# BigchainDB Python Driver Documentation

***Release 0.1.3***

**BigchainDB**

December 07, 2016

Contents

**Important: Development Status: Alpha**

CHAPTER 1

# BigchainDB Python Driver

- Free software: Apache Software License 2.0
- Documentation: https://docs.bigchaindb.com/projects/py-driver/

## 1.1 Features

- Support for preparing, fulfilling, and sending transactions to a BigchainDB node.
- Retrieval of transactions by id.
- Getting status of a transaction by id.

## 1.2 Compatibility Matrix

| BigchainDB Server | BigchainDB Driver |
|---|---|
| `0.8.x` | `0.1.x` |

*Although we do our best to keep the master branches in sync, there may be occasional delays.*

## 1.3 Credits

This package was initially created using Cookiecutter and the audreyr/cookiecutter-pypackage project template. Many BigchainDB developers have contributed since then.

CHAPTER 2

# Quickstart / Installation

The BigchainDB Python Driver depends on:

1. `libffi/ffi.h`
2. Python 3.5+
3. A recent Python 3 version of `pip`
4. A recent Python 3 version of `setuptools`

If you're missing one of those, then see below. Otherwise, you can install the BigchainDB Python Driver (`bigchaindb_driver`) using:

```
pip install bigchaindb_driver
```

and then you can try the Basic Usage Examples.

## 2.1 How to Install the Dependencies

### 2.1.1 Dependency 1: ffi.h

BigchainDB (server and driver) depends on cryptoconditions, which depends on PyNaCl (Networking and Cryptography library), which depends on `ffi.h`. Hence, depending on your setup, you may need to install the development files for `libffi`.

On Ubuntu 14.04 and 16.04, this works:

```
sudo apt-get update

sudo apt-get install libffi-dev
```

On Fedora 23 and 24, this works:

```
sudo dnf update

sudo dnf install libffi-devel
```

For other operating systems, just do some web searches for "ffi.h" with the name of your OS.

### 2.1.2 Dependency 2: Python 3.5+

The BigchainDB Python Driver uses Python 3.5+. You can check your version of Python using:

```
python --version
```

An easy way to install a specific version of Python, and to switch between versions of Python, is to use virtualenv. Another option is conda.

### 2.1.3 Dependency 3: pip

You also need to get a recent, Python 3 version of `pip`, the Python package manager.

If you're using virtualenv or conda, then each virtual environment should include an appropriate version of `pip`.

You can check your version of `pip` using:

```
pip --version
```

`pip` was at version 9.0.0 as of November 2016. If you need to upgrade your version of `pip`, then see the pip documentation or our page about that in the BigchainDB Server docs.

### 2.1.4 Dependency 4: setuptools

Once you have a recent Python 3 version of `pip`, you should be able to upgrade `setuptools` using:

```
pip install --upgrade setuptools
```

## 2.2 Installing from the Source Code

The source code for the BigchainDB Python Driver can be downloaded from the Github repo. You can either clone the public repository:

```
git clone git://github.com/bigchaindb/bigchaindb-driver
```

Or download the tarball:

```
curl  -OL https://github.com/bigchaindb/bigchaindb-driver/tarball/master
```

Once you have a copy of the source code, you can install it by going to the directory containing `setup.py` and doing:

```
python setup.py install
```

## 2.3 Installing latest master with pip

In order to work with the latest BigchainDB (server) master branch:

```
$ pip install --process-dependency-links git+https://github.com/bigchaindb/bigchaindb-driver.git
```

Point to some BigchainDB node, which is running BigchainDB server `master`:

```
from bigchaindb_driver import BigchainDB

bdb = BigchainDB('http://here.be.dragons:9984/api/v1')
```

CHAPTER 3

# Basic Usage Examples

**Note:** You must install the bigchaindb_driver Python package first.

You should use Python 3 for these examples.

The BigchainDB driver's main purpose is to connect to one or more BigchainDB server nodes, in order to perform supported API calls documented under `drivers-clients/http-client-server-api`.

Connecting to a BigchainDB node, is done via the *`BigchainDB class`*:

```
from bigchaindb_driver import BigchainDB

bdb = BigchainDB('<api_endpoint>')
```

where `<api_endpoint>` is the root URL of the BigchainDB server API you wish to connect to.

For the simplest case in which a BigchainDB node would be running locally, (and the `BIGCHAINDB_SERVER_BIND` setting wouldn't have been changed), you would connect to the local BigchainDB server this way:

```
bdb = BigchainDB('http://localhost:9984/api/v1')
```

If you are running the docker-based dev setup that comes along with the bigchaindb_driver repository (see *Development Environment with Docker* for more information), and wish to connect to it from the `bdb-driver` linked (container) service:

```
>>> bdb = BigchainDB('http://bdb-server:9984/api/v1')
```

Alternatively, you may connect to the containerized BigchainDB node from "outside", in which case you need to know the port binding:

```
$ docker-compose port bdb-server 9984
0.0.0.0:32780
```

```
>>> bdb = BigchainDB('http://0.0.0.0:32780/api/v1')
```

For the sake of this example:

```
In [1]: from bigchaindb_driver import BigchainDB

In [2]: bdb = BigchainDB('http://bdb-server:9984/api/v1')
```

## 3.1 Digital Asset Definition

As an example, let's consider the creation and transfer of a digital asset that represents a bicycle:

```
In [3]: bicycle = {
   ...:     'data': {
   ...:         'bicycle': {
   ...:             'serial_number': 'abcd1234',
   ...:             'manufacturer': 'bkfab',
   ...:         },
   ...:     },
   ...: }
   ...:
```

We'll suppose that the bike belongs to Alice, and that it will be transferred to Bob.

## 3.2 Metadata Definition (*optional*)

You can *optionally* add metadata to a transaction. Any dictionary is accepted.

For example:

```
In [4]: metadata = {'planet': 'earth'}
```

## 3.3 Cryptographic Identities Generation

Alice, and Bob are represented by signing/verifying key pairs. The signing (private) key is used to sign transactions, meanwhile the verifying (public) key is used to verify that a signed transaction was indeed signed by the one who claims to be the signee.

```
In [5]: from bigchaindb_driver.crypto import generate_keypair

In [6]: alice, bob = generate_keypair(), generate_keypair()
```

## 3.4 Asset Creation

We're now ready to create the digital asset. First we prepare the transaction:

```
In [7]: prepared_creation_tx = bdb.transactions.prepare(
   ...:     operation='CREATE',
   ...:     owners_before=alice.verifying_key,
   ...:     asset=bicycle,
   ...:     metadata=metadata,
   ...: )
   ...:
```

The `prepared_creation_tx` dictionary should be similar to:

```
In [8]: prepared_creation_tx
Out[8]:
{'id': '68710e289e7c54d74c30c2262bbb08cd252aeaa695fd930398a1fa0a63665370',
 'transaction': {'asset': {'data': {'bicycle': {'manufacturer': 'bkfab',
```

```
     'serial_number': 'abcd1234'}},
   'divisible': False,
   'id': '599f10da-8bed-45f1-b749-ec7165817c07',
   'refillable': False,
   'updatable': False},
  'conditions': [{'amount': 1,
    'cid': 0,
    'condition': {'details': {'bitmask': 32,
      'public_key': 'H18WULNJaKxz9Nh35McyuimrL1PoXoFZchawnkgoUqNy',
      'signature': None,
      'type': 'fulfillment',
      'type_id': 4},
     'uri': 'cc:4:20:7cN69AfuZxE0q5AXhoH9NZYV-A773mxcGiJ5Rf7Mj3I:96'},
    'owners_after': ['H18WULNJaKxz9Nh35McyuimrL1PoXoFZchawnkgoUqNy']}],
  'fulfillments': [{'fid': 0,
    'fulfillment': {'bitmask': 32,
     'public_key': 'H18WULNJaKxz9Nh35McyuimrL1PoXoFZchawnkgoUqNy',
     'signature': None,
     'type': 'fulfillment',
     'type_id': 4},
    'input': None,
    'owners_before': ['H18WULNJaKxz9Nh35McyuimrL1PoXoFZchawnkgoUqNy']}],
  'metadata': {'data': {'planet': 'earth'},
   'id': 'bee2cb61-15d8-4de5-87b8-adf60856cb73'},
  'operation': 'CREATE'},
 'version': 1}
```

The transaction needs to be fulfilled:

```
In [9]: fulfilled_creation_tx = bdb.transactions.fulfill(
   ...:     prepared_creation_tx, private_keys=alice.signing_key)
   ...:
```

```
In [10]: fulfilled_creation_tx
Out[10]:
{'id': '68710e289e7c54d74c30c2262bbb08cd252aeaa695fd930398a1fa0a63665370',
 'transaction': {'asset': {'data': {'bicycle': {'manufacturer': 'bkfab',
     'serial_number': 'abcd1234'}},
   'divisible': False,
   'id': '599f10da-8bed-45f1-b749-ec7165817c07',
   'refillable': False,
   'updatable': False},
  'conditions': [{'amount': 1,
    'cid': 0,
    'condition': {'details': {'bitmask': 32,
      'public_key': 'H18WULNJaKxz9Nh35McyuimrL1PoXoFZchawnkgoUqNy',
      'signature': None,
      'type': 'fulfillment',
      'type_id': 4},
     'uri': 'cc:4:20:7cN69AfuZxE0q5AXhoH9NZYV-A773mxcGiJ5Rf7Mj3I:96'},
    'owners_after': ['H18WULNJaKxz9Nh35McyuimrL1PoXoFZchawnkgoUqNy']}],
  'fulfillments': [{'fid': 0,
    'fulfillment': 'cf:4:7cN69AfuZxE0q5AXhoH9NZYV-A773mxcGiJ5Rf7Mj3KioVXc6m3HqRouOTPqcawcZTGFmX85jQYh
    'input': None,
    'owners_before': ['H18WULNJaKxz9Nh35McyuimrL1PoXoFZchawnkgoUqNy']}],
  'metadata': {'data': {'planet': 'earth'},
   'id': 'bee2cb61-15d8-4de5-87b8-adf60856cb73'},
  'operation': 'CREATE'},
 'version': 1}
```

And sent over to a BigchainDB node:

```
>>> sent_creation_tx = bdb.transactions.send(fulfilled_creation_tx)
```

Note that the response from the node should be the same as that which was sent:

```
>>> sent_creation_tx == fulfilled_creation_tx
True
```

Notice the transaction `id`:

```
In [11]: txid = fulfilled_creation_tx['id']

In [12]: txid
Out[12]: '68710e289e7c54d74c30c2262bbb08cd252aeaa695fd930398a1fa0a63665370'
```

To check the status of the transaction:

```
>>> trials = 0

>>> while bdb.transactions.status(txid).get('status') != 'valid' and trials < 100:
...     trials += 1

>>> bdb.transactions.status(txid)
{'status': 'valid'}
```

**Note:** It may take a small amount of time before a BigchainDB cluster confirms a transaction as being valid.

## 3.5 Asset Transfer

Imagine some time goes by, during which Alice is happy with her bicycle, and one day, she meets Bob, who is interested in acquiring her bicycle. The timing is good for Alice as she had been wanting to get a new bicycle.

To transfer the bicycle (asset) to Bob, Alice must consume the transaction in which the Bicycle asset was created.

Alice could retrieve the transaction:

```
>>>  creation_tx = bdb.transactions.retrieve(txid)
```

or simply use `fulfilled_creation_tx`:

```
In [13]: creation_tx = fulfilled_creation_tx
```

Preparing the transfer transaction:

```
In [14]: cid = 0

In [15]: condition = creation_tx['transaction']['conditions'][cid]

In [16]: transfer_input = {
   ....:     'fulfillment': condition['condition']['details'],
   ....:     'input': {
   ....:         'cid': cid,
   ....:         'txid': creation_tx['id'],
   ....:      },
   ....:      'owners_before': condition['owners_after'],
   ....: }
```

```
   ....:

In [17]: prepared_transfer_tx = bdb.transactions.prepare(
   ....:     operation='TRANSFER',
   ....:     asset=creation_tx['transaction']['asset'],
   ....:     inputs=transfer_input,
   ....:     owners_after=bob.verifying_key,
   ....: )
   ....:
```

and then fulfills the prepared transfer:

```
In [18]: fulfilled_transfer_tx = bdb.transactions.fulfill(
   ....:     prepared_transfer_tx,
   ....:     private_keys=alice.signing_key,
   ....: )
   ....:
```

and finally sends the fulfilled transaction to the connected BigchainDB node:

```
>>> sent_transfer_tx = bdb.transactions.send(fulfilled_transfer_tx)
```

```
>>> sent_transfer_tx == fulfilled_transfer_tx
True
```

The `fulfilled_transfer_tx` dictionary should look something like:

```
In [19]: fulfilled_transfer_tx
Out[19]:
{'id': 'fc73a3acef7656fc71043ceda32f90063a3fa3beda16dd8723da862438379b36',
 'transaction': {'asset': {'id': '599f10da-8bed-45f1-b749-ec7165817c07'},
  'conditions': [{'amount': 1,
    'cid': 0,
    'condition': {'details': {'bitmask': 32,
      'public_key': '24J1kGb9JHXYVWCf3RGLqsqbQMyYCUoaz7995TJ6fHx8',
      'signature': None,
      'type': 'fulfillment',
      'type_id': 4},
     'uri': 'cc:4:20:D7OqdbCaqgoZVRT3Sa-eVZWGH0kT2RfLSJxQdmfQJh0:96'},
    'owners_after': ['24J1kGb9JHXYVWCf3RGLqsqbQMyYCUoaz7995TJ6fHx8']}],
  'fulfillments': [{'fid': 0,
    'fulfillment': 'cf:4:7cN69AfuZxE0q5AXhoH9NZYV-A773mxcGiJ5Rf7Mj3JbkDRTygqjpuXg2BEuEDjS2PLvjPVEGtG
    'input': {'cid': 0,
     'txid': '68710e289e7c54d74c30c2262bbb08cd252aeaa695fd930398a1fa0a63665370'},
    'owners_before': ['H18WULNJaKxz9Nh35McyuimrL1PoXoFZchawnkgoUqNy']}],
  'metadata': None,
  'operation': 'TRANSFER'},
 'version': 1}
```

Bob is the new owner:

```
In [20]: fulfilled_transfer_tx['transaction']['conditions'][0]['owners_after'][0] == bob.verifying_ke
Out[20]: True
```

Alice is the former owner:

```
In [21]: fulfilled_transfer_tx['transaction']['fulfillments'][0]['owners_before'][0] == alice.verify
Out[21]: True
```

## 3.6 Transaction Status

Using the `id` of a transaction, its status can be obtained:

```
>>> bdb.transactions.status(creation_tx['id'])
{'status': 'valid'}
```

Handling cases for which the transaction `id` may not be found:

```
import logging

from bigchaindb_driver import BigchainDB
from bigchaindb_driver.exceptions import NotFoundError

logger = logging.getLogger(__name__)
logging.basicConfig(format='%(asctime)-15s %(status)-3s %(message)s')

# NOTE: You may need to change the URL.
# E.g.: 'http://localhost:9984/api/v1'
bdb = BigchainDB('http://bdb-server:9984/api/v1')
txid = '12345'
try:
    status = bdb.transactions.status(txid)
except NotFoundError as e:
    logger.error('Transaction "%s" was not found.',
                 txid,
                 extra={'status': e.status_code})
```

Running the above code should give something similar to:

```
2016-09-29 15:06:30,606 404 Transaction "12345" was not found.
```

## 3.7 Divisible Assets

In BigchainDB all assets are non-divisible by default so if we want to make a divisible asset we need to explicitly mark it as divisible.

Let's continue with the bicycle example. Bob is now the proud owner of the bicycle and he decides he wants to rent the bicycle. Bob starts by creating a time sharing token in which 1 token corresponds to 1 hour of riding time:

```
In [22]: bicycle_token = {
   ....:     'divisible': True,
   ....:     'data': {
   ....:         'token_for': {
   ....:             'bicycle': {
   ....:                 'serial_number': 'abcd1234',
   ....:                 'manufacturer': 'bkfab'
   ....:             }
   ....:         },
   ....:         'description': 'Time share token. Each token equals 1 hour of riding.'
   ....:     }
   ....: }
   ....:
```

Bob has now decided to issue 10 tokens and assign them to Carly.

```
In [23]: bob, carly = generate_keypair(), generate_keypair()

In [24]: prepared_token_tx = bdb.transactions.prepare(
   ....:     operation='CREATE',
   ....:     owners_before=bob.verifying_key,
   ....:     owners_after=[([carly.verifying_key], 10)],
   ....:     asset=bicycle_token
   ....: )
   ....:

In [25]: fulfilled_token_tx = bdb.transactions.fulfill(
   ....:     prepared_token_tx, private_keys=bob.signing_key)
   ....:
```

Sending the transaction:

```
>>> sent_token_tx = bdb.transactions.send(fulfilled_token_tx)
```

---

**Note:** Defining `owners_after`.

For divisible assets we need to specify the amounts togheter with the public keys. The way we do this is by passing a `list` of `tuples` in `owners_after` in which each `tuple` corresponds to a condition.

For instance instead of creating a transaction with 1 condition with `amount=10` we could have created a transaction with 2 conditions with `amount=5` with:

```
owners_after=[([carly.verifying_key], 5), ([carly.verifying_key], 5)]
```

The reason why the addresses are contained in `lists` is because each condition can have multiple ownership. For instance we can create a condition with `amount=10` in which both Carly and Alice are owners with:

```
owners_after=[([carly.verifying_key, alice.verifying_key], 10)]
```

---

```
>>> sent_token_tx == fulfilled_token_tx
True
```

The `fulfilled_token_tx` dictionary should look something like:

```
In [26]: fulfilled_token_tx
Out[26]:
{'id': 'af606b2f96c90fae320be01c39a10608ae8ac50e5588c1e8cbb7f9eac21a9494',
 'transaction': {'asset': {'data': {'description': 'Time share token. Each token equals 1 hour of ri
    'token_for': {'bicycle': {'manufacturer': 'bkfab',
      'serial_number': 'abcd1234'}}},
   'divisible': True,
   'id': 'd9d8d1ff-2425-4ba9-a225-c4c04cf23477',
   'refillable': False,
   'updatable': False},
  'conditions': [{'amount': 10,
    'cid': 0,
    'condition': {'details': {'bitmask': 32,
      'public_key': '35fBG12fLbvdVReDsjerQZkZmYUndo19JQeC7t4gJu9T',
      'signature': None,
      'type': 'fulfillment',
      'type_id': 4},
     'uri': 'cc:4:20:HujdIxmLwshBG_h9hbvAdYZSOXw411p4YGQ541YX1vI:96'},
    'owners_after': ['35fBG12fLbvdVReDsjerQZkZmYUndo19JQeC7t4gJu9T']}],
  'fulfillments': [{'fid': 0,
```

```
    'fulfillment': 'cf:4:3GToq5OQMzqe3hd2d3o8G62hYIph1obXHMzHoXfwnWtmiJJMATEQQyIN8ztXWO6ORAKCEZMyDNz
    'input': None,
    'owners_before': ['FqKwPqK2gF7NnSdm5UAEnsV9UY5RbR6E8ELW2P5memF8']}],
  'metadata': None,
  'operation': 'CREATE'},
 'version': 1}
```

Bob is the issuer:

```
In [27]: fulfilled_token_tx['transaction']['fulfillments'][0]['owners_before'][0] == bob.verifying_ke
Out[27]: True
```

Carly is the owner of 10 tokens:

```
In [28]: fulfilled_token_tx['transaction']['conditions'][0]['owners_after'][0] == carly.verifying_key
Out[28]: True

In [29]: fulfilled_token_tx['transaction']['conditions'][0]['amount'] == 10
\\\\\\\\\\\\\\Out[29]: True
```

Now Carly wants to ride the bicycle for 2 hours so she needs to send 2 tokens to Bob:

```
In [30]: cid = 0

In [31]: condition = prepared_token_tx['transaction']['conditions'][cid]

In [32]: transfer_input = {
   ....:     'fulfillment': condition['condition']['details'],
   ....:     'input': {
   ....:         'cid': cid,
   ....:         'txid': prepared_token_tx['id'],
   ....:     },
   ....:     'owners_before': condition['owners_after'],
   ....: }
   ....:

In [33]: prepared_transfer_tx = bdb.transactions.prepare(
   ....:     operation='TRANSFER',
   ....:     asset=prepared_token_tx['transaction']['asset'],
   ....:     inputs=transfer_input,
   ....:     owners_after=[([bob.verifying_key], 2), ([carly.verifying_key], 8)]
   ....: )
   ....:

In [34]: fulfilled_transfer_tx = bdb.transactions.fulfill(
   ....:     prepared_transfer_tx, private_keys=carly.signing_key)
   ....:
```

```
>>> sent_transfer_tx = bdb.transactions.send(fulfilled_transfer_tx)
```

```
>>> sent_transfer_tx == fulfilled_transfer_tx
True
```

When transferring divisible assets BigchainDB makes sure that the amount being used is the same as the amount being spent. This ensures that no amounts are lost. For this reason, if Carly wants to transfer 2 tokens of her 10 tokens she needs to reassign the remaining 8 tokens to herself.

The `fulfilled_transfer_tx` with 2 conditions, one with `amount=2` and the other with `amount=8` dictionary should look something like:

```
In [35]: fulfilled_transfer_tx
Out[35]:
{'id': '99f251016d462475e2ba05b2046f37ea4e032fea2f0e0d9d9cdd001f33820482',
 'transaction': {'asset': {'id': 'd9d8d1ff-2425-4ba9-a225-c4c04cf23477'},
  'conditions': [{'amount': 2,
    'cid': 0,
    'condition': {'details': {'bitmask': 32,
      'public_key': 'FqKwPqK2gF7NnSdm5UAEnsV9UY5RbR6E8ELW2P5memF8',
      'signature': None,
      'type': 'fulfillment',
      'type_id': 4},
     'uri': 'cc:4:20:3GToq5OQMzqe3hd2d3o8G62hYIph1obXHMzHoXfwnWs:96'},
    'owners_after': ['FqKwPqK2gF7NnSdm5UAEnsV9UY5RbR6E8ELW2P5memF8']},
   {'amount': 8,
    'cid': 1,
    'condition': {'details': {'bitmask': 32,
      'public_key': '35fBG12fLbvdVReDsjerQZkZmYUndo19JQeC7t4gJu9T',
      'signature': None,
      'type': 'fulfillment',
      'type_id': 4},
     'uri': 'cc:4:20:HujdIxmLwshBG_h9hbvAdYZSOXw411p4YGQ541YX1vI:96'},
    'owners_after': ['35fBG12fLbvdVReDsjerQZkZmYUndo19JQeC7t4gJu9T']}],
  'fulfillments': [{'fid': 0,
    'fulfillment': 'cf:4:HujdIxmLwshBG_h9hbvAdYZSOXw411p4YGQ541YX1vLODeSI1KDvlCip93b-3IWjev04eS807-uH
    'input': {'cid': 0,
     'txid': 'af606b2f96c90fae320be01c39a10608ae8ac50e5588c1e8cbb7f9eac21a9494'},
    'owners_before': ['35fBG12fLbvdVReDsjerQZkZmYUndo19JQeC7t4gJu9T']}],
  'metadata': None,
  'operation': 'TRANSFER'},
 'version': 1}
```

CHAPTER 4

# Advanced Usage Examples

This section has examples of using the Python Driver for more advanced use cases such as escrow.

**Todo**

Work in progress. Will gradually appear as

- https://github.com/bigchaindb/bigchaindb/issues/664
- https://github.com/bigchaindb/bigchaindb-driver/issues/108
- https://github.com/bigchaindb/bigchaindb-driver/issues/110

are taken care of.

## 4.1 Getting Started

First, make sure you have RethinkDB and BigchainDB *installed and running*, i.e. you installed them and you ran:

```
$ rethinkdb
$ bigchaindb configure
$ bigchaindb start
```

Don't shut them down! In a new terminal, open a Python shell:

```
$ python
```

Now we can import the *BigchainDB* class and create an instance:

```
In [1]: from bigchaindb_driver import BigchainDB

In [2]: bdb = BigchainDB()
```

This instantiates an object `bdb` of class *BigchainDB*. When instantiating a *BigchainDB* object without arguments (as above), it sets the server node URL to `http://localhost:9984/api/v1`.

## 4.2 Create a Digital Asset

At a high level, a "digital asset" is something which can be represented digitally and can be assigned to a user. In BigchainDB, users are identified by their public key, and the data payload in a digital asset is represented using a

generic Python dict.

In BigchainDB, digital assets can be created by doing a special kind of transaction: a CREATE transaction.

```
In [3]: from bigchaindb_driver.crypto import generate_keypair
```

Create a test user: alice

```
In [4]: alice = generate_keypair()
```

Define a digital asset data payload

```
In [5]: digital_asset_payload = {'data': {'msg': 'Hello BigchainDB!'}}

In [6]: tx = bdb.transactions.prepare(operation='CREATE',
   ...:                               owners_before=alice.verifying_key,
   ...:                               asset=digital_asset_payload)
   ...:
```

All transactions need to be signed by the user creating the transaction.

```
In [7]: signed_tx = bdb.transactions.fulfill(tx, private_keys=alice.signing_key)
```

Write the transaction to the bigchain. The transaction will be stored in a backlog where it will be validated, included in a block, and written to the bigchain.

```
>>> sent_tx = bdb.transactions.send(signed_tx)
```

Note that the transaction payload returned by the BigchainDB node is equivalent to the signed transaction payload.

```
>>> sent_tx == signed_tx
True
```

```
In [8]: signed_tx
Out[8]:
{'id': 'e90fdc78a0bee54eb1fdc25cffb95cdd37794fd64c8a403e898e2babeff53a48',
 'transaction': {'asset': {'data': {'msg': 'Hello BigchainDB!'},
   'divisible': False,
   'id': '18bba980-9159-4e79-9e3d-2ce1befebd9d',
   'refillable': False,
   'updatable': False},
  'conditions': [{'amount': 1,
    'cid': 0,
    'condition': {'details': {'bitmask': 32,
      'public_key': 'D2TW8o4MtExZu5ZYx45QemKKeZaipLwws3XqQyZN2cjk',
      'signature': None,
      'type': 'fulfillment',
      'type_id': 4},
     'uri': 'cc:4:20:sqvKSe3ugFCfmrCEx2fO1YqM3tTNPlkUHhwOLH6_WZM:96'},
    'owners_after': ['D2TW8o4MtExZu5ZYx45QemKKeZaipLwws3XqQyZN2cjk']}],
  'fulfillments': [{'fid': 0,
    'fulfillment': 'cf:4:sqvKSe3ugFCfmrCEx2fO1YqM3tTNPlkUHhwOLH6_WZMPyUrgQL3TccImsIPKkEREhO96MzT9RyP
    'input': None,
    'owners_before': ['D2TW8o4MtExZu5ZYx45QemKKeZaipLwws3XqQyZN2cjk']}],
  'metadata': None,
  'operation': 'CREATE'},
 'version': 1}
```

## 4.3 Read the Creation Transaction from the DB

After a couple of seconds, we can check if the transactions was included in the bigchain:

```
# Retrieve a transaction from the bigchain
>>> tx_retrieved = bdb.transactions.retrieve(tx['id'])
```

The new owner of the digital asset is now alice which is the public key, aka verifying key of `alice`.

```
In [9]: alice.verifying_key
Out[9]: 'D2TW8o4MtExZu5ZYx45QemKKeZaipLwws3XqQyZN2cjk'
```

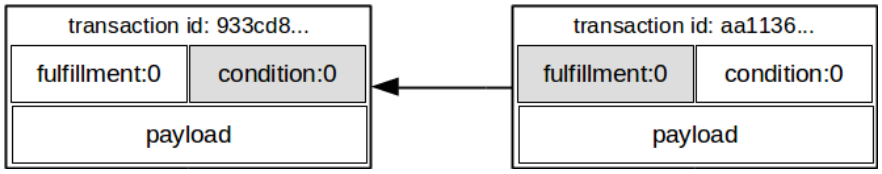
## 4.4 Transfer the Digital Asset

Now that `alice` has a digital asset assigned to her, she can transfer it to another person. Transfer transactions require an input. The input will be the transaction id of a digital asset that was assigned to `alice`, which in our case is

```
In [10]: signed_tx['id']
Out[10]: 'e90fdc78a0bee54eb1fdc25cffb95cdd37794fd64c8a403e898e2babeff53a48'
```

BigchainDB makes use of the crypto-conditions library to both cryptographically lock and unlock transactions. The locking script is referred to as a `condition` and a corresponding `fulfillment` unlocks the condition of the `input_tx`.

Since a transaction can have multiple outputs with each its own (crypto)condition, each transaction input should also refer to the condition index `cid`.




In order to prepare a transfer transaction, alice needs to provide at least three things:

1. `inputs` – one or more conditions that will be fulfilled.
2. `asset` – the asset being transferred.
3. `owners_after` – one or more public keys representing the new owner(s).

To construct the input:

```
In [11]: cid = 0

In [12]: condition = tx['transaction']['conditions'][cid]

In [13]: input_ = {
   ....:     'fulfillment': condition['condition']['details'],
   ....:     'input': {
   ....:         'cid': cid,
   ....:         'txid': tx['id'],
   ....:     },
   ....:     'owners_before': condition['owners_after'],
   ....: }
   ....:
```

The asset, can be directly retrieved from the input `tx`:

```
In [14]: asset = tx['transaction']['asset']
```

Create a second test user, `bob`:

```
In [15]: bob = generate_keypair()

In [16]: bob.verifying_key
Out[16]: 'Ep6tabzBhtJBn3ZTJc7UxMfqB2P4tn952mLiX5STyDGy'
```

And prepare the transfer transaction:

```
In [17]: tx_transfer = bdb.transactions.prepare(
   ....:     operation='TRANSFER',
   ....:     inputs=input_,
   ....:     asset=asset,
   ....:     owners_after=bob.verifying_key,
   ....: )
   ....:
```

The `tx_transfer` dictionary should look something like:

```
In [18]: tx_transfer
Out[18]:
{'id': 'ff7dd6f4be07cc18418350eec6cc99ccc6e0de8a0aaf1899e9ed60bc0a24d1fe',
 'transaction': {'asset': {'id': '18bba980-9159-4e79-9e3d-2ce1befebd9d'},
  'conditions': [{'amount': 1,
    'cid': 0,
    'condition': {'details': {'bitmask': 32,
      'public_key': 'Ep6tabzBhtJBn3ZTJc7UxMfqB2P4tn952mLiX5STyDGy',
      'signature': None,
      'type': 'fulfillment',
      'type_id': 4},
     'uri': 'cc:4:20:zTjjGl-anyx_60o5pjYRLZRUYhKq56-oGMbWtJWK7U4:96'},
    'owners_after': ['Ep6tabzBhtJBn3ZTJc7UxMfqB2P4tn952mLiX5STyDGy']}],
  'fulfillments': [{'fid': 0,
    'fulfillment': {'bitmask': 32,
     'public_key': 'D2TW8o4MtExZu5ZYx45QemKKeZaipLwws3XqQyZN2cjk',
     'signature': None,
     'type': 'fulfillment',
     'type_id': 4},
    'input': {'cid': 0,
     'txid': 'e90fdc78a0bee54eb1fdc25cffb95cdd37794fd64c8a403e898e2babeff53a48'},
    'owners_before': ['D2TW8o4MtExZu5ZYx45QemKKeZaipLwws3XqQyZN2cjk']}],
  'metadata': None,
  'operation': 'TRANSFER'},
 'version': 1}
```

Notice, `bob`'s verifying key (public key), appearing in the above `dict`.

```
In [19]: bob.verifying_key
Out[19]: 'Ep6tabzBhtJBn3ZTJc7UxMfqB2P4tn952mLiX5STyDGy'
```

The transaction now needs to be fulfilled by `alice`:

```
In [20]: signed_tx_transfer = bdb.transactions.fulfill(
   ....:     tx_transfer,
   ....:     private_keys=alice.signing_key,
   ....: )
   ....:
```

If you look at the content of `signed_tx_transfer` you should see the added fulfilment uri, holding the signature:

```
In [21]: signed_tx_transfer
Out[21]:
{'id': 'ff7dd6f4be07cc18418350eec6cc99ccc6e0de8a0aaf1899e9ed60bc0a24d1fe',
 'transaction': {'asset': {'id': '18bba980-9159-4e79-9e3d-2ce1befebd9d'},
  'conditions': [{'amount': 1,
    'cid': 0,
    'condition': {'details': {'bitmask': 32,
      'public_key': 'Ep6tabzBhtJBn3ZTJc7UxMfqB2P4tn952mLiX5STyDGy',
      'signature': None,
      'type': 'fulfillment',
      'type_id': 4},
     'uri': 'cc:4:20:zTjjGl-anyx_60o5pjYRLZRUYhKq56-oGMbWtJWK7U4:96'},
    'owners_after': ['Ep6tabzBhtJBn3ZTJc7UxMfqB2P4tn952mLiX5STyDGy']}],
  'fulfillments': [{'fid': 0,
    'fulfillment': 'cf:4:sqvKSe3ugFCfmrCEx2fO1YqM3tTNPlkUHhwOLH6_WZO2PTXLYuAj4Iie1zeHfWjC5zF4FDlHoOH
    'input': {'cid': 0,
     'txid': 'e90fdc78a0bee54eb1fdc25cffb95cdd37794fd64c8a403e898e2babeff53a48'},
    'owners_before': ['D2TW8o4MtExZu5ZYx45QemKKeZaipLwws3XqQyZN2cjk']}],
  'metadata': None,
  'operation': 'TRANSFER'},
 'version': 1}
```

More precisely:

```
In [22]: signed_tx_transfer['transaction']['fulfillments'][0]['fulfillment']
Out[22]: 'cf:4:sqvKSe3ugFCfmrCEx2fO1YqM3tTNPlkUHhwOLH6_WZO2PTXLYuAj4Iie1zeHfWjC5zF4FDlHoOHhu47_nGDfWI
```

We have yet to send the transaction over to a BigchainDB node, as both preparing and fulfilling a transaction are done "offchain", that is without the need to have a conenction to a BigchainDB federation.

```
sent_tx_transfer = bdb.transactions.send(signed_tx_transfer)
```

Again, as with the `'CREATE'` transaction, notice how the payload returned by the server is equal to the signed one.

```
>>> sent_tx_transfer == signed_tx_transfer
True
```

## 4.5 Double Spends

BigchainDB makes sure that a user can't transfer the same digital asset two or more times (i.e. it prevents double spends).

If we try to create another transaction with the same input as before, the transaction will be marked invalid and the validation will throw a double spend exception.

Let's suppose that Alice tries to re-send the asset back to her "secret" account.

```
In [23]: alice_secret_stash = generate_keypair()
```

Create another transfer transaction with the same input

```
In [24]: tx_transfer_2 = bdb.transactions.prepare(
   ....:     operation='TRANSFER',
   ....:     inputs=input_,
   ....:     asset=asset,
   ....:     owners_after=alice_secret_stash.verifying_key,
```

```
   ....: )
   ....:
```

Fulfill the transaction

```
In [25]: fulfilled_tx_transfer_2 = bdb.transactions.fulfill(
   ....:     tx_transfer_2,
   ....:     private_keys=alice.signing_key,
   ....: )
   ....:
```

Send the transaction over to the node

```
>>> from bigchaindb_driver.exceptions import BigchaindbException
>>> try:
...     bdb.transactions.send(fulfilled_tx_transfer_2)
... except BigchaindbException as e:
...     print(e.info)

{'message': 'Invalid transaction', 'status': 400}
```

**Todo**

Update the above output once https://github.com/bigchaindb/bigchaindb/issues/664 is taken care of.

## 4.6 Multiple Owners

Say `alice` and `bob` own a car together:

```
In [26]: car_asset = {'data': {'car': {'vin': '5YJRE11B781000196'}}}
```

and they agree that `alice` will be the one issuing the asset. To create a new digital asset with *multiple* owners, one can simply provide a list of `owners_after`:

```
In [27]: car_creation_tx = bdb.transactions.prepare(
   ....:     operation='CREATE',
   ....:     owners_before=alice.verifying_key,
   ....:     owners_after=(alice.verifying_key, bob.verifying_key),
   ....:     asset=car_asset,
   ....: )
   ....:

In [28]: signed_car_creation_tx = bdb.transactions.fulfill(
   ....:     car_creation_tx,
   ....:     private_keys=alice.signing_key,
   ....: )
   ....:
```

```
sent_car_tx = bdb.transactions.send(signed_car_creation_tx
```

One day, `alice` and `bob`, having figured out how to teleport themselves, and realizing they no longer need their car, wish to transfer the ownership of their car over to `carol`:

```
In [29]: carol = generate_keypair()
```

In order to prepare the transfer transaction, `alice` and `bob` need the input:

```
In [30]: cid = 0

In [31]: condition = signed_car_creation_tx['transaction']['conditions'][cid]

In [32]: input_ = {
   ....:     'fulfillment': condition['condition']['details'],
   ....:     'input': {
   ....:         'cid': cid,
   ....:         'txid': signed_car_creation_tx['id'],
   ....:     },
   ....:     'owners_before': condition['owners_after'],
   ....: }
   ....:
```

Let's take a moment to contemplate what this `input_` is:

```
In [33]: input_
Out[33]:
{'fulfillment': {'bitmask': 41,
  'subfulfillments': [{'bitmask': 32,
    'public_key': 'D2TW8o4MtExZu5ZYx45QemKKeZaipLwws3XqQyZN2cjk',
    'signature': None,
    'type': 'fulfillment',
    'type_id': 4,
    'weight': 1},
   {'bitmask': 32,
    'public_key': 'Ep6tabzBhtJBn3ZTJc7UxMfqB2P4tn952mLiX5STyDGy',
    'signature': None,
    'type': 'fulfillment',
    'type_id': 4,
    'weight': 1}],
  'threshold': 2,
  'type': 'fulfillment',
  'type_id': 2},
 'input': {'cid': 0,
  'txid': 'b9b0da358a6ed83f2b4dabf4146446ff8c798eae4640ddd236bd9c4c528b630b'},
 'owners_before': ['D2TW8o4MtExZu5ZYx45QemKKeZaipLwws3XqQyZN2cjk',
  'Ep6tabzBhtJBn3ZTJc7UxMfqB2P4tn952mLiX5STyDGy']}
```

and the asset:

```
In [34]: asset = signed_car_creation_tx['transaction']['asset']
```

then `alice` can prepare the transfer:

```
In [35]: car_transfer_tx = bdb.transactions.prepare(
   ....:     operation='TRANSFER',
   ....:     owners_after=carol.verifying_key,
   ....:     asset=asset,
   ....:     inputs=input_,
   ....: )
   ....:
```

The asset can be transfered as soon as each of the `owners_after` fulfills the transaction, that is `alice` and `bob`.

To do so, simply provide a list of all private keys to the fulfill method.

**Danger:** We are currently working to support partial fulfillments, such that not all keys of all parties involved need to be supplied at once. The issue bigchaindb/bigchaindb/issues/729 addresses the current limitation. Your feedback is welcome!

```
In [36]: signed_car_transfer_tx = bdb.transactions.fulfill(
   ....:     car_transfer_tx, private_keys=[alice.signing_key, bob.signing_key]
   ....: )
   ....:
```

Note, that if one the signing keys is missing, the fulfillment will fail. If we omit `bob`:

```
In [37]: from bigchaindb_driver.exceptions import MissingSigningKeyError

In [38]: try:
   ....:     signed_car_transfer_tx = bdb.transactions.fulfill(
   ....:         car_transfer_tx,
   ....:         private_keys=alice.signing_key,
   ....:     )
   ....: except MissingSigningKeyError as e:
   ....:     print(e, e.__cause__, sep='\n')
   ....:
A signing key is missing!
Public key Ep6tabzBhtJBn3ZTJc7UxMfqB2P4tn952mLiX5STyDGy is not a pair to any of the private keys
```

Notice `bob`'s public key in the above message:

```
In [39]:  bob.verifying_key
Out[39]: 'Ep6tabzBhtJBn3ZTJc7UxMfqB2P4tn952mLiX5STyDGy'
```

And the same goes for `alice`. Try it!

Sending the transaction over to a BigchainDB node:

```
sent_car_transfer_tx = bdb.transactions.send(signed_car_transfer_tx)
```

if `alice` and `bob` wish to check the status of the transfer they may use the `status()` endpoint:

```
>>> bdb.transactions.status(sent_car_transfer_tx['id'])
{'status': 'valid'}
```

Done!

Happy, `alice` and `bob` have successfully transferred the ownership of their car to `carol`, and can go on exploring the countless galaxies of the universe using their new teleportation skills.

## 4.7 Crypto-Conditions (Advanced)

### 4.7.1 Introduction

Crypto-conditions provide a mechanism to describe a signed message such that multiple actors in a distributed system can all verify the same signed message and agree on whether it matches the description.

This provides a useful primitive for event-based systems that are distributed on the Internet since we can describe events in a standard deterministic manner (represented by signed messages) and therefore define generic authenticated event handlers.

Crypto-conditions are part of the Interledger protocol and the full specification can be found here.

Implementations of the crypto-conditions are available in Python and JavaScript.

### 4.7.2 Threshold Conditions

Threshold conditions introduce multi-signatures, m-of-n signatures or even more complex binary Merkle trees to BigchainDB.

Setting up a generic threshold condition is a bit more elaborate than regular transaction signing but allow for flexible signing between multiple parties or groups.

The basic workflow for creating a more complex cryptocondition is the following:

1. Create a transaction template that includes the public key of all (nested) parties as `owners_after`
2. Set up the threshold condition using the cryptocondition library
3. Update the condition and hash in the transaction template

We'll illustrate this by a threshold condition where 2 out of 3 `owners_after` need to sign the transaction:

---

**Todo**

Stay tuned. Will soon be documented once

- https://github.com/bigchaindb/bigchaindb-driver/issues/109

is taken care of.

---

The transaction can now be transfered by fulfilling the threshold condition.

The fulfillment involves:

1. Create a transaction template that includes the public key of all (nested) parties as `owners_before`
2. Parsing the threshold condition into a fulfillment using the cryptocondition library
3. Signing all necessary subfulfillments and updating the fulfillment field in the transaction

---

**Todo**

Stay tuned. Will soon be documented once

- https://github.com/bigchaindb/bigchaindb-driver/issues/110

are taken care of.

---

### 4.7.3 Hash-locked Conditions

A hash-lock condition on an asset is like a password condition: anyone with the secret preimage (like a password) can fulfill the hash-lock condition and transfer the asset to themselves.

Under the hood, fulfilling a hash-lock condition amounts to finding a string (a "preimage") which, when hashed, results in a given value. It's easy to verify that a given preimage hashes to the given value, but it's computationally difficult to *find* a string which hashes to the given value. The only practical way to get a valid preimage is to get it from the original creator (possibly via intermediaries).

One possible use case is to distribute preimages as "digital vouchers." The first person to redeem a voucher will get the associated asset.

A federation node can create an asset with a hash-lock condition and no `owners_after`. Anyone who can fullfill the hash-lock condition can transfer the asset to themselves.

**Todo**

Stay tuned. Will soon be documented once

- https://github.com/bigchaindb/bigchaindb-driver/issues/110

are taken care of.

In order to redeem the asset, one needs to create a fulfillment with the correct secret:

**Todo**

Stay tuned. Will soon be documented once

- https://github.com/bigchaindb/bigchaindb-driver/issues/110

are taken care of.

### 4.7.4 Timeout Conditions

Timeout conditions allow assets to expire after a certain time. The primary use case of timeout conditions is to enable *Escrow*.

The condition can only be fulfilled before the expiry time. Once expired, the asset is lost and cannot be fulfilled by anyone.

**Note:** The timeout conditions are BigchainDB-specific and not (yet) supported by the ILP standard.

**Important:** **Caveat**: The times between nodes in a BigchainDB federation may (and will) differ slightly. In this case, the majority of the nodes will decide.

**Todo**

Stay tuned. Will soon be documented once

- https://github.com/bigchaindb/bigchaindb-driver/issues/110

are taken care of.

The following demonstrates that the transaction invalidates once the timeout occurs:

**Todo**

Stay tuned. Will soon be documented once

- https://github.com/bigchaindb/bigchaindb-driver/issues/110

are taken care of.

If you were fast enough, you should see the following output:

**Todo**

Stay tuned. Will soon be documented once

- https://github.com/bigchaindb/bigchaindb-driver/issues/110

are taken care of.

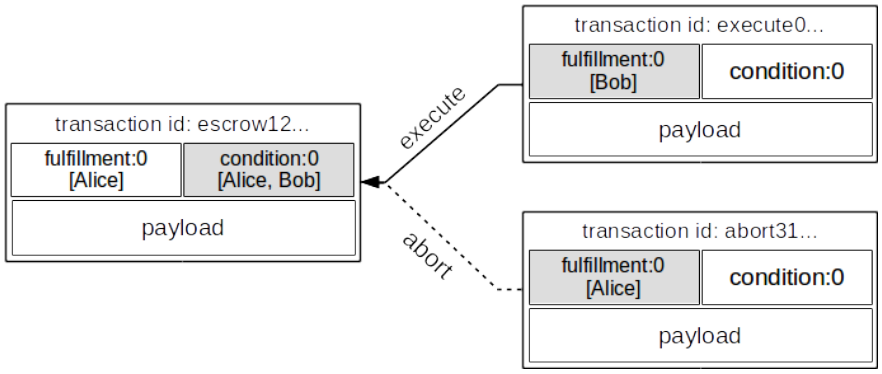
## 4.8 Escrow

Escrow is a mechanism for conditional release of assets.

This means that the assets are locked up by a trusted party until an `execute` condition is presented. In order not to tie up the assets forever, the escrow foresees an `abort` condition, which is typically an expiry time.

BigchainDB and cryptoconditions provides escrow out-of-the-box, without the need of a trusted party.

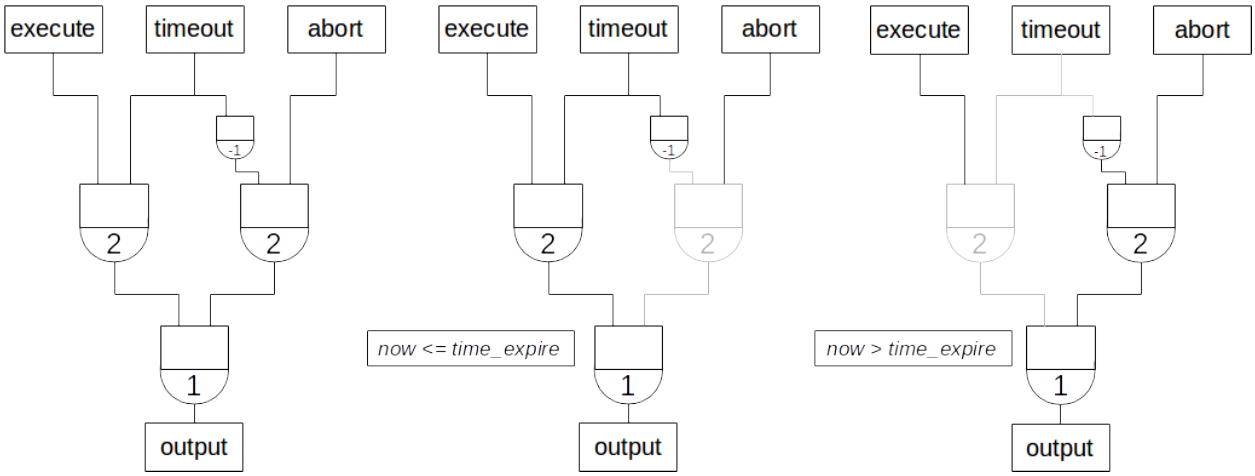
A threshold condition is used to represent the escrow, since BigchainDB transactions cannot have a *pending* state.




The logic for switching between `execute` and `abort` conditions is conceptually simple:

```
if timeout_condition.validate(utcnow()):
    execute_fulfillment.validate(msg) == True
    abort_fulfillment.validate(msg) == False
else:
    execute_fulfillment.validate(msg) == False
    abort_fulfillment.validate(msg) == True
```

The above switch can be implemented as follows using threshold cryptoconditions:




The inverted timeout is denoted by a -1 threshold, which negates the output of the fulfillment.

```
inverted_fulfillment.validate(msg) == not fulfillment.validate(msg)
```

**Note:** inverted thresholds are BigchainDB-specific and not supported by the ILP standard. The main reason is that it's difficult to tell whether the fulfillment was negated, or just omitted.

The following code snippet shows how to create an escrow condition:

**Todo**

Stay tuned. Will soon be documented once

- https://github.com/bigchaindb/bigchaindb-driver/issues/108
- https://github.com/bigchaindb/bigchaindb-driver/issues/110

are taken care of.

**Todo**

Stay tuned. Will soon be documented once

- https://github.com/bigchaindb/bigchaindb-driver/issues/108
- https://github.com/bigchaindb/bigchaindb-driver/issues/110

are taken care of.

In the case of `bob`, we create the `abort` fulfillment:

**Todo**

Stay tuned. Will soon be documented once

- https://github.com/bigchaindb/bigchaindb-driver/issues/108
- https://github.com/bigchaindb/bigchaindb-driver/issues/110

are taken care of.

The following demonstrates that the transaction validation switches once the timeout occurs:

---

**Todo**

Stay tuned. Will soon be documented once

- https://github.com/bigchaindb/bigchaindb-driver/issues/108
- https://github.com/bigchaindb/bigchaindb-driver/issues/110

are taken care of.

---

If you execute in a timely fashion, you should see the following:

---

**Todo**

Stay tuned. Will soon be documented once

- https://github.com/bigchaindb/bigchaindb-driver/issues/108
- https://github.com/bigchaindb/bigchaindb-driver/issues/110

are taken care of.

---

Of course, when the `execute` transaction was accepted in-time by bigchaindb, then writing the `abort` transaction after expiry will yield a `Doublespend` error.

CHAPTER 5

# Handcrafting Transactions

For those who wish to assemble transaction payloads "by hand", with examples in Python.

**Note:** The contents are presented for BigchainDB 0.8. The transaction schema is constantly evolving at this stage and the current contents may be outdated by a new release.



## 5.1 Overview

Submitting a transaction to a BigchainDB node consists of three main steps:

1. Preparing the transaction payload;
2. Fulfilling the prepared transaction payload; and
3. Sending the transaction payload via HTTPS.

Step 1 and 2 can be performed offline on the client. That is, they do not require any connection to any BigchainDB node.

For convenience's sake, some utilites are provided to prepare and fulfill a transaction via the `BigchainDB` class, and via the `offchain` module. For an introduction on using these utilities, see the *Basic Usage Examples* or *Advanced Usage Examples* sections.

The rest of this document will guide you through completing steps 1 and 2 manually by revisiting some of the examples provided in the usage sections. We will:

- provide all values, including the default ones;
- generate the transaction id;
- learn to use crypto-conditions to generate a condition that locks the transaction, hence protecting it from being consumed by an unauthorized user;
- learn to use crypto-conditions to generate a fulfillment that unlocks the transaction asset, and consequently enact an ownership transfer.

In order to perform all of the above, we'll use the following Python libraries:

- `json`: to serialize the transaction dictionary into a JSON formatted string;
- sha3: to hash the serialized transaction; and
- cryptoconditions: to create conditions and fulfillments

### 5.1.1 High-level view of a transaction in Python

For detailled documentation on the transaction schema, please consult The Transaction Model and The Transaction Schema.

From the point of view of Python, a transaction is simply a dictionary with a number of nested structures.

The first level has three keys:

- `id` – a `str`;
- `version` – an `int`; and
- `transaction` – a `dict`

Because a transaction must be signed before being sent, the `id` is required to be provided by the client.

When you assemble the payload you'll have:

whose `id` can be generated by hashing the above with SHA-3's SHA256 algorithm.

---

**Important: Implications of Signed Payloads**

Because transactions are signed by the client submitting them, various values that could traditionally be generated on the server side need to be generated on the client side.

These values include:

- **transaction id**, which is a hash of the entire payload, without the signature(s)
- **asset id**
- **metadata id**
- any optional value, such as `version` which defaults to `1`

This makes the assembling of a payload more involved as one needs to provide all values regardless of whether there are defaults or not.

---

#### The transaction body

The transaction body is made up of the following keys:

- `asset` – `dict`
- `metadata` – `dict`
- `operation` – `str`
- `conditions` – `list` of `dict`
- `fulfillments` – `list` of `dict`

#### asset

```
asset = {
    'data': {},
    'divisible': False,
    'refillable': False,
    'updatable': False,
    'id': '',
}
```

Example of an asset payload:

```
asset = {
    'data': {
        'bicycle': {
            'manufacturer': 'bkfab',
            'serial_number': 'abcd1234',
        },
    },
    'divisible': False,
    'refillable': False,
    'updatable': False,
    'id': '7ab63c48-4c24-41df-a1bd-934bb609a7f7',
}
```

---

**Note:** In many client-server architectures, the values for the keys:

- `'divisible'`
- `'refillable'`
- `'updatable'`
- `'id'`

could all be generated on the server side.

*In the case of BigchainDB, because we rely on cryptographic signatures, the payloads need to be fully prepared and signed on the client side. This prevents the server(s) from tempering with the provided data.*

---

#### metadata

```
metadata = {
    'data': {},
    'id': '',
}
```

Example of a metadata payload:

```
metadata = {
    'data': {
        'planet': 'earth',
    },
    'id': 'ad8c83bd-9192-43b3-b636-af93a3a6b07c',
}
```

**Note:** In many client-server architectures, the value of the `'id'` could be generated on the server side.

*In the case of BigchainDB, because we rely on cryptographic signatures, the payloads need to be fully prepared and signed on the client side. This prevents the server(s) from tempering with the provided data.*

#### operation

```
operation = '<operation>'
```

`<operation>` must be one of `'CREATE'`, `'TRANSFER'`, or `'GENESIS'`

**Important:** Case sensitive; all letters must be capitalized.

#### conditions

The purpose of the condition is to lock the transaction, such that a valid fulfillment is required to unlock it. In the case of signature-based schemes, the lock is basically a public key, such that in order to unlock the transaction one needs to have the private key.

Example of a condition payload:

```
{
    'amount': 1,
    'cid': 0,
    'condition': {
        'details': {
            'bitmask': 32,
            'public_key': '8L6ngTZ5ixuFEr1GiunrFNWtGkft4swWWArXjWJu2Uwc',
            'signature': None,
            'type': 'fulfillment',
            'type_id': 4,
        },
        'uri': 'cc:4:20:bOZjTedaOgPsbYjh3QeOEQCj1o1lIvVefR71sS8egnM:96'
    },
    'owners_after': ['8L6ngTZ5ixuFEr1GiunrFNWtGkft4swWWArXjWJu2Uwc'],
}
```

#### fulfillments

A fulfillment payload is first prepared without its fulfillment uri (e.g., containing the signature), and included in the transaction payload, which will be hashed to generate the transaction id.

In a second step, after the transaction id has been generated, the fulfillment URI (e.g. containing a signature) can be added.

Moreover, payloads for `CREATE` operations are a bit different.

**Note:** We hope to be able to simplify the payload structure and validation, such that this is no longer required.

**Todo**

Point to issues addressing the topic.

Example of a fulfillment payload **before fulfilling it**, for a *CREATE* operation:

```
fulfillment = {
    'fid': 0,
    'fulfillment': None,
    'input': None,
    'owners_before': ['8L6ngTZ5ixuFEr1GiunrFNWtGkft4swWWArXjWJu2Uwc'],
}
```

**Note:** Because it is a `CREATE` operation, the `'input'` field is set to `None`.

**Todo**

Example of a fulfillment payload **after fulfilling it**:

## 5.2 Bicycle Asset Creation Revisited

### 5.2.1 The Prepared Transaction

Recall that in order to prepare a transaction, we had to do something similar to:

```
In [1]: from bigchaindb_driver.crypto import generate_keypair

In [2]: from bigchaindb_driver.offchain import prepare_transaction

In [3]: alice = generate_keypair()

In [4]: bicycle = {
   ...:     'data': {
   ...:         'bicycle': {
   ...:             'serial_number': 'abcd1234',
   ...:             'manufacturer': 'bkfab',
   ...:         },
   ...:     },
   ...: }
   ...:

In [5]: metadata = {'planet': 'earth'}

In [6]: prepared_creation_tx = prepare_transaction(
   ...:     operation='CREATE',
   ...:     owners_before=alice.verifying_key,
   ...:     asset=bicycle,
   ...:     metadata=metadata,
   ...: )
   ...:
```

and the payload of the prepared transaction looked similar to:

```
In [7]: prepared_creation_tx
Out[7]:
{'id': 'ede453dc71ca2c716db3c78d9d8ac5b2b6e82f93d74c071c1323b45159785f8b',
 'transaction': {'asset': {'data': {'bicycle': {'manufacturer': 'bkfab',
     'serial_number': 'abcd1234'}},
   'divisible': False,
   'id': '3a1b1c72-43a8-439c-ad26-c762b8981b26',
   'refillable': False,
   'updatable': False},
  'conditions': [{'amount': 1,
    'cid': 0,
    'condition': {'details': {'bitmask': 32,
      'public_key': '6Sk1GUU6CL44znRHiY2ebRCpGknyeGx7WssgZBg4mS3f',
      'signature': None,
      'type': 'fulfillment',
      'type_id': 4},
     'uri': 'cc:4:20:UOKUj39h7TOpoMfJj6KFKooO36E7I2yJhW3msLcuNA4:96'},
    'owners_after': ['6Sk1GUU6CL44znRHiY2ebRCpGknyeGx7WssgZBg4mS3f']}],
  'fulfillments': [{'fid': 0,
    'fulfillment': {'bitmask': 32,
     'public_key': '6Sk1GUU6CL44znRHiY2ebRCpGknyeGx7WssgZBg4mS3f',
     'signature': None,
     'type': 'fulfillment',
     'type_id': 4},
    'input': None,
    'owners_before': ['6Sk1GUU6CL44znRHiY2ebRCpGknyeGx7WssgZBg4mS3f']}],
  'metadata': {'data': {'planet': 'earth'},
   'id': '6799b776-af79-4f3a-baf0-396987bb357e'},
  'operation': 'CREATE'},
 'version': 1}
```

Note `alice`'s public key:

```
In [8]: alice.verifying_key
Out[8]: '6Sk1GUU6CL44znRHiY2ebRCpGknyeGx7WssgZBg4mS3f'
```

We are now going to craft this payload by hand.

Extract asset id and metadata id:

```
In [9]: asset_id = prepared_creation_tx['transaction']['asset']['id']

In [10]: metadata_id = prepared_creation_tx['transaction']['metadata']['id']
```

### The transaction body

#### asset

```
In [11]: asset = {
   ....:     'data': {
   ....:         'bicycle': {
   ....:             'manufacturer': 'bkfab',
   ....:             'serial_number': 'abcd1234',
   ....:         },
   ....:     },
   ....:     'divisible': False,
   ....:     'refillable': False,
```

```
   ....:     'updatable': False,
   ....:     'id': asset_id,
   ....: }
   ....:
```

#### metadata

```
In [12]: metadata = {
   ....:     'data': {
   ....:         'planet': 'earth',
   ....:     },
   ....:     'id': metadata_id,
   ....: }
   ....:
```

#### operation

```
In [13]: operation = 'CREATE'
```

**Important:** Case sensitive; all letters must be capitalized.

#### conditions

The purpose of the condition is to lock the transaction, such that a valid fulfillment is required to unlock it. In the case of signature-based schemes, the lock is basically a public key, such that in order to unlock the transaction one needs to have the private key.

Let's review the condition payload of the prepared transaction, to see what we are aiming for:

```
In [14]: prepared_creation_tx['transaction']['conditions'][0]
Out[14]:
{'amount': 1,
 'cid': 0,
 'condition': {'details': {'bitmask': 32,
   'public_key': '6Sk1GUU6CL44znRHiY2ebRCpGknyeGx7WssgZBg4mS3f',
   'signature': None,
   'type': 'fulfillment',
   'type_id': 4},
  'uri': 'cc:4:20:UOKUj39h7TOpoMfJj6KFKooO36E7I2yJhW3msLcuNA4:96'},
 'owners_after': ['6Sk1GUU6CL44znRHiY2ebRCpGknyeGx7WssgZBg4mS3f']}
```

The difficult parts are the condition details and URI. We'll now see how to generate them using the `cryptoconditions` library:

```
In [15]: from cryptoconditions import Ed25519Fulfillment

In [16]: ed25519 = Ed25519Fulfillment(public_key=alice.verifying_key)
```

generate the condition uri:

```
In [17]: ed25519.condition_uri
Out[17]: 'cc:4:20:UOKUj39h7TOpoMfJj6KFKooO36E7I2yJhW3msLcuNA4:96'
```

So now you have a condition URI for Alice's public key.

As for the details:

```
In [18]: ed25519.to_dict()
Out[18]:
{'bitmask': 32,
 'public_key': '6Sk1GUU6CL44znRHiY2ebRCpGknyeGx7WssgZBg4mS3f',
 'signature': None,
 'type': 'fulfillment',
 'type_id': 4}
```

We can now easily assemble the `dict` for the condition:

```
In [19]: condition = {
   ....:     'amount': 1,
   ....:     'cid': 0,
   ....:     'condition': {
   ....:         'details': ed25519.to_dict(),
   ....:         'uri': ed25519.condition_uri,
   ....:     },
   ....:     'owners_after': (alice.verifying_key,),
   ....: }
   ....:
```

Let's recap and set the `conditions` key:

```
In [20]: from cryptoconditions import Ed25519Fulfillment

In [21]: ed25519 = Ed25519Fulfillment(public_key=alice.verifying_key)

In [22]: condition = {
   ....:     'amount': 1,
   ....:     'cid': 0,
   ....:     'condition': {
   ....:         'details': ed25519.to_dict(),
   ....:         'uri': ed25519.condition_uri,
   ....:     },
   ....:     'owners_after': (alice.verifying_key,),
   ....: }
   ....:

In [23]: conditions = (condition,)
```

The key part is the condition URI:

```
In [24]: ed25519.condition_uri
Out[24]: 'cc:4:20:UOKUj39h7TOpoMfJj6KFKooO36E7I2yJhW3msLcuNA4:96'
```

To know more about its meaning, you may read the cryptoconditions internet draft.

#### fulfillments

The fulfillment for a `CREATE` operation is somewhat special:

```
In [25]: fulfillment = {
   ....:     'fid': 0,
   ....:     'fulfillment': None,
   ....:     'input': None,
```

```
   ....:     'owners_before': (alice.verifying_key,)
   ....: }
   ....:
```

- The input field is empty because it's a `CREATE` operation;
- The `'fulfillemnt'` value is `None` as it will be set during the fulfillment step; and
- The `'owners_before'` field identifies the issuer(s) of the asset that is being created.

The `fulfillments` value is simply a list or tuple of all fulfillments:

```
In [26]: fulfillments = (fulfillment,)
```

---

**Note:** You may rightfully observe that the `prepared_creation_tx` fulfillment generated via the `prepare_transaction` function differs:

```
In [27]: prepared_creation_tx['transaction']['fulfillments'][0]
Out[27]:
{'fid': 0,
 'fulfillment': {'bitmask': 32,
  'public_key': '6Sk1GUU6CL44znRHiY2ebRCpGknyeGx7WssgZBg4mS3f',
  'signature': None,
  'type': 'fulfillment',
  'type_id': 4},
 'input': None,
 'owners_before': ['6Sk1GUU6CL44znRHiY2ebRCpGknyeGx7WssgZBg4mS3f']}
```

More precisely, the value of `'fulfillment'`:

```
In [28]: prepared_creation_tx['transaction']['fulfillments'][0]['fulfillment']
Out[28]:
{'bitmask': 32,
 'public_key': '6Sk1GUU6CL44znRHiY2ebRCpGknyeGx7WssgZBg4mS3f',
 'signature': None,
 'type': 'fulfillment',
 'type_id': 4}
```

The quick answer is that it simply is not needed, and can be set to `None`.

---

Putting it all together:

```
In [29]: handcrafted_creation_tx = {
   ....:     'transaction': {
   ....:         'asset': asset,
   ....:         'metadata': metadata,
   ....:         'operation': operation,
   ....:         'conditions': conditions,
   ....:         'fulfillments': fulfillments,
   ....:     },
   ....:     'version': 1,
   ....: }
   ....:

In [30]: handcrafted_creation_tx
Out[30]:
{'transaction': {'asset': {'data': {'bicycle': {'manufacturer': 'bkfab',
     'serial_number': 'abcd1234'}},
   'divisible': False,
```

```
   'id': '3a1b1c72-43a8-439c-ad26-c762b8981b26',
   'refillable': False,
   'updatable': False},
  'conditions': ({'amount': 1,
    'cid': 0,
    'condition': {'details': {'bitmask': 32,
      'public_key': '6Sk1GUU6CL44znRHiY2ebRCpGknyeGx7WssgZBg4mS3f',
      'signature': None,
      'type': 'fulfillment',
      'type_id': 4},
     'uri': 'cc:4:20:UOKUj39h7TOpoMfJj6KFKooO36E7I2yJhW3msLcuNA4:96'},
    'owners_after': ('6Sk1GUU6CL44znRHiY2ebRCpGknyeGx7WssgZBg4mS3f',)},),
  'fulfillments': ({'fid': 0,
    'fulfillment': None,
    'input': None,
    'owners_before': ('6Sk1GUU6CL44znRHiY2ebRCpGknyeGx7WssgZBg4mS3f',)},),
  'metadata': {'data': {'planet': 'earth'},
   'id': '6799b776-af79-4f3a-baf0-396987bb357e'},
  'operation': 'CREATE'},
 'version': 1}
```

We're missing the `id`, and we'll generate it soon, but before that, let's recap how we've put all the code together to generate the above payload:

```
from cryptoconditions import Ed25519Fulfillment
from bigchaindb_driver.crypto import CryptoKeypair

alice = CryptoKeypair(
    verifying_key=alice.verifying_key,
    signing_key=alice.signing_key,
)

operation = 'CREATE'

asset = {
    'data': {
        'bicycle': {
            'manufacturer': 'bkfab',
            'serial_number': 'abcd1234',
        },
    },
    'divisible': False,
    'refillable': False,
    'updatable': False,
    'id': asset_id,
}

metadata = {
    'data': {
        'planet': 'earth',
    },
    'id': metadata_id,
}

ed25519 = Ed25519Fulfillment(public_key=alice.verifying_key)

condition = {
    'amount': 1,
```

```
    'cid': 0,
    'condition': {
        'details': ed25519.to_dict(),
        'uri': ed25519.condition_uri,
    },
    'owners_after': (alice.verifying_key,),
}
conditions = (condition,)

fulfillment = {
    'fid': 0,
    'fulfillment': None,
    'input': None,
    'owners_before': (alice.verifying_key,)
}
fulfillments = (fulfillment,)

handcrafted_creation_tx = {
    'transaction': {
        'asset': asset,
        'metadata': metadata,
        'operation': operation,
        'conditions': conditions,
        'fulfillments': fulfillments,
    },
    'version': 1,
}
```

#### id

```
In [31]: import json

In [32]: from sha3 import sha3_256

In [33]: json_str_tx = json.dumps(
   ....:     handcrafted_creation_tx,
   ....:     sort_keys=True,
   ....:     separators=(',', ':'),
   ....:     ensure_ascii=False,
   ....: )
   ....:

In [34]: txid = sha3_256(json_str_tx.encode()).hexdigest()

In [35]: handcrafted_creation_tx['id'] = txid
```

Compare this to the txid of the transaction generated via `prepare_transaction()`:

```
In [36]: txid == prepared_creation_tx['id']
Out[36]: True
```

You may observe that

```
In [37]: handcrafted_creation_tx == prepared_creation_tx
Out[37]: False
```

```
In [38]: from copy import deepcopy

In [39]: # back up

In [40]: prepared_creation_tx_bk = deepcopy(prepared_creation_tx)

In [41]: # set fulfillment to None

In [42]: prepared_creation_tx['transaction']['fulfillments'][0]['fulfillment'] = None

In [43]: handcrafted_creation_tx == prepared_creation_tx
Out[43]: False
```

Are still not equal because we used tuples instead of lists.

```
In [44]: # serialize to json str

In [45]: json_str_handcrafted_tx = json.dumps(handcrafted_creation_tx, sort_keys=True)

In [46]: json_str_prepared_tx = json.dumps(prepared_creation_tx, sort_keys=True)
```

```
In [47]: json_str_handcrafted_tx == json_str_prepared_tx
Out[47]: True

In [48]: prepared_creation_tx = prepared_creation_tx_bk
```

The full handcrafted yet-to-be-fulfilled transaction payload:

```
In [49]: handcrafted_creation_tx
Out[49]:
{'id': 'ede453dc71ca2c716db3c78d9d8ac5b2b6e82f93d74c071c1323b45159785f8b',
 'transaction': {'asset': {'data': {'bicycle': {'manufacturer': 'bkfab',
     'serial_number': 'abcd1234'}},
   'divisible': False,
   'id': '3a1b1c72-43a8-439c-ad26-c762b8981b26',
   'refillable': False,
   'updatable': False},
  'conditions': ({'amount': 1,
    'cid': 0,
    'condition': {'details': {'bitmask': 32,
      'public_key': '6Sk1GUU6CL44znRHiY2ebRCpGknyeGx7WssgZBg4mS3f',
      'signature': None,
      'type': 'fulfillment',
      'type_id': 4},
     'uri': 'cc:4:20:UOKUj39h7TOpoMfJj6KFKooO36E7I2yJhW3msLcuNA4:96'},
    'owners_after': ('6Sk1GUU6CL44znRHiY2ebRCpGknyeGx7WssgZBg4mS3f',)},),
  'fulfillments': ({'fid': 0,
    'fulfillment': None,
    'input': None,
    'owners_before': ('6Sk1GUU6CL44znRHiY2ebRCpGknyeGx7WssgZBg4mS3f',)},),
  'metadata': {'data': {'planet': 'earth'},
   'id': '6799b776-af79-4f3a-baf0-396987bb357e'},
  'operation': 'CREATE'},
 'version': 1}
```

### 5.2.2 The Fulfilled Transaction

```
In [50]: from cryptoconditions.crypto import Ed25519SigningKey

In [51]: from bigchaindb_driver.offchain import fulfill_transaction

In [52]: fulfilled_creation_tx = fulfill_transaction(
   ....:     prepared_creation_tx,
   ....:     private_keys=alice.signing_key,
   ....: )
   ....:

In [53]: sk = Ed25519SigningKey(alice.signing_key)

In [54]: message = json.dumps(
   ....:     handcrafted_creation_tx,
   ....:     sort_keys=True,
   ....:     separators=(',', ':'),
   ....:     ensure_ascii=False,
   ....: )
   ....:

In [55]: ed25519.sign(message.encode(), sk)

In [56]: fulfillment = ed25519.serialize_uri()

In [57]: handcrafted_creation_tx['transaction']['fulfillments'][0]['fulfillment'] = fulfillment
```

Let's check this:

```
In [58]: fulfilled_creation_tx['transaction']['fulfillments'][0]['fulfillment'] == fulfillment
Out[58]: True

In [59]: json.dumps(fulfilled_creation_tx, sort_keys=True) == json.dumps(handcrafted_creation_tx, sor
\\\\\\\\\\\\\Out[59]: True
```

### 5.2.3 In a nutshell

Handcrafting a `'CREATE'` transaction can be done as follows:

```
import json
from uuid import uuid4

import sha3
import cryptoconditions

from bigchaindb_driver.crypto import generate_keypair


alice = generate_keypair()

operation = 'CREATE'

asset_id = str(uuid4())
asset = {
    'data': {
        'bicycle': {
```

```
            'manufacturer': 'bkfab',
            'serial_number': 'abcd1234',
        },
    },
    'divisible': False,
    'refillable': False,
    'updatable': False,
    'id': asset_id,
}

metadata_id = str(uuid4())
metadata = {
    'data': {
        'planet': 'earth',
    },
    'id': metadata_id,
}

ed25519 = cryptoconditions.Ed25519Fulfillment(public_key=alice.verifying_key)

condition = {
    'amount': 1,
    'cid': 0,
    'condition': {
        'details': ed25519.to_dict(),
        'uri': ed25519.condition_uri,
    },
    'owners_after': (alice.verifying_key,),
}
conditions = (condition,)

fulfillment = {
    'fid': 0,
    'fulfillment': None,
    'input': None,
    'owners_before': (alice.verifying_key,)
}
fulfillments = (fulfillment,)

handcrafted_creation_tx = {
    'transaction': {
        'asset': asset,
        'metadata': metadata,
        'operation': operation,
        'conditions': conditions,
        'fulfillments': fulfillments,
    },
    'version': 1,
}

json_str_tx = json.dumps(
    handcrafted_creation_tx,
    sort_keys=True,
    separators=(',', ':'),
    ensure_ascii=False,
)

creation_txid = sha3.sha3_256(json_str_tx.encode()).hexdigest()
```

```
handcrafted_creation_tx['id'] = creation_txid

sk = cryptoconditions.crypto.Ed25519SigningKey(alice.signing_key)

message = json.dumps(
    handcrafted_creation_tx,
    sort_keys=True,
    separators=(',', ':'),
    ensure_ascii=False,
)

ed25519.sign(message.encode(), sk)

fulfillment = ed25519.serialize_uri()

handcrafted_creation_tx['transaction']['fulfillments'][0]['fulfillment'] = fulfillment
```

Sending it over to a BigchainDB node:

```
from bigchaindb_driver import BigchainDB

bdb = BigchainDB('http://bdb-server:9984/api/v1')
returned_creation_tx = bdb.transactions.send(handcrafted_creation_tx)
```

A few checks:

```
>>> json.dumps(returned_creation_tx, sort_keys=True) == json.dumps(handcrafted_creation_tx, sort_keys
True
```

```
>>> bdb.transactions.status(creation_txid)
{'status': 'valid'}
```

**Tip:** When checking for the status of a transaction, one should keep in mind tiny delays before a transaction reaches a valid status.

## 5.3 Bicycle Asset Transfer Revisited

In the *bicycle transfer example* , we showed that the transfer transaction was prepared and fulfilled as follows:

```
In [60]: creation_tx = fulfilled_creation_tx

In [61]: bob = generate_keypair()

In [62]: cid = 0

In [63]: condition = creation_tx['transaction']['conditions'][cid]

In [64]: transfer_input = {
   ....:     'fulfillment': condition['condition']['details'],
   ....:     'input': {
   ....:         'cid': cid,
   ....:         'txid': creation_tx['id'],
   ....:      },
   ....:      'owners_before': condition['owners_after'],
```

```
   ....: }
   ....:

In [65]: prepared_transfer_tx = prepare_transaction(
   ....:     operation='TRANSFER',
   ....:     asset=creation_tx['transaction']['asset'],
   ....:     inputs=transfer_input,
   ....:     owners_after=bob.verifying_key,
   ....: )
   ....:

In [66]: fulfilled_transfer_tx = fulfill_transaction(
   ....:     prepared_transfer_tx,
   ....:     private_keys=alice.signing_key,
   ....: )
   ....:

In [67]: fulfilled_transfer_tx
Out[67]:
{'id': '04958329d67c941a9aa22d13f583939050315cd40df0ca76a7a95a3570b2ab49',
 'transaction': {'asset': {'id': '3a1b1c72-43a8-439c-ad26-c762b8981b26'},
  'conditions': [{'amount': 1,
    'cid': 0,
    'condition': {'details': {'bitmask': 32,
      'public_key': 'DRD7NfhpMmP2caqTJmn9FzxHBFnqNNHw4NU9zn2ZgVCX',
      'signature': None,
      'type': 'fulfillment',
      'type_id': 4},
     'uri': 'cc:4:20:uH_gDwjq9W31pDEV5bXexuUyBhT8_xp6BXk-ebK2CAQ:96'},
    'owners_after': ['DRD7NfhpMmP2caqTJmn9FzxHBFnqNNHw4NU9zn2ZgVCX']}],
  'fulfillments': [{'fid': 0,
    'fulfillment': 'cf:4:UOKUj39h7TOpoMfJj6KFKooO36E7I2yJhW3msLcuNA7CAQXev_9e4hmjsdcB9R8qcB717y1_GrM5
    'input': {'cid': 0,
     'txid': 'ede453dc71ca2c716db3c78d9d8ac5b2b6e82f93d74c071c1323b45159785f8b'},
    'owners_before': ['6Sk1GUU6CL44znRHiY2ebRCpGknyeGx7WssgZBg4mS3f']}],
  'metadata': None,
  'operation': 'TRANSFER'},
 'version': 1}
```

Our goal is now to handcraft a payload equal to `fulfilled_transfer_tx` with the help of

- `json`: to serialize the transaction dictionary into a JSON formatted string.
- sha3: to hash the serialized transaction
- cryptoconditions: to create conditions and fulfillments

### 5.3.1 The Prepared Transaction

#### The transaction body

##### asset

```
In [68]: asset = {'id': asset_id}
```

#### metadata

```
In [69]: metadata = None
```

#### operation

```
In [70]: operation = 'TRANSFER'
```

#### conditions

```
In [71]: from cryptoconditions import Ed25519Fulfillment

In [72]: ed25519 = Ed25519Fulfillment(public_key=bob.verifying_key)

In [73]: condition = {
   ....:     'amount': 1,
   ....:     'cid': 0,
   ....:     'condition': {
   ....:         'details': ed25519.to_dict(),
   ....:         'uri': ed25519.condition_uri,
   ....:     },
   ....:     'owners_after': (bob.verifying_key,),
   ....: }
   ....:

In [74]: conditions = (condition,)
```

#### fulfillments

```
In [75]: fulfillment = {
   ....:     'fid': 0,
   ....:     'fulfillment': None,
   ....:     'input': {
   ....:         'txid': creation_tx['id'],
   ....:         'cid': 0,
   ....:     },
   ....:     'owners_before': (alice.verifying_key,)
   ....: }
   ....:

In [76]: fulfillments = (fulfillment,)
```

A few notes:

- The `input` field points to the condition that needs to be fulfilled;
- The `'fulfillment'` value is `None` as it will be set during the fulfillment step; and
- The `'owners_before'` field identifies the fulfiller(s).

Putting it all together:

```
In [77]: handcrafted_transfer_tx = {
   ....:     'transaction': {
```

```
   ....:         'asset': asset,
   ....:         'metadata': metadata,
   ....:         'operation': operation,
   ....:         'conditions': conditions,
   ....:         'fulfillments': fulfillments,
   ....:     },
   ....:     'version': 1,
   ....: }
   ....:

In [78]: handcrafted_transfer_tx
Out[78]:
{'transaction': {'asset': {'id': '3a1b1c72-43a8-439c-ad26-c762b8981b26'},
  'conditions': ({'amount': 1,
    'cid': 0,
    'condition': {'details': {'bitmask': 32,
      'public_key': 'DRD7NfhpMmP2caqTJmn9FzxHBFnqNNHw4NU9zn2ZgVCX',
      'signature': None,
      'type': 'fulfillment',
      'type_id': 4},
     'uri': 'cc:4:20:uH_gDwjq9W31pDEV5bXexuUyBhT8_xp6BXk-ebK2CAQ:96'},
    'owners_after': ('DRD7NfhpMmP2caqTJmn9FzxHBFnqNNHw4NU9zn2ZgVCX',)},),
  'fulfillments': ({'fid': 0,
    'fulfillment': None,
    'input': {'cid': 0,
     'txid': 'ede453dc71ca2c716db3c78d9d8ac5b2b6e82f93d74c071c1323b45159785f8b'},
    'owners_before': ('6Sk1GUU6CL44znRHiY2ebRCpGknyeGx7WssgZBg4mS3f',)},),
  'metadata': None,
  'operation': 'TRANSFER'},
 'version': 1}
```

We're missing the `id`, and we'll generate it, but before, let's recap how we've put all the code together to generate the above payload:

```
from cryptoconditions import Ed25519Fulfillment
from bigchaindb_driver.crypto import CryptoKeypair

bob = CryptoKeypair(
    verifying_key=bob.verifying_key,
    signing_key=bob.signing_key,
)

operation = 'TRANSFER'
asset = {'id': asset_id}
metadata = None

ed25519 = Ed25519Fulfillment(public_key=bob.verifying_key)

condition = {
    'amount': 1,
    'cid': 0,
    'condition': {
        'details': ed25519.to_dict(),
        'uri': ed25519.condition_uri,
    },
    'owners_after': (bob.verifying_key,),
}
conditions = (condition,)
```

```
fulfillment = {
    'fid': 0,
    'fulfillment': None,
    'input': {
        'txid': creation_tx['id'],
        'cid': 0,
    },
    'owners_before': (alice.verifying_key,)
}
fulfillments = (fulfillment,)

handcrafted_transfer_tx = {
    'transaction': {
        'asset': asset,
        'metadata': metadata,
        'operation': operation,
        'conditions': conditions,
        'fulfillments': fulfillments,
    },
    'version': 1,
}
```

#### id

```
In [79]: import json

In [80]: from sha3 import sha3_256

In [81]: json_str_tx = json.dumps(
   ....:     handcrafted_transfer_tx,
   ....:     sort_keys=True,
   ....:     separators=(',', ':'),
   ....:     ensure_ascii=False,
   ....: )
   ....:

In [82]: txid = sha3_256(json_str_tx.encode()).hexdigest()

In [83]: handcrafted_transfer_tx['id'] = txid
```

Compare this to the txid of the transaction generated via `prepare_transaction()`

```
In [84]: txid == prepared_transfer_tx['id']
Out[84]: True
```

You may observe that

```
In [85]: handcrafted_transfer_tx == prepared_transfer_tx
Out[85]: False
```

```
In [86]: from copy import deepcopy

In [87]: # back up

In [88]: prepared_transfer_tx_bk = deepcopy(prepared_transfer_tx)

In [89]: # set fulfillment to None
```

```
In [90]: prepared_transfer_tx['transaction']['fulfillments'][0]['fulfillment'] = None

In [91]: handcrafted_transfer_tx == prepared_transfer_tx
Out[91]: False
```

Are still not equal because we used tuples instead of lists.

```
In [92]: # serialize to json str

In [93]: json_str_handcrafted_tx = json.dumps(handcrafted_transfer_tx, sort_keys=True)

In [94]: json_str_prepared_tx = json.dumps(prepared_transfer_tx, sort_keys=True)
```

```
In [95]: json_str_handcrafted_tx == json_str_prepared_tx
Out[95]: True

In [96]: prepared_transfer_tx = prepared_transfer_tx_bk
```

The full handcrafted yet-to-be-fulfilled transaction payload:

```
In [97]: handcrafted_transfer_tx
Out[97]:
{'id': '04958329d67c941a9aa22d13f583939050315cd40df0ca76a7a95a3570b2ab49',
 'transaction': {'asset': {'id': '3a1b1c72-43a8-439c-ad26-c762b8981b26'},
  'conditions': ({'amount': 1,
    'cid': 0,
    'condition': {'details': {'bitmask': 32,
      'public_key': 'DRD7NfhpMmP2caqTJmn9FzxHBFnqNNHw4NU9zn2ZgVCX',
      'signature': None,
      'type': 'fulfillment',
      'type_id': 4},
     'uri': 'cc:4:20:uH_gDwjq9W31pDEV5bXexuUyBhT8_xp6BXk-ebK2CAQ:96'},
    'owners_after': ('DRD7NfhpMmP2caqTJmn9FzxHBFnqNNHw4NU9zn2ZgVCX',)},),
  'fulfillments': ({'fid': 0,
    'fulfillment': None,
    'input': {'cid': 0,
     'txid': 'ede453dc71ca2c716db3c78d9d8ac5b2b6e82f93d74c071c1323b45159785f8b'},
    'owners_before': ('6Sk1GUU6CL44znRHiY2ebRCpGknyeGx7WssgZBg4mS3f',)},),
  'metadata': None,
  'operation': 'TRANSFER'},
 'version': 1}
```

### 5.3.2 The Fulfilled Transaction

```
In [98]: from cryptoconditions.crypto import Ed25519SigningKey

In [99]: from bigchaindb_driver.offchain import fulfill_transaction

In [100]: fulfilled_transfer_tx = fulfill_transaction(
   .....:     prepared_transfer_tx,
   .....:     private_keys=alice.signing_key,
   .....: )
   .....:

In [101]: sk = Ed25519SigningKey(alice.signing_key)

In [102]: message = json.dumps(
```

```
   .....:     handcrafted_transfer_tx,
   .....:     sort_keys=True,
   .....:     separators=(',', ':'),
   .....:     ensure_ascii=False,
   .....: )
   .....:

In [103]: ed25519.sign(message.encode(), sk)

In [104]: fulfillment = ed25519.serialize_uri()

In [105]: handcrafted_transfer_tx['transaction']['fulfillments'][0]['fulfillment'] = fulfillment
```

Let's check this:

```
In [106]: fulfilled_transfer_tx['transaction']['fulfillments'][0]['fulfillment'] == fulfillment
Out[106]: True

In [107]: json.dumps(fulfilled_transfer_tx, sort_keys=True) == json.dumps(handcrafted_transfer_tx, so
\\\\\\\\\\\\\\\Out[107]: True
```

### 5.3.3 In a nutshell

```
import json

import sha3
import cryptoconditions

from bigchaindb_driver.crypto import generate_keypair


bob = generate_keypair()

operation = 'TRANSFER'
asset = {'id': asset_id}
metadata = None

ed25519 = cryptoconditions.Ed25519Fulfillment(public_key=bob.verifying_key)

condition = {
    'amount': 1,
    'cid': 0,
    'condition': {
        'details': ed25519.to_dict(),
        'uri': ed25519.condition_uri,
    },
    'owners_after': (bob.verifying_key,),
}
conditions = (condition,)

fulfillment = {
    'fid': 0,
    'fulfillment': None,
    'input': {
        'txid': creation_txid,
        'cid': 0,
    },
```

```
    'owners_before': (alice.verifying_key,)
}
fulfillments = (fulfillment,)

handcrafted_transfer_tx = {
    'transaction': {
        'asset': asset,
        'metadata': metadata,
        'operation': operation,
        'conditions': conditions,
        'fulfillments': fulfillments,
    },
    'version': 1,
}

json_str_tx = json.dumps(
    handcrafted_transfer_tx,
    sort_keys=True,
    separators=(',', ':'),
    ensure_ascii=False,
)

transfer_txid = sha3.sha3_256(json_str_tx.encode()).hexdigest()

handcrafted_transfer_tx['id'] = transfer_txid

sk = cryptoconditions.crypto.Ed25519SigningKey(alice.signing_key)

message = json.dumps(
    handcrafted_transfer_tx,
    sort_keys=True,
    separators=(',', ':'),
    ensure_ascii=False,
)

ed25519.sign(message.encode(), sk)

fulfillment = ed25519.serialize_uri()

handcrafted_transfer_tx['transaction']['fulfillments'][0]['fulfillment'] = fulfillment
```

Sending it over to a BigchainDB node:

```
from bigchaindb_driver import BigchainDB

bdb = BigchainDB('http://bdb-server:9984/api/v1')
returned_transfer_tx = bdb.transactions.send(handcrafted_transfer_tx)
```

A few checks:

```
>>> json.dumps(returned_transfer_tx, sort_keys=True) == json.dumps(handcrafted_transfer_tx, sort_keys
True
```

```
>>> bdb.transactions.status(transfer_txid)
{'status': 'valid'}
```

**Tip:** When checking for the status of a transaction, one should keep in mind tiny delays before a transaction reaches

a valid status.

## 5.4 Bicycle Sharing Revisited

Handcrafting the 'CREATE' transaction:

```
import json
from uuid import uuid4

import sha3
import cryptoconditions

from bigchaindb_driver.crypto import generate_keypair


bob, carly = generate_keypair(), generate_keypair()

asset_id = str(uuid4())
asset = {
    'divisible': True,
    'data': {
        'token_for': {
            'bicycle': {
                'manufacturer': 'bkfab',
                'serial_number': 'abcd1234',
            },
            'description': 'time share token. each token equals 1 hour of riding.'
        },
    },
    'refillable': False,
    'updatable': False,
    'id': asset_id,
}

# CRYPTO-CONDITIONS: instantiate an Ed25519 crypto-condition for carly
ed25519 = cryptoconditions.Ed25519Fulfillment(public_key=carly.verifying_key)

# CRYPTO-CONDITIONS: generate the condition uri
condition_uri = ed25519.condition.serialize_uri()

# CRYPTO-CONDITIONS: get the unsigned fulfillment dictionary (details)
unsigned_fulfillment_dict = ed25519.to_dict()

condition = {
    'amount': 10,
    'cid': 0,
    'condition': {
        'details': unsigned_fulfillment_dict,
        'uri': condition_uri,
    },
    'owners_after': (carly.verifying_key,),
}

fulfillment = {
    'fid': 0,
    'fulfillment': None,
```

```
    'input': None,
    'owners_before': (bob.verifying_key,)
}

token_creation_tx = {
    'transaction': {
        'asset': asset,
        'metadata': None,
        'operation': 'CREATE',
        'conditions': (condition,),
        'fulfillments': (fulfillment,),
    },
    'version': 1,
}

# JSON: serialize the id-less transaction to a json formatted string
json_str_tx = json.dumps(
    token_creation_tx,
    sort_keys=True,
    separators=(',', ':'),
    ensure_ascii=False,
)

# SHA3: hash the serialized id-less transaction to generate the id
creation_txid = sha3.sha3_256(json_str_tx.encode()).hexdigest()

# add the id
token_creation_tx['id'] = creation_txid

# JSON: serialize the transaction-with-id to a json formatted string
message = json.dumps(
    token_creation_tx,
    sort_keys=True,
    separators=(',', ':'),
    ensure_ascii=False,
)

# CRYPTO-CONDITIONS: sign the serialized transaction-with-id
ed25519.sign(message.encode(),
             cryptoconditions.crypto.Ed25519SigningKey(bob.signing_key))

# CRYPTO-CONDITIONS: generate the fulfillment uri
fulfillment_uri = ed25519.serialize_uri()

# add the fulfillment uri (signature)
token_creation_tx['transaction']['fulfillments'][0]['fulfillment'] = fulfillment_uri
```

Sending it over to a BigchainDB node:

```
from bigchaindb_driver import BigchainDB

bdb = BigchainDB('http://bdb-server:9984/api/v1')
returned_creation_tx = bdb.transactions.send(token_creation_tx)
```

A few checks:

```
>>> json.dumps(returned_creation_tx, sort_keys=True) == json.dumps(token_creation_tx, sort_keys=True)
True
```

```
>>> token_creation_tx['transaction']['fulfillments'][0]['owners_before'][0] == bob.verifying_key
True

>>> token_creation_tx['transaction']['conditions'][0]['owners_after'][0] == carly.verifying_key
True

>>> token_creation_tx['transaction']['conditions'][0]['amount'] == 10
True
```

```
>>> bdb.transactions.status(creation_txid)
{'status': 'valid'}
```

**Tip:** When checking for the status of a transaction, one should keep in mind tiny delays before a transaction reaches a valid status.

Now Carly wants to ride the bicycle for 2 hours so she needs to send 2 tokens to Bob:

```
# CRYPTO-CONDITIONS: instantiate an Ed25519 crypto-condition for carly
bob_ed25519 = cryptoconditions.Ed25519Fulfillment(public_key=bob.verifying_key)

# CRYPTO-CONDITIONS: instantiate an Ed25519 crypto-condition for carly
carly_ed25519 = cryptoconditions.Ed25519Fulfillment(public_key=carly.verifying_key)

# CRYPTO-CONDITIONS: generate the condition uris
bob_condition_uri = bob_ed25519.condition.serialize_uri()
carly_condition_uri = carly_ed25519.condition.serialize_uri()

# CRYPTO-CONDITIONS: get the unsigned fulfillment dictionary (details)
bob_unsigned_fulfillment_dict = bob_ed25519.to_dict()
carly_unsigned_fulfillment_dict = carly_ed25519.to_dict()

bob_condition = {
    'amount': 2,
    'cid': 0,
    'condition': {
        'details': bob_unsigned_fulfillment_dict,
        'uri': bob_condition_uri,
    },
    'owners_after': (bob.verifying_key,),
}
carly_condition = {
    'amount': 8,
    'cid': 1,
    'condition': {
        'details': carly_unsigned_fulfillment_dict,
        'uri': carly_condition_uri,
    },
    'owners_after': (carly.verifying_key,),
}

fulfillment = {
    'fid': 0,
    'fulfillment': None,
    'input': {
        'txid': token_creation_tx['id'],
        'cid': 0,
    },
```

```
    'owners_before': (carly.verifying_key,)
}

token_transfer_tx = {
    'transaction': {
        'asset': {'id': asset_id},
        'metadata': None,
        'operation': 'TRANSFER',
        'conditions': (bob_condition, carly_condition),
        'fulfillments': (fulfillment,),
    },
    'version': 1,
}

# JSON: serialize the id-less transaction to a json formatted string
json_str_tx = json.dumps(
    token_transfer_tx,
    sort_keys=True,
    separators=(',', ':'),
    ensure_ascii=False,
)

# SHA3: hash the serialized id-less transaction to generate the id
transfer_txid = sha3.sha3_256(json_str_tx.encode()).hexdigest()

# add the id
token_transfer_tx['id'] = transfer_txid

# JSON: serialize the transaction-with-id to a json formatted string
message = json.dumps(
    token_transfer_tx,
    sort_keys=True,
    separators=(',', ':'),
    ensure_ascii=False,
)

# CRYPTO-CONDITIONS: sign the serialized transaction-with-id for bob
carly_ed25519.sign(message.encode(),
                   cryptoconditions.crypto.Ed25519SigningKey(carly.signing_key))

# CRYPTO-CONDITIONS: generate bob's fulfillment uri
fulfillment_uri = carly_ed25519.serialize_uri()

# add bob's fulfillment uri (signature)
token_transfer_tx['transaction']['fulfillments'][0]['fulfillment'] = fulfillment_uri
```

Sending it over to a BigchainDB node:

```
bdb = BigchainDB('http://bdb-server:9984/api/v1')
returned_transfer_tx = bdb.transactions.send(token_transfer_tx)
```

A few checks:

```
>>> json.dumps(returned_transfer_tx, sort_keys=True) == json.dumps(token_transfer_tx, sort_keys=True)
True

>>> token_transfer_tx['transaction']['fulfillments'][0]['owners_before'][0] == carly.verifying_key
True
```

```
>>> bdb.transactions.status(creation_txid)
{'status': 'valid'}
```

**Tip:** When checking for the status of a transaction, one should keep in mind tiny delays before a transaction reaches a valid status.

## 5.5 Multiple Owners Revisited

### 5.5.1 Walkthrough

We'll re-use the example, to compare our work.

Say `alice` and `bob` own a car together:

```
In [108]: from bigchaindb_driver.crypto import generate_keypair

In [109]: from bigchaindb_driver import offchain

In [110]: alice, bob = generate_keypair(), generate_keypair()

In [111]: car_asset = {'data': {'car': {'vin': '5YJRE11B781000196'}}}

In [112]: car_creation_tx = offchain.prepare_transaction(
   .....:     operation='CREATE',
   .....:     owners_before=alice.verifying_key,
   .....:     owners_after=(alice.verifying_key, bob.verifying_key),
   .....:     asset=car_asset,
   .....: )
   .....:

In [113]: signed_car_creation_tx = offchain.fulfill_transaction(
   .....:     car_creation_tx,
   .....:     private_keys=alice.signing_key,
   .....: )
   .....:

In [114]: signed_car_creation_tx
Out[114]:
{'id': '0b19aa871823e18e72a2beb4c2c12be1a64f85d5100ab7f089ff39543e248f01',
 'transaction': {'asset': {'data': {'car': {'vin': '5YJRE11B781000196'}}},
   'divisible': False,
   'id': '99731f4e-4425-43ac-93a5-e7fc8d649b10',
   'refillable': False,
   'updatable': False},
  'conditions': [{'amount': 1,
    'cid': 0,
    'condition': {'details': {'bitmask': 41,
      'subfulfillments': [{'bitmask': 32,
        'public_key': '895Syw7iDfaDVnc7pg7dKYmHaX8vyEzx7tgonPjjqdwp',
        'signature': None,
        'type': 'fulfillment',
        'type_id': 4,
        'weight': 1},
       {'bitmask': 32,
```

```
        'public_key': 'BsF3zdUfd4HeaTPVQTjTVaUsnayMnDYGSKSDkP1CJfdz',
        'signature': None,
        'type': 'fulfillment',
        'type_id': 4,
        'weight': 1}],
      'threshold': 2,
      'type': 'fulfillment',
      'type_id': 2},
     'uri': 'cc:2:29:cMXwPrS7NYEIff6BrVp194WUPqRg3fSh77CK3_-krzQ:206'},
    'owners_after': ['895Syw7iDfaDVnc7pg7dKYmHaX8vyEzx7tgonPjjqdwp',
     'BsF3zdUfd4HeaTPVQTjTVaUsnayMnDYGSKSDkP1CJfdz']}],
  'fulfillments': [{'fid': 0,
    'fulfillment': 'cf:4:ahN7-cYAAlV4vLwlQghpfruWJrDe7Tj52fTgGawOr9uRTTj7FubFNnJ9RQ3nmfVv218lmgtqe-wo
    'input': None,
    'owners_before': ['895Syw7iDfaDVnc7pg7dKYmHaX8vyEzx7tgonPjjqdwp']}],
  'metadata': None,
  'operation': 'CREATE'},
 'version': 1}
```

```
sent_car_tx = bdb.transactions.send(signed_car_creation_tx)
```

One day, `alice` and `bob`, having figured out how to teleport themselves, and realizing they no longer need their car, wish to transfer the ownership of their car over to `carol`:

```
In [115]: carol = generate_keypair()

In [116]: cid = 0

In [117]: condition = signed_car_creation_tx['transaction']['conditions'][cid]

In [118]: input_ = {
   .....:     'fulfillment': condition['condition']['details'],
   .....:     'input': {
   .....:         'cid': cid,
   .....:         'txid': signed_car_creation_tx['id'],
   .....:     },
   .....:     'owners_before': condition['owners_after'],
   .....: }
   .....:

In [119]: asset = signed_car_creation_tx['transaction']['asset']

In [120]: car_transfer_tx = offchain.prepare_transaction(
   .....:     operation='TRANSFER',
   .....:     owners_after=carol.verifying_key,
   .....:     asset=asset,
   .....:     inputs=input_,
   .....: )
   .....:

In [121]: signed_car_transfer_tx = offchain.fulfill_transaction(
   .....:     car_transfer_tx, private_keys=[alice.signing_key, bob.signing_key]
   .....: )
   .....:

In [122]: signed_car_transfer_tx
Out[122]:
{'id': '63c419696526c10dbf0304d61670911afbb32664839de67f790b880f8e8b84c9',
```

```
 'transaction': {'asset': {'id': '99731f4e-4425-43ac-93a5-e7fc8d649b10'},
  'conditions': [{'amount': 1,
    'cid': 0,
    'condition': {'details': {'bitmask': 32,
      'public_key': 'EvBGQVNKpt5wr8cuXAmRJeU7B499p3cz8PkQaafbF9XX',
      'signature': None,
      'type': 'fulfillment',
      'type_id': 4},
     'uri': 'cc:4:20:zsdR_b6BKWtFowhbNCLyeeQDz5enh2zifwDF6VyQjdo:96'},
    'owners_after': ['EvBGQVNKpt5wr8cuXAmRJeU7B499p3cz8PkQaafbF9XX']}],
  'fulfillments': [{'fid': 0,
    'fulfillment': 'cf:2:AQIBAgEBYwAEYGoTe_nGAAJVeLy8JUIIaX67liaw3u04-dn04BmsDq_b0_Faa_xuhx4UebaToyJ2
    'input': {'cid': 0,
     'txid': '0b19aa871823e18e72a2beb4c2c12be1a64f85d5100ab7f089ff39543e248f01'},
    'owners_before': ['895Syw7iDfaDVnc7pg7dKYmHaX8vyEzx7tgonPjjqdwp',
     'BsF3zdUfd4HeaTPVQTjTVaUsnayMnDYGSKSDkP1CJfdz']}],
  'metadata': None,
  'operation': 'TRANSFER'},
 'version': 1}
```

Sending the transaction to a BigchainDB node:

```
sent_car_transfer_tx = bdb.transactions.send(signed_car_transfer_tx)
```

In order to do this manually, let's first import the necessary tools (json, sha3, and cryptoconditions):

```
In [123]: import json

In [124]: from sha3 import sha3_256

In [125]: from cryptoconditions import Ed25519Fulfillment, ThresholdSha256Fulfillment

In [126]: from cryptoconditions.crypto import Ed25519SigningKey
```

Create the asset, setting all values:

```
In [127]: car_asset_id = signed_car_creation_tx['transaction']['asset']['id']

In [128]: car_asset = {
   .....:     'data': {'car': {'vin': '5YJRE11B781000196'}},
   .....:     'divisible': False,
   .....:     'refillable': False,
   .....:     'updatable': False,
   .....:     'id': car_asset_id,
   .....: }
   .....:
```

Generate the condition:

```
In [129]: alice_ed25519 = Ed25519Fulfillment(public_key=alice.verifying_key)

In [130]: bob_ed25519 = Ed25519Fulfillment(public_key=bob.verifying_key)

In [131]: threshold_sha256 = ThresholdSha256Fulfillment(threshold=2)

In [132]: threshold_sha256.add_subfulfillment(alice_ed25519)

In [133]: threshold_sha256.add_subfulfillment(bob_ed25519)

In [134]: unsigned_subfulfillments_dict = threshold_sha256.to_dict()
```

```
In [135]: condition_uri = threshold_sha256.condition.serialize_uri()

In [136]: condition = {
   .....:     'amount': 1,
   .....:     'cid': 0,
   .....:     'condition': {
   .....:         'details': unsigned_subfulfillments_dict,
   .....:         'uri': condition_uri,
   .....:     },
   .....:     'owners_after': (alice.verifying_key, bob.verifying_key),
   .....: }
   .....:
```

**Tip:** The condition `uri` could have been generated in a slightly different way, which may be more intuitive to you. You can think of the threshold condition containing sub conditions:

```
In [137]: alt_threshold_sha256 = ThresholdSha256Fulfillment(threshold=2)

In [138]: alt_threshold_sha256.add_subcondition(alice_ed25519.condition)

In [139]: alt_threshold_sha256.add_subcondition(bob_ed25519.condition)

In [140]: alt_threshold_sha256.condition.serialize_uri() == condition_uri
Out[140]: True
```

The `details` on the other hand holds the associated fulfillments not yet fulfilled.

The yet to be fulfilled fulfillment:

```
In [141]: fulfillment = {
   .....:     'fid': 0,
   .....:     'fulfillment': None,
   .....:     'input': None,
   .....:     'owners_before': (alice.verifying_key,),
   .....: }
   .....:
```

Craft the payload:

```
In [142]: handcrafted_car_creation_tx = {
   .....:     'transaction': {
   .....:         'asset': car_asset,
   .....:         'metadata': None,
   .....:         'operation': 'CREATE',
   .....:         'conditions': (condition,),
   .....:         'fulfillments': (fulfillment,),
   .....:     },
   .....:     'version': 1,
   .....: }
   .....:
```

Generate the id, by hashing the encoded json formatted string representation of the transaction:

```
In [143]: json_str_tx = json.dumps(
   .....:     handcrafted_car_creation_tx,
   .....:     sort_keys=True,
   .....:     separators=(',', ':'),
```

```
   .....:     ensure_ascii=False,
   .....: )
   .....:

In [144]: car_creation_txid = sha3_256(json_str_tx.encode()).hexdigest()

In [145]: handcrafted_car_creation_tx['id'] = car_creation_txid
```

Let's make sure our txid is the same as the one provided by the driver:

```
In [146]: handcrafted_car_creation_tx['id'] == car_creation_tx['id']
Out[146]: True
```

Sign the transaction:

```
In [147]: message = json.dumps(
   .....:     handcrafted_car_creation_tx,
   .....:     sort_keys=True,
   .....:     separators=(',', ':'),
   .....:     ensure_ascii=False,
   .....: )
   .....:

In [148]: alice_ed25519.sign(message.encode(), Ed25519SigningKey(alice.signing_key))

In [149]: fulfillment_uri = alice_ed25519.serialize_uri()

In [150]: handcrafted_car_creation_tx['transaction']['fulfillments'][0]['fulfillment'] = fulfillment_
```

Compare our signed CREATE transaction with the driver's:

```
In [151]: (json.dumps(handcrafted_car_creation_tx, sort_keys=True) ==
   .....:  json.dumps(signed_car_creation_tx, sort_keys=True))
   .....:
Out[151]: True
```

The transfer:

```
In [152]: alice_ed25519 = Ed25519Fulfillment(public_key=alice.verifying_key)

In [153]: bob_ed25519 = Ed25519Fulfillment(public_key=bob.verifying_key)

In [154]: carol_ed25519 = Ed25519Fulfillment(public_key=carol.verifying_key)

In [155]: unsigned_fulfillments_dict = carol_ed25519.to_dict()

In [156]: condition_uri = carol_ed25519.condition.serialize_uri()

In [157]: condition = {
   .....:     'amount': 1,
   .....:     'cid': 0,
   .....:     'condition': {
   .....:         'details': unsigned_fulfillments_dict,
   .....:         'uri': condition_uri,
   .....:     },
   .....:     'owners_after': (carol.verifying_key,),
   .....: }
   .....:
```

The yet to be fulfilled fulfillments:

```
In [158]: fulfillment = {
   .....:     'fid': 0,
   .....:     'fulfillment': None,
   .....:     'input': {
   .....:         'txid': handcrafted_car_creation_tx['id'],
   .....:         'cid': 0,
   .....:     },
   .....:     'owners_before': (alice.verifying_key, bob.verifying_key),
   .....: }
   .....:
```

Craft the payload:

```
In [159]: handcrafted_car_transfer_tx = {
   .....:     'transaction': {
   .....:         'asset': {'id': car_asset_id},
   .....:         'metadata': None,
   .....:         'operation': 'TRANSFER',
   .....:         'conditions': (condition,),
   .....:         'fulfillments': (fulfillment,),
   .....:     },
   .....:     'version': 1,
   .....: }
   .....:
```

Generate the id, by hashing the encoded json formatted string representation of the transaction:

```
In [160]: json_str_tx = json.dumps(
   .....:     handcrafted_car_transfer_tx,
   .....:     sort_keys=True,
   .....:     separators=(',', ':'),
   .....:     ensure_ascii=False,
   .....: )
   .....:

In [161]: car_transfer_txid = sha3_256(json_str_tx.encode()).hexdigest()

In [162]: handcrafted_car_transfer_tx['id'] = car_transfer_txid
```

Let's make sure our txid is the same as the one provided by the driver:

```
In [163]: handcrafted_car_transfer_tx['id'] == car_transfer_tx['id']
Out[163]: True
```

Sign the transaction:

```
In [164]: message = json.dumps(
   .....:     handcrafted_car_transfer_tx,
   .....:     sort_keys=True,
   .....:     separators=(',', ':'),
   .....:     ensure_ascii=False,
   .....: )
   .....:

In [165]: alice_sk = Ed25519SigningKey(alice.signing_key)

In [166]: bob_sk = Ed25519SigningKey(bob.signing_key)

In [167]: threshold_sha256 = ThresholdSha256Fulfillment(threshold=2)
```

```
In [168]: threshold_sha256.add_subfulfillment(alice_ed25519)

In [169]: threshold_sha256.add_subfulfillment(bob_ed25519)

In [170]: alice_condition = threshold_sha256.get_subcondition_from_vk(alice.verifying_key)[0]

In [171]: bob_condition = threshold_sha256.get_subcondition_from_vk(bob.verifying_key)[0]

In [172]: alice_condition.sign(message.encode(), private_key=alice_sk)

In [173]: bob_condition.sign(message.encode(), private_key=bob_sk)

In [174]: fulfillment_uri = threshold_sha256.serialize_uri()

In [175]: handcrafted_car_transfer_tx['transaction']['fulfillments'][0]['fulfillment'] = fulfillment_
```

Compare our signed TRANSFER transaction with the driver's:

```
In [176]: (json.dumps(handcrafted_car_transfer_tx, sort_keys=True) ==
   .....:  json.dumps(signed_car_transfer_tx, sort_keys=True))
   .....:
Out[176]: True
```

### 5.5.2 In a nutshell

#### Handcrafting the 'CREATE' transaction

```
import json

import sha3
import cryptoconditions

from bigchaindb_driver.crypto import generate_keypair


car_asset = {
    'data': {
        'car': {
            'vin': '5YJRE11B781000196',
        },
    },
    'divisible': False,
     'refillable': False,
     'updatable': False,
     'id': '5YJRE11B781000196',
}

alice, bob = generate_keypair(), generate_keypair()

# CRYPTO-CONDITIONS: instantiate an Ed25519 crypto-condition for alice
alice_ed25519 = cryptoconditions.Ed25519Fulfillment(public_key=alice.verifying_key)

# CRYPTO-CONDITIONS: instantiate an Ed25519 crypto-condition for bob
bob_ed25519 = cryptoconditions.Ed25519Fulfillment(public_key=bob.verifying_key)

# CRYPTO-CONDITIONS: instantiate a threshold SHA 256 crypto-condition
```

```
threshold_sha256 = cryptoconditions.ThresholdSha256Fulfillment(threshold=2)

# CRYPTO-CONDITIONS: add alice ed25519 to the threshold SHA 256 condition
threshold_sha256.add_subfulfillment(alice_ed25519)

# CRYPTO-CONDITIONS: add bob ed25519 to the threshold SHA 256 condition
threshold_sha256.add_subfulfillment(bob_ed25519)

# CRYPTO-CONDITIONS: get the unsigned fulfillment dictionary (details)
unsigned_subfulfillments_dict = threshold_sha256.to_dict()

# CRYPTO-CONDITIONS: generate the condition uri
condition_uri = threshold_sha256.condition.serialize_uri()

condition = {
    'amount': 1,
    'cid': 0,
    'condition': {
        'details': unsigned_subfulfillments_dict,
        'uri': threshold_sha256.condition_uri,
    },
    'owners_after': (alice.verifying_key, bob.verifying_key),
}

# The yet to be fulfilled fulfillment:
fulfillment = {
    'fid': 0,
    'fulfillment': None,
    'input': None,
    'owners_before': (alice.verifying_key,),
}

# Craft the payload:
handcrafted_car_creation_tx = {
    'transaction': {
        'asset': car_asset,
        'metadata': None,
        'operation': 'CREATE',
        'conditions': (condition,),
        'fulfillments': (fulfillment,),
    },
    'version': 1,
}

# JSON: serialize the id-less transaction to a json formatted string
# Generate the id, by hashing the encoded json formatted string representation of
# the transaction:
json_str_tx = json.dumps(
    handcrafted_car_creation_tx,
    sort_keys=True,
    separators=(',', ':'),
    ensure_ascii=False,
)

# SHA3: hash the serialized id-less transaction to generate the id
car_creation_txid = sha3.sha3_256(json_str_tx.encode()).hexdigest()

# add the id
```

```
handcrafted_car_creation_tx['id'] = car_creation_txid

# JSON: serialize the transaction-with-id to a json formatted string
message = json.dumps(
    handcrafted_car_creation_tx,
    sort_keys=True,
    separators=(',', ':'),
    ensure_ascii=False,
)

# CRYPTO-CONDITIONS: sign the serialized transaction-with-id
alice_ed25519.sign(message.encode(),
                   cryptoconditions.crypto.Ed25519SigningKey(alice.signing_key))

# CRYPTO-CONDITIONS: generate the fulfillment uri
fulfillment_uri = alice_ed25519.serialize_uri()

# add the fulfillment uri (signature)
handcrafted_car_creation_tx['transaction']['fulfillments'][0]['fulfillment'] = fulfillment_uri
```

Sending it over to a BigchainDB node:

```
from bigchaindb_driver import BigchainDB

bdb = BigchainDB('http://bdb-server:9984/api/v1')
returned_car_creation_tx = bdb.transactions.send(handcrafted_car_creation_tx)
```

Wait for some nano seconds, and check the status:

```
>>> bdb.transactions.status(returned_car_creation_tx['id'])
{'status': 'valid'}
```

### Handcrafting the 'TRANSFER' transaction

```
carol = generate_keypair()

alice_ed25519 = cryptoconditions.Ed25519Fulfillment(public_key=alice.verifying_key)

bob_ed25519 = cryptoconditions.Ed25519Fulfillment(public_key=bob.verifying_key)

carol_ed25519 = cryptoconditions.Ed25519Fulfillment(public_key=carol.verifying_key)

unsigned_fulfillments_dict = carol_ed25519.to_dict()

condition_uri = carol_ed25519.condition.serialize_uri()

condition = {
    'amount': 1,
    'cid': 0,
    'condition': {
        'details': unsigned_fulfillments_dict,
        'uri': condition_uri,
    },
    'owners_after': (carol.verifying_key,),
}

# The yet to be fulfilled fulfillments:
```

```
fulfillment = {
    'fid': 0,
    'fulfillment': None,
    'input': {
        'txid': handcrafted_car_creation_tx['id'],
        'cid': 0,
    },
    'owners_before': (alice.verifying_key, bob.verifying_key),
}

# Craft the payload:
handcrafted_car_transfer_tx = {
    'transaction': {
        'asset': {'id': car_asset['id']},
        'metadata': None,
        'operation': 'TRANSFER',
        'conditions': (condition,),
        'fulfillments': (fulfillment,),
    },
    'version': 1,
}

# Generate the id, by hashing the encoded json formatted string
# representation of the transaction:
json_str_tx = json.dumps(
    handcrafted_car_transfer_tx,
    sort_keys=True,
    separators=(',', ':'),
    ensure_ascii=False,
)

car_transfer_txid = sha3.sha3_256(json_str_tx.encode()).hexdigest()

handcrafted_car_transfer_tx['id'] = car_transfer_txid

# Sign the transaction:
message = json.dumps(
    handcrafted_car_transfer_tx,
    sort_keys=True,
    separators=(',', ':'),
    ensure_ascii=False,
)

alice_sk = cryptoconditions.crypto.Ed25519SigningKey(alice.signing_key)

bob_sk = cryptoconditions.crypto.Ed25519SigningKey(bob.signing_key)

threshold_sha256 = cryptoconditions.ThresholdSha256Fulfillment(threshold=2)

threshold_sha256.add_subfulfillment(alice_ed25519)

threshold_sha256.add_subfulfillment(bob_ed25519)

alice_condition = threshold_sha256.get_subcondition_from_vk(alice.verifying_key)[0]

bob_condition = threshold_sha256.get_subcondition_from_vk(bob.verifying_key)[0]

alice_condition.sign(message.encode(), private_key=alice_sk)
```

```
bob_condition.sign(message.encode(), private_key=bob_sk)

fulfillment_uri = threshold_sha256.serialize_uri()

handcrafted_car_transfer_tx['transaction']['fulfillments'][0]['fulfillment'] = fulfillment_uri
```

Sending it over to a BigchainDB node:

```
bdb = BigchainDB('http://bdb-server:9984/api/v1')
returned_car_transfer_tx = bdb.transactions.send(handcrafted_car_transfer_tx)
```

Wait for some nano seconds, and check the status:

```
>>> bdb.transactions.status(returned_car_transfer_tx['id'])
{'status': 'valid'}
```

CHAPTER 6

# Upgrading

## 6.1 Upgrading Using pip

If you installed the BigchainDB Python Driver using `pip install bigchaindb_driver`, then you can upgrade it to the latest version using:

```
pip install --upgrade bigchaindb_driver
```

CHAPTER 7

# Library Reference

## 7.1 driver

**class** bigchaindb_driver.**BigchainDB**(**nodes*, *transport_class=<class 'bigchaindb_driver.transport.Transport'>*)

BigchainDB driver class.

A *BigchainDB* driver is initialized with a keypair and is able to create, sign, and submit transactions to a Node in the Federation. At the moment, a BigchainDB driver instance is bounded to a specific `node` in the Federation. In the future, a *BigchainDB* driver instance might connect to `>1` nodes.

**__init__**(**nodes*, *transport_class=<class 'bigchaindb_driver.transport.Transport'>*)

Initialize a *BigchainDB* driver instance.

If a `verifying_key` or `signing_key` are given, this instance will be bound to the keys and applied them as defaults whenever a verifying and/or signing key are needed.

**Parameters**

- ***nodes** (*str*) – BigchainDB nodes to connect to. Currently, the full URL must be given. In the absence of any node, the default of the `transport_class` will be used, e.g.: `'http://localhost:9984/api/v1'`.
- **transport_class** – Optional transport class to use. Defaults to *Transport*.

**nodes**

`tuple` of `str`: URLs of connected nodes.

**transactions**

*TransactionsEndpoint*: Exposes functionalities of the *'/transactions'* endpoint.

**transport**

*Transport*: Object responsible for forwarding requests to a *Connection* instance (node).

**class** bigchaindb_driver.driver.**TransactionsEndpoint**(*driver*)

Endpoint for transactions.

**path**

*str*

The path of the endpoint.

**__init__**(*driver*)

Initializes an instance of *NamespacedDriver* with the given driver instance.

**Parameters** **driver** (BigchainDB) – Instance of `BigchainDB`.

**static fulfill**(*transaction*, *private_keys*)
Fulfills the given transaction.

**Parameters**

- **transaction** (*dict*) – The transaction to be fulfilled.
- **private_keys** (str | list | tuple) – One or more private keys to be used for fulfilling the transaction.

**Returns** The fulfilled transaction payload, ready to be sent to a BigchainDB federation.

**Return type** dict

**Raises** *MissingSigningKeyError* – If a private key, (aka signing key), is missing.

**static prepare**(*, *operation='CREATE'*, *owners_before=None*, *owners_after=None*, *asset=None*, *metadata=None*, *inputs=None*)
Prepares a transaction payload, ready to be fulfilled.

**Parameters**

- **operation** (*str*) – The operation to perform. Must be 'CREATE' or 'TRANSFER'. Case insensitive. Defaults to 'CREATE'.
- **owners_before** (list | tuple | str, optional) – One or more public keys representing the issuer(s) of the asset being created. Only applies for 'CREATE' operations. Defaults to None.
- **owners_after** (list | tuple | str, optional) – One or more public keys representing the new owner(s) of the asset being created or transferred. Defaults to None.
- **asset** (dict, optional) – The asset being created or transferred. MUST be supplied for 'TRANSFER' operations. Defaults to None.
- **metadata** (dict, optional) – Metadata associated with the transaction. Defaults to None.
- **inputs** (dict | list | tuple, optional) – One or more inputs holding the condition(s) that this transaction intends to fulfill. Each input is expected to be a dict. Only applies to, and MUST be supplied for, 'TRANSFER' operations.

**Returns** The prepared transaction.

**Return type** dict

**Raises** *BigchaindbException* – If operation is not 'CREATE' or 'TRANSFER'.

---

**Important: CREATE operations**

•owners_before MUST be set.

•owners_after, asset, and metadata MAY be set.

•The argument inputs is ignored.

•If owners_after is not given, or evaluates to False, it will be set equal to owners_before:

```
if not owners_after:
    owners_after = owners_before
```

**TRANSFER operations**

•owners_after, asset, and inputs MUST be set.

- `metadata` MAY be set.
- The argument `owners_before` is ignored.

---

**retrieve**(*txid*)
Retrieves the transaction with the given id.

**Parameters** **txid** (*str*) – Id of the transaction to retrieve.

**Returns** The transaction with the given id.

**Return type** dict

**send**(*transaction*)
Submit a transaction to the Federation.

**Parameters** **transaction** (*dict*) – the transaction to be sent to the Federation node(s).

**Returns** The transaction sent to the Federation node(s).

**Return type** dict

**status**(*txid*)
Retrieves the status of the transaction with the given id.

**Parameters** **txid** (*str*) – Id of the transaction to retrieve the status for.

**Returns** A dict containing a 'status' item for the transaction.

**Return type** dict

**class** bigchaindb_driver.driver.**NamespacedDriver**(*driver*)
Base class for creating endpoints (namespaced objects) that can be added under the `BigchainDB` driver.

**__init__**(*driver*)
Initializes an instance of *NamespacedDriver* with the given driver instance.

**Parameters** **driver** (BigchainDB) – Instance of `BigchainDB`.

## 7.2 offchain

Module for operations that can be performed "offchain", meaning without a connection to one or more BigchainDB federation nodes.

bigchaindb_driver.offchain.**prepare_transaction**(*, *operation='CREATE'*, *owners_before=None*, *owners_after=None*, *asset=None*, *metadata=None*, *inputs=None*)
Prepares a transaction payload, ready to be fulfilled. Depending on the value of `operation` simply dispatches to either *prepare_create_transaction()* or *prepare_transfer_transaction()*.

**Parameters**

- **operation** (*str*) – The operation to perform. Must be `'CREATE'` or `'TRANSFER'`. Case insensitive. Defaults to `'CREATE'`.
- **owners_before** (`list` | `tuple` | `str`, optional) – One or more public keys representing the issuer(s) of the asset being created. Only applies for `'CREATE'` operations. Defaults to `None`.
- **owners_after** (`list` | `tuple` | `str`, optional) – One or more public keys representing the new owner(s) of the asset being created or transferred. Defaults to `None`.

- **asset** (dict, optional) – The asset being created or transferred. MUST be supplied for 'TRANSFER' operations. Defaults to None.
- **metadata** (dict, optional) – Metadata associated with the transaction. Defaults to None.
- **inputs** (dict | list | tuple, optional) – One or more inputs holding the condition(s) that this transaction intends to fulfill. Each input is expected to be a dict. Only applies to, and MUST be supplied for, 'TRANSFER' operations.

**Returns** The prepared transaction.

**Return type** dict

**Raises** *BigchaindbException* – If operation is not 'CREATE' or 'TRANSFER'.

---

**Important: CREATE operations**

•owners_before MUST be set.

•owners_after, asset, and metadata MAY be set.

•The argument inputs is ignored.

•If owners_after is not given, or evaluates to False, it will be set equal to owners_before:

```
if not owners_after:
    owners_after = owners_before
```

**TRANSFER operations**

•owners_after, asset, and inputs MUST be set.

•metadata MAY be set.

•The argument owners_before is ignored.

---

bigchaindb_driver.offchain.**prepare_create_transaction**(*, *owners_before*, *owners_after=None*, *asset=None*, *metadata=None*)

Prepares a "CREATE" transaction payload, ready to be fulfilled.

**Parameters**

- **owners_before** (list | tuple | str) – One or more public keys representing the issuer(s) of the asset being created.
- **owners_after** (list | tuple | str, optional) – One or more public keys representing the new owner(s) of the asset being created. Defaults to None.
- **asset** (dict, optional) – The asset being created. Defaults to None.
- **metadata** (dict, optional) – Metadata associated with the transaction. Defaults to None.

**Returns** The prepared "CREATE" transaction.

**Return type** dict

---

**Important:** If owners_after is not given, or evaluates to False, it will be set equal to owners_before:

```
if not owners_after:
    owners_after = owners_before
```

---

bigchaindb_driver.offchain.**prepare_transfer_transaction**(*, *inputs*, *owners_after*, *asset*, *metadata=None*)

Prepares a "TRANSFER" transaction payload, ready to be fulfilled.

**Parameters**

- **inputs** (dict | list | tuple) – One or more inputs holding the condition(s) that this transaction intends to fulfill. Each input is expected to be a dict.
- **owners_after** (str | list | tuple) – One or more public keys representing the new owner(s) of the asset being transferred.
- **asset** (dict) – The asset being transferred.
- **metadata** (dict) – Metadata associated with the transaction. Defaults to None.

**Returns** The prepared "TRANSFER" transaction.

**Return type** dict

### Example

In case it may not be clear what an input should look like, say Alice (public key: '3Cxh1eKZk3Wp9KGBWFS7iVde465UvqUKnEqTg2MW4wNf') wishes to transfer an asset over to Bob (public key: 'EcRawy3Y22eAUSS94vLF8BVJi62wbqbD9iSUSUNU9wAA'). Let the asset creation transaction payload be denoted by tx:

```
# noqa E501
>>> tx
{'id': '57cff2b9490468bdb6d4767a1b07905fdbe18d638d9c7783f639b4b2bc165c39',
 'transaction': {'asset': {'data': {'msg': 'Hello BigchainDB!'},
   'divisible': False,
   'id': 'd04b05de-774c-4f81-9e54-6c19ed3cd18d',
   'refillable': False,
   'updatable': False},
  'conditions': [{'amount': 1,
    'cid': 0,
    'condition': {'details': {'bitmask': 32,
      'public_key': '3Cxh1eKZk3Wp9KGBWFS7iVde465UvqUKnEqTg2MW4wNf',
      'signature': None,
      'type': 'fulfillment',
      'type_id': 4},
     'uri': 'cc:4:20:IMe7QSL5xRAYIlXon76ZonWktR0NI02M8rAG1bN-ugg:96'},
    'owners_after': ['3Cxh1eKZk3Wp9KGBWFS7iVde465UvqUKnEqTg2MW4wNf']}],
  'fulfillments': [{'fid': 0,
    'fulfillment': 'cf:4:IMe7QSL5xRAYIlXon76ZonWktR0NI02M8rAG1bN-ughA8-9lUJYc_LGAB_NtyTPCCV58LfM
    'input': None,
    'owners_before': ['3Cxh1eKZk3Wp9KGBWFS7iVde465UvqUKnEqTg2MW4wNf']}],
  'metadata': None,
  'operation': 'CREATE',
  'timestamp': '1479393278'},
 'version': 1}
```

Then, the input may be constructed in this way:

```
cid = 0
condition = tx['transaction']['conditions'][cid]
input_ = {
    'fulfillment': condition['condition']['details'],
    'input': {
```

```
            'cid': cid,
            'txid': tx['id'],
        },
        'owners_before': condition['owners_after'],
    }
```

Displaying the input on the prompt would look like:

```
>>> input_
{'fulfillment': {'bitmask': 32,
  'public_key': '3Cxh1eKZk3Wp9KGBWFS7iVde465UvqUKnEqTg2MW4wNf',
  'signature': None,
  'type': 'fulfillment',
  'type_id': 4},
 'input': {'cid': 0,
  'txid': '57cff2b9490468bdb6d4767a1b07905fdbe18d638d9c7783f639b4b2bc165c39'},
 'owners_before': ['3Cxh1eKZk3Wp9KGBWFS7iVde465UvqUKnEqTg2MW4wNf']}
```

To prepare the transfer:

```
>>> prepare_transfer_transaction(
...     inputs=input_,
...     owners_after='EcRawy3Y22eAUSS94vLF8BVJi62wbqbD9iSUSUNU9wAA',
...     asset=tx['transaction']['asset'],
... )
```

bigchaindb_driver.offchain.**fulfill_transaction**(*transaction*, *, *private_keys*)

Fulfills the given transaction.

**Parameters**

- **transaction** (*dict*) – The transaction to be fulfilled.
- **private_keys** (str | list | tuple) – One or more private keys to be used for fulfilling the transaction.

**Returns** The fulfilled transaction payload, ready to be sent to a BigchainDB federation.

**Return type** dict

**Raises** *MissingSigningKeyError* – If a private key, (aka signing key), is missing.

## 7.3 transport

**class** bigchaindb_driver.transport.**Transport**(**nodes*)

Transport class.

**__init__**(**nodes*)

Initializes an instance of *Transport*.

**Parameters** **nodes** – nodes

**forward_request**(*method*, *path=None*, *json=None*)

Forwards an http request to a connection.

**Parameters**

- **method** (*str*) – HTTP method name (e.g.: 'GET'.
- **path** (*str*) – Path to be appended to the base url of a node. E.g.: '/transactions'.

- **json** (*dict*) – Payload to be sent with the HTTP request.

**Returns** Result of `requests.models.Response.json()`

**Return type** dict

**get_connection**()
Gets a connection from the pool.

**Returns** A *Connection* instance.

**init_pool**(*nodes*)
Initializes the pool of connections.

## 7.4 pool

**class** bigchaindb_driver.pool.**Pool**(*connections*, *picker_class=<class 'bigchaindb_driver.pool.RoundRobinPicker'>*)
Pool of connections.

**__init__**(*connections*, *picker_class=<class 'bigchaindb_driver.pool.RoundRobinPicker'>*)
Initializes a *Pool* instance.

**Parameters** **connections** (*list*) – List of *Connection* instances.

**get_connection**()
Gets a *Connection* instance from the pool.

**Returns** A *Connection* instance.

**class** bigchaindb_driver.pool.**RoundRobinPicker**
Object to pick a *Connection* instance from a list of connections.

**picked**
*str*

List index of *Connection* instance that has been picked.

**__init__**()
Initializes a *RoundRobinPicker* instance. Sets *picked* to *-1*.

**pick**(*connections*)
Picks a *Connection* instance from the given list of *Connection* instances.

**Parameters** **connections** (*List*) – List of *Connection* instances.

**class** bigchaindb_driver.pool.**AbstractPicker**
Abstract class for picker classes that pick connections from a pool.

**pick**(*connections*)
Picks a *Connection* instance from the given list of *Connection* instances.

**Parameters** **connections** (*List*) – List of *Connection* instances.

## 7.5 connection

**class** bigchaindb_driver.connection.**Connection**(*, *node_url*)
A Connection object to make HTTP requests.

**__init__**(*, *node_url*)
Initializes a *Connection* instance.

**Parameters** **node_url** (*str*) – Url of the node to connect to.

**request**(*method*, *, *path=None*, *json=None*, ***kwargs*)
Performs an HTTP requests for the specified arguments.

**Parameters**

- **method** (*str*) – HTTP method (e.g.: *'GET'*.
- **path** (*str*) – API endpoint path (e.g.: *'/transactions'*.
- **json** (*dict*) – JSON data to send along with the request.
- **kwargs** – Optional keyword arguments.

## 7.6 crypto

**class** bigchaindb_driver.crypto.**CryptoKeypair**(*signing_key*, *verifying_key*)

**__getnewargs__**()
Return self as a plain tuple. Used by copy and pickle.

**__getstate__**()
Exclude the OrderedDict from pickling

**static __new__**(*_cls*, *signing_key*, *verifying_key*)
Create new instance of CryptoKeypair(signing_key, verifying_key)

**__repr__**()
Return a nicely formatted representation string

**signing_key**
Alias for field number 0

**verifying_key**
Alias for field number 1

bigchaindb_driver.crypto.**generate_keypair**()
Generates a cryptographic key pair.

**Returns** A *collections.namedtuple* with named fields *signing_key* and *verifying_key*.

**Return type** *CryptoKeypair*

## 7.7 exceptions

Exceptions used by *bigchaindb_driver*.

**exception** bigchaindb_driver.exceptions.**BigchaindbException**
Base exception for all Bigchaindb exceptions.

**exception** bigchaindb_driver.exceptions.**TransportError**
Base exception for transport related errors.

This is mainly for cases where the status code denotes an HTTP error, and for cases in which there was a connection error.

**exception** bigchaindb_driver.exceptions.**ConnectionError**
Exception for errors occurring when connecting, and/or making a request to Bigchaindb.

**exception** bigchaindb_driver.exceptions.**NotFoundError**
Exception for HTTP 404 errors.

**exception** bigchaindb_driver.exceptions.**KeypairNotFoundException**
Raised if an operation cannot proceed because the keypair was not given.

**exception** bigchaindb_driver.exceptions.**InvalidSigningKey**
Raised if a signing key is invalid. E.g.: None.

**exception** bigchaindb_driver.exceptions.**InvalidVerifyingKey**
Raised if a verifying key is invalid. E.g.: None.

**exception** bigchaindb_driver.exceptions.**MissingSigningKeyError**
Raised if a signing key is missing.

## 7.8 utils

Set of utilities to support various functionalities of the driver.

bigchaindb_driver.utils.**ops_map**
*dict*

Mapping between operation strings and classes. E.g.: The string 'CREATE' is mapped to *CreateOperation*.

**class** bigchaindb_driver.utils.**CreateOperation**
Class representing the 'CREATE' transaction operation.

**class** bigchaindb_driver.utils.**TransferOperation**
Class representing the 'TRANSFER' transaction operation.

bigchaindb_driver.utils.**_normalize_asset**(*asset*, *is_transfer=False*)
Normalizes the given asset dictionary.

For now, this means converting the given asset dictionary to a Asset class.

**Parameters**

- **asset** (*dict*) – The asset to normalize.
- **is_transfer** (*boal, optional*) – Flag used to indicate whether the asset is to be used as part of a *'TRANSFER'* operation or not. Defaults to False.

**Returns**

The Asset class, instantiated from the given asset dictionary.

**Important:** If the instantiation step fails, None may be returned.

**Danger:** For specific internal usage only. The behavior is tricky, and is subject to change.

bigchaindb_driver.utils.**_normalize_operation**(*operation*)

Normalizes the given operation string. For now, this simply means converting the given string to uppercase, looking it up in *ops_map*, and returning the corresponding class if present.

**Parameters** **operation** (*str*) – The operation string to convert.

**Returns**

The class corresponding to the given string, *CreateOperation* or *TransferOperation*.

---

**Important:** If the `str.upper()` step, or the *ops_map* lookup fails, the given `operation` argument is returned.

---

**Danger:** For specific internal usage only. The behavior is tricky, and is subject to change.

CHAPTER 8

# About this Documentation

This section contains instructions to build and view the documentation locally, using the `docker-compose docs.yml` file of the `bigchaindb-driver` repository: https://github.com/bigchaindb/bigchaindb-driver.

If you do not have a clone of the repo, you need to get one.

## 8.1 Building the documentation

To build the docs, simply run

```
$ docker-compose -d docs.yml up bdocs
```

Or if you prefer, start a `bash` session,

```
$ docker-compose -f docs.yml run --rm bdocs bash
```

and build the docs:

```
root@a651959a1f2d:/usr/src/app# make -C docs html
```

## 8.2 Viewing the documentation

You can start a little web server to view the docs at http://localhost:55555/

```
$ docker-compose up -d vdocs
```

**Note:** If you are using `docker-machine` you need to replace `localhost` with the `ip` of the machine (e.g.: `docker-machine ip tm` if your machine is named `tm`).

## 8.3 Making changes

The necessary source code is mounted, which allows you to make modifications, and view the changes by simply re-building the docs, and refreshing the browser.

CHAPTER 9

# Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

## 9.1 Types of Contributions

### 9.1.1 Report Bugs

Report bugs at https://github.com/bigchaindb/bigchaindb-driver/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

### 9.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

### 9.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

### 9.1.4 Write Documentation

bigchaindb-driver could always use more documentation, whether as part of the official bigchaindb-driver docs, in docstrings, or even on the web in blog posts, articles, and such.

### 9.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/bigchaindb/bigchaindb-driver/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

## 9.2 Get Started!

Ready to contribute? Here's how to set up bigchaindb-driver for local development.

1. Fork the bigchaindb-driver repo on GitHub.
2. Clone your fork locally and enter into the project:

```
$ git clone git@github.com:your_name_here/bigchaindb-driver.git
$ cd bigchaindb-driver/
```

3. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

4. When you're done making changes, check that your changes pass flake8 and the tests. For the tests, you'll need to start the RethinkDB and BigchainDB servers:

```
$ docker-compose up -d rdb
$ docker-compose up -d bdb-server
```

5. flake8 check:

```
$ docker-compose run --rm bdb-driver flake8 bigchaindb_driver tests
```

6. To run the tests:

```
$ docker-compose run --rm bdb-driver pytest -v
```

7. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

8. Submit a pull request through the GitHub website.

## 9.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.

3. The pull request should work for Python 3.5, and pass the flake8 check. Check https://travis-ci.org/bigchaindb/bigchaindb-driver/pull_requests and make sure that the tests pass for all supported Python versions.

## 9.4 Tips

### 9.4.1 Development Environment with Docker

Depending on what you are doing, you may need to run at least one BigchainDB node. You can use the docker-compose.yml file to run a node, and perform other tasks that depend on the running node. To run a BigchainDB node, (for development), you start a RethinkDB node, followed by the linked BigchainDB node:

```
$ docker-compose up -d rdb
$ docker-compose up -d bdb-server
```

You can monitor the logs:

```
$ docker-compose logs -f
```

### 9.4.2 Tests

To run a subset of tests:

```
$ docker-compose run --rm bdb-driver pytest -v tests/test_driver.py
```

**Important:** When running tests, unless you are targeting a test that does not require a connection with the BigchainDB server, you need to run the BigchainDB and RethinkDB servers:

```
$ docker-compose up -d rdb
$ docker-compose up -d bdb-server
```

### 9.4.3 Dependency on Bigchaindb

By default, the development requirements, BigchainDB server Dockerfile, and .travis.yml are set to depend from BigchainDB's master branch to more easily track changes against BigchainDB's API.

CHAPTER 10

# Credits

## 10.1 Development Lead

- BigchainDB <dev@bigchaindb.com>

## 10.2 Contributors

None yet. Why not be the first?

CHAPTER 11

# Changelog

## 11.1 0.1.0 (2016-11-29)

### 11.1.1 Added

- Support for BigchainDB server 0.8.0.
- Support for divisible assets.

### 11.1.2 Removed

- `create()` and `transfer()` under `TransactionEndpoint`, and available via `BigchainDB.transactions`. Replaced by the three "canonical" transaction operations: `prepare()`, `fulfill()`, and `send()`.
- Support for client side timestamps.

## 11.2 0.0.3 (2016-11-25)

### 11.2.1 Added

- Support for "canonical" transaction operations:
  - `prepare`
  - `fulfill`
  - `send`

### 11.2.2 Deprecated

- `create()` and `transfer()` under `TransactionEndpoint`, and available via `BigchainDB.transactions`. Replaced by the above three "canonical" transaction operations: `prepare()`, `fulfill()`, and `send()`.

### 11.2.3 Fixed

- `BigchainDB()` default node setting on its transport class. See commit 0a80206

## 11.3 0.0.2 (2016-10-28)

### 11.3.1 Added

- Support for BigchainDB server 0.7.0

## 11.4 0.0.1dev1 (2016-08-25)

- Development (pre-alpha) release on PyPI.

### 11.4.1 Added

- Minimal support for `POST` (via `create()` and `transfer()`), and `GET` operations on the `/transactions` endpoint.

## 11.5 0.0.1a1 (2016-08-12)

- Planning release on PyPI.

CHAPTER 12

# Indices and tables

- genindex
- modindex
- search

## b

## Symbols



## A



## B



## C



## F



## G



## I



## K



## M



## N

## O



## P



## R



## S



## T



## V