pyledger Documentation

Release 0.5

Guillem Borrell

Apr 10, 2018

Contents:

1 How to create a smart contract **3**
- 1.1 Attributes . . . 4
- 1.2 Exceptions . . . 5
- 1.3 Docstrings of the classes cited in this section . . . 6

2 Options for running a ledger server **7**
- 2.1 Docstrings cited in this section . . . 7

3 The status chain **9**

4 Users and permissions **11**

5 Example. A digital currency with permissions **13**

6 Distributed ledger with the RAFT consensus protocol **15**

7 Indices and tables **17**

A simple ledger for smart contracts written in Python

Assume that you want to create a smart contract to implement a digital currency system. You have some features you consider necessary, namely creating accounts, adding currency to any account, checking the balance and transfer some amount.

A smart contract is an application, so you need to code to create one. In Pyledger you can implement your smart contract in Python. In a few words, a smart contract in Pyledger is a Python class

```
from pyledger.server.contract import SimpleContract

class DigitalCurrency(SimpleContract):
    accounts = {}

    def add_account(self, key: str):
        if key in self.accounts:
            raise Exception('Account already exists')

        self.accounts[key] = 0.0
        return key

    def increment(self, key: str, quantity: float):
        if key not in self.accounts:
            raise Exception('Account not found')

        self.accounts[key] += quantity

    def transfer(self, source: str, dest: str, quantity: float):
        if source not in self.accounts:
            raise Exception('Source account not found')
        if dest not in self.accounts:
            raise Exception('Destination account not found')
        if self.accounts[source] < quantity:
            raise Exception('Not enough funds in source account')
        if quantity < 0:
            raise Exception('You cannot transfer negative currency')

        self.accounts[source] -= quantity
        self.accounts[dest] += quantity

    def balance(self, key: str):
        if key not in self.accounts:
            print(self.accounts)
            raise Exception('Account not found')

        return str(self.accounts[key])
```

There is no need to deal with the details now, but if you are familiar with Python you more or less understand where the thing is going. Once you have finished creating your smart contract, PyLedger can get it up and running in no time.

```
from pyledger.server import run

run(DigitalCurrency)
```

Assume that the previous script is called *ledger.py*. Running the ledger is as simple as running the script with some options:

```
$> python ledger.py --sync
```

Now you have your ledger up and running, you can connect to it with a REPL client:

```
$> pyledger-shell

Connected to server: tcp:127.0.0.1:9000
Pyledger REPL client, write 'help' for help or 'help command' for help on a specific␣
↪command
PL >>> help

The Pyledger REPL is a console to interact with a Pyledger server.
The list of available commands is the following

 help          Shows this help
 disconnect    Disconnects from the server in a clean way.
 contracts     Lists the available contracts in the server
 api           Shows the api for a particular contract
 call          Calls a method of a contract
 broadcast     Broadcast message all clients

This client may have some limitations respect to a custom client.
For instance, the server may push notifications to the clients,
and using the client API, you could define callbacks to those
pushed messages.

Read the full documentation in http://pyledger.readthedocs.io

PL >>> contracts
['DigitalCurrency']
PL >>> api DigitalCurrency
{'add_account': {'key': <class 'str'>},
 'balance': {'key': <class 'str'>},
 'increment': {'key': <class 'str'>, 'quantity': <class 'float'>},
 'transfer': {'dest': <class 'str'>,
              'quantity': <class 'float'>,
              'source': <class 'str'>}}
PL >>> call DigitalCurrency add_account account1
Call with pairs of key value arguments
PL >>> call DigitalCurrency add_account key account1
'account1'
PL >>> call DigitalCurrency increment key account1 quantity 100.0
None
PL >>> call DigitalCurrency balance key account1
'100.0'
PL >>> call DigitalCurrency add_account key account2
'account2'
PL >>> call DigitalCurrency transfer source account1 dest account2 quantity 50.0
None
PL >>> call DigitalCurrency balance key account1
'50.0'
PL >>> call DigitalCurrency balance key account2
'50.0'
PL >>> disconnect
Successfully closed, you can kill this with Ctrl-C
WebSocket connection closed: 1000; None
^CBye
```

Pyledger is possible thanks to Autobahn

Now that we may have your attention, the actual docs.

CHAPTER 1

How to create a smart contract

A smart contract in pyledger is a function that returns an instance of `pyledger.contract.Builder`. This object is a helper to manage the attributes of the smart contract and the methods that may or may not modify those attributes. The simplest smart contract you may think of is one that just returns the string "Hello".

```
from pyledger.handlers import make_tornado
from pyledger.contract import Builder
from pyledger.config import args
import tornado.ioloop


def hello():
    def say_hello(attrs):
        return attrs, 'Hello'

    contract = Builder('Hello')
    contract.add_method(say_hello)

    return contract


if __name__ == '__main__':
    application = make_tornado(hello)
    application.listen(args.port)
    tornado.ioloop.IOLoop.instance().start()
```

If you run this snippet as script without options, you will be able to connect to this server with the command line client provided by pyledger, called `pyledger-shell`:

```
(env) $> pyledger-shell
PyLedger simple client
(http://localhost:8888)> contracts
     Hello
(http://localhost:8888)> api Hello
   say_hello (  )
```

```
(http://localhost:8888)> call Hello say_hello
Hello
(http://localhost:8888)>
```

This almost trival example is useful to understand the very basics about how the contracts are created. The contract is called *Hello* which is the argument of the Builder instance. The method *say_hello* gets no arguments and it modifies no attributes, but it must get the attributes as an argument and return them anyways. If an additional argument, like the `Hello` string, is returned by the method, it is given as a second return argument.

1.1 Attributes

Let's change the previous example a little by adding an attribute to the contract. For instance, we will make a counter of the amount of times the contract has greeted us.

```
def hello():
    def say_hello(attrs):
        attrs.counter += 1
        return attrs, 'Hello {}'.format(attrs.counter)

    contract = Builder('Hello')
    contract.add_attribute('counter', 0)
    contract.add_method(say_hello)

    return contract
```

A session with this new smart contract would be as follows:

```
(http://localhost:8888)> call Hello say_hello
Hello 1
(http://localhost:8888)> call Hello say_hello
Hello 2
(http://localhost:8888)> status Hello
{'counter': 2}
```

Note that the contract function pretty much looks like an object, it has attributes and methods that change those attributes. It is also quite similar as how Solidity defines the smart contracts, with attributes and methods that modify them. Pyledger is a little more explicit.

We can also define methods with arguments, and here's one of the important particularities of pyledger: *all the arguments but the first one (attrs) must be type annotated*. For instance, this is a contract that greets with a name, that is passed as a parameter.

```
def hello():
    def say_hello(attrs, name: str):
        attrs.counter += 1
        return attrs, 'Hello {} for time #{}'.format(name, attrs.counter)

    contract = Builder('Hello')
    contract.add_attribute('counter', 0)
    contract.add_method(say_hello)

    return contract
```

A smart contract must expose an API, and type annotation is needed to let the client and any user of the contract to know which type the arguments must be:

```
(env) $> pyledger-shell
PyLedger simple client
(http://localhost:8888)> api Hello
   say_hello ( name [str] )

(http://localhost:8888)> call Hello say_hello Guillem
Hello Guillem for time #1
(http://localhost:8888)> call Hello say_hello Guillem
Hello Guillem for time #2
(http://localhost:8888)> status Hello
{'counter': 2}
(http://localhost:8888)>
```

With these features, the smart contracts can be as complex as needed. One can store information of any kind within the arguments, that are the ones that define the status of the contract.

Important: If you want the contract to be fast and you want to avoid obscure bugs too, keep your attributes as primitive python types.

1.2 Exceptions

Contracts can raise only a generic exception of type `Exception`. The goal is only to inform the user that the operation has not been successful. Note that the methods that return no additional value send back to the client the string *SUCCESS*. This means that the client is always waiting for a message to come.

We will introduce some very simple exception that checks the most common mispelling of my name

```
def hello():
    def say_hello(attrs, name: str):
        if name == 'Guillen':
            raise Exception('You probably mispelled Guillem')

        attrs.counter += 1
        return attrs, 'Hello {} for time #{}'.format(name, attrs.counter)

    contract = Builder('Hello')
    contract.add_attribute('counter', 0)
    contract.add_method(say_hello)

    return contract
```

And how the exception is handled at the client side:

```
(env) $> pyledger-shell
PyLedger simple client
(http://localhost:8888)> call Hello say_hello Guillem
Hello Guillem for time #1
(http://localhost:8888)> call Hello say_hello Guillen
You probably mispelled Guillem
(http://localhost:8888)> call Hello say_hello Guillem
Hello Guillem for time #2
```

1.3 Docstrings of the classes cited in this section

CHAPTER 2

Options for running a ledger server

When a server is built using :py:

You can take a look at SQLAlchemy's engine configuration session to see how this option should be formatted.

2.1 Docstrings cited in this section

CHAPTER 3

The status chain

Pyledger does not use a blockchain or any similar protocol because it would be very inefficient for a tool that is not distributed. The internal storage for every contract is not divided in blocks, since each state is stored as a register in a SQL database.

One of the important features of the blockchain is that it is impossible for anyone, even the owner of the data, to tamper with its contents. Pyledger also has this feature, but in a slightly different fashion. All the statuses stored in the ledger for every contract are hashed with the the previous state's hash and the date and time of insertion. It is therefore impossible to modify a register of the database without leaving an obvious footprint on the sequence of hashes. This is a kind of *status chain* instead of a block chain.

All the hashes are secure, since pyledger uses SHA-256, the same as in Bitcoin. This means that one can verify that anyone hasn't been tampering with the backend database that stores the statuses. We can use the *verify* command of the client to check that the greeter smart contract works as expected. We will start an example session to understand some of the features of this *status chain* with one of the previous examples:

```
(env) $> pyledger-shell
PyLedger simple client
(http://localhost:8888)> call Hello say_hello Guillem
Hello Guillem for time #1
(http://localhost:8888)> call Hello say_hello Guillem
Hello Guillem for time #2
(http://localhost:8888)> call Hello say_hello Guillem
Hello Guillem for time #3
(http://localhost:8888)> status Hello
{'counter': 3}
(http://localhost:8888)> verify Hello
'Contract OK'
```

The *status* command checks and prints the last status of the contract attributes, while the *verify* command verifies **at the server side** that all the statuses of the attributes are consistent. If any of the statuses is inconsistent with the chain, that inconsistency and its timestamp will be printed.

Of course, you may not trust the server-side operations on the *status chain*, which is quite smart. For that reason you can dump all the statuses of the contract with their corresponding signatures and timestamps with the following command:

```
(http://localhost:8888)> status Hello dump ./hello-ledger.json
Contract data dump at ./hello-ledger.json
(http://localhost:8888)> exit
```

The dumped file looks like this:

```
[{'attributes': 'gAN9cQBYBwAAAGNvdW50ZXJxAUsAcy4=',
  'hash': 'Z2VuZXNpcw==',
  'when': '2017-03-15T18:24:27.523828'},
 {'attributes': 'gAN9cQBYBwAAAGNvdW50ZXJxAUsBcy4=',
  'hash': 'eRs+YxhvKIyUdl++TQZ5sCcMDE0aoaNKn1swFQ44bMM=',
  'when': '2017-03-15T18:24:38.846864'},
 {'attributes': 'gAN9cQBYBwAAAGNvdW50ZXJxAUsCcy4=',
  'hash': 'ZGELWR6y7n+hneBbR+8x9PwaRpBi3Bi0CI/T+9J7ccY=',
  'when': '2017-03-15T18:24:39.580593'},
 {'attributes': 'gAN9cQBYBwAAAGNvdW50ZXJxAUsDcy4=',
  'hash': '7B+OH/4xxJz6J6NOixl32F1vXrWZFNQKMR7pe/HO7gY=',
  'when': '2017-03-15T18:24:39.925244'}]
```

Every state has three properties. The first one is the hash, which is a base64-encoded SHA-526 hash; the second one is the timestamp of the addition to the database in the ISO 8601 format, while the third are all the attributes of the contract, also properly serialized.

Note: If you want to deserialize the attributes to look inside that funny string, they are pickled and then base64 encoded

---

If you want to verify the dumped statuses of the contract you can use the utility *pyledger-verify*:

```
$> pyledger-verify --data hello-ledger.json
...
DONE
```

where every dot is one successfully verified step.

If you tamper with this file, or the database that stores the information, even changing a single bit, the status chain will inform you of the inconsistency giving its timestamp.

```
 [{'attributes': 'gAN9cQBYBwAAAGNvdW50ZXJxAUsAcy4=',
   'hash': 'Z2VuZXNpcw==',
   'when': '2017-03-15T18:24:27.523828'},
  {'attributes': 'gAN9cQBYBwAABGNvdW50ZXJxAUsBcy4=',
   'hash': 'eRs+YxhvKIyUdl++TQZ5sCcMDE0aoaNKn1swFQ44bMM=',
   'when': '2017-03-15T18:24:38.846864'},
  {'attributes': 'gAN9cQBYBwAAAGNvdW50ZXJxAUsCcy4=',
   'hash': 'ZGELWR6y7n+hneBbR+8x9PwaRpBi3Bi0CI/T+9J7ccY=',
   'when': '2017-03-15T18:24:39.580593'},
  {'attributes': 'gAN9cQBYBwAAAGNvdW50ZXJxAUsDcy4=',
   'hash': '7B+OH/4xxJz6J6NOixl32F1vXrWZFNQKMR7pe/HO7gY=',
   'when': '2017-03-15T18:24:39.925244'}]
```

This is the output of the pyledger-verify tool with the manipulated file:

```
$> pyledger-verify --data hello-ledger.json
 Inconsistency 2017-03-15T18:24:38.846864..
DONE
```

CHAPTER 4

Users and permissions

Pyledger supports basic key-based authentication for clients, and the contracts may be aware if the user was previously created by the administrator of the ledger. When you run the server for the first time, the ledger server outputs an admin authentication key, that is stored within the ledger itself:

```
$> python examples/authentication/server.py --sync
Warning: Syncing database at sqlite://
Warning: Admin key is a1ee413e-0505-49a6-9902-748e87741225
```

If you start a client with this key, it will have admin privileges.

One of the important aspects of admin privileges is the key creation, which is equivalent of creating a user, since each user is identified by a random key:

```
$> pyledger-shell --user a1ee413e-0505-49a6-9902-748e87741225
PyLedger simple client
(http://localhost:8888)> key NewUser
Created user Guillem: 79ab6f2d-5fe6-4bf8-9ebd-ee359d9dfa94
(http://localhost:8888)> exit
```

This key can be used to authenticate the user, and we can make the contract aware of the authentication of a client.

```
def hello():
    def say_hello(attrs):
        if attrs.user:
            return attrs, 'Hello {}, your key is {}'.format(attrs.user.name,
                                                            attrs.user.key)
        else:
            raise Exception('Not authenticated')

    contract = Builder('Hello')
    contract.add_method(say_hello)

    return contract
```

The attrs object contains a copy of the data stored about the user, like its name or the user key. If the user was not authenticated, `attrs.user` is set as `None`.

We can now start the client with the new user key:

```
$> pyledger-shell --user 79ab6f2d-5fe6-4bf8-9ebd-ee359d9dfa94
(http://localhost:8888)> api Hello
   say_hello (  )

(http://localhost:8888)> call Hello say_hello
Hello Guillem, your key is 79ab6f2d-5fe6-4bf8-9ebd-ee359d9dfa94
(http://localhost:8888)> exit
```

Important: There is only one user called `admin`, that is assigned the key that is printed when the ledger is started for the first time with the `--sync` option. This means that, `if attrs.user.name == 'admin'` checks if the current user is in fact the owner of the ledger.

CHAPTER 5

Example. A digital currency with permissions

This example makes use of all the features that have been commented so far. It implements a more or less correct cryptocurrency.

CHAPTER 6

Distributed ledger with the RAFT consensus protocol

While having a unique and centralized database allows pyledger to significantly simplify the ledger infrastructure, it becomes a single point of failure. However, since the database is a pluggable component in pyledger, you can turn pyledger into a distributed ledger using a distributed database.

One interesting choice is rqlite, a distributed and relational database built on SQLite where all the nodes reach a consensus based on the RAFT protocol.

To integrate rqlite with pyledger you must install the packages sqlalchemy_rqlite and pyrqlite, and run pyledger with the following arguments:

```
python examples/hello/server.py --db rqlite+pyrqlite://localhost:4001 --sync
```

CHAPTER 7

Indices and tables

- genindex
- modindex
- search