wait_for_transaction(stx)
    assert(height > 0)
```

Thats it! You have successfully executed your transaction in the Aeternity Blockchain testnet network. For the mainnet network the procedure is the same except you will have to get some tokens via an exchange or via other means.

1.10 Contributing

Contributing to the Aeternity Python SDK

1.10.1 Pull Requests

- Squash your commits to ensure each resulting commit is stable and can be tested.
- Include issue numbers in the PR title, e.g. “GH-128. Resolves issue #123”.
- Add unit tests for self-contained modules.
- Add integration tests as needed.
- Document new code.

1.10.2 Git Commit Messages

- Use the present tense (“Add feature” not “Added feature”)
- Use the imperative mood (“Move cursor to…” not “Moves cursor to…”)
- Limit the first line to 72 characters or less
- Reference issues and pull requests liberally after the first line

Learn more about writing a good commit message.

See also:

If you are new to Aeternity blockchain, you may want to start by checking the website and the forum.

See also:

If you’re new to Python, you might want to start by getting an idea of what the language is like.

If you’re new to programming entirely, you might want to start with this list of Python resources for non-programmers

If you already know a few other languages and want to get up to speed with Python quickly, we recommend Dive Into Python. If that’s not quite your style, there are many other books about Python.

CHAPTER

TWO

"HOW-TO" GUIDES

Here you'll find short answers to "How do I....?" types of questions. These how-to guides don't cover topics in depth. However, these guides will help you quickly accomplish common tasks.

2.1 Account management via CLI

The Python SDK comes with a command line client (CLI) that can be used among other things to generate, store, inspect and load accounts

2.1.1 Generate a new account

The command to generate a new (non HD) account is:

```
$ aecli account create KEYSTORE_FILENAME.json
```

the command will ask for a password and will store the account in the file named `KEYSTORE_FILENAME.json`; here an example:

```
 aecli account create  BOB.json
Enter the account password []:
<account>
  Address ____________________________________________ ak_
↪2P44NhGFT7fP1TFpZ62FwabWP9djg5sDwFNDVVpiGgd3p4yA7X
  Path _______________________________________________ /howto/BOB.json
</account>
```

2.1.2 Print the secret key of an account

Warning: The secret key is printed in "clear text", by printing it on the terminal or saving it unencrypted you expose your account

the command to print the public and secret key of an account saved in a keystore file is:

```
$ aecli account address --secret-key KEYSTORE_FILENAME.json
```

Example:

```
 aecli account address BOB.json --secret-key
Enter the account password:
!Warning! this will print your secret key on the screen, are you sure? [y/N]: y
<account>
  Address ____________________________________________ ak_
↪2P44NhGFT7fP1TFpZ62FwabWP9djg5sDwFNDVVpiGgd3p4yA7X
  Secretkey __________________________________________ c226bf650f30740287f77a715.......
↪.....f49ddff758971112fb5cfb0e66975a8f
</account>
```

2.2 Validate contract bytecode

When deploying a contract on the aeternity blockchain you have to compile the source into bytecode and then create a transaction that includes this bytecode. But when the contract is deployed, aeternity node initializes the contract using the provided *init* method and then removes it from the bytecode before storing it on the chain. Therefore, given the contract source and it's address on the network you cannot verify the source.

To overcome this behaviour and validate the bytecode, you can refer to the below example:

Note: This method is only available if you are using compiler version >= 4.1.0.

2.2.1 Deploy the contract (if it is not already)

```
        "function": "main",
        "arguments":  [42],
        "response": 42
    }
]
compiler = compiler_fixture.COMPILER
for t in tests:
    result = compiler.decode_calldata_with_sourcecode(t.get("sourcecode"), t.get(
↪'function'), t.get('calldata'))
```

2.2.2 Validate the contract

```
        result.function == t.get("function") and
        result.arguments[0].value == t.get("response")
    ))
```

2.3 Amounts

The Aeternity blockchain tokens are an ERC-20 compatible token, where the unit is the Aetto (1e-18).

The `utils` submodule provides functions to manipulate the amounts.

2.3.1 Format amount to a human readable format

The following snippets show how to format an amount from `aetto` to a human readable format.

```
# import the utils submodule
from aeternity import utils

amount = 1000000000000000000
human_value = utils.format_amount(amount)
print(human_value) # prints 1AE
```

2.3.2 Parse amounts

This snippets shows how to parse amounts expressed in a human friendly, float or scientific notation.

```
# import the utils submodule
from aeternity import utils

amount = utils.amount_to_aettos(1.2)
print(amount) # prints 1200000000000000000

amount = utils.amount_to_aettos("10AE")
print(amount) # prints 10000000000000000000

amount = utils.amount_to_aettos(1e1)
print(amount) # prints 10000000000000000000

amount = utils.amount_to_aettos(1)
print(amount) # prints 1
```

Important: when using amount_to_aettos function, the maximum value as imput is 1e9, that will result in an aetto value of 1000000000000000000000000000 (1e27). Bigger values wil return in an error.

2.4 Transaction delegation signatures

The Sophia language for smart contracts allow to delegate the transaction execution to a contract by providing delegation signatures.

2.4.1 Delegate signatures for AENS

The following code snippet shows how to generate signatures for name transactions delegation to a contract

```
# import the required libraries
from aeternity.node import NodeClient, Config
from aeternity.signing import Account

# initialize the node client
node_cli = NodeClient(Config(external_url="https://mainnet.aeternal.io"))

# get an account
account = Account.from_keystore("/path/to/keystore", "keystore password")

# an example name
name = "example.chain"

# name preclaim signature delegation
sig = node_cli.delegate_name_preclaim_signature(account, contract_id)

# name claim signature delegation
sig = node_cli.delegate_name_claim_signature(account, contract_id, name)

# name revoke signature delegation
sig = node_cli.delegate_name_revoke_signature(account, contract_id, name)

# name revoke signature delegation
sig = node_cli.delegate_name_transfer_signature(account, contract_id, name)
```

See also:

Many writers in the Aeternity forum write this sort of how-to material.

CHAPTER

THREE

TOPICS

3.1 How to install the Aeternity SDK

This document will explain different way to install the Aeternity Python SDK

3.1.1 Install Python

Get the latest version of Python at https://www.python.org/downloads/ or with your operating system's package manager.

3.1.2 Install the SDK

Installation instructions are slightly different depending on whether you're installing official release or fetching the latest development version.

Installing an official release with `pip`

This is the recommended way to install the Aeternity SDK

1. Install pip. The easiest is to use the standalone pip installer. If your distribution already has `pip` installed, you might need to update it if it's outdated. If it's outdated, you'll know because installation won't work.
2. Take a look at virtualenv and virtualenvwrapper. These tools provide isolated Python environments, which are more practical than installing packages system-wide. They also allow installing packages without administrator privileges. The *contributing tutorial* walks through how to create a virtualenv.
3. After you've created and activated a virtual environment, enter the command:

```
$ python -m pip install aepp-sdk
```

Installing the development version

If you'd like to be able to update your SDK code occasionally with the latest bug fixes and improvements, follow these instructions:

1. Make sure that you have **Git_** installed and that you can run its commands from a shell. (Enter `git help` at a shell prompt to test this.)
2. Install Poetry and make sure it is available in your `PATH`
3. Check out the SDK main development branch like so:

```
$ git clone https://github.com/aeternity/aepp-sdk-python.git
```

This will create a directory `aepp-sdk-python` in your current directory.

1. Make sure that the Python interpreter can load the SDK's code. The most convenient way to do this is to use virtualenv, virtualenvwrapper, and pip.
2. After setting up and activating the virtualenv, run the following command:

```
$ poetry build
$ python -m pip install dist/$(ls -tr dist | grep whl | tail -1)
```

This will make the SDK code importable, and will also make the `aecli` utility command available. In other words, you're all set!

When you want to update your copy of the SDK source code, run the command `git pull` from within the `aepp-sdk-directory` directory. When you do this, Git will download any changes.

Where is the command line client?

The Python SDK bundles a command line client `aecli` that can be use to submit transactions and poke around the Aeternity blockchain. To be able to access the command line client you have to add it to your executable path. To find out where is your base path for installed python libraries use the command `python -m site --user-base`

3.2 Keystore format change [LEGACY]

The release 0.25.0.1 of the Python SDK change the format of the keystore used to store a secret key in an encrypted format.

The previous format, suported till version `0.25.0.1b1` is not supported anymore and secret keys encrypted with the legacy format will have to be updated manually.

> **Warning:** BEFORE YOU START
>
> The following procedure will print your secret key unencrypted on the terminal.
>
> **Make sure you perform this procedure in a secret setting.**

The follwing steps are required to upgade a keystore (named *my_account.json*).

1. Install the *aepp-sdk* v0.25.0.1b1

```
$ pip install aepp-sdk==0.25.0.1b1
```

Verify that you are using the correct version

```
$ aecli --version
aecli, version 0.25.0.1b1
```

1. Print your secret key to the terminal

```
aecli account address --secret-key my_account.json
Enter the account password []:
!Warning! this will print your secret key on the screen, are you sure? [y/N]: y
<account>
  Address ____________________________________________ ak_
↪2UeaQn7Ei7HoMvDTiq2jyDuE8ymEQMPZExzC64qWTxpUnanYsE
  Signing key ____________________________________________␣
↪507f598f2fac1a4ab57edae53650cbf7ffae9eeeea1a297cc7c3b6172052e55ec27954c4ba901cf9b3760dc12b2c313d60
</account>
```

1. Install the *aepp-sdk* v0.25.0.1

```
$ pip install aepp-sdk==0.25.0.1
```

Verify that you are using the correct version

```
$ aecli --version
aecli, version 0.25.0.1
```

1. Save the secret key to the new format

```
aecli account save my_account_new.json␣
↪507f598f2fac1a4ab57edae53650cbf7ffae9eeeea1a297cc7c3b6172052e55ec27954c4ba901cf9b3760dc12b2c313d60
Enter the account password []:
<account>
  Address ____________________________________________ ak_
↪2UeaQn7Ei7HoMvDTiq2jyDuE8ymEQMPZExzC64qWTxpUnanYsE
  Path ____________________________________________ /Users/andrea/tmp/aeternity/my_
↪account_new.json
</account>
```

5. Verify that the new keystore has the correct format

the content of the keystore file should appear as the following:

```
$ cat my_account_new.json
{"public_key": "ak_2UeaQn7Ei7HoMvDTiq2jyDuE8ymEQMPZExzC64qWTxpUnanYsE", "crypto": {
↪"secret_type": "ed25519", "symmetric_alg": "xsalsa20-poly1305", "ciphertext":
↪"f431af7e6f00da7f9acc8187900b97d42526eb135f4db0da80a7809f36a00e37f3a313f7c611784f381e58620bb2c23ef2
↪", "cipher_params": {"nonce": "1ea25d885a68adf13998e0fad17b22e7ade78f5cf1670eb1"},
↪"kdf": "argon2id", "kdf_params": {"memlimit_kib": 262144, "opslimit": 3, "salt":
↪"c3dd4a4ac8347b3ad706756b96919387", "parallelism": 1}}, "id": "44c3d693-a890-4ac1-
↪936b-0a65c8293388", "name": "", "version": 1}
```

6. Cleanup!

Once the operation is completed, cleanup the terminaly history that contains your secret key

```
history -c
```

And remove the old account file

```
rm my_account.json
```

CHAPTER

FOUR

API REFERENCE

4.1 Node Client

The `Config` class is used to intialized the `NodeClient`

class aeternity.node.**Config**(*external_url='http://localhost:3013'*, *internal_url='http://localhost:3113'*, *websocket_url='http://localhost:3014'*, *force_compatibility=False*, ***kwargs*)

__init__(*external_url='http://localhost:3013'*, *internal_url='http://localhost:3113'*, *websocket_url='http://localhost:3014'*, *force_compatibility=False*, ***kwargs*)

Initialize a configuration object to be used with the NodeClient

Args: external_url (str): the node external url internal_url (str): the node internal url websocket_url (str): the node websocket url force_compatibility (bool): ignore node version compatibility check (default False)

Kwargs:

blocking_mode (bool) whenever to block the execution when broadcasting a transaction until the transaction is mined in the chain, default: False

tx_gas_per_byte (int) the amount of gas per byte used to calculate the transaction fees, [optional, default: defaults.GAS_PER_BYTE]

tx_base_gas (int) the amount of gas per byte used to calculate the transaction fees, [optional, default: defaults.BASE_GAS]

tx_gas_price (int) the gas price used to calculate the transaction fees, it is set by consensus but can be increased by miners [optional, default: defaults.GAS_PRICE]

network_id (str) the id of the network that is the target for the transactions, if not provided it will be automatically selected by the client

contract_gas (int) default gas limit to be used for contract calls, [optional, default: defaults.CONTRACT_GAS]

contract_gas_price (int) default gas price used for contract calls, [optional, default: defaults.CONTRACT_GAS_PRICE]

oracle_ttl_type (int) default oracle ttl type

key_block_interval (int) the key block interval in minutes

key_block_confirmation_num (int) the number of key blocks to consider a transaction confirmed

poll_tx_max_retries (int) max poll retries when checking if a transaction has been included

poll_tx_retries_interval (int) the interval in seconds between retries

poll_block_max_retries (int) TODO

poll_block_retries_interval (int) TODO

offline (bool) whenever the node should not contact the node for any information

debug (bool) enable debug logging for api calls

The `NodeClient` is the main object to interact with an Aeternity node.

class aeternity.node.**NodeClient**(*config=<aeternity.node.Config object>*)

account_basic_to_ga(*account: aeternity.signing.Account*, *ga_contract: str*, *calldata: str*, *auth_fun: str = 'authorize'*, *fee=0*, *tx_ttl: int = 0*, *gas: int = 10000*, *gas_price=1000000000*) → aeternity.transactions.TxObject

Transform a Basic to a GA (Generalized Account)

Parameters

- **account** – the account to transform
- **ga_contract** – the compiled contract associated to the GA
- **calldata** – the calldata for the authentication function
- **auth_fun** – the name of the contract function to use for authorization (default: authorize)
- **gas** – the gas limit for the authorization function execution
- **gas_price** – the gas price for the contract execution
- **fee** – TODO
- **ttl** – TODO

Returns the TxObject of the transaction

Raises

- **TypeError** – if the auth_fun is missing
- **TypeError** – if the auth_fun is no found in the contract

broadcast_transaction(*tx: aeternity.transactions.TxObject*)

Post a transaction to the chain and verify that the hash match the local calculated hash It blocks for a period of time to wait for the transaction to be included if in blocking_mode

Parameters **tx** – the transaction to broadcast

Returns the transaction hash of the transaction

Raises **TransactionHashMismatch** – if the transaction hash returned by the node is different from the one calculated

compute_absolute_ttl(*relative_ttl*)

Compute the absolute ttl by adding the ttl to the current height of the chain

Parameters **relative_ttl** – the relative ttl, if 0 will set the ttl to 0

delegate_name_claim_signature(*account: aeternity.signing.Account*, *contract_id: str*, *name: str*)

Helper to generate a signature to delegate a name claim to a contract.

Args: account: the account authorizing the transaction contract_id: the if of the contract executing the transaction name: the name being claimed

Returns: the signature to use for delegation

delegate_name_preclaim_signature(*account: aeternity.signing.Account*, *contract_id: str*)
Helper to generate a signature to delegate a name preclaim to a contract.

Args: account: the account authorizing the transaction contract_id: the if of the contract executing the transaction

Returns: the signature to use for delegation

delegate_name_revoke_signature(*account: aeternity.signing.Account*, *contract_id: str*, *name: str*)
Helper to generate a signature to delegate a name revoke to a contract.

Args: account: the account authorizing the transaction contract_id: the if of the contract executing the transaction name: the name being revoked

Returns: the signature to use for delegation

delegate_name_transfer_signature(*account: aeternity.signing.Account*, *contract_id: str*, *name: str*)
Helper to generate a signature to delegate a name tranfer to a contract.

Args: account: the account authorizing the transaction contract_id: the if of the contract executing the transaction name: the name being transferred

Returns: the signature to use for delegation

get_account(*address: str*) → aeternity.signing.Account
: Retrieve an account by it's public key

get_balance(*account*) → int
Retrieve the balance of an account, return 0 if the account has not balance

Parameters **account** – either an account address or a signing.Account object

Returns the account balance or 0 if the account is not known to the network

get_block_by_hash(*hash=None*)
Retrieve a key block or a micro block header based on the block hash_prefix

Parameters **block_hash** – either a key block or micro block hash

Returns the block matching the hash

get_consensus_protocol_version(*height: int = None*) → int
Get the consensus protocol version number :param height: the height to get the protocol version for, if None the current height will be used :return: the version

get_next_nonce(*account*, *use_cached=True*)
Get the next nonce to be used for a transaction for an account. If account is an instance Account, the cached value of the account.nonce will be used unless the parameter use_cached is set to False

Args: account: the account instance or account address of get the nonce for use_cached: use the cached value in the account.nonce property in case of an account object (default True)

Returns: the next nonce for an account

get_top_block()
Override the native method to transform the get top block response object to a Block

Returns a block (either key block or micro block)

get_transaction(*transaction_hash: str*) → aeternity.transactions.TxObject
Retrieve a transaction by it's hash.

Args: transaction_hash: the hash of the transaction to retrieve

Returns: the TxObject of the transaction

Raises: ValueError: if the transaction hash is not a valid hash for transactions

get_vm_abi_versions()
Check the version of the node and retrieve the correct values for abi and vm version

sign_transaction(*account: aeternity.signing.Account*, *tx: aeternity.transactions.TxObject*, *metadata: dict = {}*, ***kwargs*) → aeternity.transactions.TxObject
The function sign a transaction to be broadcast to the chain. It automatically detect if the account is Basic or GA and return the correct transaction to be broadcast.

Parameters

- **account** – the account signing the transaction
- **tx** – the transaction to be signed
- **metadata** – additional metadata to maintain in the TxObject
- ****kwargs** – for GA accounts, see below

Kwargs:

auth_data (str) the encoded calldata for the GA auth function

gas (int) the gas limit for the GA auth function [optional]

gas_price (int) the gas price for the GA auth function [optional]

fee (int) the fee for the GA transaction [optional, automatically calculated]

abi_version (int) the abi_version to select FATE or AEVM [optional, default 3/FATE]

ttl (str) the transaction ttl expressed in relative number of blocks [optional, default 0/max_ttl]:

Returns a TxObject of the signed transaction or metat transaction for GA

Raises

- **TypeError** – if the auth_data is missing and the account is GA
- **TypeError** – if the gas for auth_func is gt defaults.GA_MAX_AUTH_FUN_GAS

spend(*account: aeternity.signing.Account*, *recipient_id: str*, *amount*, *payload: str = ''*, *fee: int = 0*, *tx_ttl: int = 0*) → aeternity.transactions.TxObject
Create and execute a spend transaction, automatically retrieve the nonce for the siging account and calculate the absolut ttl.

Parameters

- **account** – the account signing the spend transaction (sender)
- **recipient_id** – the recipient address or name_id
- **amount** – the amount to spend
- **payload** – the payload for the transaction
- **fee** – the fee for the transaction (automatically calculated if not provided)

- **tx_ttl** – the transaction ttl expressed in relative number of blocks

Returns the TxObject of the transaction

Raises **TypeError** – if the recipient_id is not a valid name_id or address

transfer_funds(*account: aeternity.signing.Account*, *recipient_id: str*, *percentage: float*, *payload: str = ''*, *tx_ttl: int = 0*, *fee: int = 0*, *include_fee=True*)
Create and execute a spend transaction

verify(*encoded_tx: str*) → aeternity.transactions.TxObject
Unpack and verify an encoded transaction

wait_for_confirmation(*tx*, *max_retries=None*, *polling_interval=None*) → int
Wait for a transaction to be confirmed by at least "key_block_confirmation_num" blocks (default 3) The amount of blocks can be configured in the Config object using key_block_confirmation_num parameter

Args: tx (TxObject|str): the TxObject or transaction hash of the transaction to wait for max_retries (int): the maximum number of retries to test for transaction polling_interval (int): the interval between transaction polls

Returns: the block height of the transaction if it has been found

Raises: TransactionWaitTimeoutExpired: if the transaction hasn't been found

wait_for_transaction(*tx*, *max_retries=None*, *polling_interval=None*) → int
Wait for a transaction to be mined for an account The method will wait for a specific transaction to be included in the chain, it will return False if one of the following conditions are met: - the chain reply with a 404 not found (the transaction was expunged) - the account nonce is >= of the transaction nonce (transaction is in an illegal state) - the ttl of the transaction or the one passed as parameter has been reached

Args: tx (TxObject|str): the TxObject or transaction hash of the transaction to wait for max_retries (int): the maximum number of retries to test for transaction polling_interval (int): the interval between transaction polls

Returns: the block height of the transaction if it has been found

Raises: TransactionWaitTimeoutExpired: if the transaction hasn't been found

4.2 TxObject

`TxObject` is one of the central entity of the Python SDK, and it represent a transaction object.

class aeternity.transactions.**TxObject**(***kwargs*)
This is a TxObject that is used throughout the SDK for transactions It contains all the info associated with a transaction

ga_meta(*name*)
Get the value of a GA meta transaction property

Parameters **name** – the name of the property to get

Returns the property value or None if there is no such property

get(*name*)
Get the value of a property of the transaction by name, searching recursively in the TxObject structure.

For GA transaction the properties of the GA use the ga_meta() method below

Parameters **name** – the name of the property to get

Returns the property value or None if there is no such property

meta(*name*)
Get the value of a meta property such as the min_fee.

Parameters **name** – the name of the meta property

Returns the value of the meta property or none if not found

The fields of a `TxObject` are

- `hash`: the transaction hash
- `data`: the transaction data, it varies for every transaction type
- `metadata`: it contains additional data that may be relevant in the transaction context
- `tx`: the rlp + base64 check encoded string of the transaction that is used to broadcast a transaction

Since a transaction can be a nested structured, the `TxObject` is nested as well: considering a simple spend transaction, the actual structure of the transaction is:

```
SignedTx(
  tag:                  - signed transaction type (11)
  version               - transaction version, 1 in this case
  [signature]           - the list of signatures for the signed transaction
  tx: SpendTx(          - the inner spend transaction
    tag                 - spend transaction type (12)
    version             - spend transaction version, 1 in this case
    sender_id           - spend sender
    recipient_id        - spend recipient
    amount              - amount being transferred
    fee                 - fee for the miner
    ttl                 - the mempool time to live
    nonce               - sender account nonce
    payload             - arbitrary payload
  )
)
```

This means that to access, for example, the spend transaction `recipient_id` from a `TxObject`, the code would be:

```
tx_object = node_cli.spend(sender, recipient, "100AE")
# access the recipient_id
tx_object.data.tx.data.recipient_id
```

unless the transaction has been posted from a generalized account, in which case there are 4 levels of nesting:

```
tx_object = node_cli.spend(sender, recipient, "100AE")
# access the recipient_id for a GA generated transaction
tx_object.data.tx.data.tx.data.tx.data.recipient_id
```

This is of course somewhat awkward, and therefore the `TxObject` provides the `get(NAME)`, `meta(NAME)`, `ga_meta(NAME)` functions.

The functions are used to access the values of the properties without worrying about the structure of the transaction, so the example above will become:

```
tx_object = node_cli.spend(sender, recipient, "100AE")
# access the recipient_id for any spend transaction
tx_object.get("recipient_id")
```

4.2.1 Metadata

Metadatas are special informations that are not part of the transaction itself but my be generated ad a additional output while creating or parsing a transaction, in particular metadata fields are:

- `min_fee` minimum fee for a transaction, this value is always calculaed and can be used to evaluate the actual fee used for the transaction.
- `contract_id` the id of a contract, only present when deploying a new contract (starts with prefix `ct_`).
- `salt` the random generated salt to prepare the `commitment_id` of a name pre-claim transaction. The salt must be used then to prepare a claim transaction.

4.2.2 TxObject data fields

Here is the complete list of transactions and available fields:

```
(idf.OBJECT_TAG_SIGNED_TRANSACTION, 1): {
    "fee": None,  # signed transactions do not have fees
    "schema": {
        "version": Fd(1),
        "signatures": Fd(2, _SG, prefix=idf.SIGNATURE),  # list
        "tx": Fd(3, _TX, prefix=idf.TRANSACTION),
    }
},
(idf.OBJECT_TAG_SPEND_TRANSACTION, 1): {
    "fee": Fee(SIZE_BASED),
    "schema": {
        "version": Fd(1),
        "sender_id": Fd(2, _ID),
        "recipient_id": Fd(3, _ID),
        "amount": Fd(4),
        "fee": Fd(5),
        "ttl": Fd(6),
        "nonce": Fd(7),
        "payload": Fd(8, _ENC, prefix=idf.BYTE_ARRAY),
    }
},
(idf.OBJECT_TAG_NAME_SERVICE_PRECLAIM_TRANSACTION, 1): {
    "fee": Fee(SIZE_BASED),
    "schema": {
        "version": Fd(1),
        "account_id": Fd(2, _ID),
        "nonce": Fd(3),
        "commitment_id": Fd(4, _ID),
        "fee": Fd(5),
        "ttl": Fd(6),
    }},
(idf.OBJECT_TAG_NAME_SERVICE_CLAIM_TRANSACTION, 1): {
    "fee": Fee(SIZE_BASED),
    "schema": {
        "version": Fd(1),
        "account_id": Fd(2, _ID),
        "nonce": Fd(3),
        "name": Fd(4, _BIN, data_type=str),
        "name_salt": Fd(5),
        "fee": Fd(6),
```

(continues on next page)

(continued from previous page)

```
        "ttl": Fd(7),
    }},
(idf.OBJECT_TAG_NAME_SERVICE_CLAIM_TRANSACTION, 2): {
    "fee": Fee(SIZE_BASED),
    "schema": {
        "version": Fd(1),
        "account_id": Fd(2, _ID),
        "nonce": Fd(3),
        "name": Fd(4, _BIN, data_type=str),
        "name_salt": Fd(5),
        "name_fee": Fd(6),
        "fee": Fd(7),
        "ttl": Fd(8),
    }},
(idf.OBJECT_TAG_NAME_SERVICE_UPDATE_TRANSACTION, 1): {
    "fee": Fee(SIZE_BASED),
    "schema": {
        "version": Fd(1),
        "account_id": Fd(2, _ID),
        "nonce": Fd(3),
        "name_id": Fd(4, _ID),
        "name_ttl": Fd(5),
        "pointers": Fd(6, _PTR),
        "client_ttl": Fd(7),
        "fee": Fd(8),
        "ttl": Fd(9),
    }},
(idf.OBJECT_TAG_NAME_SERVICE_TRANSFER_TRANSACTION, 1): {
    "fee": Fee(SIZE_BASED),
    "schema": {
        "version": Fd(1),
        "account_id": Fd(2, _ID),
        "nonce": Fd(3),
        "name_id": Fd(4, _ID),
        "recipient_id": Fd(5, _ID),
        "fee": Fd(6),
        "ttl": Fd(7),
    }},
(idf.OBJECT_TAG_NAME_SERVICE_REVOKE_TRANSACTION, 1): {
    "fee": Fee(SIZE_BASED),
    "schema": {
        "version": Fd(1),
        "account_id": Fd(2, _ID),
        "nonce": Fd(3),
        "name_id": Fd(4, _ID),
        "fee": Fd(5),
        "ttl": Fd(6),
    }},
(idf.OBJECT_TAG_CONTRACT_CREATE_TRANSACTION, 1): {
    "fee": Fee(SIZE_BASED, base_gas_multiplier=5),
    "schema": {
        "version": Fd(1),
        "owner_id": Fd(2, _ID),
        "nonce": Fd(3),
        "code": Fd(4, _ENC, prefix=idf.BYTECODE),
        # "vm_version": Fd(5),
        "abi_version": Fd(5, _VM_ABI),
```

(continues on next page)

(continued from previous page)

```
        "fee": Fd(6),
        "ttl": Fd(7),
        "deposit": Fd(8),
        "amount": Fd(9),
        "gas": Fd(10),
        "gas_price": Fd(11),
        "call_data": Fd(12, _ENC, prefix=idf.BYTECODE),
    }},
(idf.OBJECT_TAG_CONTRACT_CALL_TRANSACTION, 1): {
    "fee": Fee(SIZE_BASED, base_gas_multiplier=(
        'abi_version', {
            idf.ABI_FATE: 12,
            idf.ABI_SOPHIA: 30
        })),
    "schema": {
        "version": Fd(1),
        "caller_id": Fd(2, _ID),
        "nonce": Fd(3),
        "contract_id": Fd(4, _ID),
        "abi_version": Fd(5),
        "fee": Fd(6),
        "ttl": Fd(7),
        "amount": Fd(8),
        "gas": Fd(9),
        "gas_price": Fd(10),
        "call_data": Fd(11, _ENC, prefix=idf.BYTECODE),
    }},
(idf.OBJECT_TAG_CHANNEL_CREATE_TRANSACTION, 1): {
    "fee": Fee(SIZE_BASED),
    "schema": {
        "version": Fd(1),
        "initiator": Fd(2, _ID),
        "initiator_amount": Fd(3),
        "responder": Fd(4, _ID),
        "responder_amount": Fd(5),
        "channel_reserve": Fd(6),
        "lock_period": Fd(7),
        "ttl": Fd(8),
        "fee": Fd(9),
        "delegate_ids": Fd(10, _IDS),  # [d(d) for d in tx_native[10]], TODO: are u␣
↪sure
        "state_hash": Fd(11, _ENC, prefix=idf.STATE_HASH),
        "nonce": Fd(12),
    }},
(idf.OBJECT_TAG_CHANNEL_DEPOSIT_TRANSACTION, 1): {
    "fee": Fee(SIZE_BASED),
    "schema": {
        "version": Fd(1),
        "channel_id": Fd(2, _ID),
        "from_id": Fd(3, _ID),
        "amount": Fd(4),
        "ttl": Fd(5),
        "fee": Fd(6),
        "state_hash": Fd(7, _ENC, prefix=idf.STATE_HASH),  # _binary_decode(tx_
↪native[7]),
        "round": Fd(8),
        "nonce": Fd(9),
```

(continues on next page)

(continued from previous page)

```
    }},
(idf.OBJECT_TAG_CHANNEL_WITHDRAW_TRANSACTION, 1): {
    "fee": Fee(SIZE_BASED),
    "schema": {
        "version": Fd(1),
        "channel_id": Fd(2, _ID),
        "to_id": Fd(3, _ID),
        "amount": Fd(4),
        "ttl": Fd(5),
        "fee": Fd(6),
        "state_hash": Fd(7, _ENC, prefix=idf.STATE_HASH),  # _binary_decode(tx_
↪native[7]),
        "round": Fd(8),
        "nonce": Fd(9),
    }},
(idf.OBJECT_TAG_CHANNEL_CLOSE_MUTUAL_TRANSACTION, 1): {
    "fee": Fee(SIZE_BASED),
    "schema": {
        "version": Fd(1),
        "channel_id": Fd(2, _ID),
        "from_id": Fd(3, _ID),
        "initiator_amount_final": Fd(4),
        "responder_amount_final": Fd(5),
        "ttl": Fd(6),
        "fee": Fd(7),
        "nonce": Fd(8),
    }},
(idf.OBJECT_TAG_CHANNEL_CLOSE_SOLO_TRANSACTION, 1): {
    "fee": Fee(SIZE_BASED),
    "schema": {
        "version": Fd(1),
        "channel_id": Fd(2, _ID),
        "from_id": Fd(3, _ID),
        "payload": Fd(4, _ENC, prefix=idf.BYTE_ARRAY),
        "poi": Fd(5, _POI),
        "ttl": Fd(6),
        "fee": Fd(7),
        "nonce": Fd(8),
    }},
(idf.OBJECT_TAG_CHANNEL_SLASH_TRANSACTION, 1): {
    "fee": Fee(SIZE_BASED),
    "schema": {
        "version": Fd(1),
        "channel_id": Fd(2, _ID),
        "from_id": Fd(3, _ID),
        "payload": Fd(4, _ENC, prefix=idf.BYTE_ARRAY),
        "poi": Fd(5, _POI),
        "ttl": Fd(6),
        "fee": Fd(7),
        "nonce": Fd(8),
    }},
(idf.OBJECT_TAG_CHANNEL_SETTLE_TRANSACTION, 1): {
    "fee": Fee(SIZE_BASED),
    "schema": {
        "version": Fd(1),
        "channel_id": Fd(2, _ID),
        "from_id": Fd(3, _ID),
```

(continues on next page)

(continued from previous page)

```
        "initiator_amount_final": Fd(4),
        "responder_amount_final": Fd(5),
        "ttl": Fd(6),
        "fee": Fd(7),
        "nonce": Fd(8),
    }},
(idf.OBJECT_TAG_CHANNEL_SNAPSHOT_TRANSACTION, 1): {
    "fee": Fee(SIZE_BASED),
    "schema": {
        "version": Fd(1),
        "channel_id": Fd(2, _ID),
        "from_id": Fd(3, _ID),
        "payload": Fd(4, _ENC, prefix=idf.BYTE_ARRAY),
        "ttl": Fd(5),
        "fee": Fd(6),
        "nonce": Fd(7),
    }},
(idf.OBJECT_TAG_CHANNEL_FORCE_PROGRESS_TRANSACTION, 1): {
    "fee": Fee(SIZE_BASED),
    "schema": {
        "version": Fd(1),
        "channel_id": Fd(2, _ID),
        "from_id": Fd(3, _ID),
        "payload": Fd(4, _ENC, prefix=idf.BYTE_ARRAY),
        "round": Fd(5),
        "update": Fd(6),  # _binary_decode(tx_native[2]),
        "state_hash": Fd(7, _ENC, prefix=idf.STATE_HASH),  # _binary_decode(tx_
↪native[2]),
        "offchain_trees": Fd(8, _TR),  # TODO: implement support for _trees
        "ttl": Fd(2),
        "fee": Fd(2),
        "nonce": Fd(2),
    }},
(idf.OBJECT_TAG_ORACLE_REGISTER_TRANSACTION, 1): {
    "fee": Fee(TTL_BASED, ttl_field="oracle_ttl_value"),
    "schema": {
        "version": Fd(1),
        "account_id": Fd(2, _ID),
        "nonce": Fd(3),
        "query_format": Fd(4, _BIN),  # _binary_decode(tx_native[4]), TODO: verify␣
↪the type
        "response_format": Fd(5, _BIN),  # _binary_decode(tx_native[5]),
        "query_fee": Fd(6),
        "oracle_ttl_type": Fd(7, _OTTL_TYPE),
        "oracle_ttl_value": Fd(8),
        "fee": Fd(9),
        "ttl": Fd(10),
        "vm_version": Fd(11),
    }},
(idf.OBJECT_TAG_ORACLE_QUERY_TRANSACTION, 1): {
    "fee": Fee(TTL_BASED, ttl_field="query_ttl_value"),
    "schema": {
        "version": Fd(1),
        "sender_id": Fd(2, _ID),
        "nonce": Fd(3),
        "oracle_id": Fd(4, _ID),
        "query": Fd(5, _BIN),
```

(continues on next page)

(continued from previous page)

```
        "query_fee": Fd(6),
        "query_ttl_type": Fd(7, _OTTL_TYPE),
        "query_ttl_value": Fd(8),
        "response_ttl_type": Fd(9, _OTTL_TYPE),
        "response_ttl_value": Fd(10),
        "fee": Fd(11),
        "ttl": Fd(12),
    }},
(idf.OBJECT_TAG_ORACLE_RESPONSE_TRANSACTION, 1): {
    "fee": Fee(TTL_BASED, ttl_field="response_ttl_value"),
    "schema": {
        "version": Fd(1),
        "oracle_id": Fd(2, _ID),
        "nonce": Fd(3),
        "query_id": Fd(4, _ENC, prefix=idf.ORACLE_QUERY_ID),
        "response": Fd(5, _BIN),
        "response_ttl_type": Fd(6, _OTTL_TYPE),
        "response_ttl_value": Fd(7),
        "fee": Fd(8),
        "ttl": Fd(9),
    }},
(idf.OBJECT_TAG_ORACLE_EXTEND_TRANSACTION, 1): {
    "fee": Fee(TTL_BASED, ttl_field="oracle_ttl_value"),
    "schema": {
        "version": Fd(1),
        "oracle_id": Fd(2, _ID),
        "nonce": Fd(3),
        "oracle_ttl_type": Fd(4, _OTTL_TYPE),
        "oracle_ttl_value": Fd(5),
        "fee": Fd(6),
        "ttl": Fd(7),
    }},
(idf.OBJECT_TAG_GA_ATTACH_TRANSACTION, 1): {
    "fee": Fee(SIZE_BASED, base_gas_multiplier=5),
    "schema": {
        "version": Fd(1),
        "owner_id": Fd(2, _ID),
        "nonce": Fd(3),
        "code": Fd(4, _ENC, prefix=idf.BYTECODE),
        "auth_fun": Fd(5, _BIN),
        # "vm_version": Fd(6), # This field is retrieved via abi_version
        "abi_version": Fd(6, _VM_ABI),
        "fee": Fd(7),
        "ttl": Fd(8),
        "gas": Fd(9),
        "gas_price": Fd(10),
        "call_data": Fd(11, _ENC, prefix=idf.BYTECODE),  # _binary_decode(tx_
↪native[11]),
    }},
(idf.OBJECT_TAG_GA_META_TRANSACTION, 1): {
    "fee": Fee(SIZE_BASED, base_gas_multiplier=5),
    "schema": {
        "version": Fd(1),
        "ga_id": Fd(2, _ID),
        "auth_data": Fd(3, _ENC, prefix=idf.BYTECODE),
        "abi_version": Fd(4),
        "fee": Fd(5),
```

(continues on next page)

(continued from previous page)

```
        "gas": Fd(6),
        "gas_price": Fd(7),
        "ttl": Fd(8),
        "tx": Fd(9, _TX),
    }
},
```

4.3 TxBuilder

class aeternity.transactions.**TxBuilder**(*base_gas=15000*, *gas_per_byte=20*, *gas_price=1000000000*, *key_block_interval=3*)

TxBuilder is used to build and post transactions to the chain.

Args: base_gas (int): the base gas value as defined by protocol gas_per_byte (int): the gas per byte as defined by protocol gas_price (int): the gas price as defined by protocol or node configuration key_block_interval (int): the estimated number of minutes between blocks

compute_min_fee(*tx_data: dict*, *tx_descriptor: dict*, *tx_raw: list*) → int

Compute the minimum fee for a transaction

Args: tx_data (dict): the transaction data tx_descriptor (dict): the descriptor of the transaction fields tx_raw: the unencoded rlp data of the transaction

Returns: the minimum fee for the transaction

static compute_tx_hash(*encoded_tx: str*) → str

Generate the hash from a signed and encoded transaction

Args: encoded_tx (str): an encoded signed transaction

parse_node_reply(*tx_data*) → aeternity.transactions.TxObject

Parse a node rest api reply to a transaction object

parse_tx_string(*tx_string*) → aeternity.transactions.TxObject

Parse a transaction string to a transaction object

tx_contract_call(*caller_id*, *contract_id*, *call_data*, *function*, *amount*, *gas*, *gas_price*, *abi_version*, *fee*, *ttl*, *nonce*) → aeternity.transactions.TxObject

Create a Contract Call transaction :param caller_id: the account creating the contract :param contract_id: the contract to call :param call_data: the call data for the contract :param function: the function to execute :param amount: the amount associated to the transaction call :param gas: the gas limit for the execution of the function :param gas_price: the gas unit price :param abi_version: TODO: add definition :param fee: the transaction fee :param ttl: the ttl of the transaction :param nonce: the nonce of the account for the transaction

tx_contract_create(*owner_id*, *code*, *call_data*, *amount*, *deposit*, *gas*, *gas_price*, *vm_version*, *abi_version*, *fee*, *ttl*, *nonce*) → aeternity.transactions.TxObject

Create a contract transaction :param owner_id: the account creating the contract :param code: the binary code of the contract :param call_data: the call data for the contract :param amount: initial amount(balance) of the contract :param deposit: the deposit bound to the contract :param gas: the gas limit for the execution of the limit function :param gas_price: the gas price for the unit of gas :param vm_version: the vm version of the contract :param abi_version: TODO: add definition :param fee: the transaction fee :param ttl: the ttl of the transaction :param nonce: the nonce of the account for the transaction

tx_ga_attach(*owner_id*, *nonce*, *code*, *auth_fun*, *vm_version*, *abi_version*, *fee*, *ttl*, *gas*, *gas_price*, *call_data*) → aeternity.transactions.TxObject

Parameters

- **owner_id** – the owner of the contra
- **nonce** – the transaction nonce
- **code** – the bytecode of for the generalized account contract
- **auth_fun** – the hash of the authorization function of the ga contract
- **vm_version** – the vm version of the contract
- **abi_version** – TODO: add definition
- **fee** – the fee for the transaction
- **ttl** – the ttl for the transaction

tx_ga_meta(*ga_id*, *auth_data*, *abi_version*, *fee*, *gas*, *gas_price*, *ttl*, *tx*) → aeternity.transactions.TxObject

Parameters

- **ga_id** – the account id
- **auth_data** – the authorized data
- **abi_version** – compiler abi version
- **fee** – transaction fee
- **gas** – gas limit for the authorization function
- **gas_price** – the gas prize
- **ttl** – time to live (in height) of the transaction
- **tx** – the transaction to be authorized

tx_name_claim(*account_id*, *name*, *name_salt*, *fee*, *ttl*, *nonce*) → aeternity.transactions.TxObject

create a preclaim transaction :param account_id: the account registering the name :param name: the actual name to claim :param name_salt: the salt used to create the commitment_id during preclaim :param fee: the fee for the transaction :param ttl: the ttl for the transaction :param nonce: the nonce of the account for the transaction

tx_name_claim_v2(*account_id*, *name*, *name_salt*, *name_fee*, *fee*, *ttl*, *nonce*) → aeternity.transactions.TxObject

create a preclaim transaction :param account_id: the account registering the name :param name: the actual name to claim :param name_salt: the salt used to create the commitment_id during preclaim :param name_fee: the fee bid to claim the name :param fee: the fee for the transaction :param ttl: the ttl for the transaction :param nonce: the nonce of the account for the transaction

tx_name_preclaim(*account_id*, *commitment_id*, *fee*, *ttl*, *nonce*) → aeternity.transactions.TxObject

create a preclaim transaction :param account_id: the account registering the name :param commitment_id: the commitment id :param fee: the fee for the transaction :param ttl: the ttl for the transaction :param nonce: the nonce of the account for the transaction

tx_name_revoke(*account_id*, *name_id*, *fee*, *ttl*, *nonce*) → aeternity.transactions.TxObject

create a revoke transaction :param account_id: the account revoking the name :param name_id: the name to revoke :param fee: the transaction fee :param ttl: the ttl of the transaction :param nonce: the nonce of the account for the transaction

tx_name_transfer(*account_id*, *name_id*, *recipient_id*, *fee*, *ttl*, *nonce*) → aeternity.transactions.TxObject

create a transfer transaction :param account_id: the account transferring the name :param name_id: the name to transfer :param recipient_id: the address of the account to transfer the name to :param fee: the

transaction fee :param ttl: the ttl of the transaction :param nonce: the nonce of the account for the transaction

tx_name_update(*account_id*, *name_id*, *pointers*, *name_ttl*, *client_ttl*, *fee*, *ttl*, *nonce*) → aeternity.transactions.TxObject
create an update transaction :param account_id: the account updating the name :param name_id: the name id :param pointers: the pointers to update to :param name_ttl: the ttl for the name registration :param client_ttl: the ttl for client to cache the name :param fee: the transaction fee :param ttl: the ttl of the transaction :param nonce: the nonce of the account for the transaction

tx_oracle_extend(*oracle_id*, *ttl_type*, *ttl_value*, *fee*, *ttl*, *nonce*) → aeternity.transactions.TxObject
Create a oracle extends transaction

tx_oracle_query(*oracle_id*, *sender_id*, *query*, *query_fee*, *query_ttl_type*, *query_ttl_value*, *response_ttl_type*, *response_ttl_value*, *fee*, *ttl*, *nonce*) → aeternity.transactions.TxObject
Create a oracle query transaction

tx_oracle_register(*account_id*, *query_format*, *response_format*, *query_fee*, *ttl_type*, *ttl_value*, *vm_version*, *fee*, *ttl*, *nonce*) → aeternity.transactions.TxObject
Create a register oracle transaction

tx_oracle_respond(*oracle_id*, *query_id*, *response*, *response_ttl_type*, *response_ttl_value*, *fee*, *ttl*, *nonce*) → aeternity.transactions.TxObject
Create a oracle response transaction

tx_signed(*signatures: list*, *tx: aeternity.transactions.TxObject*, *metadata={}*) → aeternity.transactions.TxObject
Create a signed transaction. This is a special type of transaction as it wraps a normal transaction adding one or more signature.

Args: signatures: the signatures to add to the transaction tx (TxObject): the enclosed transaction metadata: additional metadata to be saved in the TxObject

Returns: the TxObject representing the signed transaction

tx_spend(*sender_id*, *recipient_id*, *amount*, *payload*, *fee*, *ttl*, *nonce*) → aeternity.transactions.TxObject
create a spend transaction :param sender_id: the public key of the sender :param recipient_id: the public key of the recipient :param amount: the amount to send :param payload: the payload associated with the data :param fee: the fee for the transaction :param ttl: the absolute ttl of the transaction :param nonce: the nonce of the transaction

4.4 TxSigner

class aeternity.transactions.**TxSigner**(*account*, *network_id*)
TxSigner is used to compute the signature for transactions

Args: account (Account): the account that will be signing the transactions network_id (str): the network id of the target network

sign_transaction(*transaction: aeternity.transactions.TxObject*, *metadata: dict = None*) → str
Sign, encode and compute the hash of a transaction

Args: tx (TxObject): the TxObject to be signed

Returns: the encoded and prefixed signature

4.5 Constants

The following are a list of constants used throughout the SDK

```
# fee calculation
BASE_GAS = 15000
GAS_PER_BYTE = 20
GAS_PRICE = 1000000000
# default relative ttl in number of blocks  for executing transaction on the chain
MAX_TX_TTL = 256
TX_TTL = 0
FEE = 0
# contracts
CONTRACT_GAS = 10000
CONTRACT_GAS_PRICE = 1000000000
CONTRACT_DEPOSIT = 0
CONTRACT_AMOUNT = 0
# https://github.com/aeternity/protocol/blob/master/oracles/oracles.md#technical-
↪aspects-of-oracle-operations
ORACLE_VM_VERSION = identifiers.NO_VM
ORACLE_TTL_TYPE = identifiers.ORACLE_TTL_TYPE_DELTA
ORACLE_QUERY_FEE = 0
ORACLE_TTL_VALUE = 500
ORACLE_QUERY_TTL_VALUE = 10
ORACLE_RESPONSE_TTL_VALUE = 10
# Chain
KEY_BLOCK_INTERVAL = 3  # average time between key-blocks in minutes
KEY_BLOCK_CONFIRMATION_NUM = 3  # number of key blocks to wait for to consider a key-
↪block confirmed
# network id
NETWORK_ID = identifiers.NETWORK_ID_MAINNET
# TUNING
POLL_TX_MAX_RETRIES = 8  # used in exponential backoff when checking a transaction
POLL_TX_RETRIES_INTERVAL = 2  # in seconds
POLL_BLOCK_MAX_RETRIES = 20  # number of retries
POLL_BLOCK_RETRIES_INTERVAL = 30  # in seconds
# channels
CHANNEL_ENDPOINT = 'channel'
CHANNEL_URL = 'ws://127.0.0.1:3014'
# Generalized accounts
GA_AUTH_FUNCTION = "authorize"
GA_MAX_AUTH_FUN_GAS = 50000
GA_ACCOUNTS_NONCE = 0  # for tx in ga transactions the nonce must be 0
# Aens
# max number of block into the future that the name is going to be available
# https://github.com/aeternity/protocol/blob/master/AENS.md#aens-entry
NAME_MAX_TTL = 50000  # in blocks
NAME_MAX_CLIENT_TTL = 84600  # in seconds
NAME_FEE = 0
# see https://github.com/aeternity/aeternity/blob/
↪72e440b8731422e335f879a31ecbbee7ac23a1cf/apps/aecore/src/aec_governance.erl#L67
NAME_FEE_MULTIPLIER = 100000000000000
NAME_FEE_BID_INCREMENT = 0.05  # the increment is in percentage
# see https://github.com/aeternity/aeternity/blob/
↪72e440b8731422e335f879a31ecbbee7ac23a1cf/apps/aecore/src/aec_governance.erl#L272
NAME_BID_TIMEOUT_BLOCKS = 480  # ~1 day
NAME_BID_MAX_LENGTH = 12  # this is the max length for a domain to be part of a bid
```

(continues on next page)

(continued from previous page)

```
# ref: https://github.com/aeternity/aeternity/blob/
↪72e440b8731422e335f879a31ecbbee7ac23a1cf/apps/aecore/src/aec_governance.erl#L290
# bid ranges:
NAME_BID_RANGES = {
    31: 3 * NAME_FEE_MULTIPLIER,
    30: 5 * NAME_FEE_MULTIPLIER,
    29: 8 * NAME_FEE_MULTIPLIER,
    28: 13 * NAME_FEE_MULTIPLIER,
    27: 21 * NAME_FEE_MULTIPLIER,
    26: 34 * NAME_FEE_MULTIPLIER,
    25: 55 * NAME_FEE_MULTIPLIER,
    24: 89 * NAME_FEE_MULTIPLIER,
    23: 144 * NAME_FEE_MULTIPLIER,
    22: 233 * NAME_FEE_MULTIPLIER,
    21: 377 * NAME_FEE_MULTIPLIER,
    20: 610 * NAME_FEE_MULTIPLIER,
    19: 987 * NAME_FEE_MULTIPLIER,
    18: 1597 * NAME_FEE_MULTIPLIER,
    17: 2584 * NAME_FEE_MULTIPLIER,
    16: 4181 * NAME_FEE_MULTIPLIER,
    15: 6765 * NAME_FEE_MULTIPLIER,
    14: 10946 * NAME_FEE_MULTIPLIER,
    13: 17711 * NAME_FEE_MULTIPLIER,
    12: 28657 * NAME_FEE_MULTIPLIER,
    11: 46368 * NAME_FEE_MULTIPLIER,
    10: 75025 * NAME_FEE_MULTIPLIER,
    9: 121393 * NAME_FEE_MULTIPLIER,
    8: 196418 * NAME_FEE_MULTIPLIER,
    7: 317811 * NAME_FEE_MULTIPLIER,
    6: 514229 * NAME_FEE_MULTIPLIER,
    5: 832040 * NAME_FEE_MULTIPLIER,
    4: 1346269 * NAME_FEE_MULTIPLIER,
    3: 2178309 * NAME_FEE_MULTIPLIER,
    2: 3524578 * NAME_FEE_MULTIPLIER,
    1: 5702887 * NAME_FEE_MULTIPLIER,
}

# ref: https://github.com/aeternity/aeternity/blob/
↪72e440b8731422e335f879a31ecbbee7ac23a1cf/apps/aecore/src/aec_governance.erl#L273
# name bid timeouts
NAME_BID_TIMEOUTS = {
    13: 0,
    8: 1 * NAME_BID_TIMEOUT_BLOCKS,  # 480 blocks
    4: 31 * NAME_BID_TIMEOUT_BLOCKS,  # 14880 blocks
    1: 62 * NAME_BID_TIMEOUT_BLOCKS,  # 29760 blocks
}

# dry run
DRY_RUN_ADDRESS = "ak_11111111111111111111111111111273Yts"
DRY_RUN_AMOUNT = 100000000000000000000000000000000
```

CHAPTER

FIVE

FAQ

TODO

CHAPTER

SIX

CODE SNIPPETS

This is a collection of code snippets and scripts that may be used for copy/paste or quick references.

6.1 Top up account from the Faucet

Code to programmatically top-up an account using the Faucet

```
def top_up_account(account_address):

  print(f"top up account {account_address} using the testnet.faucet.aepps.com app")
  r = requests.post(f"https://testnet.faucet.aepps.com/account/{account_address}").
↪json()
  tx_hash = r.get("tx_hash")
  balance = utils.format_amount(r.get("balance"))
  print(f"account {account_address} has now a balance of {balance}")
  print(f"faucet transaction hash {tx_hash}")
```

6.2 Generate multiple accounts

The following is a command line tool to generate multiple accounts and to export the accounts secret/public keys. Useful for testing

```
#!/usr/bin/env python
import argparse
import json
from aeternity.signing import Account

"""
Example app to deal with common dev issues:
- export secret/public key from a keystore
- generate a number of accounts to be used
"""

# max number of account to generate
MAX_N_ACCOUNTS = 1000

def cmd_export(args):
    try:
        a = Account.from_keystore(args.keystore_path, args.password)
        print(json.dumps(
```

(continues on next page)

(continued from previous page)

```
            {
                "keystore": args.keystore_path,
                "secret_key": a.get_secret_key(),
                "address": a.get_address()
            }, indent=2))
    except Exception as e:
        print(f"Invalid keystore or password: {e}")

def cmd_generate(args):
    try:
        if args.n > MAX_N_ACCOUNTS:
            print(f"Max number of accounts to generate is {MAX_N_ACCOUNTS},␣
↪requested: {args.n}")
        accounts = []
        for i in range(args.n):
            a = Account.generate()
            accounts.append({
                "index": i,
                "secret_key": a.get_secret_key(),
                "address": a.get_address()
            })
        print(json.dumps(accounts, indent=2))
    except Exception as e:
        print(f"Generation error: {e}")


if __name__ == "__main__":
    commands = [
        {
            'name': 'export',
            'help': 'export the secret/public key of a encrypted keystore as plain␣
↪text WARNING! THIS IS UNSAFE, USE FOR DEV ONLY',
            'target': cmd_export,
            'opts': [
                {
                    "names": ["keystore_path"],
                    "help": "the keystore to use export",
                },
                {
                    "names": ["-p", "--password"],
                    "help": "the keystore password (default blank)",
                    "default": ""
                }
            ]
        },
        {
            'name': 'generate',
            'help': 'generate one or more accounts and print them on the stdout',
            'target': cmd_generate,
            'opts': [
                {
                    "names": ["-n"],
                    "help": "number of accounts to generate (default 10)",
                    "default": 10,
                }
            ]
        }
```

(continues on next page)

(continued from previous page)

```
    ]
    parser = argparse.ArgumentParser()
    subparsers = parser.add_subparsers()
    subparsers.required = True
    subparsers.dest = 'command'
    # register all the commands
    for c in commands:
        subparser = subparsers.add_parser(c['name'], help=c['help'])
        subparser.set_defaults(func=c['target'])
        # add the sub arguments
        for sa in c.get('opts', []):
            subparser.add_argument(*sa['names'],
                                   help=sa['help'],
                                   action=sa.get('action'),
                                   default=sa.get('default'))

    # parse the arguments
    args = parser.parse_args()
    # call the function
    args.func(args)
```

CHAPTER

SEVEN

HOW THE DOCUMENTATION IS ORGANIZED

A high-level overview of how it's organized will help you know where to look for certain things:

- *Tutorials* take you by the hand through a series of steps to create a python aepp. Start here if you're new to Aeternity application development. Also look at the "*First steps*" below.
- *Topic guides* discuss key topics and concepts at a fairly high level and provide useful background information and explanation.
- *Reference guides* contain technical reference for APIs and other aspects of Aeternity SDK machinery. They describe how it works and how to use it but assume that you have a basic understanding of key concepts.
- *How-to guides* are recipes. They guide you through the steps involved in addressing key problems and use-cases. They are more advanced than tutorials and assume some knowledge of how the Aeternity SDK works.
- *Code snippets* are various pieces of codes to copy/paste. They may be useful to find pieces of codes to get stuff done.

CHAPTER

EIGHT

FIRST STEPS

Are you new to the Aeternity SDK? This is the place to start!

- **From scratch:**
  - *Installation*
- **Tutorial:**
  - *Spend transactions*
  - *Deploy a contract*
  - *Call a contract*
  - *Claiming a name*
  - *Using Generalized accounts*
  - *The CLI*
  - *Generating and using a HD Wallet*
- **How to:**
  - Coming soon...
- **Reference:**
  - *The NodeClient and Config*
  - *The TxObject*

CHAPTER

NINE

GETTING HELP

Having trouble? We'd like to help!

- Try the *FAQ* – it's got answers to many common questions.
- Looking for specific information? Try the genindex or modindex
- Search for information in the Aeternity forum, or post a question.
- Report bugs with the Python SDK in our issue tracker.

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

A

B

C

D

G

M

N

P

S

T

V

W