BigchainDB Server Documentation

Release 1.0.1

BigchainDB Contributors

Jul 14, 2017

Contents

CHAPTER 1

Introduction

This is the documentation for BigchainDB Server, the BigchainDB software that one runs on servers (but not on clients).

If you want to use BigchainDB Server, then you should first understand what BigchainDB is, plus some of the specialized BigchaindB terminology. You can read about that in the overall BigchainDB project documentation.

Note that there are a few kinds of nodes:

- A **dev/test node** is a node created by a developer working on BigchainDB Server, e.g. for testing new or changed code. A dev/test node is typically run on the developer's local machine.
- A **bare-bones node** is a node deployed in the cloud, either as part of a testing cluster or as a starting point before upgrading the node to be production-ready.
- A **production node** is a node that is part of a consortium's BigchainDB cluster. A production node has the most components and requirements.

Setup Instructions for Various Cases

- Set up a local stand-alone BigchainDB node for learning and experimenting: Quickstart
- Set up and run a local dev/test node for developing and testing BigchainDB Server
- Set up and run a BigchainDB cluster

There are some old RethinkDB-based deployment instructions as well:

- Deploy a bare-bones RethinkDB-based node on Azure
- Deploy a bare-bones RethinkDB-based node on any Ubuntu machine with Ansible
- Deploy a RethinkDB-based testing cluster on AWS

Instructions for setting up a client will be provided once there's a public test net.

Can I Help?

Yes! BigchainDB is an open-source project; we welcome contributions of all kinds. If you want to request a feature, file a bug report, make a pull request, or help in some other way, please see the CONTRIBUTING.md file.

CHAPTER 2

Quickstart

This page has instructions to set up a single stand-alone BigchainDB node for learning or experimenting. Instructions for other cases are elsewhere. We will assume you're using Ubuntu 16.04 or similar. If you're not using Linux, then you might try running BigchainDB with Docker.

A. Install MongoDB as the database backend. (There are other options but you can ignore them for now.)

Install MongoDB Server 3.4+

B. To run MongoDB with default database path i.e. /data/db, open a Terminal and run the following command:

```
$ sudo mkdir -p /data/db
```

C. Assign rwx(read/write/execute) permissions to the user for default database directory:

```
$ sudo chmod -R 700 /data/db
```

D. Run MongoDB:

```
$ sudo mongod --replSet=bigchain-rs
```

E. Ubuntu 16.04 already has Python 3.5, so you don't need to install it, but you do need to install some other things:

```
$ sudo apt-get update
$ sudo apt-get install g++ python3-dev libffi-dev build-essential libssl-dev
```

F. Get the latest version of pip and setuptools:

```
$ sudo apt-get install python3-pip
$ sudo pip3 install --upgrade pip setuptools
```

G. Install the `bigchaindb` Python package from PyPI:

```
$ sudo pip3 install bigchaindb
```

In case you are having problems with installation or package/module versioning, please upgrade the relevant packages on your host by running one the following commands:

```
$ sudo pip3 install [packageName]==[packageVersion]

OR

$ sudo pip3 install [packageName] --upgrade
```

H. Configure BigchainDB Server:

```
$ bigchaindb -y configure mongodb
```

I. Run BigchainDB Server:

```
$ bigchaindb start
```

You now have a running BigchainDB Server and can post transactions to it. One way to do that is to use the BigchainDB Python Driver.

Install the BigchainDB Python Driver (link)

CHAPTER 3

Production Nodes

Production Node Assumptions

Be sure you know the key BigchainDB terminology:

- BigchainDB node, BigchainDB cluster and BigchainDB consortum
- dev/test node, bare-bones node and production node

We make some assumptions about production nodes:

1. Production nodes use MongoDB, not RethinkDB.
2. Each production node is set up and managed by an experienced professional system administrator or a team of them.
3. Each production node in a cluster is managed by a different person or team.

You can use RethinkDB when building prototypes, but we don't advise or support using it in production.

We don't provide a detailed cookbook explaining how to secure a server, or other things that a sysadmin should know. We do provide some templates, but those are just starting points.

Production Node Components

A production BigchainDB node must include:

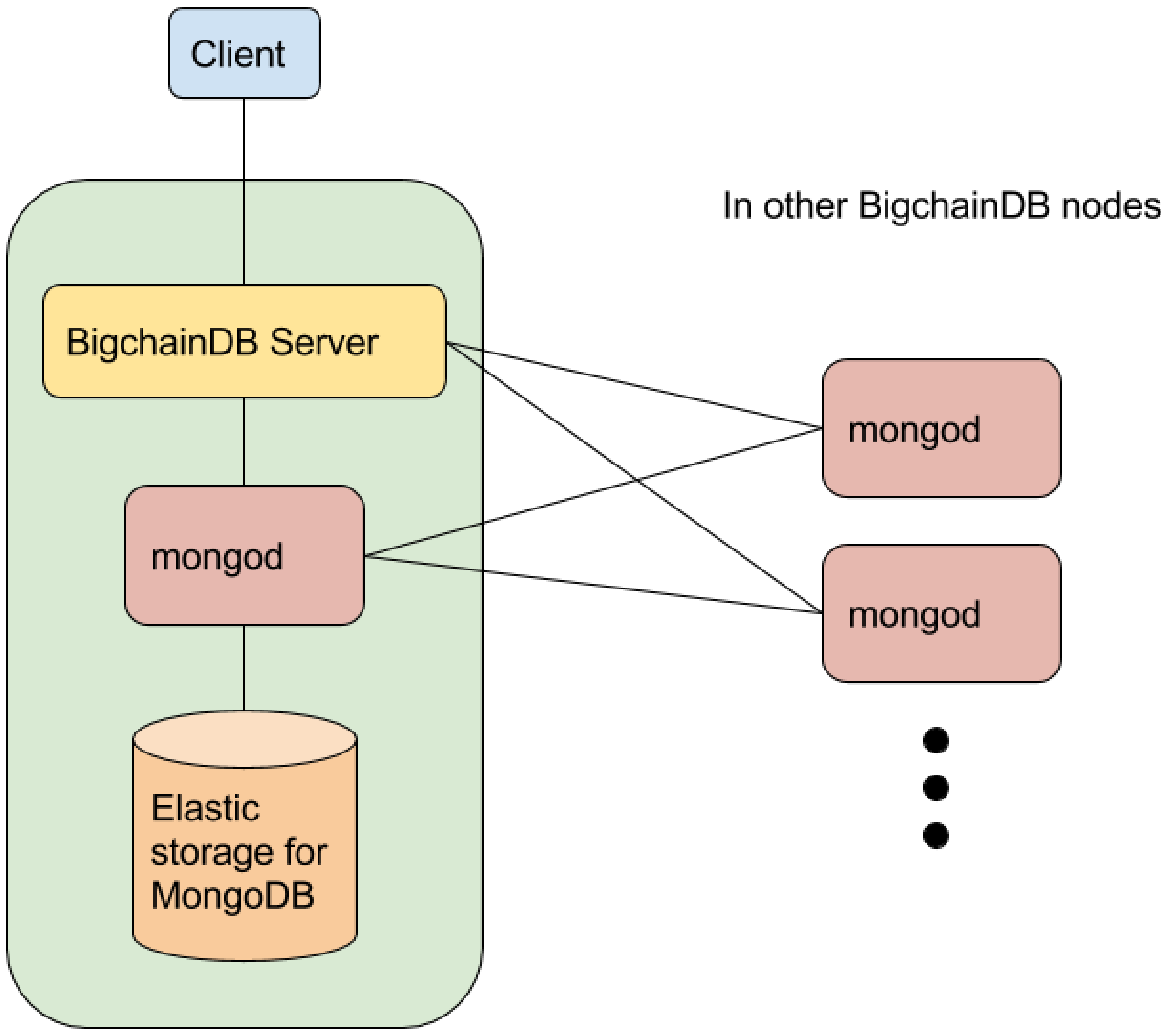
- BigchainDB Server
- MongoDB Server 3.4+ (mongod)
- Scalable storage for MongoDB

It could also include several other components, including:

- NGINX or similar, to provide authentication, rate limiting, etc.
- An NTP daemon running on all machines running BigchainDB Server or mongod, and possibly other machines

- **Not** MongoDB Automation Agent. It's for automating the deployment of an entire MongoDB cluster, not just one MongoDB node within a cluster.
- MongoDB Monitoring Agent
- MongoDB Backup Agent
- Log aggregation software
- Monitoring software
- Maybe more

The relationship between the main components is illustrated below. Note that BigchainDB Server must be able to communicate with the *primary* MongoDB instance, and any of the MongoDB instances might be the primary, so BigchainDB Server must be able to communicate with all the MongoDB instances. Also, all MongoDB instances must be able to communicate with each other.



Production Node Requirements

This page is about the requirements of BigchainDB Server. You can find the requirements of MongoDB, NGINX, your NTP daemon, your monitoring software, and other production node components in the documentation for that software.

OS Requirements

BigchainDB Server requires Python 3.5+ and Python 3.5+ will run on any modern OS, but we recommend using an LTS version of Ubuntu Server or a similarly server-grade Linux distribution.

Don't use macOS (formerly OS X, formerly Mac OS X), because it's not a server-grade operating system. Also, BigchaindB Server uses the Python multiprocessing package and some functionality in the multiprocessing package doesn't work on Mac OS X.

General Considerations

BigchainDB Server runs many concurrent processes, so more RAM and more CPU cores is better.

As mentioned on the page about production node components, every machine running BigchainDB Server should be running an NTP daemon.

Set Up and Run a Cluster Node

This is a page of general guidelines for setting up a production BigchainDB node. Before continuing, make sure you've read the pages about production node assumptions, components and requirements.

Note: These are just guidelines. You can modify them to suit your needs. For example, if you want to initialize the MongoDB replica set before installing BigchainDB, you *can* do that. If you'd prefer to use Docker and Kubernetes, you can (and we have a template). We don't cover all possible setup procedures here.

Security Guidelines

There are many articles, websites and books about securing servers, virtual machines, networks, etc. Consult those. There are some notes on BigchainDB-specific firewall setup in the Appendices.

Sync Your System Clock

A BigchainDB node uses its system clock to generate timestamps for blocks and votes, so that clock should be kept in sync with some standard clock(s). The standard way to do that is to run an NTP daemon (Network Time Protocol daemon) on the node.

MongoDB also recommends having an NTP daemon running on all MongoDB nodes.

NTP is a standard protocol. There are many NTP daemons implementing it. We don't recommend a particular one. On the contrary, we recommend that different nodes in a cluster run different NTP daemons, so that a problem with one daemon won't affect all nodes.

Please see the notes on NTP daemon setup in the Appendices.

Set Up Storage for MongoDB

We suggest you set up a separate storage device (partition, RAID array, or logical volume) to store the data in the MongoDB database. Here are some questions to ask:

- How easy will it be to add storage in the future? Will I have to shut down my server?
- How big can the storage get? (Remember that RAID can be used to make several physical drives look like one.)
- How fast can it read & write data? How many input/output operations per second (IOPS)?
- How does IOPS scale as more physical hard drives are added?
- What's the latency?
- What's the reliability? Is there replication?
- What's in the Service Level Agreement (SLA), if applicable?
- What's the cost?

There are many options and tradeoffs.

Consult the MongoDB documentation for its recommendations regarding storage hardware, software and settings, e.g. in the MongoDB Production Notes.

Install and Run MongoDB

- Install MongoDB 3.4+. (BigchainDB only works with MongoDB 3.4+.)
- Run MongoDB (mongod)

Install BigchainDB Server

Install BigchainDB Server Dependencies

Before you can install BigchainDB Server, you must install its OS-level dependencies and you may have to install Python 3.5+.

How to Install BigchainDB Server with pip

BigchainDB is distributed as a Python package on PyPI so you can install it using `pip`. First, make sure you have an up-to-date Python 3.5+ version of `pip` installed:

```
pip -V
```

If it says that `pip` isn't installed, or it says `pip` is associated with a Python version less than 3.5, then you must install a `pip` version associated with Python 3.5+. In the following instructions, we call it `pip3` but you may be able to use `pip` if that refers to the same thing. See the `pip` installation instructions.

On Ubuntu 16.04, we found that this works:

```
sudo apt-get install python3-pip
```

That should install a Python 3 version of `pip` named `pip3`. If that didn't work, then another way to get `pip3` is to do `sudo apt-get install python3-setuptools` followed by `sudo easy_install3 pip`.

You can upgrade `pip` (`pip3`) and `setuptools` to the latest versions using:

```
pip3 install --upgrade pip setuptools
pip3 -V
```

Now you can install BigchainDB Server using:

```
pip3 install bigchaindb
```

(If you're not in a virtualenv and you want to install bigchaindb system-wide, then put `sudo` in front.)

Note: You can use `pip3` to upgrade the `bigchaindb` package to the latest version using `pip3 install --upgrade bigchaindb`.

How to Install BigchainDB Server from Source

If you want to install BitchainDB from source because you want to use the very latest bleeding-edge code, clone the public repository:

```
git clone git@github.com:bigchaindb/bigchaindb.git
cd bigchaindb
python setup.py install
```

Configure BigchainDB Server

Start by creating a default BigchainDB config file for a MongoDB backend:

```
bigchaindb -y configure mongodb
```

(There's documentation for the `bigchaindb` command is in the section on the BigchainDB Command Line Interface (CLI).)

Edit the created config file by opening `$HOME/.bigchaindb` (the created config file) in your text editor:

- Change `"server": {"bind": "localhost:9984", ... }` to `"server": {"bind": "0.0.0.0:9984", ... }`. This makes it so traffic can come from any IP address to port 9984 (the HTTP Client-Server API port).
- Change `"keyring": []` to `"keyring": ["public_key_of_other_node_A", "public_key_of_other_node_B", "..."]` i.e. a list of the public keys of all the other nodes in the cluster. The keyring should *not* include your node's public key.
- Ensure that `database.host` and `database.port` are set to the hostname and port of your MongoDB instance. (The port is usually 27017, unless you changed it.)

For more information about the BigchainDB config file, see the page about the BigchainDB configuration settings.

Get All Other Nodes to Update Their Keyring

All other BigchainDB nodes in the cluster must add your new node's public key to their BigchainDB keyring. Currently, the only way to get BigchainDB Server to "notice" a changed keyring is to shut it down and start it back up again (with the new keyring).

Maybe Update the MongoDB Replica Set

If this isn't the first node in the BigchainDB cluster, then someone with an existing BigchainDB node (not you) must add your MongoDB instance to the MongoDB replica set. They can do so (on their node) using:

```
bigchaindb add-replicas your-mongod-hostname:27017
```

where they must replace `your-mongod-hostname` with the actual hostname of your MongoDB instance, and they may have to replace `27017` with the actual port.

Start BigchainDB

Warning: If you're not deploying the first node in the BigchainDB cluster, then don't start BigchainDB before your MongoDB instance has been added to the MongoDB replica set (as outlined above).

```
# See warning above
bigchaindb start
```

Using a Reverse Proxy

You may want to:

- rate limit inbound HTTP requests,
- authenticate/authorize inbound HTTP requests,
- block requests with an HTTP request body that's too large, or
- enable HTTPS (TLS) between your users and your node.

While we could have built all that into BigchainDB Server, we didn't, because you can do all that (and more) using a reverse proxy such as NGINX or HAProxy. (You would put it in front of your BigchainDB Server, so that all inbound HTTP requests would arrive at the reverse proxy before *maybe* being proxied onwards to your BigchainDB Server.) For detailed instructions, see the documentation for your reverse proxy.

Below, we note how a reverse proxy can be used to do some BigchainDB-specific things.

You may also be interested in our NGINX configuration file template (open source, on GitHub).

Enforcing a Max Transaction Size

The BigchainDB HTTP API has several endpoints, but only one of them, the `POST /transactions` endpoint, expects a non-empty HTTP request body: the transaction (JSON) being submitted by the user.

If you want to enforce a maximum-allowed transaction size (discarding any that are larger), then you can do so by configuring a maximum request body size in your reverse proxy. For example, NGINX has the `client_max_body_size` configuration setting. You could set it to 15 kB with the following line in your NGINX config file:

```
client_max_body_size 15k;
```

For more information, see the NGINX docs about client_max_body_size.

Note: By enforcing a maximum transaction size, you indirectly enforce a maximum crypto-conditions complexity.

Aside: Why 15 kB?

Both RethinkDB and MongoDB have a maximum document size of 16 MB. In BigchainDB, the biggest documents are the blocks. A BigchainDB block can contain up to 1000 transactions, plus some other data (e.g. the timestamp). If we ignore the other data as negligible relative to all the transactions, then a block of size 16 MB will have an average transaction size of (16 MB)/1000 = 16 kB. Therefore by limiting the max transaction size to 15 kB, you can be fairly sure that no blocks will ever be bigger than 16 MB.

Note: Technically, the documents that MongoDB stores aren't the JSON that BigchainDB users think of; they're JSON converted to BSON. Moreover, one can use GridFS with MongoDB to store larger documents. Therefore the above calculation shoud be seen as a rough guide, not the last word.

CHAPTER 4

Clusters

A **BigchainDB Cluster** is a set of connected **BigchainDB Nodes**, managed by a **BigchainDB Consortium** (i.e. an organization). Those terms are defined in the BigchainDB Terminology page.

Consortium Structure & Governance

The consortium might be a company, a foundation, a cooperative, or some other form of organization. It must make many decisions, e.g. How will new members be added? Who can read the stored data? What kind of data will be stored? A governance process is required to make those decisions, and therefore one of the first steps for any new consortium is to specify its governance process (if one doesn't already exist). This documentation doesn't explain how to create a consortium, nor does it outline the possible governance processes.

It's worth noting that the decentralization of a BigchainDB cluster depends, to some extent, on the decentralization of the associated consortium. See the pages about decentralization and node diversity.

Relevant Technical Documentation

There are some pages and sections that will be of particular interest to anyone building or managing a BigchainDB cluster. In particular:

- the page about how to set up and run a cluster node,
- our production deployment template, and
- our old RethinkDB-based AWS deployment template.

Cluster DNS Records and SSL Certificates

We now describe how *we* set up the external (public-facing) DNS records for a BigchainDB cluster. Your consortium may opt to do it differently. There were several goals:

- Allow external users/clients to connect directly to any BigchainDB node in the cluster (over the internet), if they want.
- Each BigchainDB node operator should get an SSL certificate for their BigchainDB node, so that their BigchainDB node can serve the BigchainDB HTTP API via HTTPS. (The same certificate might also be used to serve the WebSocket API.)
- There should be no sharing of SSL certificates among BigchainDB node operators.
- Optional: Allow clients to connect to a "random" BigchainDB node in the cluster at one particular domain (or subdomain).

Node Operator Responsibilities

1. Register a domain (or use one that you already have) for your BigchainDB node. You can use a subdomain if you like. For example, you might opt to use `abc-org73.net`, `api.dynabob8.io` or `figmentdb3.ninja`.
2. Get an SSL certificate for your domain or subdomain, and properly install it in your node (e.g. in your NGINX instance).
3. Create a DNS A Record mapping your domain or subdomain to the public IP address of your node (i.e. the one that serves the BigchainDB HTTP API).

Consortium Responsibilities

Optional: The consortium managing the BigchainDB cluster could register a domain name and set up CNAME records mapping that domain name (or one of its subdomains) to each of the nodes in the cluster. For example, if the consortium registered `bdbcluster.io`, they could set up CNAME records like the following:

- CNAME record mapping `api.bdbcluster.io` to `abc-org73.net`
- CNAME record mapping `api.bdbcluster.io` to `api.dynabob8.io`
- CNAME record mapping `api.bdbcluster.io` to `figmentdb3.ninja`

CHAPTER 5

Production Deployment Template

This section outlines how *we* deploy production BigchainDB nodes and clusters on Microsoft Azure using Kubernetes. We improve it constantly. You may choose to use it as a template or reference for your own deployment, but *we make no claim that it is suitable for your purposes*. Feel free change things to suit your needs or preferences.

Overview

This page summarizes the steps *we* go through to set up a production BigchainDB cluster. We are constantly improving them. You can modify them to suit your needs.

Things the Managing Organization Must Do First

1. Set Up a Self-Signed Certificate Authority

We use SSL/TLS and self-signed certificates for MongoDB authentication (and message encryption). The certificates are signed by the organization managing the cluster. If your organization already has a process for signing certificates (i.e. an internal self-signed certificate authority [CA]), then you can skip this step. Otherwise, your organization must *set up its own self-signed certificate authority*.

2. Register a Domain and Get an SSL Certificate for It

The BigchainDB APIs (HTTP API and WebSocket API) should be served using TLS, so the organization running the cluster should choose an FQDN for their API (e.g. api.organization-x.com), register the domain name, and buy an SSL/TLS certificate for the FQDN.

Things Each Node Operator Must Do

Every MongoDB instance in the cluster must have a unique (one-of-a-kind) name. Ask the organization managing your cluster if they have a standard way of naming instances in the cluster. For example, maybe they assign a unique number to each node, so that if you're operating node 12, your MongoDB instance would be named `mdb-instance-12`. Similarly, other instances must also have unique names in the cluster.

1. Name of the MongoDB instance (`mdb-instance-*`)
2. Name of the BigchainDB instance (`bdb-instance-*`)
3. Name of the NGINX instance (`ngx-instance-*`)
4. Name of the MongoDB monitoring agent instance (`mdb-mon-instance-*`)
5. Name of the MongoDB backup agent instance (`mdb-bak-instance-*`)

Generate four keys and corresponding certificate signing requests (CSRs):

1. Server Certificate (a.k.a. Member Certificate) for the MongoDB instance
2. Client Certificate for BigchainDB Server to identify itself to MongoDB
3. Client Certificate for MongoDB Monitoring Agent to identify itself to MongoDB
4. Client Certificate for MongoDB Backup Agent to identify itself to MongoDB

Ask the managing organization to use its self-signed CA to sign those four CSRs. They should send you:

- Four certificates (one for each CSR you sent them).
- One `ca.crt` file: their CA certificate.
- One `crl.pem` file: a certificate revocation list.

For help, see the pages:

- *How to Generate a Server Certificate for MongoDB*
- *How to Generate a Client Certificate for MongoDB*

Every node in a BigchainDB cluster needs its own BigchainDB keypair (i.e. a public key and corresponding private key). You can generate a BigchainDB keypair for your node, for example, using the BigchainDB Python Driver.

```
from bigchaindb_driver.crypto import generate_keypair
print(generate_keypair())
```

Share your BigchaindB *public* key with all the other nodes in the BigchainDB cluster. Don't share your private key.

Get the BigchainDB public keys of all the other nodes in the cluster. That list of public keys is known as the BigchainDB "keyring."

Make up an FQDN for your BigchainDB node (e.g. `mynode.mycorp.com`). Make sure you've registered the associated domain name (e.g. `mycorp.com`), and have an SSL certificate for the FQDN. (You can get an SSL certificate from any SSL certificate provider.)

Ask the managing organization for the FQDN used to serve the BigchainDB APIs (e.g. `api.orgname.net` or `bdb.clustername.com`) and for a copy of the associated SSL/TLS certificate. Also, ask for the user name to use for authenticating to MongoDB.

If the cluster uses 3scale for API authentication, monitoring and billing, you must ask the managing organization for all relevant 3scale credentials.

If the cluster uses MongoDB Cloud Manager for monitoring and backup, you must ask the managing organization for the `Group ID` and the `Agent API Key`. (Each Cloud Manager "group" has its own `Group ID`. A `Group ID` can contain a number of `Agent API Key` s. It can be found under **Settings - Group Settings**. It was recently added to the Cloud Manager to allow easier periodic rotation of the `Agent API Key` with a constant `Group ID`)

Deploy a Kubernetes cluster on Azure.

You can now proceed to set up your BigchainDB node based on whether it is the *first node in a new cluster* or a *node that will be added to an existing cluster*.

How to Set Up a Self-Signed Certificate Authority

This page enumerates the steps *we* use to set up a self-signed certificate authority (CA). This is something that only needs to be done once per cluster, by the organization managing the cluster, i.e. the CA is for the whole cluster. We use Easy-RSA.

Step 1: Install & Configure Easy-RSA

First create a directory for the CA and cd into it:

```
mkdir bdb-cluster-ca

cd bdb-cluster-ca
```

Then *install and configure Easy-RSA in that directory*.

Step 2: Create a Self-Signed CA

You can create a self-signed CA by going to the `bdb-cluster-ca/easy-rsa-3.0.1/easyrsa3` directory and using:

```
./easyrsa init-pki

./easyrsa build-ca
```

You will also be asked to enter a PEM pass phrase (for encrypting the `ca.key` file). Make sure to securely store that PEM pass phrase. If you lose it, you won't be able to add or remove entities from your PKI infrastructure in the future.

You will be prompted to enter the Distinguished Name (DN) information for this CA. For each field, you can accept the default value [in brackets] by pressing Enter.

Warning: Don't accept the default value of OU (`IT`). Instead, enter the value `ROOT-CA`.

While `Easy-RSA CA` *is* a valid and acceptable Common Name, you should probably enter a name based on the name of the managing organization, e.g. `Omega Ledger CA`.

Tip: You can get help with the `easyrsa` command (and its subcommands) by using the subcommand `./easyrsa help`

Step 3: Create an Intermediate CA

TODO

Step 4: Generate a Certificate Revocation List

You can generate a Certificate Revocation List (CRL) using:

```
./easyrsa gen-crl
```

You will need to run this command every time you revoke a certificate. The generated `crl.pem` needs to be uploaded to your infrastructure to prevent the revoked certificate from being used again.

Step 5: Secure the CA

The security of your infrastructure depends on the security of this CA.

- Ensure that you restrict access to the CA and enable only legitimate and required people to sign certificates and generate CRLs.
- Restrict access to the machine where the CA is hosted.
- Many certificate providers keep the CA offline and use a rotating intermediate CA to sign and revoke certificates, to mitigate the risk of the CA getting compromised.
- In case you want to destroy the machine where you created the CA (for example, if this was set up on a cloud provider instance), you can backup the entire `easyrsa` directory to secure storage. You can always restore it to a trusted instance again during the times when you want to sign or revoke certificates. Remember to backup the directory after every update.

How to Generate a Server Certificate for MongoDB

This page enumerates the steps *we* use to generate a server certificate for a MongoDB instance. A server certificate is also referred to as a “member certificate” in the MongoDB documentation. We use Easy-RSA.

Step 1: Install & Configure Easy–RSA

First create a directory for the server certificate (member cert) and cd into it:

```
mkdir member-cert

cd member-cert
```

Then *install and configure Easy-RSA in that directory*.

Step 2: Create the Server Private Key and CSR

You can create the server private key and certificate signing request (CSR) by going into the directory `member-cert/easy-rsa-3.0.1/easyrsa3` and using something like:

```
./easyrsa init-pki

./easyrsa --req-cn=mdb-instance-0 --subject-alt-name=DNS:localhost,DNS:mdb-instance-0␣
↪gen-req mdb-instance-0 nopass
```

You should replace the Common Name (`mdb-instance-0` above) with the correct name for *your* MongoDB instance in the cluster, e.g. `mdb-instance-5` or `mdb-instance-12`. (This name is decided by the organization managing the cluster.)

You will be prompted to enter the Distinguished Name (DN) information for this certificate. For each field, you can accept the default value [in brackets] by pressing Enter.

Warning: Don't accept the default value of OU (`IT`). Instead, enter the value `MongoDB-Instance`.

Aside: You need to provide the `DNS:localhost` SAN during certificate generation for using the `localhost exception` in the MongoDB instance. All certificates can have this attribute without compromising security as the `localhost exception` works only the first time.

Step 3: Get the Server Certificate Signed

The CSR file created in the last step should be located in `pki/reqs/mdb-instance-0.req` (where the integer `0` may be different for you). You need to send it to the organization managing the cluster so that they can use their CA to sign the request. (The managing organization should already have a self-signed CA.)

If you are the admin of the managing organization's self-signed CA, then you can import the CSR and use Easy-RSA to sign it. Go to your `bdb-cluster-ca/easy-rsa-3.0.1/easyrsa3/` directory and do something like:

```
./easyrsa import-req mdb-instance-0.req mdb-instance-0

./easyrsa --subject-alt-name=DNS:localhost,DNS:mdb-instance-0 sign-req server mdb-
↪instance-0
```

Once you have signed it, you can send the signed certificate and the CA certificate back to the requestor. The files are `pki/issued/mdb-instance-0.crt` and `pki/ca.crt`.

Step 4: Generate the Consolidated Server PEM File

MongoDB requires a single, consolidated file containing both the public and private keys.

```
cat mdb-instance-0.crt mdb-instance-0.key > mdb-instance-0.pem
```

How to Generate a Client Certificate for MongoDB

This page enumerates the steps *we* use to generate a client certificate to be used by clients who want to connect to a TLS-secured MongoDB cluster. We use Easy-RSA.

Step 1: Install and Configure Easy-RSA

First create a directory for the client certificate and cd into it:

```
mkdir client-cert

cd client-cert
```

Then *install and configure Easy-RSA in that directory*.

Step 2: Create the Client Private Key and CSR

You can create the client private key and certificate signing request (CSR) by going into the directory `client-cert/easy-rsa-3.0.1/easyrsa3` and using:

```
./easyrsa init-pki

./easyrsa gen-req bdb-instance-0 nopass
```

You should change the Common Name (e.g. `bdb-instance-0`) to a value that reflects what the client certificate is being used for, e.g. `mdb-mon-instance-3` or `mdb-bak-instance-4`. (The final integer is specific to your BigchainDB node in the BigchainDB cluster.)

You will be prompted to enter the Distinguished Name (DN) information for this certificate. For each field, you can accept the default value [in brackets] by pressing Enter.

Warning: Don't accept the default value of OU (`IT`). Instead, enter the value `BigchainDB-Instance`, `MongoDB-Mon-Instance` or `MongoDB-Backup-Instance` as appropriate.

Aside: The `nopass` option means "do not encrypt the private key (default is encrypted)". You can get help with the `easyrsa` command (and its subcommands) by using the subcommand `./easyrsa help`.

Step 3: Get the Client Certificate Signed

The CSR file created in the previous step should be located in `pki/reqs/bdb-instance-0.req` (or whatever Common Name you used in the `gen-req` command above). You need to send it to the organization managing the cluster so that they can use their CA to sign the request. (The managing organization should already have a self-signed CA.)

If you are the admin of the managing organization's self-signed CA, then you can import the CSR and use Easy-RSA to sign it. Go to your `bdb-cluster-ca/easy-rsa-3.0.1/easyrsa3/` directory and do something like:

```
./easyrsa import-req bdb-instance-0.req bdb-instance-0

./easyrsa sign-req client bdb-instance-0
```

Once you have signed it, you can send the signed certificate and the CA certificate back to the requestor. The files are `pki/issued/bdb-instance-0.crt` and `pki/ca.crt`.

Step 4: Generate the Consolidated Client PEM File

MongoDB requires a single, consolidated file containing both the public and private keys.

```
cat bdb-instance-0.crt bdb-instance-0.key > bdb-instance-0.pem
```

How to Revoke an SSL/TLS Certificate

This page enumerates the steps *we* take to revoke a self-signed SSL/TLS certificate in a cluster. It can only be done by someone with access to the self-signed CA associated with the cluster's managing organization.

Step 1: Revoke a Certificate

Since we used Easy-RSA version 3 to *set up the CA*, we use it to revoke certificates too.

Go to the following directory (associated with the self-signed CA): `.../bdb-cluster-ca/easy-rsa-3.0.1/easyrsa3`. You need to be aware of the file name used to import the certificate using the `./easyrsa import-req` before. Run the following command to revoke a certificate:

```
./easyrsa revoke <filename>
```

This will update the CA database with the revocation details. The next step is to use the updated database to issue an up-to-date certificate revocation list (CRL).

Step 2: Generate a New CRL

Generate a new CRL for your infrastructure using:

```
./easyrsa gen-crl
```

The generated `crl.pem` file needs to be uploaded to your infrastructure to prevent the revoked certificate from being used again.

In particlar, the generated `crl.pem` file should be sent to all BigchainDB node operators in your BigchainDB cluster, so that they can update it in their MongoDB instance and their BigchainDB Server instance.

Template: Deploy a Kubernetes Cluster on Azure

A BigchainDB node can be run inside a Kubernetes cluster. This page describes one way to deploy a Kubernetes cluster on Azure.

Step 1: Get a Pay-As-You-Go Azure Subscription

Microsoft Azure has a Free Trial subscription (at the time of writing), but it's too limited to run an advanced BigchainDB node. Sign up for a Pay-As-You-Go Azure subscription via the Azure website.

You may find that you have to sign up for a Free Trial subscription first. That's okay: you can have many subscriptions.

Step 2: Create an SSH Key Pair

You'll want an SSH key pair so you'll be able to SSH to the virtual machines that you'll deploy in the next step. (If you already have an SSH key pair, you *could* reuse it, but it's probably a good idea to make a new SSH key pair for your Kubernetes VMs and nothing else.)

See the *page about how to generate a key pair for SSH*.

Step 3: Deploy an Azure Container Service (ACS)

It's *possible* to deploy an Azure Container Service (ACS) from the Azure Portal (i.e. online in your web browser) but it's actually easier to do it using the Azure Command-Line Interface (CLI).

Microsoft has instructions to install the Azure CLI 2.0 on most common operating systems. Do that.

If you already *have* the Azure CLI installed, you may want to update it.

Warning: `az component update` isn't supported if you installed the CLI using some of Microsoft's provided installation instructions. See the Microsoft docs for update instructions.

Next, login to your account using:

```
$ az login
```

It will tell you to open a web page and to copy a code to that page.

If the login is a success, you will see some information about all your subscriptions, including the one that is currently enabled (`"state": "Enabled"`). If the wrong one is enabled, you can switch to the right one using:

```
$ az account set --subscription <subscription name or ID>
```

Next, you will have to pick the Azure data center location where you'd like to deploy your cluster. You can get a list of all available locations using:

```
$ az account list-locations
```

Next, create an Azure "resource group" to contain all the resources (virtual machines, subnets, etc.) associated with your soon-to-be-deployed cluster. You can name it whatever you like but avoid fancy characters because they may confuse some software.

```
$ az group create --name <resource group name> --location <location name>
```

Example location names are `koreacentral` and `westeurope`.

Finally, you can deploy an ACS using something like:

```
$ az acs create --name <a made-up cluster name> \
--resource-group <name of resource group created earlier> \
--master-count 3 \
--agent-count 2 \
--admin-username ubuntu \
--agent-vm-size Standard_D2_v2 \
--dns-prefix <make up a name> \
--ssh-key-value ~/.ssh/<name>.pub \
--orchestrator-type kubernetes \
--debug --output json
```

There are more options. For help understanding all the options, use the built-in help:

```
$ az acs create --help
```

It takes a few minutes for all the resources to deploy. You can watch the progress in the Azure Portal: go to **Resource groups** (with the blue cube icon) and click on the one you created to see all the resources in it.

Optional: SSH to Your New Kubernetes Cluster Nodes

You can SSH to one of the just-deployed Kubernetes "master" nodes (virtual machines) using:

```
$ ssh -i ~/.ssh/<name>.pub ubuntu@<master-ip-address-or-hostname>
```

where you can get the IP address or hostname of a master node from the Azure Portal. For example:

```
$ ssh -i ~/.ssh/mykey123.pub ubuntu@mydnsprefix.westeurope.cloudapp.azure.com
```

Note: All the master nodes should have the *same* public IP address and hostname (also called the Master FQDN).

The "agent" nodes shouldn't get public IP addresses or hostnames, so you can't SSH to them *directly*, but you can first SSH to the master and then SSH to an agent from there. To do that, you could copy your SSH key pair to the master (a bad idea), or use SSH agent forwarding (better). To do the latter, do the following on the machine you used to SSH to the master:

```
$ echo -e "Host <FQDN of the cluster from Azure Portal>\n  ForwardAgent yes" >> ~/.
↪ssh/config
```

To verify that SSH agent forwarding works properly, SSH to the one of the master nodes and do:

```
$ echo "$SSH_AUTH_SOCK"
```

If you get an empty response, then SSH agent forwarding hasn't been set up correctly. If you get a non-empty response, then SSH agent forwarding should work fine and you can SSH to one of the agent nodes (from a master) using something like:

```
$ ssh ubuntu@k8s-agent-4AC80E97-0
```

where `k8s-agent-4AC80E97-0` is the name of a Kubernetes agent node in your Kubernetes cluster. You will have to replace it by the name of an agent node in your cluster.

Optional: Delete the Kubernetes Cluster

```
$ az acs delete \
--name <ACS cluster name> \
--resource-group <name of resource group containing the cluster>
```

Optional: Delete the Resource Group

CAUTION: You might end up deleting resources other than the ACS cluster.

```
$ az group delete \
--name <name of resource group containing the cluster>
```

Next, you can *run a BigchainDB node on your new Kubernetes cluster*.

Kubernetes Template: Deploy a Single BigchainDB Node

This page describes how to deploy the first BigchainDB node in a BigchainDB cluster, or a stand-alone BigchainDB node, using Kubernetes. It assumes you already have a running Kubernetes cluster.

If you want to add a new BigchainDB node to an existing BigchainDB cluster, refer to *the page about that*.

Below, we refer to many files by their directory and filename, such as `configuration/config-map.yaml`. Those files are files in the bigchaindb/bigchaindb repository on GitHub in the `k8s/` directory. Make sure you're getting those files from the appropriate Git branch on GitHub, i.e. the branch for the version of BigchainDB that your BigchainDB cluster is using.

Step 1: Install and Configure kubectl

kubectl is the Kubernetes CLI. If you don't already have it installed, then see the Kubernetes docs to install it.

The default location of the kubectl configuration file is `~/.kube/config`. If you don't have that file, then you need to get it.

Azure. If you deployed your Kubernetes cluster on Azure using the Azure CLI 2.0 (as per *our template*), then you can get the `~/.kube/config` file using:

```
$ az acs kubernetes get-credentials \
--resource-group <name of resource group containing the cluster> \
--name <ACS cluster name>
```

If it asks for a password (to unlock the SSH key) and you enter the correct password, but you get an error message, then try adding `--ssh-key-file ~/.ssh/<name>` to the above command (i.e. the path to the private key).

Note: **About kubectl contexts.** You might manage several Kubernetes clusters. To make it easy to switch from one to another, kubectl has a notion of "contexts," e.g. the context for cluster 1 or the context for cluster 2. To find out the current context, do:

```
$ kubectl config view
```

and then look for the `current-context` in the output. The output also lists all clusters, contexts and users. (You might have only one of each.) You can switch to a different context using:

```
$ kubectl config use-context <new-context-name>
```

You can also switch to a different context for just one command by inserting `--context <context-name>` into any kubectl command. For example:

```
$ kubectl --context k8s-bdb-test-cluster-0 get pods
```

will get a list of the pods in the Kubernetes cluster associated with the context named `k8s-bdb-test-cluster-0`.

Step 2: Connect to Your Cluster's Web UI (Optional)

You can connect to your cluster's Kubernetes Dashboard (also called the Web UI) using:

```
$ kubectl proxy -p 8001
```

or, if you prefer to be explicit about the context (explained above):

```
$ kubectl --context k8s-bdb-test-cluster-0 proxy -p 8001
```

The output should be something like `Starting to serve on 127.0.0.1:8001`. That means you can visit the dashboard in your web browser at http://127.0.0.1:8001/ui.

Step 3: Configure Your BigchainDB Node

See the page titled *How to Configure a BigchainDB Node*.

Step 4: Start the NGINX Service

- This will will give us a public IP for the cluster.
- Once you complete this step, you might need to wait up to 10 mins for the public IP to be assigned.
- You have the option to use vanilla NGINX without HTTPS support or an OpenResty NGINX integrated with 3scale API Gateway.

Step 4.1: Vanilla NGINX

- This configuration is located in the file `nginx/nginx-svc.yaml`.
- Set the `metadata.name` and `metadata.labels.name` to the value set in `ngx-instance-name` in the ConfigMap above.
- Set the `spec.selector.app` to the value set in `ngx-instance-name` in the ConfigMap followed by `-dep`. For example, if the value set in the `ngx-instance-name` is `ngx-instance-0`, set the `spec.selector.app` to `ngx-instance-0-dep`.
- Set `ngx-public-mdb-port.port` to 27017, or the port number on which you want to expose MongoDB service. Set the `ngx-public-mdb-port.targetPort` to the port number on which the Kubernetes MongoDB service will be present.
- Set `ngx-public-api-port.port` to 80, or the port number on which you want to expose BigchainDB API service. Set the `ngx-public-api-port.targetPort` to the port number on which the Kubernetes BigchainDB API service will present.
- Set `ngx-public-ws-port.port` to 81, or the port number on which you want to expose BigchainDB Websocket service. Set the `ngx-public-ws-port.targetPort` to the port number on which the BigchainDB Websocket service will be present.
- Start the Kubernetes Service:

```
$ kubectl --context k8s-bdb-test-cluster-0 apply -f nginx/nginx-svc.yaml
```

Step 4.2: OpenResty NGINX + 3scale

- You have to enable HTTPS for this one and will need an HTTPS certificate for your domain.
- You should have already created the necessary Kubernetes Secrets in the previous step (e.g. `https-certs` and `threescale-credentials`).
- This configuration is located in the file `nginx-3scale/nginx-3scale-svc.yaml`.
- Set the `metadata.name` and `metadata.labels.name` to the value set in `ngx-instance-name` in the ConfigMap above.
- Set the `spec.selector.app` to the value set in `ngx-instance-name` in the ConfigMap followed by `-dep`. For example, if the value set in the `ngx-instance-name` is `ngx-instance-0`, set the `spec.selector.app` to `ngx-instance-0-dep`.

- Set `ngx-public-mdb-port.port` to 27017, or the port number on which you want to expose MongoDB service. Set the `ngx-public-mdb-port.targetPort` to the port number on which the Kubernetes MongoDB service will be present.
- Set `ngx-public-3scale-port.port` to 8080, or the port number on which you want to let 3scale communicate with Openresty NGINX for authenctication. Set the `ngx-public-3scale-port.targetPort` to the port number on which this Openresty NGINX service will be listening to for communication with 3scale.
- Set `ngx-public-bdb-port.port` to 443, or the port number on which you want to expose BigchainDB API service. Set the `ngx-public-api-port.targetPort` to the port number on which the Kubernetes BigchainDB API service will present.
- Set `ngx-public-bdb-port-http.port` to 80, or the port number on which you want to expose BigchainDB Websocket service. Set the `ngx-public-bdb-port-http.targetPort` to the port number on which the BigchainDB Websocket service will be present.
- Start the Kubernetes Service:

```
$ kubectl --context k8s-bdb-test-cluster-0 apply -f nginx-3scale/nginx-3scale-svc.
↪yaml
```

Step 5: Assign DNS Name to the NGINX Public IP

- This step is required only if you are planning to set up multiple BigchainDB nodes or are using HTTPS certificates tied to a domain.
- The following command can help you find out if the NGINX service started above has been assigned a public IP or external IP address:

```
$ kubectl --context k8s-bdb-test-cluster-0 get svc -w
```

- Once a public IP is assigned, you can map it to a DNS name. We usually assign `bdb-test-cluster-0`, `bdb-test-cluster-1` and so on in our documentation. Let's assume that we assign the unique name of `bdb-test-cluster-0` here.

Set up DNS mapping in Azure. Select the current Azure resource group and look for the `Public IP` resource. You should see at least 2 entries there - one for the Kubernetes master and the other for the MongoDB instance. You may have to `Refresh` the Azure web page listing the resources in a resource group for the latest changes to be reflected. Select the `Public IP` resource that is attached to your service (it should have the Azure DNS prefix name along with a long random string, without the `master-ip` string), select `Configuration`, add the DNS assigned above (for example, `bdb-test-cluster-0`), click `Save`, and wait for the changes to be applied.

To verify the DNS setting is operational, you can run `nslookup <DNS name added in ConfigMap>` from your local Linux shell.

This will ensure that when you scale the replica set later, other MongoDB members in the replica set can reach this instance.

Step 6: Start the MongoDB Kubernetes Service

- This configuration is located in the file `mongodb/mongo-svc.yaml`.
- Set the `metadata.name` and `metadata.labels.name` to the value set in `mdb-instance-name` in the ConfigMap above.
- Set the `spec.selector.app` to the value set in `mdb-instance-name` in the ConfigMap followed by `-ss`. For example, if the value set in the `mdb-instance-name` is `mdb-instance-0`, set the `spec.selector.app` to `mdb-instance-0-ss`.

- Start the Kubernetes Service:

```
$ kubectl --context k8s-bdb-test-cluster-0 apply -f mongodb/mongo-svc.yaml
```

Step 7: Start the BigchainDB Kubernetes Service

- This configuration is located in the file `bigchaindb/bigchaindb-svc.yaml`.
- Set the `metadata.name` and `metadata.labels.name` to the value set in `bdb-instance-name` in the ConfigMap above.
- Set the `spec.selector.app` to the value set in `bdb-instance-name` in the ConfigMap followed by `-dep`. For example, if the value set in the `bdb-instance-name` is `bdb-instance-0`, set the `spec.selector.app` to `bdb-instance-0-dep`.
- Start the Kubernetes Service:

```
$ kubectl --context k8s-bdb-test-cluster-0 apply -f bigchaindb/bigchaindb-svc.yaml
```

Step 8: Start the NGINX Kubernetes Deployment

- NGINX is used as a proxy to both the BigchainDB and MongoDB instances in the node. It proxies HTTP requests on port 80 to the BigchainDB backend, and TCP connections on port 27017 to the MongoDB backend.
- As in step 4, you have the option to use vanilla NGINX or an OpenResty NGINX integrated with 3scale API Gateway.

Step 8.1: Vanilla NGINX

- This configuration is located in the file `nginx/nginx-dep.yaml`.
- Set the `metadata.name` and `spec.template.metadata.labels.app` to the value set in `ngx-instance-name` in the ConfigMap followed by a `-dep`. For example, if the value set in the `ngx-instance-name` is `ngx-instance-0`, set the fields to `ngx-instance-0-dep`.
- Set `MONGODB_BACKEND_HOST` env var to the value set in `mdb-instance-name` in the ConfigMap, followed by `.default.svc.cluster.local`. For example, if the value set in the `mdb-instance-name` is `mdb-instance-0`, set the `MONGODB_BACKEND_HOST` env var to `mdb-instance-0.default.svc.cluster.local`.
- Set `BIGCHAINDB_BACKEND_HOST` env var to the value set in `bdb-instance-name` in the ConfigMap, followed by `.default.svc.cluster.local`. For example, if the value set in the `bdb-instance-name` is `bdb-instance-0`, set the `BIGCHAINDB_BACKEND_HOST` env var to `bdb-instance-0.default.svc.cluster.local`.
- Start the Kubernetes Deployment:

```
$ kubectl --context k8s-bdb-test-cluster-0 apply -f nginx/nginx-dep.yaml
```

Step 8.2: OpenResty NGINX + 3scale

- This configuration is located in the file `nginx-3scale/nginx-3scale-dep.yaml`.

- Set the `metadata.name` and `spec.template.metadata.labels.app` to the value set in `ngx-instance-name` in the ConfigMap followed by a `-dep`. For example, if the value set in the `ngx-instance-name` is `ngx-instance-0`, set the fields to `ngx-instance-0-dep`.
- Set `MONGODB_BACKEND_HOST` env var to the value set in `mdb-instance-name` in the ConfigMap, followed by `.default.svc.cluster.local`. For example, if the value set in the `mdb-instance-name` is `mdb-instance-0`, set the `MONGODB_BACKEND_HOST` env var to `mdb-instance-0.default.svc.cluster.local`.
- Set `BIGCHAINDB_BACKEND_HOST` env var to the value set in `bdb-instance-name` in the ConfigMap, followed by `.default.svc.cluster.local`. For example, if the value set in the `bdb-instance-name` is `bdb-instance-0`, set the `BIGCHAINDB_BACKEND_HOST` env var to `bdb-instance-0.default.svc.cluster.local`.
- Start the Kubernetes Deployment:

```
$ kubectl --context k8s-bdb-test-cluster-0 apply -f nginx-3scale/nginx-3scale-dep.
↪yaml
```

Step 9: Create Kubernetes Storage Classes for MongoDB

MongoDB needs somewhere to store its data persistently, outside the container where MongoDB is running. Our MongoDB Docker container (based on the official MongoDB Docker container) exports two volume mounts with correct permissions from inside the container:

- The directory where the mongod instance stores its data: `/data/db`. There's more explanation in the MongoDB docs about storage.dbpath.
- The directory where the mongodb instance stores the metadata for a sharded cluster: `/data/configdb/`. There's more explanation in the MongoDB docs about sharding.configDB.

Explaining how Kubernetes handles persistent volumes, and the associated terminology, is beyond the scope of this documentation; see the Kubernetes docs about persistent volumes.

The first thing to do is create the Kubernetes storage classes.

Set up Storage Classes in Azure. First, you need an Azure storage account. If you deployed your Kubernetes cluster on Azure using the Azure CLI 2.0 (as per *our template*), then the *az acs create* command already created two storage accounts in the same location and resource group as your Kubernetes cluster. Both should have the same "storage account SKU": `Standard_LRS`. Standard storage is lower-cost and lower-performance. It uses hard disk drives (HDD). LRS means locally-redundant storage: three replicas in the same data center. Premium storage is higher-cost and higher-performance. It uses solid state drives (SSD). At the time of writing, when we created a storage account with SKU `Premium_LRS` and tried to use that, the PersistentVolumeClaim would get stuck in a "Pending" state. For future reference, the command to create a storage account is az storage account create.

The Kubernetes template for configuration of Storage Class is located in the file `mongodb/mongo-sc.yaml`.

You may have to update the `parameters.location` field in the file to specify the location you are using in Azure.

Create the required storage classes using:

```
$ kubectl --context k8s-bdb-test-cluster-0 apply -f mongodb/mongo-sc.yaml
```

You can check if it worked using `kubectl get storageclasses`.

Azure. Note that there is no line of the form `storageAccount: <azure storage account name>` under `parameters:`. When we included one and then created a PersistentVolumeClaim based on it, the PersistentVolumeClaim would get stuck in a "Pending" state. Kubernetes just looks for a storageAccount with the specified skuName and location.

Step 10: Create Kubernetes Persistent Volume Claims

Next, you will create two PersistentVolumeClaim objects `mongo-db-claim` and `mongo-configdb-claim`.

This configuration is located in the file `mongodb/mongo-pvc.yaml`.

Note how there's no explicit mention of Azure, AWS or whatever. `ReadWriteOnce` (RWO) means the volume can be mounted as read-write by a single Kubernetes node. (`ReadWriteOnce` is the *only* access mode supported by AzureDisk.) `storage:  20Gi` means the volume has a size of 20 gibibytes.

You may want to update the `spec.resources.requests.storage` field in both the files to specify a different disk size.

Create the required Persistent Volume Claims using:

```
$ kubectl --context k8s-bdb-test-cluster-0 apply -f mongodb/mongo-pvc.yaml
```

You can check its status using: `kubectl get pvc -w`

Initially, the status of persistent volume claims might be "Pending" but it should become "Bound" fairly quickly.

Step 11: Start a Kubernetes StatefulSet for MongoDB

- This configuration is located in the file `mongodb/mongo-ss.yaml`.
- Set the `spec.serviceName` to the value set in `mdb-instance-name` in the ConfigMap. For example, if the value set in the `mdb-instance-name` is `mdb-instance-0`, set the field to `mdb-instance-0`.
- Set `metadata.name`, `spec.template.metadata.name` and `spec.template.metadata.labels.app` to the value set in `mdb-instance-name` in the ConfigMap, followed by `-ss`. For example, if the value set in the `mdb-instance-name` is `mdb-instance-0`, set the fields to the value `mdb-insance-0-ss`.
- Note how the MongoDB container uses the `mongo-db-claim` and the `mongo-configdb-claim` PersistentVolumeClaims for its `/data/db` and `/data/configdb` directories (mount paths).
- Note also that we use the pod's `securityContext.capabilities.add` specification to add the `FOWNER` capability to the container. That is because the MongoDB container has the user `mongodb`, with uid `999` and group `mongodb`, with gid `999`. When this container runs on a host with a mounted disk, the writes fail when there is no user with uid `999`. To avoid this, we use the Docker feature of `--cap-add=FOWNER`. This bypasses the uid and gid permission checks during writes and allows data to be persisted to disk. Refer to the Docker docs for details.
- As we gain more experience running MongoDB in testing and production, we will tweak the `resources.limits.cpu` and `resources.limits.memory`.
- Create the MongoDB StatefulSet using:

  ```
  $ kubectl --context k8s-bdb-test-cluster-0 apply -f mongodb/mongo-ss.yaml
  ```

- It might take up to 10 minutes for the disks, specified in the Persistent Volume Claims above, to be created and attached to the pod. The UI might show that the pod has errored with the message "timeout expired waiting for volumes to attach/mount". Use the CLI below to check the status of the pod in this case, instead of the UI. This happens due to a bug in Azure ACS.

  ```
  $ kubectl --context k8s-bdb-test-cluster-0 get pods -w
  ```

Step 12: Configure Users and Access Control for MongoDB

- In this step, you will create a user on MongoDB with authorization to create more users and assign roles to them. Note: You need to do this only when setting up the first MongoDB node of the cluster.
- Find out the name of your MongoDB pod by reading the output of the `kubectl ... get pods` command at the end of the last step. It should be something like `mdb-instance-0-ss-0`.
- Log in to the MongoDB pod using:

```
$ kubectl --context k8s-bdb-test-cluster-0 exec -it <name of your MongoDB pod>␣
↪bash
```

- Open a mongo shell using the certificates already present at `/etc/mongod/ssl/`

```
$ mongo --host localhost --port 27017 --verbose --ssl \
  --sslCAFile /etc/mongod/ssl/ca.pem \
  --sslPEMKeyFile /etc/mongod/ssl/mdb-instance.pem
```

- Initialize the replica set using:

```
> rs.initiate( {
    _id : "bigchain-rs",
    members: [ {
      _id : 0,
      host  :"<hostname>:27017"
    } ]
  } )
```

  The `hostname` in this case will be the value set in `mdb-instance-name` in the ConfigMap. For example, if the value set in the `mdb-instance-name` is `mdb-instance-0`, set the `hostname` above to the value `mdb-instance-0`.

- The instance should be voted as the `PRIMARY` in the replica set (since this is the only instance in the replica set till now). This can be observed from the mongo shell prompt, which will read `PRIMARY>`.
- Create a user `adminUser` on the `admin` database with the authorization to create other users. This will only work the first time you log in to the mongo shell. For further details, see localhost exception in MongoDB.

```
PRIMARY> use admin
PRIMARY> db.createUser( {
           user: "adminUser",
           pwd: "superstrongpassword",
           roles: [ { role: "userAdminAnyDatabase", db: "admin" } ]
         } )
```

- Exit and restart the mongo shell using the above command. Authenticate as the `adminUser` we created earlier:

```
PRIMARY> use admin
PRIMARY> db.auth("adminUser", "superstrongpassword")
```

  `db.auth()` returns 0 when authentication is not successful, and 1 when successful.

- We need to specify the user name *as seen in the certificate* issued to the BigchainDB instance in order to authenticate correctly. Use the following `openssl` command to extract the user name from the certificate:

```
$ openssl x509 -in <path to the bigchaindb certificate> \
  -inform PEM -subject -nameopt RFC2253
```

You should see an output line that resembles:

```
subject= emailAddress=dev@bigchaindb.com,CN=test-bdb-ssl,OU=BigchainDB-Instance,
↪O=BigchainDB GmbH,L=Berlin,ST=Berlin,C=DE
```

The `subject` line states the complete user name we need to use for creating the user on the mongo shell as follows:

```
PRIMARY> db.getSiblingDB("$external").runCommand( {
           createUser: 'emailAddress=dev@bigchaindb.com,CN=test-bdb-ssl,
↪OU=BigchainDB-Instance,O=BigchainDB GmbH,L=Berlin,ST=Berlin,C=DE',
           writeConcern: { w: 'majority' , wtimeout: 5000 },
           roles: [
             { role: 'clusterAdmin', db: 'admin' },
             { role: 'readWriteAnyDatabase', db: 'admin' }
           ]
         } )
```

- You can similarly create users for MongoDB Monitoring Agent and MongoDB Backup Agent. For example:

```
PRIMARY> db.getSiblingDB("$external").runCommand( {
           createUser: 'emailAddress=dev@bigchaindb.com,CN=test-mdb-mon-ssl,
↪OU=MongoDB-Mon-Instance,O=BigchainDB GmbH,L=Berlin,ST=Berlin,C=DE',
           writeConcern: { w: 'majority' , wtimeout: 5000 },
           roles: [
             { role: 'clusterMonitor', db: 'admin' }
           ]
         } )

PRIMARY> db.getSiblingDB("$external").runCommand( {
           createUser: 'emailAddress=dev@bigchaindb.com,CN=test-mdb-bak-ssl,
↪OU=MongoDB-Bak-Instance,O=BigchainDB GmbH,L=Berlin,ST=Berlin,C=DE',
           writeConcern: { w: 'majority' , wtimeout: 5000 },
           roles: [
             { role: 'backup',    db: 'admin' }
           ]
         } )
```

Step 13: Start a Kubernetes Deployment for MongoDB Monitoring Agent

- This configuration is located in the file `mongodb-monitoring-agent/mongo-mon-dep.yaml`.
- Set `metadata.name`, `spec.template.metadata.name` and `spec.template.metadata.labels.app` to the value set in `mdb-mon-instance-name` in the ConfigMap, followed by `-dep`. For example, if the value set in the `mdb-mon-instance-name` is `mdb-mon-instance-0`, set the fields to the value `mdb-mon-instance-0-dep`.
- Start the Kubernetes Deployment using:

```
$ kubectl --context k8s-bdb-test-cluster-0 apply -f mongodb-monitoring-agent/
↪mongo-mon-dep.yaml
```

Step 14: Start a Kubernetes Deployment for MongoDB Backup Agent

- This configuration is located in the file `mongodb-backup-agent/mongo-backup-dep.yaml`.

- Set `metadata.name`, `spec.template.metadata.name` and `spec.template.metadata.labels.app` to the value set in `mdb-bak-instance-name` in the ConfigMap, followed by `-dep`. For example, if the value set in the `mdb-bak-instance-name` is `mdb-bak-instance-0`, set the fields to the value `mdb-bak-instance-0-dep`.
- Start the Kubernetes Deployment using:

```
$ kubectl --context k8s-bdb-test-cluster-0 apply -f mongodb-backup-agent/mongo-
↪backup-dep.yaml
```

Step 15: Start a Kubernetes Deployment for BigchainDB

- This configuration is located in the file `bigchaindb/bigchaindb-dep.yaml`.
- Set `metadata.name` and `spec.template.metadata.labels.app` to the value set in `bdb-instance-name` in the ConfigMap, followed by `-dep`. For example, if the value set in the `bdb-instance-name` is `bdb-instance-0`, set the fields to the value `bdb-insance-0-dep`.
- Set the value of `BIGCHAINDB_KEYPAIR_PRIVATE` (not base64-encoded). (In the future, we'd like to pull the BigchainDB private key from the Secret named `bdb-private-key`, but a Secret can only be mounted as a file, so BigchainDB Server would have to be modified to look for it in a file.)
- As we gain more experience running BigchainDB in testing and production, we will tweak the `resources.limits` values for CPU and memory, and as richer monitoring and probing becomes available in BigchainDB, we will tweak the `livenessProbe` and `readinessProbe` parameters.
- Create the BigchainDB Deployment using:

```
$ kubectl --context k8s-bdb-test-cluster-0 apply -f bigchaindb/bigchaindb-dep.yaml
```

- You can check its status using the command `kubectl get deployments -w`

Step 16: Configure the MongoDB Cloud Manager

Refer to the *documentation* for details on how to configure the MongoDB Cloud Manager to enable monitoring and backup.

Step 17: Verify the BigchainDB Node Setup

Step 17.1: Testing Internally

To test the setup of your BigchainDB node, you could use a Docker container that provides utilities like `nslookup`, `curl` and `dig`. For example, you could use a container based on our bigchaindb/toolbox image. (The corresponding Dockerfile is in the `bigchaindb/bigchaindb` repository on GitHub.) You can use it as below to get started immediately:

```
$ kubectl --context k8s-bdb-test-cluster-0 \
   run -it toolbox \
   --image bigchaindb/toolbox \
   --image-pull-policy=Always \
   --restart=Never --rm
```

It will drop you to the shell prompt.

To test the MongoDB instance:

```
$ nslookup mdb-instance-0

$ dig +noall +answer _mdb-port._tcp.mdb-instance-0.default.svc.cluster.local SRV

$ curl -X GET http://mdb-instance-0:27017
```

The `nslookup` command should output the configured IP address of the service (in the cluster). The `dig` command should return the configured port numbers. The `curl` command tests the availability of the service.

To test the BigchainDB instance:

```
$ nslookup bdb-instance-0

$ dig +noall +answer _bdb-port._tcp.bdb-instance-0.default.svc.cluster.local SRV

$ curl -X GET http://bdb-instance-0:9984
```

To test the NGINX instance:

```
$ nslookup ngx-instance-0

$ dig +noall +answer _ngx-public-mdb-port._tcp.ngx-instance-0.default.svc.cluster.
↪local SRV

$ dig +noall +answer _ngx-public-bdb-port._tcp.ngx-instance-0.default.svc.cluster.
↪local SRV

$ curl -X GET http://ngx-instance-0:27017
```

The curl command should result get the response `curl: (7) Failed to connect to ngx-instance-0 port 27017: Connection refused`.

If you ran the vanilla NGINX instance, run:

```
$ curl -X GET http://ngx-instance-0:80
```

If you ran the OpenResty NGINX + 3scale instance, run:

```
$ curl -X GET https://ngx-instance-0
```

Step 17.2: Testing Externally

Check the MongoDB monitoring and backup agent on the MongoDB Cloud Manager portal to verify they are working fine.

Try to access the `<DNS/IP of your exposed BigchainDB service endpoint>:80` on your browser. You should receive a JSON response that shows the BigchainDB server version, among other things.

Use the Python Driver to send some transactions to the BigchainDB node and verify that your node or cluster works as expected.

Kubernetes Template: Add a BigchainDB Node to an Existing BigchainDB Cluster

This page describes how to deploy a BigchainDB node using Kubernetes, and how to add that node to an existing BigchainDB cluster. It assumes you already have a running Kubernetes cluster where you can deploy the new BigchainDB node.

If you want to deploy the first BigchainDB node in a BigchainDB cluster, or a stand-alone BigchainDB node, then see *the page about that*.

Terminology Used

`existing cluster` will refer to one of the existing Kubernetes clusters hosting one of the existing BigchainDB nodes.

`ctx-1` will refer to the kubectl context of the existing cluster.

`new cluster` will refer to the new Kubernetes cluster that will run a new BigchainDB node (including a BigchainDB instance and a MongoDB instance).

`ctx-2` will refer to the kubectl context of the new cluster.

`new MongoDB instance` will refer to the MongoDB instance in the new cluster.

`existing MongoDB instance` will refer to the MongoDB instance in the existing cluster.

`new BigchainDB instance` will refer to the BigchainDB instance in the new cluster.

`existing BigchainDB instance` will refer to the BigchainDB instance in the existing cluster.

Step 1: Prerequisites

- A public/private key pair for the new BigchainDB instance.
- The public key should be shared offline with the other existing BigchainDB nodes in the existing BigchainDB cluster.
- You will need the public keys of all the existing BigchainDB nodes.
- A new Kubernetes cluster setup with kubectl configured to access it.
- Some familiarity with deploying a BigchainDB node on Kubernetes. See our *other docs about that*.

Note: If you are managing multiple Kubernetes clusters, from your local system, you can run `kubectl config view` to list all the contexts that are available for the local kubectl. To target a specific cluster, add a `--context` flag to the kubectl CLI. For example:

```
$ kubectl --context ctx-1 apply -f example.yaml
$ kubectl --context ctx-2 apply -f example.yaml
$ kubectl --context ctx-1 proxy --port 8001
$ kubectl --context ctx-2 proxy --port 8002
```

Step 2: Prepare the New Kubernetes Cluster

Follow the steps in the sections to set up Storage Classes and Persistent Volume Claims, and to run MongoDB in the new cluster:

1. *Add Storage Classes*.
2. *Add Persistent Volume Claims*.
3. *Create the Config Map*.
4. *Run MongoDB instance*.

Step 3: Add the New MongoDB Instance to the Existing Replica Set

Note that by `replica set`, we are referring to the MongoDB replica set, not a Kubernetes' `ReplicaSet`.

If you are not the administrator of an existing BigchainDB node, you will have to coordinate offline with an existing administrator so that they can add the new MongoDB instance to the replica set.

Add the new instance of MongoDB from an existing instance by accessing the `mongo` shell.

```
$ kubectl --context ctx-1 exec -it mdb-0 -c mongodb -- /bin/bash
root@mdb-0# mongo --port 27017
```

One can only add members to a replica set from the `PRIMARY` instance. The `mongo` shell prompt should state that this is the primary member in the replica set. If not, then you can use the `rs.status()` command to find out who the primary is and login to the `mongo` shell in the primary.

Run the `rs.add()` command with the FQDN and port number of the other instances:

```
PRIMARY> rs.add("<fqdn>:<port>")
```

Step 4: Verify the Replica Set Membership

You can use the `rs.conf()` and the `rs.status()` commands available in the mongo shell to verify the replica set membership.

The new MongoDB instance should be listed in the membership information displayed.

Step 5: Start the New BigchainDB Instance

Get the file `bigchaindb-dep.yaml` from GitHub using:

```
$ wget https://raw.githubusercontent.com/bigchaindb/bigchaindb/master/k8s/bigchaindb/
↪bigchaindb-dep.yaml
```

Note that we set the `BIGCHAINDB_DATABASE_HOST` to `mdb` which is the name of the MongoDB service defined earlier.

Edit the `BIGCHAINDB_KEYPAIR_PUBLIC` with the public key of this instance, the `BIGCHAINDB_KEYPAIR_PRIVATE` with the private key of this instance and the `BIGCHAINDB_KEYRING` with a `:` delimited list of all the public keys in the BigchainDB cluster.

Create the required Deployment using:

```
$ kubectl --context ctx-2 apply -f bigchaindb-dep.yaml
```

You can check its status using the command `kubectl get deploy -w`

Step 6: Restart the Existing BigchainDB Instance(s)

Add the public key of the new BigchainDB instance to the keyring of all the existing BigchainDB instances and update the BigchainDB instances using:

```
$ kubectl --context ctx-1 replace -f bigchaindb-dep.yaml
```

This will create a "rolling deployment" in Kubernetes where a new instance of BigchainDB will be created, and if the health check on the new instance is successful, the earlier one will be terminated. This ensures that there is zero downtime during updates.

You can SSH to an existing BigchainDB instance and run the `bigchaindb show-config` command to check that the keyring is updated.

Step 7: Run NGINX as a Deployment

Please see *this page* to set up NGINX in your new node.

Step 8: Test Your New BigchainDB Node

Please refer to the testing steps *here* to verify that your new BigchainDB node is working as expected.

Kubernetes Template: Upgrade all Software in a BigchainDB Node

This page outlines how to upgrade all the software associated with a BigchainDB node running on Kubernetes, including host operating systems, Docker, Kubernetes, and BigchainDB-related software.

Upgrade Host OS, Docker and Kubernetes

Some Kubernetes installation & management systems can do full or partial upgrades of host OSes, Docker, or Kubernetes, e.g. Tectonic, Rancher, and Kubo. Consult the documentation for your system.

Azure Container Service (ACS). On Dec. 15, 2016, a Microsoft employee wrote: "In the coming months we [the Azure Kubernetes team] will be building managed updates in the ACS service." At the time of writing, managed updates were not yet available, but you should check the latest ACS documentation to see what's available now. Also at the time of writing, ACS only supported Ubuntu as the host (master and agent) operating system. You can upgrade Ubuntu and Docker on Azure by SSHing into each of the hosts, as documented on *another page*.

In general, you can SSH to each host in your Kubernetes Cluster to update the OS and Docker.

Note: Once you are in an SSH session with a host, the `docker info` command is a handy way to detemine the host OS (including version) and the Docker version.

When you want to upgrade the software on a Kubernetes node, you should "drain" the node first, i.e. tell Kubernetes to gracefully terminate all pods on the node and mark it as unscheduleable (so no new pods get put on the node during its downtime).

```
kubectl drain $NODENAME
```

There are more details in the Kubernetes docs, including instructions to make the node scheduleable again.

To manually upgrade the host OS, see the docs for that OS.

To manually upgrade Docker, see the Docker docs.

To manually upgrade all Kubernetes software in your Kubernetes cluster, see the Kubernetes docs.

Upgrade BigchainDB-Related Software

We use Kubernetes "Deployments" for NGINX, BigchainDB, and most other BigchainDB-related software. The only exception is MongoDB; we use a Kubernetes StatefulSet for that.

The nice thing about Kubernetes Deployments is that Kubernetes can manage most of the upgrade process. A typical upgrade workflow for a single Deployment would be:

```
$ KUBE_EDITOR=nano kubectl edit deployment/<name of Deployment>
```

The `kubectl edit` command opens the specified editor (nano in the above example), allowing you to edit the specified Deployment *in the Kubernetes cluster*. You can change the version tag on the Docker image, for example. Don't forget to save your edits before exiting the editor. The Kubernetes docs have more information about Deployments (including updating them).

The upgrade story for the MongoDB StatefulSet is *different*. (This is because MongoDB has persistent state, which is stored in some storage associated with a PersistentVolumeClaim.) At the time of writing, StatefulSets were still in beta, and they did not support automated image upgrade (Docker image tag upgrade). We expect that to change. Rather than trying to keep these docs up-to-date, we advise you to check out the current Kubernetes docs about updating containers in StatefulSets.

Log Analytics on Azure

This section documents how to create and configure a Log Analytics workspace on Azure, for a Kubernetes-based deployment. The documented approach is based on an integration of Microsoft's Operations Management Suite (OMS) with a Kubernetes-based Azure Container Service cluster.

The *References* section (below) contains links to more detailed documentation on Azure, and Kubernetes.

There are three main steps involved:

1. Create a workspace (`LogAnalyticsOMS`).
2. Create a `ContainersOMS` solution under the workspace.
3. Deploy the OMS agent(s).

Steps 1 and 2 rely on Azure Resource Manager templates and can be done with one template so we'll cover them together. Step 3 relies on a Kubernetes DaemonSet and will be covered separately.

Minimum Requirements

This document assumes that you have already deployed a Kubernetes cluster, and that you have the Kubernetes command line interface `kubectl` installed.

Creating a Workspace and Adding a Containers Solution

For the sake of this document and example, we'll assume an existing resource group named:

- `resource_group`

and the workspace we'll create will be named:

- `work_space`

If you feel creative you may replace these names by more interesting ones.

```
$ az group deployment create --debug \
    --resource-group resource_group \
    --name "Microsoft.LogAnalyticsOMS" \
    --template-file log_analytics_oms.json \
    --parameters @log_analytics_oms.parameters.json
```

An example of a simple template file (`--template-file`):

```
{
    "$schema": "http://schema.management.azure.com/schemas/2014-04-01-preview/
↪deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
    "sku": {
        "type": "String"
    },
    "workspaceName": {
        "type": "String"
    },
    "solutionType": {
        "type": "String"
    },
    },
    "resources": [
    {
        "apiVersion": "2015-03-20",
        "type": "Microsoft.OperationalInsights/workspaces",
        "name": "[parameters('workspaceName')]",
        "location": "[resourceGroup().location]",
        "properties": {
            "sku": {
                "name": "[parameters('sku')]"
            }
        },
        "resources": [
            {
                "apiVersion": "2015-11-01-preview",
                "location": "[resourceGroup().location]",
                "name": "[Concat(parameters('solutionType'), '(', parameters(
↪'workspaceName'), ')')]",
                "type": "Microsoft.OperationsManagement/solutions",
                "id": "[Concat(resourceGroup().id, '/providers/Microsoft.
↪OperationsManagement/solutions/', parameters('solutionType'), '(', parameters(
↪'workspaceName'), ')')]",
                "dependsOn": [
                    "[concat('Microsoft.OperationalInsights/workspaces/', parameters(
↪'workspaceName'))]"
                ],
                "properties": {
                    "workspaceResourceId": "[resourceId('Microsoft.
↪OperationalInsights/workspaces/', parameters('workspaceName'))]"
                },
                "plan": {
                    "publisher": "Microsoft",
                    "product": "[Concat('OMSGallery/', parameters('solutionType'))]",
                    "name": "[Concat(parameters('solutionType'), '(', parameters(
↪'workspaceName'), ')')]",
                    "promotionCode": ""
                }
            }
        ]
    }
    ]
}
```

An example of the associated parameter file (`--parameters`):

```
{
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/
↪deploymentParameters.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
    "sku": {
        "value": "Free"
    },
    "workspaceName": {
        "value": "work_space"
    },
    "solutionType": {
        "value": "Containers"
    },
    }
}
```

Deploy the OMS Agents

To deploy an OMS agent, two important pieces of information are needed:

- workspace id
- workspace key

You can obtain the workspace id using:

```
$ az resource show \
    --resource-group resource_group
    --resource-type Microsoft.OperationalInsights/workspaces
    --name work_space \
    | grep customerId
"customerId": "12345678-1234-1234-1234-123456789012",
```

Until we figure out a way to obtain the *workspace key* via the command line, you can get it via the OMS Portal. To get to the OMS Portal, go to the Azure Portal and click on:

Resource Groups > (Your k8s cluster's resource group) > Log analytics (OMS) > (Name of the only item listed) > OMS Workspace > OMS Portal

(Let us know if you find a faster way.) Then see Microsoft's instructions to obtain your workspace ID and key (via the OMS Portal).

Once you have the workspace id and key, you can include them in the following YAML file (`oms-daemonset.yaml`):

```
# oms-daemonset.yaml
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: omsagent
spec:
  template:
    metadata:
      labels:
        app: omsagent
    spec:
      containers:
      - env:
        - name: WSID
          value: <workspace_id>
        - name: KEY
          value: <workspace_key>
        image: microsoft/oms
        name: omsagent
        ports:
        - containerPort: 25225
          protocol: TCP
        securityContext:
          privileged: true
        volumeMounts:
        - mountPath: /var/run/docker.sock
          name: docker-sock
      volumes:
      - name: docker-sock
        hostPath:
          path: /var/run/docker.sock
```

To deploy the OMS agents (one per Kubernetes node, i.e. one per computer), simply run the following command:

```
$ kubectl create -f oms-daemonset.yaml
```

Search the OMS Logs

OMS should now be getting, storing and indexing all the logs from all the containers in your Kubernetes cluster. You can search the OMS logs from the Azure Portal or the OMS Portal, but at the time of writing, there was more functionality in the OMS Portal (e.g. the ability to create an Alert based on a search).

There are instructions to get to the OMS Portal in the section titled *Deploy the OMS Agents* above. Once you're in the OMS Portal, click on **Log Search** and enter a query. Here are some example queries:

All logging messages containing the strings "critical" or "error" (not case-sensitive):

```
Type=ContainerLog (critical OR error)
```

Note: You can filter the results even more by clicking on things in the left sidebar. For OMS Log Search syntax help, see the Log Analytics search reference.

All logging messages containing the string "error" but not "404":

```
Type=ContainerLog error NOT(404)
```

All logging messages containing the string "critical" but not "CriticalAddonsOnly":

```
Type=ContainerLog critical NOT(CriticalAddonsOnly)
```

All logging messages from containers running the Docker image bigchaindb/nginx_3scale:1.3, containing the string "GET" but not the strings "Go-http-client" or "runscope" (where those exclusions filter out tests by Kubernetes and Runscope):

```
Type=ContainerLog Image="bigchaindb/nginx_3scale:1.3" GET
NOT("Go-http-client") NOT(runscope)
```

Note: We wrote a small Python 3 script to analyze the logs found by the above NGINX search. It's in `k8s/logging-and-monitoring/analyze.py`. The docsting at the top of the script explains how to use it.

Create an Email Alert

Once you're satisfied with an OMS Log Search query string, click the **Alert** icon in the top menu, fill in the form, and click **Save** when you're done.

Some Useful Management Tasks

List workspaces:

```
$ az resource list \
    --resource-group resource_group \
    --resource-type Microsoft.OperationalInsights/workspaces
```

List solutions:

```
$ az resource list \
    --resource-group resource_group \
    --resource-type Microsoft.OperationsManagement/solutions
```

Delete the containers solution:

```
$ az group deployment delete --debug \
    --resource-group resource_group \
    --name Microsoft.ContainersOMS
```

```
$ az resource delete \
    --resource-group resource_group \
    --resource-type Microsoft.OperationsManagement/solutions \
    --name "Containers(work_space)"
```

Delete the workspace:

```
$ az group deployment delete --debug \
    --resource-group resource_group \
    --name Microsoft.LogAnalyticsOMS
```

```
$ az resource delete \
    --resource-group resource_group \
    --resource-type Microsoft.OperationalInsights/workspaces \
    --name work_space
```

References

- Monitor an Azure Container Service cluster with Microsoft Operations Management Suite (OMS)
- Manage Log Analytics using Azure Resource Manager templates
- azure commands for deployments (`az group deployment`)
- Understand the structure and syntax of Azure Resource Manager templates
- Kubernetes DaemonSet

How to Install & Configure Easy-RSA

We use Easy-RSA version 3, a wrapper over complex `openssl` commands. Easy-RSA is available on GitHub and licensed under GPLv2.

Step 1: Install Easy-RSA Dependencies

The only dependency for Easy-RSA v3 is `openssl`, which is available from the `openssl` package on Ubuntu and other Debian-based operating systems, i.e. you can install it using:

```
sudo apt-get update

sudo apt-get install openssl
```

Step 2: Install Easy-RSA

Make sure you're in the directory where you want Easy-RSA to live, then download it and extract it within that directory:

```
wget https://github.com/OpenVPN/easy-rsa/archive/3.0.1.tar.gz

tar xzvf 3.0.1.tar.gz

rm 3.0.1.tar.gz
```

There should now be a directory named `easy-rsa-3.0.1` in your current directory.

Step 3: Customize the Easy-RSA Configuration

We now create a config file named `vars` by copying the existing `vars.example` file and then editing it. You should change the country, province, city, org and email to the correct values for your organisation. (Note: The country, province, city, org and email are part of the Distinguished Name (DN).) The comments in the file explain what each of the variables mean.

```
cd easy-rsa-3.0.1/easyrsa3

cp vars.example vars

echo 'set_var EASYRSA_DN "org"' >> vars
echo 'set_var EASYRSA_KEY_SIZE 4096' >> vars

echo 'set_var EASYRSA_REQ_COUNTRY "DE"' >> vars
echo 'set_var EASYRSA_REQ_PROVINCE "Berlin"' >> vars
echo 'set_var EASYRSA_REQ_CITY "Berlin"' >> vars
echo 'set_var EASYRSA_REQ_ORG "BigchainDB GmbH"' >> vars
echo 'set_var EASYRSA_REQ_OU "IT"' >> vars
echo 'set_var EASYRSA_REQ_EMAIL "dev@bigchaindb.com"' >> vars
```

Note: Later, when building a CA or generating a certificate signing request, you will be prompted to enter a value for the OU (or to accept the default). You should change the default OU from `IT` to one of the following, as appropriate: `ROOT-CA`, `MongoDB-Instance`, `BigchainDB-Instance`, `MongoDB-Mon-Instance` or `MongoDB-Backup-Instance`. To understand why, see the MongoDB Manual. There are reminders to do this in the relevant docs.

Step 4: Maybe Edit x509-types/server

Warning: Only do this step if you are setting up a self-signed CA.

Edit the file `x509-types/server` and change `extendedKeyUsage = serverAuth` to `extendedKeyUsage = serverAuth,clientAuth`. See the MongoDB documentation about x.509 authentication to understand why.

Configure MongoDB Cloud Manager for Monitoring and Backup

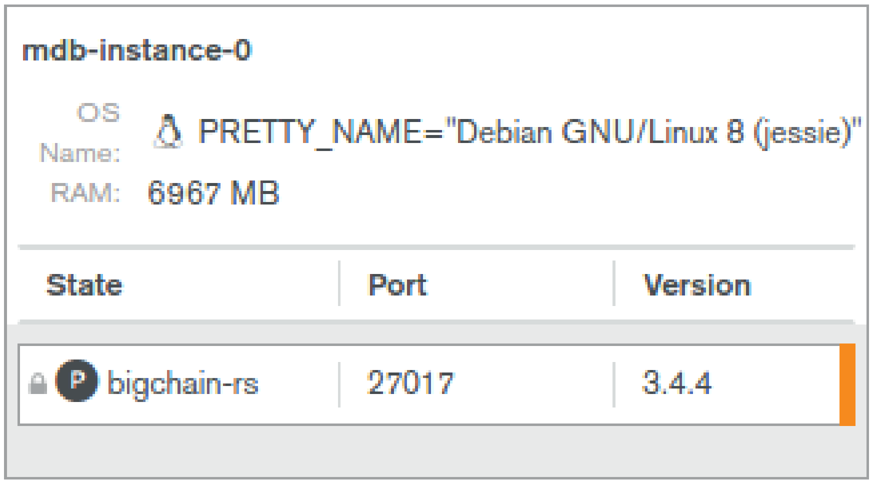
This document details the steps required to configure MongoDB Cloud Manager to enable monitoring and backup of data in a MongoDB Replica Set.

Configure MongoDB Cloud Manager for Monitoring

- Once the Monitoring Agent is up and running, open MongoDB Cloud Manager.
- Click `Login` under `MongoDB Cloud Manager` and log in to the Cloud Manager.
- Select the group from the dropdown box on the page.
- Go to Settings, Group Settings and add a `Preferred Hostnames` entry as a regexp based on the `mdb-instance-name` of the nodes in your cluster. It may take up to 5 mins till this setting takes effect. You may refresh the browser window and verify whether the changes have been saved or not.

For example, for the nodes in a cluster that are named `mdb-instance-0`, `mdb-instance-1` and so on, a regex like `^mdb-instance-[0-9]{1,2}$` is recommended.

- Next, click the `Deployment` tab, and then the `Manage Existing` button.
- On the `Import your deployment for monitoring` page, enter the hostname to be the same as the one set for `mdb-instance-name` in the global ConfigMap for a node. For example, if the `mdb-instance-name` is set to `mdb-instance-0`, enter `mdb-instance-0` as the value in this field.
- Enter the port number as `27017`, with no authentication.
- If you have authentication enabled, select the option to enable authentication and specify the authentication mechanism as per your deployment. The default BigchainDB production deployment currently supports `X.509 Client Certificate` as the authentication mechanism.
- If you have TLS enabled, select the option to enable TLS/SSL for MongoDB connections, and click `Continue`. This should already be selected for you in case you selected `X.509 Client Certificate` above.
- Wait a minute or two for the deployment to be found and then click the `Continue` button again.
- Verify that you see your process on the Cloud Manager UI. It should look something like this:



mdb-instance-0

OS Name: PRETTY_NAME="Debian GNU/Linux 8 (jessie)"
RAM: 6967 MB

State	Port	Version
P bigchain-rs	27017	3.4.4

- Click `Continue`.
- Verify on the UI that data is being sent by the monitoring agent to the Cloud Manager. It may take upto 5 minutes for data to appear on the UI.

Configure MongoDB Cloud Manager for Backup

- Once the Backup Agent is up and running, open MongoDB Cloud Manager.
- Click `Login` under `MongoDB Cloud Manager` and log in to the Cloud Manager.
- Select the group from the dropdown box on the page.
- Click `Backup` tab.
- Hover over the `Status` column of your backup and click `Start` to start the backup.
- Select the replica set on the side pane.
- If you have authentication enabled, select the authentication mechanism as per your deployment. The default BigchainDB production deployment currently supports `X.509 Client Certificate` as the authentication mechanism.

- If you have TLS enabled, select the checkbox `Replica set allows TLS/SSL connections`. This should be selected by default in case you selected `X.509 Client Certificate` as the auth mechanism above.
- Choose the `WiredTiger` storage engine.
- Verify the details of your MongoDB instance and click on `Start`.
- It may take up to 5 minutes for the backup process to start. During this process, the UI will show the status of the backup process.
- Verify that data is being backed up on the UI.

How to Configure a BigchainDB Node

This page outlines the steps to set a bunch of configuration settings in your BigchainDB node. They are pushed to the Kubernetes cluster in two files, named `config-map.yaml` (a set of ConfigMaps) and `secret.yaml` (a set of Secrets). They are stored in the Kubernetes cluster's key-value store (etcd).

Make sure you did all the things listed in the section titled *Things Each Node Operator Must Do* (including generation of all the SSL certificates needed for MongoDB auth).

Edit config-map.yaml

Make a copy of the file `k8s/configuration/config-map.yaml` and edit the data values in the various ConfigMaps. That file already contains many comments to help you understand each data value, but we make some additional remarks on some of the values below.

Note: None of the data values in `config-map.yaml` need to be base64-encoded. (This is unlike `secret.yaml`, where all data values must be base64-encoded. This is true of all Kubernetes ConfigMaps and Secrets.)

vars.mdb-instance-name and Similar

Your BigchainDB cluster organization should have a standard way of naming instances, so the instances in your BigchainDB node should conform to that standard (i.e. you can't just make up some names). There are some things worth noting about the `mdb-instance-name`:

- MongoDB reads the local `/etc/hosts` file while bootstrapping a replica set to resolve the hostname provided to the `rs.initiate()` command. It needs to ensure that the replica set is being initialized in the same instance where the MongoDB instance is running.
- We use the value in the `mdb-instance-name` field to achieve this.
- This field will be the DNS name of your MongoDB instance, and Kubernetes maps this name to its internal DNS.
- This field will also be used by other MongoDB instances when forming a MongoDB replica set.
- We use `mdb-instance-0`, `mdb-instance-1` and so on in our documentation. Your BigchainDB cluster may use a different naming convention.

bdb-config.bdb-keyring

This lists the BigchainDB public keys of all *other* nodes in your BigchainDB cluster (not including the public key of your BigchainDB node). Cases:

- If you're deploying the first node in the cluster, the value should be `""` (an empty string).
- If you're deploying the second node in the cluster, the value should be the BigchainDB public key of the first/original node in the cluster. For example, `"EPQk5i5yYpoUwGVM8VKZRjM8CYxB6j8Lu8i8SG7kGGce"`
- If there are two or more other nodes already in the cluster, the value should be a colon-separated list of the BigchainDB public keys of those other nodes. For example, `"DPjpKbmbPYPKVAuf6VSkqGCf5jzrEh69Ldef6TrLwsEQ:EPQk5i5yYpoUwGVM8VKZRjM8CYxB6j8Lu8i8SG7k`

bdb-config.bdb-user

This is the user name that BigchainDB uses to authenticate itself to the backend MongoDB database.

We need to specify the user name *as seen in the certificate* issued to the BigchainDB instance in order to authenticate correctly. Use the following `openssl` command to extract the user name from the certificate:

```
$ openssl x509 -in <path to the bigchaindb certificate> \
  -inform PEM -subject -nameopt RFC2253
```

You should see an output line that resembles:

```
subject= emailAddress=dev@bigchaindb.com,CN=test-bdb-ssl,OU=BigchainDB-Instance,
↪O=BigchainDB GmbH,L=Berlin,ST=Berlin,C=DE
```

The `subject` line states the complete user name we need to use for this field (`bdb-config.bdb-user`), i.e.

```
emailAddress=dev@bigchaindb.com,CN=test-bdb-ssl,OU=BigchainDB-Instance,O=BigchainDB␣
↪GmbH,L=Berlin,ST=Berlin,C=DE
```

Edit secret.yaml

Make a copy of the file `k8s/configuration/secret.yaml` and edit the data values in the various Secrets. That file includes many comments to explain the required values. **In particular, note that all values must be base64-encoded.** There are tips at the top of the file explaining how to convert values into base64-encoded values.

Your BigchainDB node might not need all the Secrets. For example, if you plan to access the BigchainDB API over HTTP, you don't need the `https-certs` Secret. You can delete the Secrets you don't need, or set their data values to `""`.

Note that `ca.pem` is just another name for `ca.crt` (the certificate of your BigchainDB cluster's self-signed CA).

threescale-credentials.*

If you're not using 3scale, you can delete the `threescale-credentials` Secret or leave all the values blank (`""`).

If you *are* using 3scale, you can get the value for `frontend-api-dns-name` using something like `echo "your.nodesubdomain.net" | base64 -w 0`

To get the values for `secret-token`, `service-id`, `version-header` and `provider-key`, login to your 3scale admin, then click **APIs** and click on **Integration** for the relevant API. Scroll to the bottom of the page and click the small link in the lower right corner, labelled **Download the NGINX Config files**. You'll get a `.zip` file. Unzip it, then open the `.conf` file and the `.lua` file. You should be able to find all the values in those files. You have to be careful because it will have values for *all* your APIs, and some values vary from API to API. The `version-header` is the timestamp in a line that looks like:

```
proxy_set_header  X-3scale-Version "2017-06-28T14:57:34Z";
```

Deploy Your config-map.yaml and secret.yaml

You can deploy your edited `config-map.yaml` and `secret.yaml` files to your Kubernetes cluster using the commands:

```
$ kubectl apply -f config-map.yaml

$ kubectl apply -f secret.yaml
```

CHAPTER 6

Develop & Test BigchainDB Server

Set Up & Run a Dev/Test Node

This page explains how to set up a minimal local BigchainDB node for development and testing purposes.

The BigchainDB core dev team develops BigchainDB on recent Ubuntu and Fedora distributions, so we recommend you use one of those. BigchainDB Server doesn't work on Windows and Mac OS X (unless you use a VM or containers).

Option A: Using a Local Dev Machine

Read through the BigchainDB CONTRIBUTING.md file. It outlines the steps to setup a machine for developing and testing BigchainDB.

With RethinkDB

Create a default BigchainDB config file (in `$HOME/.bigchaindb`):

```
$ bigchaindb -y configure rethinkdb
```

Note: The BigchainDB CLI and the BigchainDB Configuration Settings are documented elsewhere. (Click the links.)

Start RethinkDB using:

```
$ rethinkdb
```

You can verify that RethinkDB is running by opening the RethinkDB web interface in your web browser. It should be at http://localhost:8080/

To run BigchainDB Server, do:

```
$ bigchaindb start
```

You can run all the unit tests to test your installation.

The BigchainDB CONTRIBUTING.md file has more details about how to contribute.

With MongoDB

Create a default BigchainDB config file (in `$HOME/.bigchaindb`):

```
$ bigchaindb -y configure mongodb
```

Note: The BigchainDB CLI and the BigchainDB Configuration Settings are documented elsewhere. (Click the links.)

Start MongoDB **3.4+** using:

```
$ mongod --replSet=bigchain-rs
```

You can verify that MongoDB is running correctly by checking the output of the previous command for the line:

```
waiting for connections on port 27017
```

To run BigchainDB Server, do:

```
$ bigchaindb start
```

You can run all the unit tests to test your installation.

The BigchainDB CONTRIBUTING.md file has more details about how to contribute.

Option B: Using a Local Dev Machine and Docker

You need to have recent versions of Docker Engine and (Docker) Compose.

Build the images:

```
docker-compose build
```

Docker with RethinkDB

Note: If you're upgrading BigchainDB and have previously already built the images, you may need to rebuild them after the upgrade to install any new dependencies.

Start RethinkDB:

```
docker-compose up -d rdb
```

The RethinkDB web interface should be accessible at http://localhost:58080/. Depending on which platform, and/or how you are running docker, you may need to change `localhost` for the `ip` of the machine that is running docker. As a dummy example, if the `ip` of that machine was `0.0.0.0`, you would access the web interface at: http://0.0.0.0:58080/.

Start a BigchainDB node:

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

You can monitor the logs:

```
docker-compose logs -f bdb-rdb
```

If you wish to run the tests:

```
docker-compose run --rm bdb-rdb py.test -v -n auto
```

Docker with MongoDB

Start MongoDB:

```
docker-compose up -d mdb
```

MongoDB should now be up and running. You can check the port binding for the MongoDB driver port using:

```
$ docker-compose port mdb 27017
```

Start a BigchainDB node:

```
docker-compose up -d bdb
```

You can monitor the logs:

```
docker-compose logs -f bdb
```

If you wish to run the tests:

```
docker-compose run --rm bdb py.test -v --database-backend=mongodb
```

Accessing the HTTP API

A quick check to make sure that the BigchainDB server API is operational:

```
curl $(docker-compose port bdb 9984)
```

should give you something like:

```
{
  "keyring": [],
  "public_key": "Brx8g4DdtEhccsENzNNV6yvQHR8s9ebhKyXPFkWUXh5e",
  "software": "BigchainDB",
  "version": "0.6.0"
}
```

How does the above curl command work? Inside the Docker container, BigchainDB exposes the HTTP API on port `9984`. First we get the public port where that port is bound:

```
docker-compose port bdb 9984
```

The port binding will change whenever you stop/restart the `bdb` service. You should get an output similar to:

```
0.0.0.0:32772
```

but with a port different from `32772`.

Knowing the public port we can now perform a simple `GET` operation against the root:

```
curl 0.0.0.0:32772
```

Option C: Using a Dev Machine on Cloud9

Ian Worrall of Encrypted Labs wrote a document (PDF) explaining how to set up a BigchainDB (Server) dev machine on Cloud9:

Download that document from GitHub

Running All Tests

All documentation about writing and running tests (unit and integration tests) was moved to the file `bigchaindb/tests/README.md`.

CHAPTER 7

Settings & CLI

Configuration Settings

The value of each BigchainDB Server configuration setting is determined according to the following rules:

- If it's set by an environment variable, then use that value
- Otherwise, if it's set in a local config file, then use that value
- Otherwise, use the default value

For convenience, here's a list of all the relevant environment variables (documented below):

`BIGCHAINDB_KEYPAIR_PUBLIC` `BIGCHAINDB_KEYPAIR_PRIVATE` `BIGCHAINDB_KEYRING` `BIGCHAINDB_DATABASE_BACKEND` `BIGCHAINDB_DATABASE_HOST` `BIGCHAINDB_DATABASE_PORT` `BIGCHAINDB_DATABASE_NAME` `BIGCHAINDB_DATABASE_REPLICASET` `BIGCHAINDB_DATABASE_CONNECTION_TIMEO` `BIGCHAINDB_DATABASE_MAX_TRIES` `BIGCHAINDB_SERVER_BIND` `BIGCHAINDB_SERVER_LOGLEVEL` `BIGCHAINDB_SERVER_WORKERS` `BIGCHAINDB_WSSERVER_SCHEME` `BIGCHAINDB_WSSERVER_HOST` `BIGCHAINDB_WSSERVER_PORT` `BIGCHAINDB_CONFIG_PATH` `BIGCHAINDB_BACKLOG_REASSIGN_DELAY` `BIGCHAINDB_LOG` `BIGCHAINDB_LOG_FILE` `BIGCHAINDB_LOG_ERROR_FILE` `BIGCHAINDB_LOG_LEVEL_CONSOLE` `BIGCHAINDB_LOG_LEVEL_LOGFILE` `BIGCHAINDB_LOG_DATEFMT_CONSOLE` `BIGCHAINDB_LOG_DATEFMT_LOGFILE` `BIGCHAINDB_LOG_FMT_CONSOLE` `BIGCHAINDB_LOG_FMT_LOGFILE` `BIGCHAINDB_LOG_GRANULAR_LEVELS` `BIGCHAINDB_DATABASE_SSL` `BIGCHAINDB_DATABASE_LOGIN` `BIGCHAINDB_DATABASE_PASSWORD` `BIGCHAINDB_DATABASE_CA_CERT` `BIGCHAINDB_DATABASE_CERTFILE` `BIGCHAINDB_DATABASE_KEYFILE` `BIGCHAINDB_DATABASE_KEYFILE_PASSPHRASE` `BIGCHAINDB_DATABASE_CRLFILE` `BIGCHAINDB_GRAPHITE_HOST`

The local config file is `$HOME/.bigchaindb` by default (a file which might not even exist), but you can tell BigchainDB to use a different file by using the `-c` command-line option, e.g. `bigchaindb -c path/to/config_file.json start` or using the `BIGCHAINDB_CONFIG_PATH` environment variable, e.g. `BIGHAINDB_CONFIG_PATH=.my_bigchaindb_config bigchaindb start`. Note that the `-c` command line option will always take precedence if both the `BIGCHAINDB_CONFIG_PATH` and the `-c` command line option are used.

You can read the current default values in the file bigchaindb/__init__.py. (The link is to the latest version.)

Running `bigchaindb -y configure mongodb` will generate a local config file in `$HOME/.bigchaindb` with all the default values (for using MongoDB as the database backend), with two exceptions: it will generate a valid private/public keypair, rather than using the default keypair (`None` and `None`).

keypair.public & keypair.private

The cryptographic keypair used by the node. The public key is how the node idenifies itself to the world. The private key is used to generate cryptographic signatures. Anyone with the public key can verify that the signature was generated by whoever had the corresponding private key.

Example using environment variables

```
export BIGCHAINDB_KEYPAIR_PUBLIC=8wHUvvraRo5yEoJAt66UTZaFq9YZ9tFFwcauKPDtjkGw
export BIGCHAINDB_KEYPAIR_PRIVATE=5C5Cknco7YxBRP9AgB1cbUVTL4FAcooxErLygw1DeG2D
```

Example config file snippet

```
"keypair": {
  "public": "8wHUvvraRo5yEoJAt66UTZaFq9YZ9tFFwcauKPDtjkGw",
  "private": "5C5Cknco7YxBRP9AgB1cbUVTL4FAcooxErLygw1DeG2D"
}
```

Internally (i.e. in the Python code), both keys have a default value of `None`, but that's not a valid key. Therefore you can't rely on the defaults for the keypair. If you want to run BigchainDB, you must provide a valid keypair, either in the environment variables or in the local config file. You can generate a local config file with a valid keypair (and default everything else) using `bigchaindb -y configure mongodb`.

keyring

A list of the public keys of all the nodes in the cluster, excluding the public key of this node.

Example using an environment variable

```
export BIGCHAINDB_
↪KEYRING=BnCsre9MPBeQK8QZBFznU2dJJ2GwtvnSMdemCmod2XPB:4cYQHoQrvPiut3Sjs8fVR1BMZZpJjMTC4bsMTt9V71aQ
```

Note how the keys in the list are separated by colons.

Example config file snippet

```
"keyring": ["BnCsre9MPBeQK8QZBFznU2dJJ2GwtvnSMdemCmod2XPB",
            "4cYQHoQrvPiut3Sjs8fVR1BMZZpJjMTC4bsMTt9V71aQ"]
```

Default value (from a config file)

```
"keyring": []
```

database.*

The settings with names of the form `database.*` are for the database backend (currently either MongoDB or RethinkDB). They are:

- `database.backend` is either `mongodb` or `rethinkdb`.
- `database.host` is the hostname (FQDN) of the backend database.

- database.port is self-explanatory.
- database.name is a user-chosen name for the database inside MongoDB or RethinkDB, e.g. bigchain.
- database.replicaset is only relevant if using MongoDB; it's the name of the MongoDB replica set, e.g. bigchain-rs.
- database.connection_timeout is the maximum number of milliseconds that BigchainDB will wait before giving up on one attempt to connect to the database backend.
- database.max_tries is the maximum number of times that BigchainDB will try to establish a connection with the database backend. If 0, then it will try forever.
- database.ssl is a flag that determines if BigchainDB connects to the backend database over TLS/SSL or not. This can be set to either true or false (the default). Note: This parameter is only supported for the MongoDB backend currently.
- database.login and database.password are the login and password used to authenticate to the database before performing any operations, specified in plaintext. The default values for both are currently null, which means that BigchainDB will not authenticate with the backend database. Note: These parameters are only supported for the MongoDB backend currently.
- database.ca_cert, database.certfile, database.keyfile and database.crlfile are the paths to the CA, signed certificate, private key and certificate revocation list files respectively. Note: These parameters are only supported for the MongoDB backend currently.
- database.keyfile_passphrase is the private key decryption passphrase, specified in plaintext. Note: This parameter is only supported for the MongoDB backend currently.

Example using environment variables

```
export BIGCHAINDB_DATABASE_BACKEND=mongodb
export BIGCHAINDB_DATABASE_HOST=localhost
export BIGCHAINDB_DATABASE_PORT=27017
export BIGCHAINDB_DATABASE_NAME=bigchain
export BIGCHAINDB_DATABASE_REPLICASET=bigchain-rs
export BIGCHAINDB_DATABASE_CONNECTION_TIMEOUT=5000
export BIGCHAINDB_DATABASE_MAX_TRIES=3
```

Default values

If (no environment variables were set and there's no local config file), or you used bigchaindb -y configure rethinkdb to create a default local config file for a RethinkDB backend, then the defaults will be:

```
"database": {
    "backend": "rethinkdb",
    "host": "localhost",
    "port": 28015,
    "name": "bigchain",
    "connection_timeout": 5000,
    "max_tries": 3
}
```

If you used bigchaindb -y configure mongodb to create a default local config file for a MongoDB backend, then the defaults will be:

```
"database": {
    "backend": "mongodb",
    "host": "localhost",
    "port": 27017,
    "name": "bigchain",
    "replicaset": "bigchain-rs",
    "connection_timeout": 5000,
    "max_tries": 3,
    "login": null,
    "password": null
    "ssl": false,
    "ca_cert": null,
    "crlfile": null,
    "certfile": null,
    "keyfile": null,
    "keyfile_passphrase": null,
}
```

server.bind, server.loglevel & server.workers

These settings are for the Gunicorn HTTP server, which is used to serve the HTTP client-server API.

`server.bind` is where to bind the Gunicorn HTTP server socket. It's a string. It can be any valid value for Gunicorn's bind setting. If you want to allow IPv4 connections from anyone, on port 9984, use `0.0.0.0:9984`. In a production setting, we recommend you use Gunicorn behind a reverse proxy server. If Gunicorn and the reverse proxy are running on the same machine, then use `localhost:PORT` where PORT is *not* 9984 (because the reverse proxy needs to listen on port 9984). Maybe use PORT=9983 in that case because we know 9983 isn't used. If Gunicorn and the reverse proxy are running on different machines, then use `A.B.C.D:9984` where A.B.C.D is the IP address of the reverse proxy. There's more information about deploying behind a reverse proxy in the Gunicorn documentation. (They call it a proxy.)

`server.loglevel` sets the log level of Gunicorn's Error log outputs. See Gunicorn's documentation for more information.

`server.workers` is the number of worker processes for handling requests. If `None` (the default), the value will be (2 × cpu_count + 1). Each worker process has a single thread. The HTTP server will be able to handle `server.workers` requests simultaneously.

Example using environment variables

```
export BIGCHAINDB_SERVER_BIND=0.0.0.0:9984
export BIGCHAINDB_SERVER_LOGLEVEL=debug
export BIGCHAINDB_SERVER_WORKERS=5
```

Example config file snippet

```
"server": {
    "bind": "0.0.0.0:9984",
    "loglevel": "debug",
    "workers": 5,
}
```

Default values (from a config file)

```
"server": {
    "bind": "localhost:9984",
    "loglevel": "info",
    "workers": null,
}
```

wsserver.scheme, wsserver.host and wsserver.port

These settings are for the aiohttp server, which is used to serve the WebSocket Event Stream API. `wsserver.scheme` should be either `"ws"` or `"wss"` (but setting it to `"wss"` does *not* enable SSL/TLS). `wsserver.host` is where to bind the aiohttp server socket and `wsserver.port` is the corresponding port. If you want to allow connections from anyone, on port 9985, set `wsserver.host` to 0.0.0.0 and `wsserver.port` to 9985.

Example using environment variables

```
export BIGCHAINDB_WSSERVER_SCHEME=ws
export BIGCHAINDB_WSSERVER_HOST=0.0.0.0
export BIGCHAINDB_WSSERVER_PORT=9985
```

Example config file snippet

```
"wsserver": {
    "scheme": "wss",
    "host": "0.0.0.0",
    "port": 65000
}
```

Default values (from a config file)

```
"wsserver": {
    "scheme": "ws",
    "host": "localhost",
    "port": 9985
}
```

backlog_reassign_delay

Specifies how long, in seconds, transactions can remain in the backlog before being reassigned. Long-waiting transactions must be reassigned because the assigned node may no longer be responsive. The default duration is 120 seconds.

Example using environment variables

```
export BIGCHAINDB_BACKLOG_REASSIGN_DELAY=30
```

Default value (from a config file)

```
"backlog_reassign_delay": 120
```

log

The `log` key is expected to point to a mapping (set of key/value pairs) holding the logging configuration.

Example:

```
{
    "log": {
        "file": "/var/log/bigchaindb.log",
        "error_file": "/var/log/bigchaindb-errors.log",
        "level_console": "info",
        "level_logfile": "info",
        "datefmt_console": "%Y-%m-%d %H:%M:%S",
        "datefmt_logfile": "%Y-%m-%d %H:%M:%S",
        "fmt_console": "%(asctime)s [%(levelname)s] (%(name)s) %(message)s",
        "fmt_logfile": "%(asctime)s [%(levelname)s] (%(name)s) %(message)s",
        "granular_levels": {
            "bichaindb.backend": "info",
            "bichaindb.core": "info"
        }
}
```

Defaults to:

```
{
    "log": {
        "file": "~/bigchaindb.log",
        "error_file": "~/bigchaindb-errors.log",
        "level_console": "info",
        "level_logfile": "info",
        "datefmt_console": "%Y-%m-%d %H:%M:%S",
        "datefmt_logfile": "%Y-%m-%d %H:%M:%S",
        "fmt_logfile": "[%(asctime)s] [%(levelname)s] (%(name)s) %(message)s (
↪%(processName)-10s - pid: %(process)d)",
        "fmt_console": "[%(asctime)s] [%(levelname)s] (%(name)s) %(message)s (
↪%(processName)-10s - pid: %(process)d)",
        "granular_levels": {}
}
```

The next subsections explain each field of the `log` configuration.

log.file & log.error_file

The full paths to the files where logs and error logs should be written to.

Example:

```
{
    "log": {
        "file": "/var/log/bigchaindb/bigchaindb.log"
        "error_file": "/var/log/bigchaindb/bigchaindb-errors.log"
    }
}
```

Defaults to:

```
* `"~/bigchaindb.log"`
* `"~/bigchaindb-errors.log"`
```

Please note that the user running `bigchaindb` must have write access to the locations.

Log rotation

Log files have a size limit of 200 MB and will be rotated up to five times.

For example if we consider the log file setting:

```
{
    "log": {
        "file": "~/bigchain.log"
    }
}
```

logs would always be written to `bigchain.log`. Each time the file `bigchain.log` reaches 200 MB it would be closed and renamed `bigchain.log.1`. If `bigchain.log.1` and `bigchain.log.2` already exist they would be renamed `bigchain.log.2` and `bigchain.log.3`. This pattern would be applied up to `bigchain.log.5` after which `bigchain.log.5` would be overwritten by `bigchain.log.4`, thus ending the rotation cycle of whatever logs were in `bigchain.log.5`.

log.level_console

The log level used to log to the console. Possible allowed values are the ones defined by Python, but case insensitive for convenience's sake:

```
"critical", "error", "warning", "info", "debug", "notset"
```

Example:

```
{
    "log": {
        "level_console": "info"
    }
}
```

Defaults to: `"info"`.

log.level_logfile

The log level used to log to the log file. Possible allowed values are the ones defined by Python, but case insensitive for convenience's sake:

```
"critical", "error", "warning", "info", "debug", "notset"
```

Example:

```
{
    "log": {
        "level_file": "info"
    }
}
```

Defaults to: `"info"`.

log.datefmt_console

The format string for the date/time portion of a message, when logged to the console.

Example:

```
{
    "log": {
        "datefmt_console": "%x %X %Z"
    }
}
```

Defaults to: `"%Y-%m-%d %H:%M:%S"`.

For more information on how to construct the format string please consult the table under Python's documentation of `time.strftime(format[, t])`

log.datefmt_logfile

The format string for the date/time portion of a message, when logged to a log file.

Example:

```
{
    "log": {
        "datefmt_logfile": "%c %z"
    }
}
```

Defaults to: `"%Y-%m-%d %H:%M:%S"`.

For more information on how to construct the format string please consult the table under Python's documentation of `time.strftime(format[, t])`

log.fmt_console

A string used to format the log messages when logged to the console.

Example:

```
{
    "log": {
        "fmt_console": "%(asctime)s [%(levelname)s] %(message)s %(process)d"
    }
}
```

Defaults to: `"[%(asctime)s] [%(levelname)s] (%(name)s) %(message)s (%(processName)-10s - pid: %(process)d)"`

For more information on possible formatting options please consult Python's documentation on LogRecord attributes

log.fmt_logfile

A string used to format the log messages when logged to a log file.

Example:

```
{
    "log": {
        "fmt_logfile": "%(asctime)s [%(levelname)s] %(message)s %(process)d"
    }
}
```

Defaults to: `"[%(asctime)s] [%(levelname)s] (%(name)s) %(message)s (%(processName)-10s - pid: %(process)d)"`

For more information on possible formatting options please consult Python's documentation on LogRecord attributes

log.granular_levels

Log levels for BigchainDB's modules. This can be useful to control the log level of specific parts of the application. As an example, if you wanted the logging of the `core.py` module to be more verbose, you would set the configuration shown in the example below.

Example:

```
{
    "log": {
        "granular_levels": {
            "bichaindb.core": "debug"
        }
}
```

Defaults to: `"{}"`

graphite.host

The host name or IP address of a server listening for statsd events on UDP port 8125. This defaults to `localhost`, and if no statsd collector is running, the events are simply dropped by the operating system.

Example using environment variables

```
export BIGCHAINDB_GRAPHITE_HOST=10.0.0.5
```

Example config file snippet

```
"graphite": {
    "host": "10.0.0.5"
}
```

Default values (from a config file)

```
"graphite": {
    "host": "localhost"
}
```

Command Line Interface (CLI)

The command-line command to interact with BigchainDB Server is `bigchaindb`.

bigchaindb –help

Show help for the `bigchaindb` command. `bigchaindb -h` does the same thing.

bigchaindb –version

Show the version number. `bigchaindb -v` does the same thing.

bigchaindb configure

Generate a local configuration file (which can be used to set some or all BigchainDB node configuration settings). It will auto-generate a public-private keypair and then ask you for the values of other configuration settings. If you press Enter for a value, it will use the default value.

Since BigchainDB supports multiple databases you need to always specify the database backend that you want to use. At this point only two database backends are supported: `rethinkdb` and `mongodb`.

If you use the `-c` command-line option, it will generate the file at the specified path:

```
bigchaindb -c path/to/new_config.json configure rethinkdb
```

If you don't use the `-c` command-line option, the file will be written to `$HOME/.bigchaindb` (the default location where BigchainDB looks for a config file, if one isn't specified).

If you use the `-y` command-line option, then there won't be any interactive prompts: it will just generate a keypair and use the default values for all the other configuration settings.

```
bigchaindb -y configure rethinkdb
```

bigchaindb show-config

Show the values of the BigchainDB node configuration settings.

bigchaindb export-my-pubkey

Write the node's public key (i.e. one of its configuration values) to standard output (stdout).

bigchaindb init

Create a backend database (RethinkDB or MongoDB), all database tables/collections, various backend database indexes, and the genesis block.

Note: The `bigchaindb start` command (see below) always starts by trying a `bigchaindb init` first. If it sees that the backend database already exists, then it doesn't re-initialize the database. One doesn't have to do `bigchaindb init` before `bigchaindb start`. `bigchaindb init` is useful if you only want to initialize (but not start).

bigchaindb drop

Drop (erase) the backend database (a RethinkDB or MongoDB database). You will be prompted to make sure. If you want to force-drop the database (i.e. skipping the yes/no prompt), then use `bigchaindb -y drop`

bigchaindb start

Start BigchainDB. It always begins by trying a `bigchaindb init` first. See the note in the documentation for `bigchaindb init`. You can also use the `--dev-start-rethinkdb` command line option to automatically start rethinkdb with bigchaindb if rethinkdb is not already running, e.g. `bigchaindb --dev-start-rethinkdb start`. Note that this will also shutdown rethinkdb when the bigchaindb process stops. The option `--dev-allow-temp-keypair` will generate a keypair on the fly if no keypair is found, this is useful when you want to run a temporary instance of BigchainDB in a Docker container, for example.

Options

The log level for the console can be set via the option `--log-level` or its abbreviation `-l`. Example:

```
$ bigchaindb --log-level INFO start
```

The allowed levels are `DEBUG`, `INFO` , `WARNING`, `ERROR`, and `CRITICAL`. For an explanation regarding these levels please consult the Logging Levels section of Python's documentation.

For a more fine-grained control over the logging configuration you can use the configuration file as documented under Configuration Settings.

bigchaindb set-shards

This command is specific to RethinkDB so it will only run if BigchainDB is configured with `rethinkdb` as the backend.

If RethinkDB is the backend database, then:

```
$ bigchaindb set-shards 4
```

will set the number of shards (in all RethinkDB tables) to 4.

bigchaindb set-replicas

This command is specific to RethinkDB so it will only run if BigchainDB is configured with `rethinkdb` as the backend.

If RethinkDB is the backend database, then:

```
$ bigchaindb set-replicas 3
```

will set the number of replicas (of each shard) to 3 (i.e. it will set the replication factor to 3).

bigchaindb add-replicas

This command is specific to MongoDB so it will only run if BigchainDB is configured with `mongodb` as the backend.

This command is used to add nodes to a BigchainDB cluster. It accepts a list of space separated hosts in the form *hostname:port*:

```
$ bigchaindb add-replicas server1.com:27017 server2.com:27017 server3.com:27017
```

bigchaindb remove-replicas

This command is specific to MongoDB so it will only run if BigchainDB is configured with `mongodb` as the backend.

This command is used to remove nodes from a BigchainDB cluster. It accepts a list of space separated hosts in the form *hostname:port*:

```
$ bigchaindb remove-replicas server1.com:27017 server2.com:27017 server3.com:27017
```

CHAPTER 8

The HTTP Client-Server API

This page assumes you already know an API Root URL for a BigchainDB node or reverse proxy. It should be something like `https://example.com:9984` or `https://12.34.56.78:9984`.

If you set up a BigchainDB node or reverse proxy yourself, and you're not sure what the API Root URL is, then see the last section of this page for help.

BigchainDB Root URL

If you send an HTTP GET request to the BigchainDB Root URL e.g. `http://localhost:9984` or `https://example.com:9984` (with no `/api/v1/` on the end), then you should get an HTTP response with something like the following in the body:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "api": {
    "v1": {
      "assets": "/api/v1/assets/",
      "docs": "https://docs.bigchaindb.com/projects/server/en/v1.0.1/http-client-
↪server-api.html",
      "outputs": "/api/v1/outputs/",
      "statuses": "/api/v1/statuses/",
      "streams": "ws://localhost:9985/api/v1/streams/valid_transactions",
      "transactions": "/api/v1/transactions/"
    }
  },
  "docs": "https://docs.bigchaindb.com/projects/server/en/v1.0.1/",
  "keyring": [
    "6qHyZew94NMmUTYyHnkZsB8cxJYuRNEiEpXHe1ih9QX3",
    "AdDuyrTyjrDt935YnFu4VBCVDhHtY2Y6rcy7x2TFeiRi"
  ],
  "public_key": "NC8c8rYcAhyKVpx1PCV65CBmyq4YUbLysy3Rqrg8L8mz",
  "software": "BigchainDB",
  "version": "1.0.1"
}
```

API Root Endpoint

If you send an HTTP GET request to the API Root Endpoint e.g. `http://localhost:9984/api/v1/` or `https://example.com:9984/api/v1/`, then you should get an HTTP response that allows you to discover the BigchainDB API endpoints:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "assets": "/assets/",
  "docs": "https://docs.bigchaindb.com/projects/server/en/v1.0.1/http-client-server-
↪api.html",
  "outputs": "/outputs/",
  "statuses": "/statuses/",
  "streams": "ws://localhost:9985/api/v1/streams/valid_transactions",
  "transactions": "/transactions/"
}
```

Transactions

GET /api/v1/transactions/{transaction_id}
Get the transaction with the ID `transaction_id`.

This endpoint returns a transaction if it was included in a `VALID` block. All instances of a transaction in invalid/undecided blocks or the backlog are ignored and treated as if they don't exist. If a request is made for a transaction and instances of that transaction are found only in invalid/undecided blocks or the backlog, then the response will be `404 Not Found`.

Parameters

- **transaction_id** (*hex string*) – transaction ID

Example request:

```
GET /api/v1/transactions/
↪8b20dbe164badd5ca0611b0e233aef9acce609fbca20f787fc7d926f300d0102 HTTP/1.1
Host: example.com
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "asset": {
    "data": {
      "msg": "Hello BigchainDB!"
    }
  },
  "id": "8b20dbe164badd5ca0611b0e233aef9acce609fbca20f787fc7d926f300d0102",
  "inputs": [
    {
      "fulfillment": "pGSAIDE5i63cn4X8T8N1sZ2mGkJD5lNRnBM4PZgI_zvzbr-
↪cgUCGvCc2HO2uB4IKix6INRzGIM10r7VsKFMPM9cT7uVJ1xFLOJ9bn6UioepBMLIrrwTlk2CkTolIPonf7BnzriQL
↪",
      "fulfills": null,
      "owners_before": [
        "4K9sWUMFwTgaDGPfdynrbxWqWS6sWmKbZoTjxLtVUibD"
      ]
    }
  ],
  "metadata": {
    "sequence": 0
  },
  "operation": "CREATE",
  "outputs": [
    {
      "amount": "1",
      "condition": {
        "details": {
          "public_key": "4K9sWUMFwTgaDGPfdynrbxWqWS6sWmKbZoTjxLtVUibD",
          "type": "ed25519-sha-256"
        },
        "uri": "ni:///sha-256;PNYwdxaRaNw60N6LDFzOWO97b8tJeragczakL8PrAPc?
↪fpt=ed25519-sha-256&cost=131072"
      },
      "public_keys": [
        "4K9sWUMFwTgaDGPfdynrbxWqWS6sWmKbZoTjxLtVUibD"
      ]
    }
  ],
  "version": "1.0"
}
```

Response Headers

- Content-Type – `application/json`

Status Codes

- 200 OK – A transaction with that ID was found.
- 404 Not Found – A transaction with that ID was not found.

GET /api/v1/transactions

The unfiltered `/api/v1/transactions` endpoint without any query parameters returns a status code *400*. For valid filters, see the sections below.

There are however filtered requests that might come of use, given the endpoint is queried correctly. Some of them include retrieving a list of transactions that include:

•*Transactions related to a specific asset*

In this section, we've listed those particular requests, as they will likely to be very handy when implementing your application on top of BigchainDB.

Note: Looking up transactions with a specific `metadata` field is currently not supported, however, providing a way to query based on `metadata` data is on our roadmap.

A generalization of those parameters follows:

Query Parameters

- **asset_id** (*string*) – The ID of the asset.
- **operation** (*string*) – (Optional) One of the two supported operations of a transaction: `CREATE`, `TRANSFER`.

GET /api/v1/transactions?asset_id={asset_id}&operation={CREATE|TRANSFER}

Get a list of transactions that use an asset with the ID `asset_id`. Every `TRANSFER` transaction that originates from a `CREATE` transaction with `asset_id` will be included. This allows users to query the entire history or provenance of an asset.

This endpoint returns transactions only if they are decided `VALID` by the server.

Query Parameters

- **operation** (*string*) – (Optional) One of the two supported operations of a transaction: `CREATE`, `TRANSFER`.
- **asset_id** (*string*) – asset ID.

Example request:

```
GET /api/v1/transactions?operation=TRANSFER&asset_
↪id=8b20dbe164badd5ca0611b0e233aef9acce609fbca20f787fc7d926f300d0102 HTTP/1.1
Host: example.com
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

[{
  "asset": {
    "id": "8b20dbe164badd5ca0611b0e233aef9acce609fbca20f787fc7d926f300d0102"
  },
  "id": "7d3ed7e5bcad27b878a4e3f25363c8b03f49fa5007f6e6032d9ec38e36bc2e83",
  "inputs": [
    {
      "fulfillment": "pGSAIDE5i63cn4X8T8N1sZ2mGkJD5lNRnBM4PZgI_zvzbr-
↪cgUA11pDN83PjcBhuH-
↪ICqy6cbyxeXrHQBgHXhbulDInXoMPsVeOJp65Wsxr0WO6kmJvwgA7Je1UgzNJZ6pWb3kcL",
      "fulfills": {
        "output_index": 0,
        "transaction_id":
↪"8b20dbe164badd5ca0611b0e233aef9acce609fbca20f787fc7d926f300d0102"
      },
      "owners_before": [
        "4K9sWUMFwTgaDGPfdynrbxWqWS6sWmKbZoTjxLtVUibD"
      ]
    }
  ],
  "metadata": {
    "sequence": 1
  },
  "operation": "TRANSFER",
  "outputs": [
    {
      "amount": "1",
      "condition": {
        "details": {
          "public_key": "3yfQPHeWAa1MxTX9Zf9176QqcpcnWcanVZZbaHb8B3h9",
          "type": "ed25519-sha-256"
        },
        "uri": "ni:///sha-256;lu6ov4AKkee6KWGnyjOVLBeyuP0bz4-O6_dPi15eYUc?
↪fpt=ed25519-sha-256&cost=131072"
      },
      "public_keys": [
        "3yfQPHeWAa1MxTX9Zf9176QqcpcnWcanVZZbaHb8B3h9"
      ]
    }
  ],
  "version": "1.0"
},
{
  "asset": {
    "id": "8b20dbe164badd5ca0611b0e233aef9acce609fbca20f787fc7d926f300d0102"
  },
  "id": "d07285a60352838ff263a46ba8cca64e18d36888aee0ba76d5d601137b492fc6",
  "inputs": [
    {
      "fulfillment": "pGSAICw7Ul-c2lG6NFbHp3FbKRC7fivQcNGO7GS4wV3A-
↪1QggUBRFFWoFwJhWGhbt02I3NPIBT84qzNB1-
↪dTyuj1zvUfVmY7fn1GAqI6A6pPRch36hYF4Gup2R0DFdAitEHxhB4K",
      "fulfills": {
        "output_index": 0,
        "transaction_id":
↪"7d3ed7e5bcad27b878a4e3f25363c8b03f49fa5007f6e6032d9ec38e36bc2e83"
      },
      "owners_before": [
        "3yfQPHeWAa1MxTX9Zf9176QqcpcnWcanVZZbaHb8B3h9"
      ]
    }
  ],
  "metadata": {
    "sequence": 2
  },
  "operation": "TRANSFER",
  "outputs": [
    {
      "amount": "1",
      "condition": {
        "details": {
          "public_key": "3Af3fhhjU6d9WecEM9Uw5hfom9kNEwE7YuDWdqAUssqm",
          "type": "ed25519-sha-256"
        },
        "uri": "ni:///sha-256;Ll1r0LzgHUvWB87yIrNFYo731MMUEypqvrbPATTbuD4?
↪fpt=ed25519-sha-256&cost=131072"
      },
      "public_keys": [
        "3Af3fhhjU6d9WecEM9Uw5hfom9kNEwE7YuDWdqAUssqm"
      ]
    }
  ],
  "version": "1.0"
```

```
}]
```

Response Headers

- Content-Type – `application/json`

Status Codes

- 200 OK – A list of transactions containing an asset with ID `asset_id` was found and returned.
- 400 Bad Request – The request wasn't understood by the server, e.g. the `asset_id` querystring was not included in the request.

POST /api/v1/transactions

Push a new transaction.

Note: The posted transaction should be structurally valid and not spending an already spent output. The steps to build a valid transaction are beyond the scope of this page. One would normally use a driver such as the BigchainDB Python Driver to build a valid transaction.

Example request:

```
POST /api/v1/transactions/ HTTP/1.1
Host: example.com
Content-Type: application/json

{
  "asset": {
    "data": {
      "msg": "Hello BigchainDB!"
    }
  },
  "id": "8b20dbe164badd5ca0611b0e233aef9acce609fbca20f787fc7d926f300d0102",
  "inputs": [
    {
      "fulfillment": "pGSAIDE5i63cn4X8T8N1sZ2mGkJD5lNRnBM4PZgI_zvzbr-
↪cgUCGvCc2HO2uB4IKix6INRzGIM10r7VsKFMPM9cT7uVJ1xFLOJ9bn6UioepBMLIrrwTlk2CkTolIPonf7BnzriQL
↪",
      "fulfills": null,
      "owners_before": [
        "4K9sWUMFwTgaDGPfdynrbxWqWS6sWmKbZoTjxLtVUibD"
      ]
    }
  ],
  "metadata": {
    "sequence": 0
  },
  "operation": "CREATE",
  "outputs": [
    {
      "amount": "1",
      "condition": {
        "details": {
          "public_key": "4K9sWUMFwTgaDGPfdynrbxWqWS6sWmKbZoTjxLtVUibD",
          "type": "ed25519-sha-256"
        },
        "uri": "ni:///sha-256;PNYwdxaRaNw60N6LDFzOWO97b8tJeragczakL8PrAPc?
↪fpt=ed25519-sha-256&cost=131072"
      },
      "public_keys": [
        "4K9sWUMFwTgaDGPfdynrbxWqWS6sWmKbZoTjxLtVUibD"
      ]
    }
  ],
  "version": "1.0"
}
```

Example response:

```
HTTP/1.1 202 Accepted
Location: ../statuses?transaction_
↪id=8b20dbe164badd5ca0611b0e233aef9acce609fbca20f787fc7d926f300d0102
Content-Type: application/json

{
  "asset": {
    "data": {
      "msg": "Hello BigchainDB!"
    }
  },
  "id": "8b20dbe164badd5ca0611b0e233aef9acce609fbca20f787fc7d926f300d0102",
  "inputs": [
    {
      "fulfillment": "pGSAIDE5i63cn4X8T8N1sZ2mGkJD5lNRnBM4PZgI_zvzbr-
↪cgUCGvCc2HO2uB4IKix6INRzGIM10r7VsKFMPM9cT7uVJ1xFLOJ9bn6UioepBMLIrrwTlk2CkTolIPonf7BnzriQL
↪",
      "fulfills": null,
      "owners_before": [
        "4K9sWUMFwTgaDGPfdynrbxWqWS6sWmKbZoTjxLtVUibD"
      ]
    }
  ],
  "metadata": {
    "sequence": 0
  },
  "operation": "CREATE",
  "outputs": [
    {
      "amount": "1",
      "condition": {
        "details": {
          "public_key": "4K9sWUMFwTgaDGPfdynrbxWqWS6sWmKbZoTjxLtVUibD",
          "type": "ed25519-sha-256"
        },
        "uri": "ni:///sha-256;PNYwdxaRaNw60N6LDFzOWO97b8tJeragczakL8PrAPc?
↪fpt=ed25519-sha-256&cost=131072"
      },
      "public_keys": [
        "4K9sWUMFwTgaDGPfdynrbxWqWS6sWmKbZoTjxLtVUibD"
      ]
    }
  ],
  "version": "1.0"
}
```

Note: If the server is returning a 202 HTTP status code, then the transaction has been accepted for processing. To check the status of the transaction, poll the link to the *status monitor* provided in the Location header or listen to server's *WebSocket Event Stream API*.

Response Headers

- Content-Type – application/json
- Location – Relative link to a status monitor for the submitted transaction.

Status Codes

- 202 Accepted – The pushed transaction was accepted in the BACKLOG, but the processing has not been completed.
- 400 Bad Request – The transaction was malformed and not accepted in the BACKLOG.

Transaction Outputs

The /api/v1/outputs endpoint returns transactions outputs filtered by a given public key, and optionally filtered to only include either spent or unspent outputs.

GET /api/v1/outputs

Get transaction outputs by public key. The public_key parameter must be a base58 encoded ed25519 public key associated with transaction output ownership.

Returns a list of transaction outputs.

Parameters

- **public_key** – Base58 encoded public key associated with output ownership. This parameter is mandatory and without it the endpoint will return a 400 response code.
- **spent** – Boolean value ("true" or "false") indicating if the result set should include only spent or only unspent outputs. If not specified the result includes all the outputs (both spent and unspent) associated with the public_key.

GET /api/v1/outputs?public_key={public_key}

Return all outputs, both spent and unspent, for the public_key.

Example request:

```
GET /api/v1/outputs?public_key=1AAAbbb...ccc HTTP/1.1
Host: example.com
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  {
    "output_index": 0,
    "transaction_id":
↪"2d431073e1477f3073a4693ac7ff9be5634751de1b8abaa1f4e19548ef0b4b0e"
  },
  {
    "output_index": 1,
    "transaction_id":
↪"2d431073e1477f3073a4693ac7ff9be5634751de1b8abaa1f4e19548ef0b4b0e"
  }
]
```

Status Codes

- 200 OK – A list of outputs were found and returned in the body of the response.
- 400 Bad Request – The request wasn't understood by the server, e.g. the `public_key` querystring was not included in the request.

GET /api/v1/outputs?public_key={public_key}&spent=true

Return all **spent** outputs for `public_key`.

Example request:

```
GET /api/v1/outputs?public_key=1AAAbbb...ccc&spent=true HTTP/1.1
Host: example.com
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  {
    "output_index": 0,
    "transaction_id":
↪"2d431073e1477f3073a4693ac7ff9be5634751de1b8abaa1f4e19548ef0b4b0e"
  }
]
```

Status Codes

- 200 OK – A list of outputs were found and returned in the body of the response.
- 400 Bad Request – The request wasn't understood by the server, e.g. the `public_key` querystring was not included in the request.

GET /api/v1/outputs?public_key={public_key}&spent=false

Return all **unspent** outputs for `public_key`.

Example request:

```
GET /api/v1/outputs?public_key=1AAAbbb...ccc&spent=false HTTP/1.1
Host: example.com
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  {
    "output_index": 1,
    "transaction_id":
↪"2d431073e1477f3073a4693ac7ff9be5634751de1b8abaa1f4e19548ef0b4b0e"
  }
]
```

Status Codes

- 200 OK – A list of outputs were found and returned in the body of the response.
- 400 Bad Request – The request wasn't understood by the server, e.g. the `public_key` querystring was not included in the request.

Statuses

GET /api/v1/statuses

Get the status of an asynchronously written transaction or block by their id.

Query Parameters

- **transaction_id** (*string*) – transaction ID
- **block_id** (*string*) – block ID

Note: Exactly one of the `transaction_id` or `block_id` query parameters must be used together with this endpoint (see below for getting *transaction statuses* and *block statuses*).

GET /api/v1/statuses?transaction_id={transaction_id}

Get the status of a transaction.

The possible status values are `undecided`, `valid` or `backlog`. If a transaction in neither of those states is found, a `404 Not Found` HTTP status code is returned. We're currently looking into ways to unambigously let the user know about a transaction's status that was included in an invalid block.

Example request:

```
GET /statuses?transaction_
↪id=8b20dbe164badd5ca0611b0e233aef9acce609fbca20f787fc7d926f300d0102 HTTP/1.1
Host: example.com
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "status": "valid"
}
```

Response Headers

- Content-Type – `application/json`

Status Codes

- 200 OK – A transaction with that ID was found.

- 404 Not Found – A transaction with that ID was not found.

GET /api/v1/statuses?block_id={block_id}

Get the status of a block.

The possible status values are `undecided`, `valid` or `invalid`.

Example request:

```
GET /api/v1/statuses?block_
↪id=80c6fedbe2960470a3af2684167a555a5de16caf4e6d4b217e2847fe394da069 HTTP/1.1
Host: example.com
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "status": "valid"
}
```

Response Headers

- Content-Type – `application/json`

Status Codes

- 200 OK – A block with that ID was found.
- 404 Not Found – A block with that ID was not found.

Assets

GET /api/v1/assets

Return all the assets that match a given text search.

Query Parameters

- **text search** (*string*) – Text search string to query.
- **limit** (*int*) – (Optional) Limit the number of returned assets. Defaults to `0` meaning return all matching assets.

Note: Currently this enpoint is only supported if the server is running MongoDB as the backend.

GET /api/v1/assets?search={text_search}

Return all assets that match a given text search. The `id` of the asset is the same `id` of the transaction that created the asset.

If no assets match the text search it returns an empty list.

If the text string is empty or the server does not support text search, a `400` is returned.

The results are sorted by text score. For more information about the behavior of text search see MongoDB text search behavior

Example request:

```
GET /api/v1/assets/?search=bigchaindb HTTP/1.1
Host: example.com
```

Example response:

```
HTTP/1.1 200 OK
Content-type: application/json

[
    {
        "data": {"msg": "Hello BigchainDB 1!"},
        "id": "51ce82a14ca274d43e4992bbce41f6fdeb755f846e48e710a3bbb3b0cf8e4204"
    },
    {
        "data": {"msg": "Hello BigchainDB 2!"},
        "id": "b4e9005fa494d20e503d916fa87b74fe61c079afccd6e084260674159795ee31"
    },
    {
        "data": {"msg": "Hello BigchainDB 3!"},
        "id": "fa6bcb6a8fdea3dc2a860fcdc0e0c63c9cf5b25da8b02a4db4fb6a2d36d27791"
    }
]
```

Response Headers

- Content-Type – `application/json`

Status Codes

- 200 OK – The query was executed successfully.
- 400 Bad Request – The query was not executed successfully. Returned if the text string is empty or the server does not support text search.

GET /api/v1/assets?search={text_search}&limit={n_documents}

Return at most `n` assets that match a given text search.

If no assets match the text search it returns an empty list.

If the text string is empty or the server does not support text search, a `400` is returned.

The results are sorted by text score. For more information about the behavior of text search see MongoDB text search behavior

Example request:

```
GET /api/v1/assets/?search=bigchaindb&limit=2 HTTP/1.1
Host: example.com
```

Example response:

```
HTTP/1.1 200 OK
Content-type: application/json

[
    {
        "data": {"msg": "Hello BigchainDB 1!"},
        "id": "51ce82a14ca274d43e4992bbce41f6fdeb755f846e48e710a3bbb3b0cf8e4204"
    },
    {
        "data": {"msg": "Hello BigchainDB 2!"},
        "id": "b4e9005fa494d20e503d916fa87b74fe61c079afccd6e084260674159795ee31"
    },
]
```

Response Headers

- Content-Type – `application/json`

Status Codes

- 200 OK – The query was executed successfully.
- 400 Bad Request – The query was not executed successfully. Returned if the text string is empty or the server does not support text search.

Advanced Usage

The following endpoints are more advanced and meant for debugging and transparency purposes.

More precisely, the *blocks endpoint* allows you to retrieve a block by `block_id` as well the list of blocks that a certain transaction with `transaction_id` occured in (a transaction can occur in multiple `invalid` blocks until it either gets rejected or validated by the system). This endpoint gives the ability to drill down on the lifecycle of a transaction

The *votes endpoint* contains all the voting information for a specific block. So after retrieving the `block_id` for a given `transaction_id`, one can now simply inspect the votes that happened at a specific time on that block.

Blocks

GET /api/v1/blocks/{block_id}

Get the block with the ID `block_id`. Any blocks, be they `VALID`, `UNDECIDED` or `INVALID` will be returned. To check a block's status independently, use the *Statuses endpoint*. To check the votes on a block, have a look at the *votes endpoint*.

Parameters

- **block_id** (*hex string*) – block ID

Example request:

```
GET /api/v1/blocks/
↪80c6fedbe2960470a3af2684167a555a5de16caf4e6d4b217e2847fe394da069 HTTP/1.1
Host: example.com
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "block": {
    "node_pubkey": "DngBurxfeNVKZWCEcDnLj1eMPAS7focUZTE5FndFGuHT",
    "timestamp": "1500018524",
    "transactions": [
      {
        "asset": {
          "data": {
            "msg": "Hello BigchainDB!"
          }
        },
        "id": "8b20dbe164badd5ca0611b0e233aef9acce609fbca20f787fc7d926f300d0102",
        "inputs": [
          {
            "fulfillment": "pGSAIDE5i63cn4X8T8N1sZ2mGkJD5lNRnBM4PZgI_zvzbr-
↪cgUCGvCc2HO2uB4IKix6INRzGIM10r7VsKFMPM9cT7uVJ1xFLOJ9bn6UioepBMLIrrwTlk2CkTolIPonf7BnzriQL
↪",
            "fulfills": null,
            "owners_before": [
              "4K9sWUMFwTgaDGPfdynrbxWqWS6sWmKbZoTjxLtVUibD"
            ]
          }
        ],
        "metadata": {
          "sequence": 0
        },
        "operation": "CREATE",
        "outputs": [
          {
            "amount": "1",
            "condition": {
              "details": {
                "public_key": "4K9sWUMFwTgaDGPfdynrbxWqWS6sWmKbZoTjxLtVUibD",
                "type": "ed25519-sha-256"
              },
              "uri": "ni:///sha-256;PNYwdxaRaNw60N6LDFzOWO97b8tJeragczakL8PrAPc?
↪fpt=ed25519-sha-256&cost=131072"
            },
            "public_keys": [
              "4K9sWUMFwTgaDGPfdynrbxWqWS6sWmKbZoTjxLtVUibD"
            ]
          }
        ],
        "version": "1.0"
      }
    ],
    "voters": [
      "DngBurxfeNVKZWCEcDnLj1eMPAS7focUZTE5FndFGuHT"
    ]
  },
  "id": "80c6fedbe2960470a3af2684167a555a5de16caf4e6d4b217e2847fe394da069",
  "signature":
↪"53wxrEQDYk1dXzmvNSytbCfmNVnPqPkDQaTnAe8Jf43s6ssejPxezkCvUnGTnduNUmaLjhaan1iRLi3peu6s5DzA
↪"
}
```

Response Headers

- Content-Type – `application/json`

Status Codes

- 200 OK – A block with that ID was found.

- 400 Bad Request – The request wasn't understood by the server, e.g. just requesting `/blocks` without the `block_id`.
- 404 Not Found – A block with that ID was not found.

GET /api/v1/blocks

The unfiltered `/blocks` endpoint without any query parameters returns a *400* status code. The list endpoint should be filtered with a `transaction_id` query parameter, see the `/blocks?transaction_id={transaction_id}&status={UNDECIDED|VALID|INVALID}` *endpoint*.

Example request:

```
GET /api/v1/blocks HTTP/1.1
Host: example.com
```

Example response:

```
HTTP/1.1 400 Bad Request
```

Status Codes

- 400 Bad Request – The request wasn't understood by the server, e.g. just requesting `/blocks` without the `block_id`.

GET /api/v1/blocks?transaction_id={transaction_id}&status={UNDECIDED|VALID|INVALID}

Retrieve a list of `block_id` with their corresponding status that contain a transaction with the ID `transaction_id`.

Any blocks, be they `UNDECIDED`, `VALID` or `INVALID` will be returned if no status filter is provided.

Note: In case no block was found, an empty list and an HTTP status code `200 OK` is returned, as the request was still successful.

Query Parameters

- **transaction_id** (*string*) – transaction ID *(required)*
- **status** (*string*) – Filter blocks by their status. One of `VALID`, `UNDECIDED` or `INVALID`.

Example request:

```
GET /api/v1/blocks?transaction_
↪id=8b20dbe164badd5ca0611b0e233aef9acce609fbca20f787fc7d926f300d0102 HTTP/1.1
Host: example.com
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  "8f9a1d1d2d3e57a5151074a4e5c352dfe7c0276d4e469db3da11dcad2266c9c4",
  "80c6fedbe2960470a3af2684167a555a5de16caf4e6d4b217e2847fe394da069"
]
```

Response Headers

- Content-Type – `application/json`

Status Codes

- 200 OK – A list of blocks containing a transaction with ID `transaction_id` was found and returned.
- 400 Bad Request – The request wasn't understood by the server, e.g. just requesting `/blocks`, without defining `transaction_id`.

Votes

GET /api/v1/votes?block_id={block_id}

Retrieve a list of votes for a certain block with ID `block_id`. To check for the validity of a vote, a user of this endpoint needs to perform the following steps:

1. Check if the vote's `node_pubkey` is allowed to vote.
2. Verify the vote's signature against the vote's body (`vote.vote`) and `node_pubkey`.

Query Parameters

- **block_id** (*string*) – The block ID to filter the votes.

Example request:

```
GET /api/v1/votes?block_
↪id=80c6fedbe2960470a3af2684167a555a5de16caf4e6d4b217e2847fe394da069 HTTP/1.1
Host: example.com
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

[{
  "node_pubkey": "DngBurxfeNVKZWCEcDnLj1eMPAS7focUZTE5FndFGuHT",
  "signature":
↪"zoaBfntAv87aKnc2dT2uqABW8HMzfFVB8WiSiN7U9Sw1tSJgjBNMagJYFDicjVvWdRnvz4Qt3YVyM8AzBGUoe3Z
↪",
  "vote": {
    "invalid_reason": null,
    "is_block_valid": true,
    "previous_block":
↪"0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcdef",
    "timestamp": "1500018524",
    "voting_for_block":
↪"80c6fedbe2960470a3af2684167a555a5de16caf4e6d4b217e2847fe394da069"
  }
}]
```

Response Headers

- Content-Type – `application/json`

Status Codes

- 200 OK – A list of votes voting for a block with ID `block_id` was found and returned.

- 400 Bad Request – The request wasn't understood by the server, e.g. just requesting `/votes`, without defining `block_id`.

Determining the API Root URL

When you start BigchainDB Server using `bigchaindb start`, an HTTP API is exposed at some address. The default is:

`http://localhost:9984/api/v1/`

It's bound to `localhost`, so you can access it from the same machine, but it won't be directly accessible from the outside world. (The outside world could connect via a SOCKS proxy or whatnot.)

The documentation about BigchainDB Server *Configuration Settings* has a section about how to set `server.bind` so as to make the HTTP API publicly accessible.

If the API endpoint is publicly accessible, then the public API Root URL is determined as follows:

- The public IP address (like 12.34.56.78) is the public IP address of the machine exposing the HTTP API to the public internet (e.g. either the machine hosting Gunicorn or the machine running the reverse proxy such as Nginx). It's determined by AWS, Azure, Rackspace, or whoever is hosting the machine.
- The DNS hostname (like example.com) is determined by DNS records, such as an "A Record" associating example.com with 12.34.56.78
- The port (like 9984) is determined by the `server.bind` setting if Gunicorn is exposed directly to the public Internet. If a reverse proxy (like Nginx) is exposed directly to the public Internet instead, then it could expose the HTTP API on whatever port it wants to. (It should expose the HTTP API on port 9984, but it's not bound to do that by anything other than convention.)

CHAPTER 9

The WebSocket Event Stream API

Important: The WebSocket Event Stream runs on a different port than the Web API. The default port for the Web API is *9984*, while the one for the Event Stream is *9985*.

BigchainDB provides real-time event streams over the WebSocket protocol with the Event Stream API. Connecting to an event stream from your application enables a BigchainDB node to notify you as events occur, such as new *validated transactions*.

Demoing the API

You may be interested in demoing the Event Stream API with the WebSocket echo test to familiarize yourself before attempting an integration.

Determining Support for the Event Stream API

It's a good idea to make sure that the node you're connecting with has advertised support for the Event Stream API. To do so, send a HTTP GET request to the node's *API Root Endpoint* (e.g. `http://localhost:9984/api/v1/`) and check that the response contains a `streams` property:

```
{
 ...,
 "streams": "ws://example.com:9985/api/v1/streams/valid_transactions",
 ...
}
```

Connection Keep-Alive

The Event Stream API initially does not provide any mechanisms for connection keep-alive other than enabling TCP keepalive on each open WebSocket connection. In the future, we may add additional functionality to handle ping/pong frames or payloads designed for keep-alive.

Streams

Each stream is meant as a unidirectional communication channel, where the BigchainDB node is the only party sending messages. Any messages sent to the BigchainDB node will be ignored.

Streams will always be under the WebSocket protocol (so `ws://` or `wss://`) and accessible as extensions to the `/api/v<version>/streams/` API root URL (for example, *validated transactions* would be accessible under `/api/v1/streams/valid_transactions`). If you're running your own BigchainDB instance and need help determining its root URL, then see the page titled *Determining the API Root URL*.

All messages sent in a stream are in the JSON format.

Note: For simplicity, BigchainDB initially only provides a stream for all validated transactions. In the future, we may provide streams for other information, such as new blocks, new votes, or invalid transactions. We may also provide the ability to filter the stream for specific qualities, such as a specific `output`'s `public_key`.

If you have specific use cases that you think would fit as part of this API, feel free to reach out via Gitter or email.

Valid Transactions

`/valid_transactions`

Streams an event for any newly validated transactions. Message bodies contain the transaction's ID, associated asset ID, and containing block's ID.

Example message:

```
{
    "transaction_id": "<sha3-256 hash>",
    "asset_id": "<sha3-256 hash>",
    "block_id": "<sha3-256 hash>"
}
```

Note: Transactions in BigchainDB are validated in batches ("blocks") and will, therefore, be streamed in batches. Each block can contain up to a 1000 transactions, ordered by the time at which they were included in the block. The `/valid_transactions` stream will send these transactions in the same order that the block stored them in, but this does **NOT** guarantee that you will recieve the events in that same order.

CHAPTER 10

Drivers & Tools

Libraries and Tools Maintained by the BigchainDB Team

- Python Driver
- JavaScript / Node.js Driver
- The Transaction CLI is a command-line interface for building BigchainDB transactions. You may be able to call it from inside the language of your choice, and then use *the HTTP API* to post transactions.

Community-Driven Libraries and Tools

Note: Some of these projects are a work in progress, but may still be useful.

- Haskell transaction builder
- Go driver
- Java driver
- Ruby driver

CHAPTER 11

Data Models

BigchainDB stores all data in the underlying database as JSON documents (conceptually, at least). There are three main kinds:

1. Transactions, which contain assets, inputs, outputs, and other things
2. Blocks
3. Votes

This section unpacks each one in turn.

The Transaction Model

A transaction has the following structure:

```
{
    "id": "<ID of the transaction>",
    "version": "<Transaction schema version number>",
    "inputs": ["<List of inputs>"],
    "outputs": ["<List of outputs>"],
    "operation": "<String>",
    "asset": {"<Asset model; see below>"},
    "metadata": {"<Arbitrary transaction metadata>"}
}
```

Here's some explanation of the contents:

- **id**: The ID of the transaction and also the hash of the transaction (loosely speaking). See below for an explanation of how it's computed. It's also the database primary key.
- **version**: The version-number of *the transaction schema*. As of BigchainDB Server 1.0.0, the only allowed value is "1.0".

- **inputs**: List of inputs. Each input spends/transfers a previous output by satisfying/fulfilling the crypto-conditions on that output. A CREATE transaction should have exactly one input. A TRANSFER transaction should have at least one input (i.e. ≥1). For more details, see the subsection about *inputs*.
- **outputs**: List of outputs. Each output indicates the crypto-conditions which must be satisfied by anyone wishing to spend/transfer that output. It also indicates the number of shares of the asset tied to that output. For more details, see the subsection about *outputs*.
- **operation**: A string indicating what kind of transaction this is, and how it should be validated. It can only be `"CREATE"`, `"TRANSFER"` or `"GENESIS"` (but there should only be one transaction whose operation is `"GENESIS"`: the one in the GENESIS block).
- **asset**: A JSON document for the asset associated with the transaction. (A transaction can only be associated with one asset.) See *the page about the asset model*.
- **metadata**: User-provided transaction metadata. It can be any valid JSON document, or `null`.

How the transaction ID is computed. 1) Build a Python dictionary containing `version`, `inputs`, `outputs`, `operation`, `asset`, `metadata` and their values, 2) In each of the inputs, replace the value of each `fulfillment` with `null`, 3) *Serialize* that dictionary, 4) The transaction ID is just *the SHA3-256 hash* of the serialized dictionary.

About signing the transaction. Later, when we get to the models for the block and the vote, we'll see that both include a signature (from the node which created it). You may wonder why transactions don't have signatures… The answer is that they do! They're just hidden inside the `fulfillment` string of each input. What gets signed (as of version 1.0.0) is everything inside the transaction, including the `id`, but the value of each `fulfillment` is replaced with `null`.

There are example BigchainDB transactions in *the HTTP API documentation* and the Python Driver documentation.

The Asset Model

To avoid redundant data in transactions, the asset model is different for `CREATE` and `TRANSFER` transactions.

In a `CREATE` transaction, the `"asset"` must contain exactly one key-value pair. The key must be `"data"` and the value can be any valid JSON document, or `null`. For example:

```
{
    "data": {
                "desc": "Gold-inlay bookmark owned by Xavier Bellomat Dickens III",
                "xbd_collection_id": 1857
            }
}
```

In a `TRANSFER` transaction, the `"asset"` must contain exactly one key-value pair. They key must be `"id"` and the value must contain a transaction ID (i.e. a SHA3-256 hash: the ID of the `CREATE` transaction which created the asset, which also serves as the asset ID). For example:

```
{
    "id": "38100137cea87fb9bd751e2372abb2c73e7d5bcf39d940a5516a324d9c7fb88d"
}
```

Inputs and Outputs

There's a high-level overview of inputs and outputs in the root docs page about transaction concepts.

BigchainDB is modelled around *assets*, and *inputs* and *outputs* are the mechanism by which control of an asset (or shares of an asset) is transferred. Amounts of an asset are encoded in the outputs of a transaction, and each output may be spent separately. To spend an output, the output's `condition` must be met by an `input` that provides a corresponding `fulfillment`. Each output may be spent at most once, by a single input. Note that any asset associated with an output holding an amount greater than one is considered a divisible asset that may be split up in future transactions.

Inputs

An input has the following structure:

```
{
    "owners_before": ["<The public_keys list in the output being spent>"],
    "fulfillment": "<Fulfillment URI fulfilling the condition of the output being␣
↪spent>",
    "fulfills": {
        "output_index": "<Index of the output being spent (an integer)>",
        "transaction_id": "<ID of the transaction containing the output being spent>"
    }
}
```

You can think of the `fulfills` object as a pointer to an output on another transaction: the output that this input is spending/transferring. A CREATE transaction should have exactly one input. That input can contain one or more `owners_before`, a `fulfillment` (with one signature from each of the owners-before), and the value of `fulfills` should be `null`). A TRANSFER transaction should have at least one input, and the value of `fulfills` should not be `null`. See the reference on *inputs* for more description about the meaning of each field.

To calculate a fulfillment URI, you can use one of the *BigchainDB drivers or transaction-builders*, or use a low-level crypto-conditions library as illustrated in the page about Handcrafting Transactions.

Outputs

An output has the following structure:

```
{
    "condition": {"<Condition object>"},
    "public_keys": ["<List of all public keys associated with the condition object>"],
    "amount": "<Number of shares of the asset (an integer in a string)>"
}
```

The list of `public_keys` is always the "owners" of the asset at the time the transaction completed, but before the next transaction started. See the reference on *outputs* for more description about the meaning of each field.

Below is a high-level description of what goes into building a `condition` object. To construct an actual `condition` object, you can use one of the *BigchainDB drivers or transaction-builders*, or use a low-level crypto-conditions library as illustrated in the page about Handcrafting Transactions.

Conditions

At a high level, a condition is like a lock on an output. If can you satisfy the condition, you can unlock the output and transfer/spend it. BigchainDB Server v1.0 supports a subset of the ILP Crypto-Conditions (version 02 of Crypto-Conditions).

The simplest supported condition is a simple signature condition. Such a condition could be stated as, "You can satisfy this condition if you send me a message and a cryptographic signature of that message, produced using the private key corresponding to this public key." The public key is put in the output. BigchainDB currently only supports ED25519 signatures.

A more complex condition can be composed by using n simple signature conditions as inputs to an m-of-n threshold condition (a logic gate which outputs TRUE if and only if m or more inputs are TRUE). If there are n inputs to a threshold condition:

- 1-of-n is the same as a logical OR of all the inputs
- n-of-n is the same as a logical AND of all the inputs

For example, one could create a condition requiring m (of n) signatures before their asset can be transferred.

The (single) output of a threshold condition can be used as one of the inputs of other threshold conditions. This means that one can combine threshold conditions to build complex logical expressions, e.g. (x OR y) AND (u OR v).

When one creates a condition, one can calculate its cost, an estimate of the resources that would be required to validate the fulfillment. A BigchainDB federation can put an upper limit on the complexity of each condition, either directly by setting a maximum allowed cost, or indirectly by *setting a maximum allowed transaction size* which would limit the overall complexity accross all inputs and outputs of a transaction. Note: At the time of writing, there was no configuration setting to set a maximum allowed cost, so the only real option was to *set a maximum allowed transaction size*.

Note: The BigchainDB documentation and code talks about control of an asset in terms of "owners" and "ownership." The language is chosen to represent the most common use cases, but in some more complex scenarios, it may not be accurate to say that the output is owned by the controllers of those public keys—it would only be correct to say that those public keys are associated with the ability to fulfill the conditions on the output. Also, depending on the use case, the entity controlling an output via a private key may not be the legal owner of the asset in the corresponding legal domain. However, since we aim to use language that is simple to understand and covers the majority of use cases, we talk in terms of "owners" of an output that have the ability to "spend" that output.

The Block Model

A block has the following structure:

```
{
  "id": "<hash of block>",
  "block": {
    "timestamp": "<block-creation timestamp>",
    "transactions": ["<list of transactions>"],
    "node_pubkey": "<public key of the node creating the block>",
    "voters": ["<list of public keys of all nodes in the cluster>"]
  },
  "signature": "<signature of block>"
}
```

- `id`: The *hash* of the serialized inner `block` (i.e. the `timestamp`, `transactions`, `node_pubkey`, and `voters`). It's used as a unique index in the database backend (e.g. RethinkDB or MongoDB).
- **`block`:**
  - `timestamp`: The Unix time when the block was created. It's provided by the node that created the block.
  - `transactions`: A list of the transactions included in the block.
  - `node_pubkey`: The public key of the node that created the block.

  - voters: A list of the public keys of all cluster nodes at the time the block was created. It's the list of nodes which can cast a vote on this block. This list can change from block to block, as nodes join and leave the cluster.

- signature: *Cryptographic signature* of the block by the node that created the block (i.e. the node with public key node_pubkey). To generate the signature, the node signs the serialized inner block (the same thing that was hashed to determine the id) using the private key corresponding to node_pubkey.

Working with Blocks

There's a **Block** class for creating and working with Block objects; look in /bigchaindb/models.py. (The link is to the latest version on the master branch on GitHub.)

The Vote Model

A vote has the following structure:

```
{
    "node_pubkey": "<The public key of the voting node>",
    "vote": {
        "voting_for_block": "<ID of the block the node is voting on>",
        "previous_block": "<ID of the block previous to the block being voted on>",
        "is_block_valid": "<true OR false>",
        "invalid_reason": null,
        "timestamp": "<Unix time when the vote was generated, provided by the voting␣
↪node>"
    },
    "signature": "<Cryptographic signature of vote>"
}
```

Notes

- Votes have no ID (or "id"), as far as users are concerned. (The backend database uses one internally, but it's of no concern to users and it's never reported to them via BigchainDB APIs.)
- At the time of writing, the value of "invalid_reason" was always null. In other words, it wasn't being used. It may be used or dropped in a future version of BigchainDB. See Issue #217 on GitHub.
- For more information about the vote "timestamp", see the page about timestamps in BigchainDB.
- For more information about how the "signature" is calculated, see the page about cryptography in BigchainDB.

CHAPTER 12

Transaction Schema

Transaction

A transaction represents the creation or transfer of assets in BigchainDB.

Transaction.id

type: string

A sha3 digest of the transaction. The ID is calculated by removing all derived hashes and signatures from the transaction, serializing it to JSON with keys in sorted order and then hashing the resulting string with sha3.

Transaction.operation

type: string

Type of the transaction:

A `CREATE` transaction creates an asset in BigchainDB. This transaction has outputs but no inputs, so a dummy input is created.

A `TRANSFER` transaction transfers ownership of an asset, by providing an input that meets the conditions of an earlier transaction's outputs.

A `GENESIS` transaction is a special case transaction used as the sole member of the first block in a BigchainDB ledger.

Transaction.asset

type: object

Description of the asset being transacted.

See: *Asset*.

Transaction.inputs

type: array (object)

Array of the inputs of a transaction.

See: *Input*.

Transaction.outputs

type: array (object)

Array of outputs provided by this transaction.

See: *Output*.

Transaction.metadata

type: object or null

User provided transaction metadata. This field may be `null` or may contain an id and an object with freeform metadata.

See: *Metadata*.

Transaction.version

type: string

BigchainDB transaction schema version.

Input

An input spends a previous output, by providing one or more fulfillments that fulfill the conditions of the previous output.

Input.owners_before

type: array (string) or null

List of public keys of the previous owners of the asset.

Input.fulfillment

type: string or object or object

Fulfillment of an *Output.condition*, or, put a different way, a payload that satisfies the condition of a previous output to prove that the creator(s) of this transaction have control over the listed asset.

Input.fulfills

type: object or null

Reference to the output that is being spent.

Output

A transaction output. Describes the quantity of an asset and the requirements that must be met to spend the output.

See also: *Input*.

Output.amount

type: string

Integral amount of the asset represented by this output. In the case of a non divisible asset, this will always be 1.

Output.condition

type: object

Describes the condition that needs to be met to spend the output. Has the properties:

- **details**: Details of the condition.
- **uri**: Condition encoded as an ASCII string.

Output.public_keys

type: array (string) or null

List of public keys associated with the conditions on an output.

Asset

Description of the asset being transacted. In the case of a `TRANSFER` transaction, this field contains only the ID of asset. In the case of a `CREATE` transaction, this field contains only the user-defined payload.

Asset.id

type: string

ID of the transaction that created the asset.

Asset.data

type: object or null

User provided metadata associated with the asset. May also be `null`.

Metadata

User provided transaction metadata. This field may be `null` or may contain an non empty object with freeform metadata.

CHAPTER 13

Vote Schema

Vote

A Vote is an endorsement of a Block (identified by a hash) by a node (identified by a public key).

The outer Vote object contains the details of the vote being made as well as the signature and identifying information of the node passing the vote.

Vote.node_pubkey

type: string

Ed25519 public key identifying the voting node.

Vote.signature

type: string

Ed25519 signature of the *Vote Details* object.

Vote.vote

type: object

Vote Details to be signed.

Vote Details

Vote Details to be signed.

Vote.previous_block

type: string

ID (SHA3 hash) of the block that precedes the block being voted on. The notion of a "previous" block is subject to vote.

Vote.voting_for_block

type: string

ID (SHA3 hash) of the block being voted on.

Vote.is_block_valid

type: boolean

This field is `true` if the block was deemed valid by the node.

Vote.invalid_reason

type: string or null

Reason the block is voted invalid, or `null`.

Note: The invalid_reason was not being used and may be dropped in a future version of BigchainDB. See Issue #217 on GitHub.

Vote.timestamp

type: string

Unix timestamp that the vote was created by the node, according to the system time of the node.

CHAPTER 14

Release Notes

You can find a list of all BigchainDB Server releases and release notes on GitHub at:

https://github.com/bigchaindb/bigchaindb/releases

The CHANGELOG.md file contains much the same information, but it also has notes about what to expect in the *next* release.

We also have a roadmap document in ROADMAP.md.

CHAPTER 15

Appendices

How to Install OS-Level Dependencies

BigchainDB Server has some OS-level dependencies that must be installed.

On Ubuntu 16.04, we found that the following was enough:

```
sudo apt-get update
sudo apt-get install g++ python3-dev libffi-dev build-essential libssl-dev
```

On Fedora 23–25, we found that the following was enough:

```
sudo dnf update
sudo dnf install gcc-c++ redhat-rpm-config python3-devel libffi-devel
```

(If you're using a version of Fedora before version 22, you may have to use `yum` instead of `dnf`.)

How to Install the Latest pip and setuptools

You can check the version of `pip` you're using (in your current virtualenv) by doing:

```
pip -V
```

If it says that `pip` isn't installed, or it says `pip` is associated with a Python version less than 3.5, then you must install a `pip` version associated with Python 3.5+. In the following instructions, we call it `pip3` but you may be able to use `pip` if that refers to the same thing. See the `pip` installation instructions.

On Ubuntu 16.04, we found that this works:

```
sudo apt-get install python3-pip
```

That should install a Python 3 version of `pip` named `pip3`. If that didn't work, then another way to get `pip3` is to do `sudo apt-get install python3-setuptools` followed by `sudo easy_install3 pip`.

You can upgrade `pip` (`pip3`) and `setuptools` to the latest versions using:

```
pip3 install --upgrade pip setuptools
```

Run BigchainDB with Docker

NOT for Production Use

For those who like using Docker and wish to experiment with BigchainDB in non-production environments, we currently maintain a Docker image and a `Dockerfile` that can be used to build an image for `bigchaindb`.

Pull and Run the Image from Docker Hub

Assuming you have Docker installed, you would proceed as follows.

In a terminal shell, pull the latest version of the BigchainDB Docker image using:

```
docker pull bigchaindb/bigchaindb
```

Configuration

A one-time configuration step is required to create the config file; we will use the `-y` option to accept all the default values. The configuration file will be stored in a file on your host machine at `~/bigchaindb_docker/.bigchaindb`:

```
docker run \
  --interactive \
  --rm \
  --tty \
  --volume $HOME/bigchaindb_docker:/data \
  bigchaindb/bigchaindb \
  -y configure \
  [mongodb|rethinkdb]

Generating keypair
Configuration written to /data/.bigchaindb
Ready to go!
```

Let's analyze that command:

- `docker run` tells Docker to run some image
- `--interactive` keep STDIN open even if not attached
- `--rm` remove the container once we are done
- `--tty` allocate a pseudo-TTY
- `--volume "$HOME/bigchaindb_docker:/data"` map the host directory `$HOME/bigchaindb_docker` to the container directory `/data`; this allows us to have the data persisted on the host machine, you can read more in the official Docker documentation
- `bigchaindb/bigchaindb` the image to use. All the options after the container name are passed on to the entrypoint inside the container.

- `-y configure` execute the `configure` sub-command (of the `bigchaindb` command) inside the container, with the `-y` option to automatically use all the default config values
- `mongodb` or `rethinkdb` specifies the database backend to use with bigchaindb

To ensure that BigchainDB connects to the backend database bound to the virtual interface `172.17.0.1`, you must edit the BigchainDB configuration file (`~/bigchaindb_docker/.bigchaindb`) and change database.host from `localhost` to `172.17.0.1`.

Run the backend database

From v0.9 onwards, you can run either RethinkDB or MongoDB.

We use the virtual interface created by the Docker daemon to allow communication between the BigchainDB and database containers. It has an IP address of 172.17.0.1 by default.

You can also use docker host networking or bind to your primary (eth) interface, if needed.

For RethinkDB

```
docker run \
  --detach \
  --name=rethinkdb \
  --publish=172.17.0.1:28015:28015 \
  --publish=172.17.0.1:58080:8080 \
  --restart=always \
  --volume $HOME/bigchaindb_docker:/data \
  rethinkdb:2.3
```

You can also access the RethinkDB dashboard at http://172.17.0.1:58080/

For MongoDB

Note: MongoDB runs as user `mongodb` which had the UID `999` and GID `999` inside the container. For the volume to be mounted properly, as user `mongodb` in your host, you should have a `mongodb` user with UID and GID `999`. If you have another user on the host with UID `999`, the mapped files will be owned by this user in the host. If there is no owner with UID 999, you can create the corresponding user and group.

`useradd -r --uid 999 mongodb` OR `groupadd -r --gid 999 mongodb && useradd -r --uid 999 -g mongodb mongodb` should work.

```
docker run \
  --detach \
  --name=mongodb \
  --publish=172.17.0.1:27017:27017 \
  --restart=always \
  --volume=/tmp/mongodb_docker/db:/data/db \
  --volume=/tmp/mongodb_docker/configdb:/data/configdb \
  mongo:3.4.1 --replSet=bigchain-rs
```

Run BigchainDB

```
docker run \
  --detach \
  --name=bigchaindb \
  --publish=59984:9984 \
  --restart=always \
  --volume=$HOME/bigchaindb_docker:/data \
  bigchaindb/bigchaindb \
  start
```

The command is slightly different from the previous one, the differences are:

- `--detach` run the container in the background
- `--name bigchaindb` give a nice name to the container so it's easier to refer to it later
- `--publish "59984:9984"` map the host port `59984` to the container port `9984` (the BigchainDB API server)
- `start` start the BigchainDB service

Another way to publish the ports exposed by the container is to use the `-P` (or `--publish-all`) option. This will publish all exposed ports to random ports. You can always run `docker ps` to check the random mapping.

If that doesn't work, then replace `localhost` with the IP or hostname of the machine running the Docker engine. If you are running docker-machine (e.g. on Mac OS X) this will be the IP of the Docker machine (`docker-machine ip machine_name`).

Building Your Own Image

Assuming you have Docker installed, you would proceed as follows.

In a terminal shell:

```
git clone git@github.com:bigchaindb/bigchaindb.git
```

Build the Docker image:

```
docker build --tag local-bigchaindb .
```

Now you can use your own image to run BigchainDB containers.

Run BigchainDB with Docker On Mac

NOT for Production Use

Those developing on Mac can follow this document to run BigchainDB in docker containers for a quick dev setup. Running BigchainDB on Mac (Docker or otherwise) is not officially supported.

Support is very much limited as there are certain things that work differently in Docker for Mac than Docker for other platforms. Also, we do not use mac for our development and testing. :)

This page may not be up to date with various settings and docker updates at all the times.

These steps work as of this writing (2017.Mar.09) and might break in the future with updates to Docker for mac. Community contribution to make BigchainDB run on Docker for Mac will always be welcome.

Prerequisite

Install Docker for Mac.

(Optional) For a clean start

1. Stop all BigchainDB and RethinkDB/MongoDB containers.
2. Delete all BigchainDB docker images.
3. Delete the ~/bigchaindb_docker folder.

Pull the images

Pull the bigchaindb and other required docker images from docker hub.

```
docker pull bigchaindb/bigchaindb:master
docker pull [rethinkdb:2.3|mongo:3.4.1]
```

Create the BigchainDB configuration file on Mac

```
docker run \
  --rm \
  --volume $HOME/bigchaindb_docker:/data \
  bigchaindb/bigchaindb:master \
  -y configure \
  [mongodb|rethinkdb]
```

To ensure that BigchainDB connects to the backend database bound to the virtual interface `172.17.0.1`, you must edit the BigchainDB configuration file (`~/bigchaindb_docker/.bigchaindb`) and change database.host from `localhost` to `172.17.0.1`.

Run the backend database on Mac

From v0.9 onwards, you can run RethinkDB or MongoDB.

We use the virtual interface created by the Docker daemon to allow communication between the BigchainDB and database containers. It has an IP address of 172.17.0.1 by default.

You can also use docker host networking or bind to your primary (eth) interface, if needed.

For RethinkDB backend

```
docker run \
  --name=rethinkdb \
  --publish=28015:28015 \
  --publish=8080:8080 \
  --restart=always \
  --volume $HOME/bigchaindb_docker:/data \
  rethinkdb:2.3
```

For MongoDB backend

```
docker run \
  --name=mongodb \
  --publish=27017:27017 \
  --restart=always \
  --volume=$HOME/bigchaindb_docker/db:/data/db \
  --volume=$HOME/bigchaindb_docker/configdb:/data/configdb \
  mongo:3.4.1 --replSet=bigchain-rs
```

Run BigchainDB on Mac

```
docker run \
  --name=bigchaindb \
  --publish=9984:9984 \
  --restart=always \
  --volume=$HOME/bigchaindb_docker:/data \
  bigchaindb/bigchaindb \
  start
```

JSON Serialization

We needed to clearly define how to serialize a JSON object to calculate the hash.

The serialization should produce the same byte output independently of the architecture running the software. If there are differences in the serialization, hash validations will fail although the transaction is correct.

For example, consider the following two methods of serializing `{'a': 1}`:

```
# Use a serializer provided by RethinkDB
a = r.expr({'a': 1}).to_json().run(b.connection)
u'{"a":1}'

# Use the serializer in Python's json module
b = json.dumps({'a': 1})
'{"a": 1}'

a == b
False
```

The results are not the same. We want a serialization and deserialization so that the following is always true:

```
deserialize(serialize(data)) == data
True
```

Since BigchainDB performs a lot of serialization we decided to use python-rapidjson which is a python wrapper for rapidjson a fast and fully RFC complient JSON parser.

```
import rapidjson

rapidjson.dumps(data, skipkeys=False,
                ensure_ascii=False,
                sort_keys=True)
```

- `skipkeys`: With skipkeys `False` if the provided keys are not a string the serialization will fail. This way we enforce all keys to be strings
- `ensure_ascii`: The RFC recommends `utf-8` for maximum interoperability. By setting `ensure_ascii` to `False` we allow unicode characters and python-rapidjson forces the encoding to `utf-8`.
- `sort_keys`: Sorted output by keys.

Every time we need to perform some operation on the data like calculating the hash or signing/verifying the transaction, we need to use the previous criteria to serialize the data and then use the `byte` representation of the serialized data (if we treat the data as bytes we eliminate possible encoding errors e.g. unicode characters). For example:

```
# calculate the hash of a transaction
# the transaction is a dictionary
tx_serialized = bytes(serialize(tx))
tx_hash = hashlib.sha3_256(tx_serialized).hexdigest()

# signing a transaction
tx_serialized = bytes(serialize(tx))
signature = sk.sign(tx_serialized)

# verify signature
tx_serialized = bytes(serialize(tx))
pk.verify(signature, tx_serialized)
```

Cryptography

The section documents the cryptographic algorithms and Python implementations that we use.

Before hashing or computing the signature of a JSON document, we serialize it as described in the section on JSON serialization.

Hashes

BigchainDB computes transaction and block hashes using an implementation of the SHA3-256 algorithm provided by the **pysha3** package, which is a wrapper around the optimized reference implementation from http://keccak.noekeon.org.

Important: Since selecting the Keccak hashing algorithm for SHA-3 in 2012, NIST released a new version of the hash using the same algorithm but slightly different parameters. As of version 0.9, BigchainDB is using the latest version, supported by pysha3 1.0b1. See below for an example output of the hash function.

Here's the relevant code from 'bigchaindb/bigchaindb/common/crypto.py:

```
import sha3

def hash_data(data):
    """Hash the provided data using SHA3-256"""
    return sha3.sha3_256(data.encode()).hexdigest()
```

The incoming `data` is understood to be a Python 3 string, which may contain Unicode characters such as `'ü'` or `''`. The Python 3 `encode()` method converts `data` to a bytes object. `sha3.sha3_256(data.encode())` is a _sha3.SHA3 object; the `hexdigest()` method converts it to a hexadecimal string. For example:

```
>>> import sha3
>>> data = ''
>>> sha3.sha3_256(data.encode()).hexdigest()
'2b38731ba4ef72d4034bef49e87c381d1fbe75435163b391dd33249331f91fe7'
>>> data = 'hello world'
>>> sha3.sha3_256(data.encode()).hexdigest()
'644bcc7e564373040999aac89e7622f3ca71fba1d972fd94a31c3bfbf24e3938'
```

Note: Hashlocks (which are one kind of crypto-condition) may use a different hash function.

Signature Algorithm and Keys

BigchainDB uses the Ed25519 public-key signature system for generating its public/private key pairs. Ed25519 is an instance of the Edwards-curve Digital Signature Algorithm (EdDSA). As of December 2016, EdDSA was an "Internet-Draft" with the IETF but was already widely used.

BigchainDB uses the the **cryptoconditions** package to do signature and keypair-related calculations. That package, in turn, uses the **PyNaCl** package, a Python binding to the Networking and Cryptography (NaCl) library.

All keys are represented with a Base58 encoding. The cryptoconditions package uses the **base58** package to calculate a Base58 encoding. (There's no standard for Base58 encoding.) Here's an example public/private key pair:

```
"keypair": {
    "public": "9WYFf8T65bv4S8jKU8wongKPD4AmMZAwvk1absFDbYLM",
    "private": "3x7MQpPq8AEUGEuzAxSVHjU1FhLWVQJKFNNkvHhJPGCX"
}
```

The Bigchain class

The Bigchain class is the top-level Python API for BigchainDB. If you want to create and initialize a BigchainDB database, you create a Bigchain instance (object). Then you can use its various methods to create transactions, write transactions (to the object/database), read transactions, etc.

class `bigchaindb.`**`Bigchain`**(*public_key=None*, *private_key=None*, *keyring=[]*, *connection=None*, *backlog_reassign_delay=None*)

Bigchain API

Create, read, sign, write transactions to the database

`__init__`(*public_key=None*, *private_key=None*, *keyring=[]*, *connection=None*, *backlog_reassign_delay=None*)

Initialize the Bigchain instance

A Bigchain instance has several configuration parameters (e.g. host). If a parameter value is passed as an argument to the Bigchain __init__ method, then that is the value it will have. Otherwise, the parameter value will come from an environment variable. If that environment variable isn't set, then the value will come from the local configuration file. And if that variable isn't in the local configuration file, then the parameter will have its default value (defined in bigchaindb.__init__).

Parameters

- **`public_key`** (`str`) – the base58 encoded public key for the ED25519 curve.
- **`private_key`** (`str`) – the base58 encoded private key for the ED25519 curve.
- **`keyring`** (`list[str]`) – list of base58 encoded public keys of the federation nodes.

- **`connection`** (*`Connection`*) – A connection to the database.

`BLOCK_INVALID` = **'invalid'**
return if a block has been voted invalid

`BLOCK_VALID` = **'valid'**
return if a block is valid, or tx is in valid block

`TX_VALID` = **'valid'**
return if a block is valid, or tx is in valid block

`BLOCK_UNDECIDED` = **'undecided'**
return if block is undecided, or tx is in undecided block

`TX_UNDECIDED` = **'undecided'**
return if block is undecided, or tx is in undecided block

`TX_IN_BACKLOG` = **'backlog'**
return if transaction is in backlog

`federation`
Set of federation member public keys

`write_transaction`(*signed_transaction*)
Write the transaction to bigchain.

When first writing a transaction to the bigchain the transaction will be kept in a backlog until it has been validated by the nodes of the federation.

Parameters **`signed_transaction`** (*`Transaction`*) – transaction with the *signature* included.

Returns database response

Return type dict

`reassign_transaction`(*transaction*)
Assign a transaction to a new node

Parameters **`transaction`** (*`dict`*) – assigned transaction

Returns database response or None if no reassignment is possible

Return type dict

`delete_transaction`(**transaction_id*)
Delete a transaction from the backlog.

Parameters **`*transaction_id`** (*`str`*) – the transaction(s) to delete

Returns The database response.

`get_stale_transactions`()
Get a cursor of stale transactions.

Transactions are considered stale if they have been assigned a node, but are still in the backlog after some amount of time specified in the configuration

`validate_transaction`(*transaction*)
Validate a transaction.

Parameters **`transaction`** (*`Transaction`*) – transaction to validate.

Returns The transaction if the transaction is valid else it raises an exception describing the reason why the transaction is invalid.

is_new_transaction(*txid*, *exclude_block_id=None*)
Return True if the transaction does not exist in any VALID or UNDECIDED block. Return False otherwise.

Parameters

- **txid** (*str*) – Transaction ID
- **exclude_block_id** (*str*) – Exclude block from search

get_block(*block_id*, *include_status=False*)
Get the block with the specified *block_id* (and optionally its status)

Returns the block corresponding to *block_id* or None if no match is found.

Parameters

- **block_id** (*str*) – transaction id of the transaction to get
- **include_status** (*bool*) – also return the status of the block the return value is then a tuple: (block, status)

get_transaction(*txid*, *include_status=False*)
Get the transaction with the specified *txid* (and optionally its status)

This query begins by looking in the bigchain table for all blocks containing a transaction with the specified *txid*. If one of those blocks is valid, it returns the matching transaction from that block. Else if some of those blocks are undecided, it returns a matching transaction from one of them. If the transaction was found in invalid blocks only, or in no blocks, then this query looks for a matching transaction in the backlog table, and if it finds one there, it returns that.

Parameters

- **txid** (*str*) – transaction id of the transaction to get
- **include_status** (*bool*) – also return the status of the transaction the return value is then a tuple: (tx, status)

Returns A `Transaction` instance if the transaction was found in a valid block, an undecided block, or the backlog table, otherwise `None`. If `include_status` is `True`, also returns the transaction's status if the transaction was found.

get_status(*txid*)
Retrieve the status of a transaction with *txid* from bigchain.

Parameters **txid** (*str*) – transaction id of the transaction to query

Returns transaction status ('valid', 'undecided', or 'backlog'). If no transaction with that *txid* was found it returns *None*

Return type (string)

get_blocks_status_containing_tx(*txid*)
Retrieve block ids and statuses related to a transaction

Transactions may occur in multiple blocks, but no more than one valid block.

Parameters **txid** (*str*) – transaction id of the transaction to query

Returns A dict of blocks containing the transaction, e.g. {block_id_1: 'valid', block_id_2: 'invalid' ...}, or None

get_asset_by_id(*asset_id*)
Returns the asset associated with an asset_id.

Parameters **asset_id** (*str*) – The asset id.

Returns dict if the asset exists else None.

get_spent(*txid*, *output*)
Check if a *txid* was already used as an input.

A transaction can be used as an input for another transaction. Bigchain needs to make sure that a given *(txid, output)* is only used once.

This method will check if the *(txid, output)* has already been spent in a transaction that is in either the *VALID*, *UNDECIDED* or *BACKLOG* state.

Parameters

- **txid** (*str*) – The id of the transaction
- **output** (*num*) – the index of the output in the respective transaction

Returns The transaction (Transaction) that used the *(txid, output)* as an input else *None*

Raises

- `CriticalDoubleSpend` – If the given *(txid, output)* was spent in
- more than one valid transaction.

get_owned_ids(*owner*)
Retrieve a list of `txid` s that can be used as inputs.

Parameters **owner** (*str*) – base58 encoded public key.

Returns list of `txid` s and `output` s pointing to another transaction's condition

Return type `list` of TransactionLink

get_outputs_filtered(*owner*, *spent=None*)
Get a list of output links filtered on some criteria

Parameters

- **owner** (*str*) – base58 encoded public_key.
- **spent** (*bool*) – If `True` return only the spent outputs. If `False` return only unspent outputs. If spent is not specified (`None`) return all outputs.

Returns list of `txid` s and `output` s pointing to another transaction's condition

Return type `list` of TransactionLink

get_transactions_filtered(*asset_id*, *operation=None*)
Get a list of transactions filtered on some criteria

create_block(*validated_transactions*)
Creates a block given a list of *validated_transactions*.

Note that this method does not validate the transactions. Transactions should be validated before calling create_block.

Parameters **validated_transactions** (*list(Transaction)*) – list of validated transactions.

Returns created block.

Return type Block

validate_block(*block*)
Validate a block.

Parameters **block** (*Block*) – block to validate.

Returns The block if the block is valid else it raises and exception describing the reason why the block is invalid.

has_previous_vote(*block_id*)
Check for previous votes from this node

Parameters **block_id** (*str*) – the id of the block to check

Returns `True` if this block already has a valid vote from this node, `False` otherwise.

Return type bool

write_block(*block*)
Write a block to bigchain.

Parameters **block** (*Block*) – block to write to bigchain.

prepare_genesis_block()
Prepare a genesis block.

create_genesis_block()
Create the genesis block

Block created when bigchain is first initialized. This method is not atomic, there might be concurrency problems if multiple instances try to write the genesis block when the BigchainDB Federation is started, but it's a highly unlikely scenario.

vote(*block_id*, *previous_block_id*, *decision*, *invalid_reason=None*)
Create a signed vote for a block given the `previous_block_id` and the `decision` (valid/invalid).

Parameters

- **block_id** (*str*) – The id of the block to vote on.
- **previous_block_id** (*str*) – The id of the previous block.
- **decision** (*bool*) – Whether the block is valid or invalid.
- **invalid_reason** (*Optional[str]*) – Reason the block is invalid

write_vote(*vote*)
Write the vote to the database.

get_last_voted_block()
Returns the last block that this node voted on.

block_election_status(*block*)
Tally the votes on a block, and return the status: valid, invalid, or undecided.

get_assets(*asset_ids*)
Return a list of assets that match the asset_ids

Parameters **asset_ids** (`list` of `str`) – A list of asset_ids to retrieve from the database.

Returns The list of assets returned from the database.

Return type list

write_assets(*assets*)
Writes a list of assets into the database.

Parameters **assets** (`list` of `dict`) – A list of assets to write to the database.

text_search(*search*, *, *limit=0*)
Return an iterator of assets that match the text search

Parameters

- **search** (*str*) – Text search string to query the text index
- **limit** (*int, optional*) – Limit the number of returned documents.

Returns An iterator of assets that match the text search.

Return type iter

Pipelines

Block Creation

This module takes care of all the logic related to block creation.

The logic is encapsulated in the `BlockPipeline` class, while the sequence of actions to do on transactions is specified in the `create_pipeline` function.

class `bigchaindb.pipelines.block.`**`BlockPipeline`**

This class encapsulates the logic to create blocks.

Note: Methods of this class will be executed in different processes.

`filter_tx`(*tx*)

Filter a transaction.

Parameters **tx** (*dict*) – the transaction to process.

Returns The transaction if assigned to the current node, `None` otherwise.

Return type dict

`validate_tx`(*tx*)

Validate a transaction.

Also checks if the transaction already exists in the blockchain. If it does, or it's invalid, it's deleted from the backlog immediately.

Parameters **tx** (*dict*) – the transaction to validate.

Returns The transaction if valid, `None` otherwise.

Return type `Transaction`

`create`(*tx*, *timeout=False*)

Create a block.

This method accumulates transactions to put in a block and outputs a block when one of the following conditions is true: - the size limit of the block has been reached, or - a timeout happened.

Parameters

- **tx** (`Transaction`) – the transaction to validate, might be None if a timeout happens.
- **timeout** (*bool*) – `True` if a timeout happened (Default: `False`).

Returns The block, if a block is ready, or `None`.

Return type `Block`

`write`(*block*)

Write the block to the Database.

Parameters **block** (Block) – the block of transactions to write to the database.

Returns The Block.

Return type Block

delete_tx(*block*)
Delete transactions.

Parameters **block** (Block) – the block containg the transactions to delete.

Returns The block.

Return type Block

bigchaindb.pipelines.block.**tx_collector**()
A helper to deduplicate transactions

bigchaindb.pipelines.block.**create_pipeline**()
Create and return the pipeline of operations to be distributed on different processes.

bigchaindb.pipelines.block.**start**()
Create, start, and return the block pipeline.

Block Voting

This module takes care of all the logic related to block voting.

The logic is encapsulated in the Vote class, while the sequence of actions to do on transactions is specified in the create_pipeline function.

class bigchaindb.pipelines.vote.**Vote**
This class encapsulates the logic to vote on blocks.

Note: Methods of this class will be executed in different processes.

ungroup(*block_id*, *transactions*)
Given a block, ungroup the transactions in it.

Parameters

- **block_id** (*str*) – the id of the block in progress.
- **transactions** (*list (dict)*) – transactions of the block in progress.

Returns None if the block has been already voted, an iterator that yields a transaction, block id, and the total number of transactions contained in the block otherwise.

validate_tx(*tx_dict*, *block_id*, *num_tx*)

Validate a transaction. Transaction must also not be in any VALID block.

Parameters

- **tx_dict** (*dict*) – the transaction to validate
- **block_id** (*str*) – the id of block containing the transaction
- **num_tx** (*int*) – the total number of transactions to process

Returns Three values are returned, the validity of the transaction, block_id, num_tx.

vote (*tx_validity*, *block_id*, *num_tx*)
Collect the validity of transactions and cast a vote when ready.

Parameters

- **tx_validity** (*bool*) – the validity of the transaction
- **block_id** (*str*) – the id of block containing the transaction
- **num_tx** (*int*) – the total number of transactions to process

Returns None, or a vote if a decision has been reached.

write_vote (*vote*, *num_tx*)
Write vote to the database.

Parameters **vote** – the vote to write.

bigchaindb.pipelines.vote.**create_pipeline**()
Create and return the pipeline of operations to be distributed on different processes.

bigchaindb.pipelines.vote.**get_changefeed**()
Create and return ordered changefeed of blocks starting from last voted block

bigchaindb.pipelines.vote.**start**()
Create, start, and return the block pipeline.

Block Status

This module takes care of all the logic related to block status.

Specifically, what happens when a block becomes invalid. The logic is encapsulated in the `Election` class, while the sequence of actions is specified in `create_pipeline`.

class bigchaindb.pipelines.election.**Election** (*events_queue=None*)
Election class.

check_for_quorum (*next_vote*)
Checks if block has enough invalid votes to make a decision

Parameters **next_vote** – The next vote.

requeue_transactions (*invalid_block*)
Liquidates transactions from invalid blocks so they can be processed again

Stale Transaction Monitoring

This module monitors for stale transactions.

It reassigns transactions which have been assigned a node but remain in the backlog past a certain amount of time.

class bigchaindb.pipelines.stale.**StaleTransactionMonitor** (*timeout=5*, *backlog_reassign_delay=None*)
This class encapsulates the logic for re-assigning stale transactions.

Note: Methods of this class will be executed in different processes.

check_transactions()
Poll backlog for stale transactions

Returns txs to be re assigned

Return type txs (list)

`reassign_transactions`(*tx*)
Put tx back in backlog with new assignee

Returns transaction

`bigchaindb.pipelines.stale.`**`create_pipeline`**(*timeout=5*, *backlog_reassign_delay=5*)
Create and return the pipeline of operations to be distributed on different processes.

`bigchaindb.pipelines.stale.`**`start`**(*timeout=5*, *backlog_reassign_delay=None*)
Create, start, and return the block pipeline.

Database Backend Interfaces

Generic backend database interfaces expected by BigchainDB.

The interfaces in this module allow BigchainDB to be agnostic about its database backend. One can configure BigchainDB to use different databases as its data store by setting the `database.backend` property in the configuration or the `BIGCHAINDB_DATABASE_BACKEND` environment variable.

Generic Interfaces

`bigchaindb.backend.connection`

`bigchaindb.backend.connection.`**`connect`**(*backend=None*, *host=None*, *port=None*, *name=None*, *max_tries=None*, *connection_timeout=None*, *replicaset=None*, *ssl=None*, *login=None*, *password=None*, *ca_cert=None*, *certfile=None*, *keyfile=None*, *keyfile_passphrase=None*, *crlfile=None*)
Create a new connection to the database backend.

All arguments default to the current configuration's values if not given.

Parameters

- **backend** (*str*) – the name of the backend to use.
- **host** (*str*) – the host to connect to.
- **port** (*int*) – the port to connect to.
- **name** (*str*) – the name of the database to use.
- **replicaset** (*str*) – the name of the replica set (only relevant for MongoDB connections).

Returns An instance of `Connection` based on the given (or defaulted) `backend`.

Raises

- `ConnectionError` – If the connection to the database fails.
- `ConfigurationError` – If the given (or defaulted) `backend` is not supported or could not be loaded.
- `AuthenticationError` – If there is a OperationFailure due to Authentication failure after connecting to the database.

class `bigchaindb.backend.connection.`**`Connection`**(*host=None*, *port=None*, *dbname=None*, *connection_timeout=None*, *max_tries=None*, ***kwargs*)

Connection class interface.

All backend implementations should provide a connection class that inherits from and implements this class.

`__init__`(*host=None*, *port=None*, *dbname=None*, *connection_timeout=None*, *max_tries=None*, ***kwargs*)

Create a new `Connection` instance.

Parameters

- **`host`** (`str`) – the host to connect to.
- **`port`** (`int`) – the port to connect to.
- **`dbname`** (`str`) – the name of the database to use.
- **`connection_timeout`** (`int, optional`) – the milliseconds to wait until timing out the database connection attempt. Defaults to 5000ms.
- **`max_tries`** (`int, optional`) – how many tries before giving up, if 0 then try forever. Defaults to 3.
- **`**kwargs`** – arbitrary keyword arguments provided by the configuration's `database` settings

`run`(*query*)

Run a query.

Parameters **`query`** – the query to run

Raises

- `DuplicateKeyError` – If the query fails because of a duplicate key constraint.
- `OperationFailure` – If the query fails for any other reason.
- `ConnectionError` – If the connection to the database fails.

`connect`()

Try to connect to the database.

Raises `ConnectionError` – If the connection to the database fails.

`bigchaindb.backend.changefeed`

Changefeed interfaces for backends.

class `bigchaindb.backend.changefeed.`**`ChangeFeed`**(*table*, *operation*, ***, *prefeed=None*, *connection=None*)

Create a new changefeed.

It extends `multipipes.Node` to make it pluggable in other Pipelines instances, and makes usage of `self.outqueue` to output the data.

A changefeed is a real time feed on inserts, updates, and deletes, and is volatile. This class is a helper to create changefeeds. Moreover, it provides a way to specify a `prefeed` of iterable data to output before the actual changefeed.

`run_forever`()

Main loop of the `multipipes.Node`

This method is responsible for first feeding the prefeed to the outqueue and after that starting the changefeed and recovering from any errors that may occur in the backend.

run_changefeed()

Backend specific method to run the changefeed.

The changefeed is usually a backend cursor that is not closed when all the results are exausted. Instead it remains open waiting for new results.

This method should also filter each result based on the `operation` and put all matching results on the outqueue of `multipipes.Node`.

`bigchaindb.backend.changefeed.`**get_changefeed**(*connection*, *table*, *operation*, ***, *prefeed=None*)

Return a ChangeFeed.

Parameters

- **connection** (*Connection*) – A connection to the database.
- **table** (*str*) – name of the table to listen to for changes.
- **operation** (*int*) – can be ChangeFeed.INSERT, ChangeFeed.DELETE, or ChangeFeed.UPDATE. Combining multiple operation is possible with the bitwise `|` operator (e.g. `ChangeFeed.INSERT | ChangeFeed.UPDATE`)
- **prefeed** (*iterable*) – whatever set of data you want to be published first.

bigchaindb.backend.query

Query interfaces for backends.

`bigchaindb.backend.query.`**write_transaction**(*connection*, *signed_transaction*)

Write a transaction to the backlog table.

Parameters **signed_transaction** (*dict*) – a signed transaction.

Returns The result of the operation.

`bigchaindb.backend.query.`**update_transaction**(*connection*, *transaction_id*, *doc*)

Update a transaction in the backlog table.

Parameters

- **transaction_id** (*str*) – the id of the transaction.
- **doc** (*dict*) – the values to update.

Returns The result of the operation.

`bigchaindb.backend.query.`**delete_transaction**(*connection*, **transaction_id*)

Delete a transaction from the backlog.

Parameters ***transaction_id** (*str*) – the transaction(s) to delete.

Returns The database response.

`bigchaindb.backend.query.`**get_stale_transactions**(*connection*, *reassign_delay*)

Get a cursor of stale transactions.

Transactions are considered stale if they have been assigned a node, but are still in the backlog after some amount of time specified in the configuration.

Parameters **reassign_delay** (*int*) – threshold (in seconds) to mark a transaction stale.

Returns A cursor of transactions.

`bigchaindb.backend.query.`**`get_transaction_from_block`**`(`*connection*, *transaction_id*, *block_id*`)`

Get a transaction from a specific block.

Parameters

- **`transaction_id`** (*`str`*) – the id of the transaction.
- **`block_id`** (*`str`*) – the id of the block.

Returns The matching transaction.

`bigchaindb.backend.query.`**`get_transaction_from_backlog`**`(`*connection*, *transaction_id*`)`

Get a transaction from backlog.

Parameters **`transaction_id`** (*`str`*) – the id of the transaction.

Returns The matching transaction.

`bigchaindb.backend.query.`**`get_blocks_status_from_transaction`**`(`*connection*, *transaction_id*`)`

Retrieve block election information given a secondary index and value.

Parameters

- **`value`** – a value to search (e.g. transaction id string, payload hash string)
- **`index`** (*`str`*) – name of a secondary index, e.g. 'transaction_id'

Returns A list of blocks with with only election information

Return type `list` of `dict`

`bigchaindb.backend.query.`**`get_asset_by_id`**`(`*conneciton*, *asset_id*`)`

Returns the asset associated with an asset_id.

Parameters **`asset_id`** (*`str`*) – The asset id.

Returns Returns a rethinkdb cursor.

`bigchaindb.backend.query.`**`get_spent`**`(`*connection*, *transaction_id*, *condition_id*`)`

Check if a *txid* was already used as an input.

A transaction can be used as an input for another transaction. Bigchain needs to make sure that a given *txid* is only used once.

Parameters

- **`transaction_id`** (*`str`*) – The id of the transaction.
- **`condition_id`** (*`int`*) – The index of the condition in the respective transaction.

Returns The transaction that used the *txid* as an input else *None*

`bigchaindb.backend.query.`**`get_spending_transactions`**`(`*connection*, *inputs*`)`

Return transactions which spend given inputs

Parameters **`inputs`** (*`list`*) – list of {txid, output}

Returns Iterator of (block_ids, transaction) for transactions that spend given inputs.

`bigchaindb.backend.query.`**`get_owned_ids`**`(`*connection*, *owner*`)`

Retrieve a list of *txids* that can we used has inputs.

Parameters **`owner`** (*`str`*) – base58 encoded public key.

Returns Iterator of (block_id, transaction) for transactions that list given owner in conditions.

`bigchaindb.backend.query.`**`get_votes_by_block_id`**(*connection*, *block_id*)

Get all the votes casted for a specific block.

Parameters **`block_id`** (*`str`*) – the block id to use.

Returns A cursor for the matching votes.

`bigchaindb.backend.query.`**`get_votes_by_block_id_and_voter`**(*connection*, *block_id*, *node_pubkey*)

Get all the votes casted for a specific block by a specific voter.

Parameters

- **`block_id`** (*`str`*) – the block id to use.
- **`node_pubkey`** (*`str`*) – base58 encoded public key

Returns A cursor for the matching votes.

`bigchaindb.backend.query.`**`get_votes_for_blocks_by_voter`**(*connection*, *block_ids*, *pubkey*)

Return votes for many block_ids

Parameters

- **`block_ids`** (*`set`*) – block_ids
- **`pubkey`** (*`str`*) – public key of voting node

Returns A cursor of votes matching given block_ids and public key

`bigchaindb.backend.query.`**`write_block`**(*connection*, *block*)

Write a block to the bigchain table.

Parameters **`block`** (*`dict`*) – the block to write.

Returns The database response.

`bigchaindb.backend.query.`**`get_block`**(*connection*, *block_id*)

Get a block from the bigchain table.

Parameters **`block_id`** (*`str`*) – block id of the block to get

Returns the block or *None*

Return type *block* (dict)

`bigchaindb.backend.query.`**`write_assets`**(*connection*, *assets*)

Write a list of assets to the assets table.

Parameters **`assets`** (*`list`*) – a list of assets to write.

Returns The database response.

`bigchaindb.backend.query.`**`get_assets`**(*connection*, *asset_ids*)

Get a list of assets from the assets table.

Parameters

- **`asset_ids`** (*`list`*) – a list of ids for the assets to be retrieved from
- **`database.`** (*`the`*) –

Returns the list of returned assets.

Return type assets (list)

`bigchaindb.backend.query.`**`count_blocks`**(*connection*)

Count the number of blocks in the bigchain table.

Returns The number of blocks.

`bigchaindb.backend.query.`**`count_backlog`**(*connection*)

Count the number of transactions in the backlog table.

Returns The number of transactions in the backlog.

`bigchaindb.backend.query.`**`write_vote`**(*connection*, *vote*)

Write a vote to the votes table.

Parameters **`vote`** (*`dict`*) – the vote to write.

Returns The database response.

`bigchaindb.backend.query.`**`get_genesis_block`**(*connection*)

Get the genesis block.

Returns The genesis block

`bigchaindb.backend.query.`**`get_last_voted_block_id`**(*connection*, *node_pubkey*)

Get the last voted block for a specific node.

Parameters **`node_pubkey`** (*`str`*) – base58 encoded public key.

Returns The id of the last block the node has voted on. If the node didn't cast any vote then the genesis block id is returned.

`bigchaindb.backend.query.`**`get_txids_filtered`**(*connection*, *asset_id*, *operation=None*)

Return all transactions for a particular asset id and optional operation.

Parameters

- **`asset_id`** (*`str`*) – ID of transaction that defined the asset
- **`operation`** (*`str) (optional`*) – Operation to filter on

`bigchaindb.backend.query.`**`get_new_blocks_feed`**(*connection*, *start_block_id*)

Return a generator that yields change events of the blocks feed

Parameters **`start_block_id`** (*`str`*) – ID of block to resume from

Returns Generator of change events

`bigchaindb.backend.query.`**`text_search`**(*conn*, *search*, ***, *language='english'*, *case_sensitive=False*, *diacritic_sensitive=False*, *text_score=False*, *limit=0*)

Return all the assets that match the text search.

The results are sorted by text score. For more information about the behavior of text search on MongoDB see https://docs.mongodb.com/manual/reference/operator/query/text/#behavior

Parameters

- **`search`** (*`str`*) – Text search string to query the text index
- **`language`** (*`str, optional`*) – The language for the search and the rules for stemmer and tokenizer. If the language is `None` text search uses simple tokenization and no stemming.
- **`case_sensitive`** (*`bool, optional`*) – Enable or disable case sensitive search.
- **`diacritic_sensitive`** (*`bool, optional`*) – Enable or disable case sensitive diacritic search.
- **`text_score`** (*`bool, optional`*) – If `True` returns the text score with each document.
- **`limit`** (*`int, optional`*) – Limit the number of returned documents.

Returns a list of assets

Return type `list` of `dict`

Raises `OperationError` – If the backend does not support text search

`bigchaindb.backend.schema`

Database creation and schema-providing interfaces for backends.

`bigchaindb.backend.schema.`**`TABLES`**

tuple – The three standard tables BigchainDB relies on:

- `backlog` for incoming transactions awaiting to be put into a block.
- `bigchain` for blocks.
- `votes` to store votes for each block by each federation node.

`bigchaindb.backend.schema.`**`create_database`**(*connection*, *dbname*)

Create database to be used by BigchainDB.

Parameters **`dbname`** (*`str`*) – the name of the database to create.

Raises `DatabaseAlreadyExists` – If the given `dbname` already exists as a database.

`bigchaindb.backend.schema.`**`create_tables`**(*connection*, *dbname*)

Create the tables to be used by BigchainDB.

Parameters **`dbname`** (*`str`*) – the name of the database to create tables for.

`bigchaindb.backend.schema.`**`create_indexes`**(*connection*, *dbname*)

Create the indexes to be used by BigchainDB.

Parameters **`dbname`** (*`str`*) – the name of the database to create indexes for.

`bigchaindb.backend.schema.`**`drop_database`**(*connection*, *dbname*)

Drop the database used by BigchainDB.

Parameters **`dbname`** (*`str`*) – the name of the database to drop.

Raises `DatabaseDoesNotExist` – If the given `dbname` does not exist as a database.

`bigchaindb.backend.schema.`**`init_database`**(*connection=None*, *dbname=None*)

Initialize the configured backend for use with BigchainDB.

Creates a database with `dbname` with any required tables and supporting indexes.

Parameters

- **`connection`** (*`Connection`*) – an existing connection to use to initialize the database. Creates one if not given.
- **`dbname`** (*`str`*) – the name of the database to create. Defaults to the database name given in the BigchainDB configuration.

Raises `DatabaseAlreadyExists` – If the given `dbname` already exists as a database.

`bigchaindb.backend.admin`

Database configuration functions.

`bigchaindb.backend.utils`

exception `bigchaindb.backend.utils.`**`ModuleDispatchRegistrationError`**

Raised when there is a problem registering dispatched functions for a module

RethinkDB Backend

RethinkDB backend implementation.

Contains a RethinkDB-specific implementation of the *`changefeed`*, *`query`*, and *`schema`* interfaces.

You can specify BigchainDB to use RethinkDB as its database backend by either setting `database.backend` to `'rethinkdb'` in your configuration file, or setting the `BIGCHAINDB_DATABASE_BACKEND` environment variable to `'rethinkdb'`.

If configured to use RethinkDB, BigchainDB will automatically return instances of `RethinkDBConnection` for *`connect()`* and dispatch calls of the generic backend interfaces to the implementations in this module.

`bigchaindb.backend.rethinkdb.connection`

class `bigchaindb.backend.rethinkdb.connection.`**`RethinkDBConnection`**(*host=None, port=None, dbname=None, connection_timeout=None, max_tries=None, **kwargs*)

This class is a proxy to run queries against the database, it is:

- lazy, since it creates a connection only when needed
- resilient, because before raising exceptions it tries more times to run the query or open a connection.

`run`(*query*)

Run a RethinkDB query.

Parameters **`query`** – the RethinkDB query.

Raises `rethinkdb.ReqlDriverError` – After `max_tries`.

`bigchaindb.backend.rethinkdb.schema`

`bigchaindb.backend.rethinkdb.query`

`bigchaindb.backend.rethinkdb.query.`**`unwind_block_transactions`**(*block*)

Yield a block for each transaction in given block

`bigchaindb.backend.rethinkdb.changefeed`

class `bigchaindb.backend.rethinkdb.changefeed.`**`RethinkDBChangeFeed`**(*table, operation, *, prefeed=None, connection=None*)

This class wraps a RethinkDB changefeed as a multipipes Node.

`bigchaindb.backend.rethinkdb.changefeed.`**`run_changefeed`**(*connection*, *table*)

Encapsulate operational logic of tailing changefeed from RethinkDB

`bigchaindb.backend.rethinkdb.changefeed.`**`get_changefeed`**(*connection*, *table*, *operation*, ***, *prefeed=None*)

Return a RethinkDB changefeed.

Returns An instance of `RethinkDBChangeFeed`.

`bigchaindb.backend.rethinkdb.admin`

Database configuration functions.

`bigchaindb.backend.rethinkdb.admin.`**`get_config`**(*connection*, ***, *table*)

Get the configuration of the given table.

Parameters

- **`connection`** (*`Connection`*) – A connection to the database.
- **`table`** (*`str`*) – The name of the table to get the configuration for.

Returns The configuration of the given table

Return type dict

`bigchaindb.backend.rethinkdb.admin.`**`reconfigure`**(*connection*, ***, *table*, *shards*, *replicas*, *primary_replica_tag=None*, *dry_run=False*, *nonvoting_replica_tags=None*)

Reconfigures the given table.

Parameters

- **`connection`** (*`Connection`*) – A connection to the database.
- **`table`** (*`str`*) – The name of the table to reconfigure.
- **`shards`** (*`int`*) – The number of shards, an integer from 1-64.
- **`replicas`** (`int` | `dict`) –
  – If replicas is an integer, it specifies the number of replicas per shard. Specifying more replicas than there are servers will return an error.
  – If replicas is a dictionary, it specifies key-value pairs of server tags and the number of replicas to assign to those servers:

```
{'africa': 2, 'asia': 4, 'europe': 2, ...}
```

- **`primary_replica_tag`** (*`str`*) – The primary server specified by its server tag. Required if `replicas` is a dictionary. The tag must be in the `replicas` dictionary. This must not be specified if `replicas` is an integer. Defaults to `None`.
- **`dry_run`** (*`bool`*) – If `True` the generated configuration will not be applied to the table, only returned. Defaults to `False`.
- **`nonvoting_replica_tags`** (`list` of `str`) – Replicas with these server tags will be added to the `nonvoting_replicas` list of the resulting configuration. Defaults to `None`.

Returns

A dictionary with possibly three keys:

- `reconfigured`: the number of tables reconfigured. This will be `0` if `dry_run` is `True`.
- `config_changes`: a list of new and old table configuration values.
- `status_changes`: a list of new and old table status values.

For more information please consult RethinkDB's documentation ReQL command: reconfigure.

Return type dict

Raises `OperationError` – If the reconfiguration fails due to a RethinkDB `ReqlOpFailedError` or `ReqlQueryLogicError`.

`bigchaindb.backend.rethinkdb.admin.`**`set_shards`**(*connection*, *, *shards*, *dry_run=False*)

Sets the shards for the tables *`TABLES`*.

Parameters

- **`connection`** (*`Connection`*) – A connection to the database.
- **`shards`** (*`int`*) – The number of shards, an integer from 1-64.
- **`dry_run`** (*`bool`*) – If `True` the generated configuration will not be applied to the table, only returned. Defaults to `False`.

Returns

A dictionary with the configuration and status changes. For more details please see *`reconfigure()`*.

Return type dict

`bigchaindb.backend.rethinkdb.admin.`**`set_replicas`**(*connection*, *, *replicas*, *dry_run=False*)

Sets the replicas for the tables *`TABLES`*.

Parameters

- **`connection`** (*`Connection`*) – A connection to the database.
- **`replicas`** (*`int`*) – The number of replicas per shard. Specifying more replicas than there are servers will return an error.
- **`dry_run`** (*`bool`*) – If `True` the generated configuration will not be applied to the table, only returned. Defaults to `False`.

Returns

A dictionary with the configuration and status changes. For more details please see *`reconfigure()`*.

Return type dict

MongoDB Backend

Stay tuned!

Command Line Interface

bigchaindb.commands.bigchaindb

Implementation of the *bigchaindb* command, the command-line interface (CLI) for BigchainDB Server.

`bigchaindb.commands.bigchaindb.`**`run_show_config`**(*args*)
Show the current configuration

`bigchaindb.commands.bigchaindb.`**`run_configure`**(*args*, *skip_if_exists=False*)
Run a script to configure the current node.

Parameters **`skip_if_exists`** (*`bool`*) – skip the function if a config file already exists

`bigchaindb.commands.bigchaindb.`**`run_export_my_pubkey`**(*args*)
Export this node's public key to standard output

`bigchaindb.commands.bigchaindb.`**`run_init`**(*args*)
Initialize the database

`bigchaindb.commands.bigchaindb.`**`run_drop`**(*args*)
Drop the database

`bigchaindb.commands.bigchaindb.`**`run_start`**(*args*)
Start the processes to run the node

bigchaindb.commands.utils

Utility functions and basic common arguments for `argparse.ArgumentParser`.

`bigchaindb.commands.utils.`**`configure_bigchaindb`**(*command*)
Decorator to be used by command line functions, such that the configuration of bigchaindb is performed before the execution of the command.

Parameters **`command`** – The command to decorate.

Returns The command wrapper function.

`bigchaindb.commands.utils.`**`start_logging_process`**(*command*)
Decorator to start the logging subscriber process.

Parameters **`command`** – The command to decorate.

Returns The command wrapper function.

Important: Configuration, if needed, should be applied before invoking this decorator, as starting the subscriber process for logging will configure the root logger for the child process based on the state of `bigchaindb.config` at the moment this decorator is invoked.

`bigchaindb.commands.utils.`**`input_on_stderr`**(*prompt='‘*, *default=None*, *convert=None*)
Output a string to stderr and wait for input.

Parameters

- **`prompt`** (*`str`*) – the message to display.
- **`default`** – the default value to return if the user leaves the field empty
- **`convert`** (*`callable`*) – a callable to be used to convert the value the user inserted. If None, the type of `default` will be used.

`bigchaindb.commands.utils.`**`start_rethinkdb`**`()`

Start RethinkDB as a child process and wait for it to be available.

Raises *:class:'~bigchaindb.common.exceptions.StartupError' if* – RethinkDB cannot be started.

`bigchaindb.commands.utils.`**`start`**`(`*parser*, *argv*, *scope*`)`

Utility function to execute a subcommand.

The function will look up in the `scope` if there is a function called `run_<parser.args.command>` and will run it using `parser.args` as first positional argument.

Parameters

- **`parser`** – an ArgumentParser instance.
- **`argv`** – the list of command line arguments without the script name.
- **`scope`** (*`dict`*) – map containing (eventually) the functions to be called.

Raises `NotImplementedError` – if `scope` doesn't contain a function called `run_<parser.args.command>`.

`bigchaindb.commands.utils.`**`mongodb_host`**`(`*host*`)`

Utility function that works as a type for mongodb `host` args.

This function validates the `host` args provided by to the `add-replicas` and `remove-replicas` commands and checks if each arg is in the form "host:port"

Parameters **`host`** (*`str`*) – A string containing hostname and port (e.g. "host:port")

Raises `ArgumentTypeError` – if it fails to parse the argument

Basic AWS Setup

Before you can deploy anything on AWS, you must do a few things.

Get an AWS Account

If you don't already have an AWS account, you can sign up for one for free at aws.amazon.com.

Install the AWS Command-Line Interface

To install the AWS Command-Line Interface (CLI), just do:

```
pip install awscli
```

Create an AWS Access Key

The next thing you'll need is AWS access keys (access key ID and secret access key). If you don't have those, see the AWS documentation about access keys.

You should also pick a default AWS region name (e.g. `eu-central-1`). That's where your cluster will run. The AWS documentation has a list of them.

Once you've got your AWS access key, and you've picked a default AWS region name, go to a terminal session and enter:

```
aws configure
```

and answer the four questions. For example:

```
AWS Access Key ID [None]: AKIAIOSFODNN7EXAMPLE
AWS Secret Access Key [None]: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
Default region name [None]: eu-central-1
Default output format [None]: [Press Enter]
```

This writes two files: `~/.aws/credentials` and `~/.aws/config`. AWS tools and packages look for those files.

Generate an RSA Key Pair for SSH

Eventually, you'll have one or more instances (virtual machines) running on AWS and you'll want to SSH to them. To do that, you need a public/private key pair. The public key will be sent to AWS, and you can tell AWS to put it in any instances you provision there. You'll keep the private key on your local workstation.

See the page about how to generate a key pair for SSH.

Send the Public Key to AWS

To send the public key to AWS, use the AWS Command-Line Interface:

```
aws ec2 import-key-pair \
--key-name "<key-name>" \
--public-key-material file://~/.ssh/<key-name>.pub
```

If you're curious why there's a `file://` in front of the path to the public key, see issue aws/aws-cli#41 on GitHub.

If you want to verify that your key pair was imported by AWS, go to the Amazon EC2 console, select the region you gave above when you did `aws configure` (e.g. eu-central-1), click on **Key Pairs** in the left sidebar, and check that `<key-name>` is listed.

Deploy a RethinkDB-Based Testing Cluster on AWS

This section explains a way to deploy a *RethinkDB-based* cluster of BigchainDB nodes on Amazon Web Services (AWS) for testing purposes.

Why?

Why would anyone want to deploy a centrally-controlled BigchainDB cluster? Isn't BigchainDB supposed to be decentralized, where each node is controlled by a different person or organization?

Yes! These scripts are for deploying a testing cluster, not a production cluster.

How?

We use some Bash and Python scripts to launch several instances (virtual servers) on Amazon Elastic Compute Cloud (EC2). Then we use Fabric to install RethinkDB and BigchainDB on all those instances.

Python Setup

The instructions that follow have been tested on Ubuntu 16.04. Similar instructions should work on similar Linux distros.

Note: Our Python scripts for deploying to AWS use Python 2 because Fabric doesn't work with Python 3.

You must install the Python package named `fabric`, but it depends on the `cryptography` package, and that depends on some OS-level packages. On Ubuntu 16.04, you can install those OS-level packages using:

```
sudo apt-get install build-essential libssl-dev libffi-dev python-dev
```

For other operating systems, see the installation instructions for the `cryptography` package.

Maybe create a Python 2 virtual environment and activate it. Then install the following Python packages (in that virtual environment):

```
pip install fabric fabtools requests boto3 awscli
```

What did you just install?

- "Fabric is a Python (2.5-2.7) library and command-line tool for streamlining the use of SSH for application deployment or systems administration tasks."
- fabtools are "tools for writing awesome Fabric files"
- requests is a Python package/library for sending HTTP requests
- "Boto is the Amazon Web Services (AWS) SDK for Python, which allows Python developers to write software that makes use of Amazon services like S3 and EC2." (`boto3` is the name of the latest Boto package.)
- The aws-cli package, which is an AWS Command Line Interface (CLI).

Setting up in AWS

See the page about basic AWS Setup in the Appendices.

Get Enough Amazon Elastic IP Addresses

The AWS cluster deployment scripts use elastic IP addresses (although that may change in the future). By default, AWS accounts get five elastic IP addresses. If you want to deploy a cluster with more than five nodes, then you will need more than five elastic IP addresses; you may have to apply for those; see the AWS documentation on elastic IP addresses.

Create an Amazon EC2 Security Group

Go to the AWS EC2 Console and select "Security Groups" in the left sidebar. Click the "Create Security Group" button. You can name it whatever you like. (Notes: The default name in the example AWS deployment configuration file is `bigchaindb`. We had problems with names containing dashes.) The description should be something to help you remember what the security group is for.

For a super lax, somewhat risky, anything-can-enter security group, add these rules for Inbound traffic:

- Type = All TCP, Protocol = TCP, Port Range = 0-65535, Source = 0.0.0.0/0
- Type = SSH, Protocol = SSH, Port Range = 22, Source = 0.0.0.0/0
- Type = All UDP, Protocol = UDP, Port Range = 0-65535, Source = 0.0.0.0/0

- Type = All ICMP, Protocol = ICMP, Port Range = 0-65535, Source = 0.0.0.0/0

(Note: Source = 0.0.0.0/0 is CIDR notation for "allow this traffic to come from *any* IP address.")

If you want to set up a more secure security group, see the Notes for Firewall Setup.

Deploy a BigchainDB Cluster

Step 1

Suppose *N* is the number of nodes you want in your BigchainDB cluster. If you already have a set of *N* BigchainDB configuration files in the `deploy-cluster-aws/confiles` directory, then you can jump to the next step. To create such a set, you can do something like:

```
# in a Python 3 virtual environment where bigchaindb is installed
cd bigchaindb
cd deploy-cluster-aws
./make_confiles.sh confiles 3
```

That will create three (3) *default* BigchainDB configuration files in the `deploy-cluster-aws/confiles` directory (which will be created if it doesn't already exist). The three files will be named `bcdb_conf0`, `bcdb_conf1`, and `bcdb_conf2`.

You can look inside those files if you're curious. For example, the default keyring is an empty list. Later, the deployment script automatically changes the keyring of each node to be a list of the public keys of all other nodes. Other changes are also made. That is, the configuration files generated in this step are *not* what will be sent to the deployed nodes; they're just a starting point.

Step 2

Step 2 is to make an AWS deployment configuration file, if necessary. There's an example AWS configuration file named `example_deploy_conf.py`. It has many comments explaining each setting. The settings in that file are (or should be):

```
NUM_NODES=3
BRANCH="master"
SSH_KEY_NAME="not-set-yet"
USE_KEYPAIRS_FILE=False
IMAGE_ID="ami-8504fdea"
INSTANCE_TYPE="t2.medium"
SECURITY_GROUP="bigchaindb"
USING_EBS=True
EBS_VOLUME_SIZE=30
EBS_OPTIMIZED=False
ENABLE_WEB_ADMIN=True
BIND_HTTP_TO_LOCALHOST=True
```

Make a copy of that file and call it whatever you like (e.g. `cp example_deploy_conf.py my_deploy_conf.py`). You can leave most of the settings at their default values, but you must change the value of `SSH_KEY_NAME` to the name of your private SSH key. You can do that with a text editor. Set `SSH_KEY_NAME` to the name you used for `<key-name>` when you generated an RSA key pair for SSH (in basic AWS setup).

You'll also want to change the `IMAGE_ID` to one that's up-to-date and available in your AWS region. If you don't remember your AWS region, then look in your `$HOME/.aws/config` file. You can find an up-to-date Ubuntu image ID for your region at https://cloud-images.ubuntu.com/locator/ec2/. An example search string is "eu-central-1 16.04 LTS amd64 hvm:ebs-ssd". You should replace "eu-central-1" with your region name.

If you want your nodes to have a predictable set of pre-generated keypairs, then you should 1) set `USE_KEYPAIRS_FILE=True` in the AWS deployment configuration file, and 2) provide a `keypairs.py` file containing enough keypairs for all of your nodes. You can generate a `keypairs.py` file using the `write_keypairs_file.py` script. For example:

```
# in a Python 3 virtual environment where bigchaindb is installed
cd bigchaindb
cd deploy-cluster-aws
python3 write_keypairs_file.py 100
```

The above command generates a `keypairs.py` file with 100 keypairs. You can generate more keypairs than you need, so you can use the same list over and over again, for different numbers of servers. The deployment scripts will only use the first NUM_NODES keypairs.

Step 3

Step 3 is to launch the nodes ("instances") on AWS, to install all the necessary software on them, configure the software, run the software, and more. Here's how you'd do that:

```
# in a Python 2.5-2.7 virtual environment where fabric, boto3, etc. are installed
cd bigchaindb
cd deploy-cluster-aws
./awsdeploy.sh my_deploy_conf.py
# Only if you want to set the replication factor to 3
fab set_replicas:3
# Only if you want to start BigchainDB on all the nodes:
fab start_bigchaindb
```

`awsdeploy.sh` is a Bash script which calls some Python and Fabric scripts. If you're curious what it does, the source code has many explanatory comments.

It should take a few minutes for the deployment to finish. If you run into problems, see the section on **Known Deployment Issues** below.

The EC2 Console has a section where you can see all the instances you have running on EC2. You can `ssh` into a running instance using a command like:

```
ssh -i pem/bigchaindb.pem ubuntu@ec2-52-29-197-211.eu-central-1.compute.amazonaws.com
```

except you'd replace the `ec2-52-29-197-211.eu-central-1.compute.amazonaws.com` with the public DNS name of the instance you want to `ssh` into. You can get that from the EC2 Console: just click on an instance and look in its details pane at the bottom of the screen. Some commands you might try:

```
ip addr show
sudo service rethinkdb status
bigchaindb --help
bigchaindb show-config
```

If you enabled the RethinkDB web interface (by setting `ENABLE_WEB_ADMIN=True` in your AWS configuration file), then you can also check that. The way to do that depends on how `BIND_HTTP_TO_LOCALHOST` was set (in your AWS deployment configuration file):

- If it was set to `False`, then just go to your web browser and visit a web address like `http://ec2-52-29-197-211.eu-central-1.compute.amazonaws.com:8080/`. (Replace `ec2-...aws.com` with the hostname of one of your instances.)
- If it was set to `True` (the default in the example config file), then follow the instructions in the "Via a SOCKS proxy" section of the "Secure your cluster" page of the RethinkDB documentation.

Server Monitoring with New Relic

New Relic is a business that provides several monitoring services. One of those services, called Server Monitoring, can be used to monitor things like CPU usage and Network I/O on BigchainDB instances. To do that:

1. Sign up for a New Relic account
2. Get your New Relic license key
3. Put that key in an environment variable named `NEWRELIC_KEY`. For example, you might add a line like the following to your `~/.bashrc` file (if you use Bash): `export NEWRELIC_KEY=<insert your key here>`
4. Once you've deployed a BigchainDB cluster on AWS as above, you can install a New Relic system monitor (agent) on all the instances using:

```
# in a Python 2.5-2.7 virtual environment where fabric, boto3, etc. are installed
fab install_newrelic
```

Once the New Relic system monitor (agent) is installed on the instances, it will start sending server stats to New Relic on a regular basis. It may take a few minutes for data to show up in your New Relic dashboard (under New Relic Servers).

Shutting Down a Cluster

There are fees associated with running instances on EC2, so if you're not using them, you should terminate them. You can do that using the AWS EC2 Console.

The same is true of your allocated elastic IP addresses. There's a small fee to keep them allocated if they're not associated with a running instance. You can release them using the AWS EC2 Console, or by using a handy little script named `release_eips.py`. For example:

```
$ python release_eips.py
You have 2 allocated elactic IPs which are not associated with instances
0: Releasing 52.58.110.110
(It has Domain = vpc.)
1: Releasing 52.58.107.211
(It has Domain = vpc.)
```

Known Deployment Issues

NetworkError

If you tested with a high sequence it might be possible that you run into an error message like this:

```
NetworkError: Host key for ec2-xx-xx-xx-xx.eu-central-1.compute.amazonaws.com
did not match pre-existing key! Server's key was changed recently, or possible
man-in-the-middle attack.
```

If so, just clean up your `known_hosts` file and start again. For example, you might copy your current `known_hosts` file to `old_known_hosts` like so:

```
mv ~/.ssh/known_hosts ~/.ssh/old_known_hosts
```

Then terminate your instances and try deploying again with a different tag.

Failure of sudo apt-get update

The first thing that's done on all the instances, once they're running, is basically `sudo apt-get update`. Sometimes that fails. If so, just terminate your instances and try deploying again with a different tag. (These problems seem to be time-bounded, so maybe wait a couple of hours before retrying.)

Failure when Installing Base Software

If you get an error with installing the base software on the instances, then just terminate your instances and try deploying again with a different tag.

Template: Using Terraform to Provision an Ubuntu Machine on AWS

This page explains a way to use Terraform to provision an Ubuntu machine (i.e. an EC2 instance with Ubuntu 16.04) and other resources on AWS. That machine can then be used to host a one-machine BigchainDB node, for example.

Note: We're not actively maintaining the associated Terraform files. You may find them useful nevertheless, which is why we moved this page to the Appendices rather than deleting it.

Install Terraform

The Terraform documentation has installation instructions for all common operating systems.

If you don't want to run Terraform on your local machine, you can install it on a cloud machine under your control (e.g. on AWS).

Note: Hashicorp has an enterprise version of Terraform called "Terraform Enterprise." You can license it by itself or get it as part of Atlas. If you decide to license Terraform Enterprise or Atlas, be sure to install it on your own hosting (i.e. "on premise"), not on the hosting provided by Hashicorp. The reason is that BigchainDB clusters are supposed to be decentralized. If everyone used Hashicorp's hosted Atlas, then that would be a point of centralization.

Ubuntu Installation Tips

If you want to install Terraform on Ubuntu, first download the .zip file. Then install it in `/opt`:

```
sudo mkdir -p /opt/terraform
sudo unzip path/to/zip-file.zip -d /opt/terraform
```

Why install it in `/opt`? See the answers at Ask Ubuntu.

Next, add `/opt/terraform` to your path. If you use bash for your shell, then you could add this line to `~/.bashrc`:

```
export PATH="/opt/terraform:$PATH"
```

After doing that, relaunch your shell or force it to read `~/.bashrc` again, e.g. by doing `source ~/.bashrc`. You can verify that terraform is installed and in your path by doing:

```
terraform --version
```

It should say the current version of Terraform.

Get Set Up to Use Terraform

First, do the basic AWS setup steps outlined in the Appendices.

Then go to the `.../bigchaindb/ntools/one-m/aws/` directory and open the file `variables.tf`. Most of the variables have sensible default values, but you can change them if you like. In particular, you may want to change `aws_region`. (Terraform looks in `~/.aws/credentials` to get your AWS credentials, so you don't have to enter those anywhere.)

The `ssh_key_name` has no default value, so Terraform will prompt you every time it needs it.

To see what Terraform will do, run:

```
terraform plan
```

It should ask you the value of `ssh_key_name`.

It figured out the plan by reading all the `.tf` Terraform files in the directory.

If you don't want to be asked for the `ssh_key_name`, you can change the default value of `ssh_key_name` (in the file `variables.tf`) or you can set an environmen variable named `TF_VAR_ssh_key_name`.

Use Terraform to Provision Resources

To provision all the resources specified in the plan, do the following. **Note: This will provision actual resources on AWS, and those cost money. Be sure to shut down the resources you don't want to keep running later, otherwise the cost will keep growing.**

```
terraform apply
```

Terraform will report its progress as it provisions all the resources. Once it's done, you can go to the Amazon EC2 web console and see the instance, its security group, its elastic IP, and its attached storage volumes (one for the root directory and one for RethinkDB storage).

At this point, there is no software installed on the instance except for Ubuntu 16.04 and whatever else came with the Amazon Machine Image (AMI) specified in the Terraform configuration (files).

The next step is to install, configure and run all the necessary software for a BigchainDB node. You could use our example Ansible playbook to do that.

Optional: "Destroy" the Resources

If you want to shut down all the resources just provisioned, you must first disable termination protection on the instance:

1. Go to the EC2 console and select the instance you just launched. It should be named `BigchainDB_node`.
2. Click **Actions** > **Instance Settings** > **Change Termination Protection** > **Yes, Disable**
3. Back in your terminal, do `terraform destroy`

Terraform should "destroy" (i.e. terminate or delete) all the AWS resources you provisioned above.

If it fails (e.g. because of an attached and mounted EBS volume), then you can terminate the instance using the EC2 console: **Actions** > **Instance State** > **Terminate** > **Yes, Terminate**. Once the instance is terminated, you should still do `terraform destroy` to make sure that all the other resources are destroyed.

Template: Ansible Playbook to Run a BigchainDB Node on an Ubuntu Machine

This page explains how to use Ansible to install, configure and run all the software needed to run a one-machine BigchainDB node on a server running Ubuntu 16.04.

Note: We're not actively maintaining the associated Ansible files (e.g. playbooks). They are RethinkDB-specific, even though we now recommend using MongoDB. You may find the old Ansible stuff useful nevertheless, which is why we moved this page to the Appendices rather than deleting it.

Install Ansible

The Ansible documentation has installation instructions. Note the control machine requirements: at the time of writing, Ansible required Python 2.6 or 2.7. (Python 3 support is coming: "Ansible 2.2 features a tech preview of Python 3 support." and the latest version, as of January 31, 2017, was 2.2.1.0. For now, it's probably best to use it with Python 2.)

For example, you could create a special Python 2.x virtualenv named `ansenv` and then install Ansible in it:

```
cd repos/bigchaindb/ntools
virtualenv -p /usr/local/lib/python2.7.11/bin/python ansenv
source ansenv/bin/activate
pip install ansible
```

About Our Example Ansible Playbook

Our example Ansible playbook installs, configures and runs a basic BigchainDB node on an Ubuntu 16.04 machine. That playbook is in `.../bigchaindb/ntools/one-m/ansible/one-m-node.yml`.

When you run the playbook (as per the instructions below), it ensures all the necessary software is installed, configured and running. It can be used to get a BigchainDB node set up on a bare Ubuntu 16.04 machine, but it can also be used to ensure that everything is okay on a running BigchainDB node. (If you run the playbook against a host where everything is okay, then it won't change anything on that host.)

Create an Ansible Inventory File

An Ansible "inventory" file is a file which lists all the hosts (machines) you want to manage using Ansible. (Ansible will communicate with them via SSH.) Right now, we only want to manage one host.

First, determine the public IP address of the host (i.e. something like `192.0.2.128`).

Then create a one-line text file named `hosts` by doing this:

```
# cd to the directory .../bigchaindb/ntools/one-m/ansible
echo "192.0.2.128" > hosts
```

but replace `192.0.2.128` with the IP address of the host.

Run the Ansible Playbook(s)

The latest Ubuntu 16.04 AMIs from Canonical don't include Python 2 (which is required by Ansible), so the first step is to run a small Ansible playbook to install Python 2 on the managed node:

```
# cd to the directory .../bigchaindb/ntools/one-m/ansible
ansible-playbook -i hosts --private-key ~/.ssh/<key-name> install-python2.yml
```

where `<key-name>` should be replaced by the name of the SSH private key you created earlier (for SSHing to the host machine at your cloud hosting provider).

The next step is to run the Ansible playbook named `one-m-node.yml`:

```
# cd to the directory .../bigchaindb/ntools/one-m/ansible
ansible-playbook -i hosts --private-key ~/.ssh/<key-name> one-m-node.yml
```

What did you just do? Running that playbook ensures all the software necessary for a one-machine BigchainDB node is installed, configured, and running properly. You can run that playbook on a regular schedule to ensure that the system stays properly configured. If something is okay, it does nothing; it only takes action when something is not as-desired.

Some Notes on the One-Machine Node You Just Got Running

- It ensures that the installed version of RethinkDB is the latest. You can change that by changing the installation task.
- It uses a very basic RethinkDB configuration file based on `bigchaindb/ntools/one-m/ansible/roles/rethinkdb/templates/rethinkdb.conf.j2`.
- If you edit the RethinkDB configuration file, then running the Ansible playbook will **not** restart RethinkDB for you. You must do that manually. (You can stop RethinkDB using `sudo /etc/init.d/rethinkdb stop`; run the playbook to get RethinkDB started again. This assumes you're using init.d, which is what the Ansible playbook assumes. If you want to use systemd, you'll have to edit the playbook accordingly, and stop RethinkDB using `sudo systemctl stop rethinkdb@<name_instance>`.)
- It generates and uses a default BigchainDB configuration file, which it stores in `~/.bigchaindb` (the default location).
- If you edit the BigchainDB configuration file, then running the Ansible playbook will **not** restart BigchainDB for you. You must do that manually. (You could stop it using `sudo killall -9 bigchaindb`. Run the playbook to get it started again.)

Optional: Create an Ansible Config File

The above command (`ansible-playbook -i ...`) is fairly long. You can omit the optional arguments if you put their values in an Ansible configuration file (config file) instead. There are many places where you can put a config file, but to make one specifically for the "one-m" case, you should put it in `.../bigchaindb/ntools/one-m/ansible/`. In that directory, create a file named `ansible.cfg` with the following contents:

```
[defaults]
private_key_file = $HOME/.ssh/<key-name>
inventory = hosts
```

where, as before, `<key-name>` must be replaced.

Next Steps

You could make changes to the Ansible playbook (and the resources it uses) to make the node more production-worthy. See the section on production node assumptions, components and requirements.

Azure Quickstart Template

This page outlines how to run a single BigchainDB node on the Microsoft Azure public cloud, with RethinkDB as the database backend. It uses an Azure Quickstart Template. That template is dated because we now recommend using MongoDB instead of RethinkDB. That's why we moved this page to the Appendices.

Note: There was an Azure quickstart template in the `blockchain` directory of Microsoft's `Azure/azure-quickstart-templates` repository on GitHub. It's gone now; it was replaced by the one described here.

One can deploy a BigchainDB node on Azure using the template in the `bigchaindb-on-ubuntu` directory of Microsoft's `Azure/azure-quickstart-templates` repository on GitHub. Here's how:

1. Go to that directory on GitHub.
2. Click the button labelled **Deploy to Azure**.
3. If you're not already logged in to Microsoft Azure, then you'll be prompted to login. If you don't have an account, then you'll have to create one.
4. Once you are logged in to the Microsoft Azure Portal, you should be taken to a form titled **BigchainDB**. Some notes to help with filling in that form are available below.
5. Deployment takes a few minutes. You can follow the notifications by clicking the bell icon at the top of the screen. At the time of writing, the final deployment operation (running the `init.sh` script) was failing, but a pull request (#2884) has been made to fix that and these instructions say what you can do before that pull request gets merged...
6. Find out the public IP address of the virtual machine in the Azure Portal. Example: `40.69.87.250`
7. ssh in to the virtual machine at that IP address, i.e. do `ssh <Admin_username>@<machine-ip>` where `<Admin_username>` is the admin username you entered into the form and `<machine-ip>` is the virtual machine IP address determined in the last step. Example: `ssh bcdbadmin@40.69.87.250`
8. You should be prompted for a password. Give the `<Admin_password>` you entered into the form.
9. Configure BigchainDB Server by doing:

```
bigchaindb configure rethinkdb
```

It will ask you several questions. You can press `Enter` (or `Return`) to accept the default for all of them *except for one*. When it asks **API Server bind? (default ‘localhost:9984‘):**, you should answer:

```
API Server bind? (default `localhost:9984`): 0.0.0.0:9984
```

Finally, run BigchainDB Server by doing:

```
bigchaindb start
```

BigchainDB Server should now be running on the Azure virtual machine.

Remember to shut everything down when you're done (via the Azure Portal), because it generally costs money to run stuff on Azure.

Notes on the Blockchain Template Form Fields

BASICS

Resource group - You can use an existing resource group (if you have one) or create a new one named whatever you like, but avoid using fancy characters in the name because Azure might have problems if you do.

Location is the Microsoft Azure data center where you want the BigchainDB node to run. Pick one close to where you are located.

SETTINGS

You can use whatever **Admin_username** and **Admin_password** you like (provided you don't get too fancy). It will complain if your password is too simple. You'll need these later to `ssh` into the virtual machine.

Dns_label_prefix - Once your virtual machine is deployed, it will have a public IP address and a DNS name (hostname) something like `<DNSprefix>.northeurope.cloudapp.azure.com`. The `<DNSprefix>` will be whatever you enter into this field.

Virtual_machine_size - This should be one of Azure's standard virtual machine sizes, such as `Standard_D1_v2`. There's a list of virtual machine sizes in the Azure docs.

_artifacts Location - Leave this alone.

_artifacts Location Sas Token - Leave this alone (blank).

TERMS AND CONDITIONS

Read the terms and conditions. If you agree to them, then check the checkbox.

Finally, click the button labelled **Purchase**. (Generally speaking, it costs money to run stuff on Azure.)

Generate a Key Pair for SSH

This page describes how to use `ssh-keygen` to generate a public/private RSA key pair that can be used with SSH. (Note: `ssh-keygen` is found on most Linux and Unix-like operating systems; if you're using Windows, then you'll have to use another tool, such as PuTTYgen.)

By convention, SSH key pairs get stored in the `~/.ssh/` directory. Check what keys you already have there:

```
ls -1 ~/.ssh/
```

Next, make up a new key pair name (called `<name>` below). Here are some ideas:

- `aws-bdb-2`
- `tim-bdb-azure`
- `chris-bcdb-key`

Next, generate a public/private RSA key pair with that name:

```
ssh-keygen -t rsa -C "<name>" -f ~/.ssh/<name>
```

It will ask you for a passphrase. You can use whatever passphrase you like, but don't lose it. Two keys (files) will be created in `~/.ssh/`:

1. `~/.ssh/<name>.pub` is the public key
2. `~/.ssh/<name>` is the private key

Notes for Firewall Setup

This is a page of notes on the ports potentially used by BigchainDB nodes and the traffic they should expect, to help with firewall setup (and security group setup on AWS). This page is *not* a firewall tutorial or step-by-step guide.

Expected Unsolicited Inbound Traffic

Assuming you aren't exposing the RethinkDB web interface on port 8080 (or any other port, because there are more secure ways to access it), there are only three ports that should expect unsolicited inbound traffic:

1. **Port 22** can expect inbound SSH (TCP) traffic from the node administrator (i.e. a small set of IP addresses).
2. **Port 9984** can expect inbound HTTP (TCP) traffic from BigchainDB clients sending transactions to the BigchainDB HTTP API.
3. **Port 9985** can expect inbount WebSocket traffic from BigchainDB clients.
4. If you're using RethinkDB, **Port 29015** can expect inbound TCP traffic from other RethinkDB nodes in the RethinkDB cluster (for RethinkDB intracluster communications).
5. If you're using MongoDB, **Port 27017** can expect inbound TCP traffic from other nodes.

All other ports should only get inbound traffic in response to specific requests from inside the node.

Port 22

Port 22 is the default SSH port (TCP) so you'll at least want to make it possible to SSH in from your remote machine(s).

Port 53

Port 53 is the default DNS port (UDP). It may be used, for example, by some package managers when look up the IP address associated with certain package sources.

Port 80

Port 80 is the default HTTP port (TCP). It's used by some package managers to get packages. It's *not* used by the RethinkDB web interface (see Port 8080 below) or the BigchainDB client-server HTTP API (Port 9984).

Port 123

Port 123 is the default NTP port (UDP). You should be running an NTP daemon on production BigchainDB nodes. NTP daemons must be able to send requests to external NTP servers and accept the respones.

Port 161

Port 161 is the default SNMP port (usually UDP, sometimes TCP). SNMP is used, for example, by some server monitoring systems.

Port 443

Port 443 is the default HTTPS port (TCP). You may need to open it up for outbound requests (and inbound responses) temporarily because some RethinkDB installation instructions use wget over HTTPS to get the RethinkDB GPG key. Package managers might also get some packages using HTTPS.

Port 8080

Port 8080 is the default port used by RethinkDB for its adminstrative web (HTTP) interface (TCP). While you *can*, you shouldn't allow traffic arbitrary external sources. You can still use the RethinkDB web interface by binding it to localhost and then accessing it via a SOCKS proxy or reverse proxy; see "Binding the web interface port" on the RethinkDB page about securing your cluster.

Port 9984

Port 9984 is the default port for the BigchainDB client-server HTTP API (TCP), which is served by Gunicorn HTTP Server. It's *possible* allow port 9984 to accept inbound traffic from anyone, but we recommend against doing that. Instead, set up a reverse proxy server (e.g. using Nginx) and only allow traffic from there. Information about how to do that can be found in the Gunicorn documentation. (They call it a proxy.)

If Gunicorn and the reverse proxy are running on the same server, then you'll have to tell Gunicorn to listen on some port other than 9984 (so that the reverse proxy can listen on port 9984). You can do that by setting `server.bind` to 'localhost:PORT' in the BigchainDB Configuration Settings, where PORT is whatever port you chose (e.g. 9983).

You may want to have Gunicorn and the reverse proxy running on different servers, so that both can listen on port 9984. That would also help isolate the effects of a denial-of-service attack.

Port 9985

Port 9985 is the default port for the BigchainDB WebSocket Event Stream API.

Port 28015

Port 28015 is the default port used by RethinkDB client driver connections (TCP). If your BigchainDB node is just one server, then Port 28015 only needs to listen on localhost, because all the client drivers will be running on localhost. Port 28015 doesn't need to accept inbound traffic from the outside world.

Port 29015

Port 29015 is the default port for RethinkDB intracluster connections (TCP). It should only accept incoming traffic from other RethinkDB servers in the cluster (a list of IP addresses that you should be able to find out).

Other Ports

On Linux, you can use commands such as `netstat -tunlp` or `lsof -i` to get a sense of currently open/listening ports and connections, and the associated processes.

Notes on NTP Daemon Setup

There are several NTP daemons available, including:

- The reference NTP daemon (`ntpd`) from ntp.org; see their support website
- chrony
- OpenNTPD
- Maybe NTPsec, once it's production-ready
- Maybe Ntimed, once it's production-ready
- More

We suggest you run your NTP daemon in a mode which will tell your OS kernel to handle leap seconds in a particular way: the default NTP way, so that system clock adjustments are localized and not spread out across the minutes, hours, or days surrounding leap seconds (e.g. "slewing" or "smearing"). There's a nice Red Hat Developer Blog post about the various options.

Use the default mode with `ntpd` and `chronyd`. For another NTP daemon, consult its documentation.

It's tricky to make an NTP daemon setup secure. Always install the latest version and read the documentation about how to configure and run it securely. See the notes on firewall setup.

Amazon Linux Instances

If your BigchainDB node is running on an Amazon Linux instance (i.e. a Linux instance packaged by Amazon, not Canonical, Red Hat, or someone else), then an NTP daemon should already be installed and configured. See the EC2 documentation on Setting the Time for Your Linux Instance.

That said, you should check *which* NTP daemon is installed. Is it recent? Is it configured securely?

The Ubuntu ntp Packages

The Ubuntu `ntp` packages are based on the reference implementation of NTP.

The following commands will uninstall the `ntp` and `ntpdate` packages, install the latest `ntp` package (which *might not be based on the latest ntpd code*), and start the NTP daemon (a local NTP server). (`ntpdate` is not reinstalled because it's deprecated and you shouldn't use it.)

```
sudo apt-get --purge remove ntp ntpdate
sudo apt-get autoremove
sudo apt-get update
sudo apt-get install ntp
# That should start the NTP daemon too, but just to be sure:
sudo service ntp restart
```

You can check if `ntpd` is running using `sudo ntpq -p`.

You may want to use different NTP time servers. You can change them by editing the NTP config file `/etc/ntp.conf`.

Note: A server running an NTP daemon can be used by others for DRDoS amplification attacks. The above installation procedure should install a default NTP configuration file `/etc/ntp.conf` with the lines:

```
restrict -4 default kod notrap nomodify nopeer noquery
restrict -6 default kod notrap nomodify nopeer noquery
```

Those lines should prevent the NTP daemon from being used in an attack. (The first line is for IPv4, the second for IPv6.)

There are additional things you can do to make NTP more secure. See the NTP Support Website for more details.

RethinkDB Requirements

The RethinkDB documentation should be your first source of information about its requirements. This page serves mostly to document some of its more obscure requirements.

RethinkDB Server will run on any modern OS. Note that the Fedora package isn't officially supported. Also, official support for Windows is fairly recent (April 2016).

Storage Requirements

When it comes to storage for RethinkDB, there are many things that are nice to have (e.g. SSDs, high-speed input/output [IOPS], replication, reliability, scalability, pay-for-what-you-use), but there are few *requirements* other than:

1. have enough storage to store all your data (and its replicas), and
2. make sure your storage solution (hardware and interconnects) can handle your expected read & write rates.

For RethinkDB's failover mechanisms to work, every RethinkDB table must have at least three replicas (i.e. a primary replica and two others). For example, if you want to store 10 GB of unique data, then you need at least 30 GB of storage. (Indexes and internal metadata are stored in RAM.)

As for the read & write rates, what do you expect those to be for your situation? It's not enough for the storage system alone to handle those rates: the interconnects between the nodes must also be able to handle them.

Storage Notes Specific to RethinkDB

- The RethinkDB storage engine has a number of SSD optimizations, so you *can* benefit from using SSDs. (source)
- If you have an N-node RethinkDB cluster and 1) you want to use it to store an amount of data D (unique records, before replication), 2) you want the replication factor to be R (all tables), and 3) you want N shards (all tables), then each BigchainDB node must have storage space of at least R×D/N.
- RethinkDB tables can have at most 64 shards. What does that imply? Suppose you only have one table, with 64 shards. How big could that table be? It depends on how much data can be stored in each node. If the maximum amount of data that a node can store is d, then the biggest-possible shard is d, and the biggest-possible table size is 64 times that. (All shard replicas would have to be stored on other nodes beyond the initial 64.) If there are two tables, the second table could also have 64 shards, stored on 64 other maxed-out nodes, so the total amount of unique data in the database would be (64 shards/table)×(2 tables)×d. In general, if you have T tables, the maximum amount of unique data that can be stored in the database (i.e. the amount of data before replication) is 64×T×d.
- When you set up storage for your RethinkDB data, you may have to select a filesystem. (Sometimes, the filesystem is already decided by the choice of storage.) We recommend using a filesystem that supports direct I/O (Input/Output). Many compressed or encrypted file systems don't support direct I/O. The ext4 filesystem supports direct I/O (but be careful: if you enable the data=journal mode, then direct I/O support will be disabled; the default is data=ordered). If your chosen filesystem supports direct I/O and you're using Linux, then you don't need to do anything to request or enable direct I/O. RethinkDB does that.
- RethinkDB stores its data in a specific directory. You can tell RethinkDB *which* directory using the RethinkDB config file, as explained below. In this documentation, we assume the directory is `/data`. If you set up a

separate device (partition, RAID array, or logical volume) to store the RethinkDB data, then mount that device on `/data`.

Memory (RAM) Requirements

In their FAQ, RethinkDB recommends that, "RethinkDB servers have at least 2GB of RAM..." (source)

In particular: "RethinkDB requires data structures in RAM on each server proportional to the size of the data on that server's disk, usually around 1% of the size of the total data set." (source) We asked what they meant by "total data set" and they said it's "referring to only the data stored on the particular server."

Also, "The storage engine is used in conjunction with a custom, B-Tree-aware caching engine which allows file sizes many orders of magnitude greater than the amount of available memory. RethinkDB can operate on a terabyte of data with about ten gigabytes of free RAM." (source) (In this case, it's the *cluster* which has a total of one terabyte of data, and it's the *cluster* which has a total of ten gigabytes of RAM. That is, if you add up the RethinkDB RAM on all the servers, it's ten gigabytes.)

In reponse to our questions about RAM requirements, @danielmewes (of RethinkDB) wrote:

> ... If you replicate the data, the amount of data per server increases accordingly, because multiple copies of the same data will be held by different servers in the cluster.

For example, if you increase the data replication factor from 1 to 2 (i.e. the primary plus one copy), then that will double the RAM needed for metadata. Also from @danielmewes:

> **For reasonable performance, you should probably aim at something closer to 5-10% of the data size.** [Emphasis added] The 1% is the bare minimum and doesn't include any caching. If you want to run near the minimum, you'll also need to manually lower RethinkDB's cache size through the `--cache-size` parameter to free up enough RAM for the metadata overhead...

RethinkDB has documentation about its memory requirements. You can use that page to get a better estimate of how much memory you'll need. In particular, note that RethinkDB automatically configures the cache size limit to be about half the available memory, but it can be no lower than 100 MB. As @danielmewes noted, you can manually change the cache size limit (e.g. to free up RAM for queries, metadata, or other things).

If a RethinkDB process (on a server) runs out of RAM, the operating system will start swapping RAM out to disk, slowing everything down. According to @danielmewes:

> Going into swap is usually pretty bad for RethinkDB, and RethinkDB servers that have gone into swap often become so slow that other nodes in the cluster consider them unavailable and terminate the connection to them. I recommend adjusting RethinkDB's cache size conservatively to avoid this scenario. RethinkDB will still make use of additional RAM through the operating system's block cache (though less efficiently than when it can keep data in its own cache).

Filesystem Requirements

RethinkDB "supports most commonly used file systems" (source) but it has issues with BTRFS (B-tree file system).

It's best to use a filesystem that supports direct I/O, because that will improve RethinkDB performance (if you tell RethinkDB to use direct I/O). Many compressed or encrypted filesystems don't support direct I/O.

Backing Up and Restoring Data

This page was written when BigchainDB only worked with RethinkDB, so its focus is on RethinkDB-based backup. BigchainDB now supports MongoDB as a backend database and we recommend that you use MongoDB in production.

Nevertheless, some of the following backup ideas are still relevant regardless of the backend database being used, so we moved this page to the Appendices.

RethinkDB's Replication as a form of Backup

RethinkDB already has internal replication: every document is stored on *R* different nodes, where *R* is the replication factor (set using `bigchaindb set-replicas R`). Those replicas can be thought of as "live backups" because if one node goes down, the cluster will continue to work and no data will be lost.

At this point, there should be someone saying, "But replication isn't backup¡'

It's true. Replication alone isn't enough, because something bad might happen *inside* the database, and that could affect the replicas. For example, what if someone logged in as a RethinkDB admin and did a "drop table"? We currently plan for each node to be protected by a next-generation firewall (or something similar) to prevent such things from getting very far. For example, see issue #240.

Nevertheless, you should still consider having normal, "cold" backups, because bad things can still happen.

Live Replication of RethinkDB Data Files

Each BigchainDB node stores its subset of the RethinkDB data in one directory. You could set up the node's file system so that directory lives on its own hard drive. Furthermore, you could make that hard drive part of a RAID array, so that a second hard drive would always have a copy of the original. If the original hard drive fails, then the second hard drive could take its place and the node would continue to function. Meanwhile, the original hard drive could be replaced.

That's just one possible way of setting up the file system so as to provide extra reliability.

Another way to get similar reliability would be to mount the RethinkDB data directory on an Amazon EBS volume. Each Amazon EBS volume is, "automatically replicated within its Availability Zone to protect you from component failure, offering high availability and durability."

See the section on setting up storage for RethinkDB for more details.

As with shard replication, live file-system replication protects against many failure modes, but it doesn't protect against them all. You should still consider having normal, "cold" backups.

rethinkdb dump (to a File)

RethinkDB can create an archive of all data in the cluster (or all data in specified tables), as a compressed file. According to the RethinkDB blog post when that functionality became available:

> Since the backup process is using client drivers, it automatically takes advantage of the MVCC [multiversion concurrency control] functionality built into RethinkDB. It will use some cluster resources, but will not lock out any of the clients, so you can safely run it on a live cluster.

To back up all the data in a BigchainDB cluster, the RethinkDB admin user must run a command like the following on one of the nodes:

```
rethinkdb dump -e bigchain.bigchain -e bigchain.votes
```

That should write a file named `rethinkdb_dump_<date>_<time>.tar.gz`. The `-e` option is used to specify which tables should be exported. You probably don't need to export the backlog table, but you definitely need to export the bigchain and votes tables. `bigchain.votes` means the `votes` table in the RethinkDB database named `bigchain`. It's possible that your database has a different name: the database name is a BigchainDB configuration setting. The default name is `bigchain`. (Tip: you can see the values of all configuration settings using the `bigchaindb show-config` command.)

There's more information about the `rethinkdb dump` command in the RethinkDB documentation. It also explains how to restore data to a cluster from an archive file.

Notes

- If the `rethinkdb dump` subcommand fails and the last line of the Traceback says "NameError: name 'file' is not defined", then you need to update your RethinkDB Python driver; do a `pip install --upgrade rethinkdb`
- It might take a long time to backup data this way. The more data, the longer it will take.
- You need enough free disk space to store the backup file.
- If a document changes after the backup starts but before it ends, then the changed document may not be in the final backup. This shouldn't be a problem for BigchainDB, because blocks and votes can't change anyway.
- `rethinkdb dump` saves data and secondary indexes, but does *not* save cluster metadata. You will need to recreate your cluster setup yourself after you run `rethinkdb restore`.
- RethinkDB also has subcommands to import/export collections of JSON or CSV files. While one could use those for backup/restore, it wouldn't be very practical.

Client-Side Backup

In the future, it will be possible for clients to query for the blocks containing the transactions they care about, and for the votes on those blocks. They could save a local copy of those blocks and votes.

How could we be sure blocks and votes from a client are valid?

All blocks and votes are signed by cluster nodes (owned and operated by consortium members). Only cluster nodes can produce valid signatures because only cluster nodes have the necessary private keys. A client can't produce a valid signature for a block or vote.

Could we restore an entire BigchainDB database using client-saved blocks and votes?

Yes, in principle, but it would be difficult to know if you've recovered every block and vote. Votes link to the block they're voting on and to the previous block, so one could detect some missing blocks. It would be difficult to know if you've recovered all the votes.

Backup by Copying RethinkDB Data Files

It's *possible* to back up a BigchainDB database by creating a point-in-time copy of the RethinkDB data files (on all nodes, at roughly the same time). It's not a very practical approach to backup: the resulting set of files will be much larger (collectively) than what one would get using `rethinkdb dump`, and there are no guarantees on how consistent that data will be, especially for recently-written data.

If you're curious about what's involved, see the MongoDB documentation about "Backup by Copying Underlying Data Files". (Yes, that's documentation for MongoDB, but the principles are the same.)

See the last subsection of this page for a better way to use this idea.

Incremental or Continuous Backup

Incremental backup is where backup happens on a regular basis (e.g. daily), and each one only records the changes since the last backup.

Continuous backup might mean incremental backup on a very regular basis (e.g. every ten minutes), or it might mean backup of every database operation as it happens. The latter is also called transaction logging or continuous archiving.

At the time of writing, RethinkDB didn't have a built-in incremental or continuous backup capability, but the idea was raised in RethinkDB issues #89 and #5890. On July 5, 2016, Daniel Mewes (of RethinkDB) wrote the following comment on issue #5890: "We would like to add this feature [continuous backup], but haven't started working on it yet."

To get a sense of what continuous backup might look like for RethinkDB, one can look at the continuous backup options available for MongoDB. MongoDB, the company, offers continuous backup with Ops Manager (self-hosted) or Cloud Manager (fully managed). Features include:

- It "continuously maintains backups, so if your MongoDB deployment experiences a failure, the most recent backup is only moments behind..."
- It "offers point-in-time backups of replica sets and cluster-wide snapshots of sharded clusters. You can restore to precisely the moment you need, quickly and safely."
- "You can rebuild entire running clusters, just from your backups."
- It enables, "fast and seamless provisioning of new dev and test environments."

The MongoDB documentation has more details about how Ops Manager Backup works.

Considerations for BigchainDB:

- We'd like the cost of backup to be low. To get a sense of the cost, MongoDB Cloud Manager backup costed $30 / GB / year prepaid. One thousand gigabytes backed up (i.e. about a terabyte) would cost 30 thousand US dollars per year. (That's just for the backup; there's also a cost per server per year.)
- We'd like the backup to be decentralized, with no single point of control or single point of failure. (Note: some file systems have a single point of failure. For example, HDFS has one Namenode.)
- We only care to back up blocks and votes, and once written, those never change. There are no updates or deletes, just new blocks and votes.

Combining RethinkDB Replication with Storage Snapshots

Although it's not advertised as such, RethinkDB's built-in replication feature is similar to continous backup, except the "backup" (i.e. the set of replica shards) is spread across all the nodes. One could take that idea a bit farther by creating a set of backup-only servers with one full backup:

- Give all the original BigchainDB nodes (RethinkDB nodes) the server tag `original`. This is the default if you used the RethinkDB config file suggested in the section titled Configure RethinkDB Server.
- Set up a group of servers running RethinkDB only, and give them the server tag `backup`. The `backup` servers could be geographically separated from all the `original` nodes (or not; it's up to the consortium to decide).
- Clients shouldn't be able to read from or write to servers in the `backup` set.
- Send a RethinkDB reconfigure command to the RethinkDB cluster to make it so that the `original` set has the same number of replicas as before (or maybe one less), and the `backup` set has one replica. Also, make sure the `primary_replica_tag='original'` so that all primary shards live on the `original` nodes.

The RethinkDB documentation on sharding and replication has the details of how to set server tags and do RethinkDB reconfiguration.

Once you've set up a set of backup-only RethinkDB servers, you could make a point-in-time snapshot of their storage devices, as a form of backup.

You might want to disconnect the `backup` set from the `original` set first, and then wait for reads and writes in the `backup` set to stop. (The `backup` set should have only one copy of each shard, so there's no opportunity for inconsistency between shards of the `backup` set.)

You will want to re-connect the `backup` set to the `original` set as soon as possible, so it's able to catch up.

If something bad happens to the entire original BigchainDB cluster (including the `backup` set) and you need to restore it from a snapshot, you can, but before you make BigchainDB live, you should 1) delete all entries in the backlog table, 2) delete all blocks after the last voted-valid block, 3) delete all votes on the blocks deleted in part 2, and 4) rebuild the RethinkDB indexes.

NOTE: Sometimes snapshots are *incremental*. For example, Amazon EBS snapshots are incremental, meaning "only the blocks on the device that have changed after your most recent snapshot are saved. **This minimizes the time required to create the snapshot and saves on storage costs.**" [Emphasis added]

Licenses

Information about how the BigchainDB Server code and documentation are licensed can be found in the LICENSES.md file of the bigchaindb/bigchaindb repository on GitHub.

Installing BigchainDB on LXC containers using LXD

Note: This page was contributed by an external contributor and is not actively maintained. We include it in case someone is interested.

You can visit this link to install LXD (instructions here): LXD Install

(assumption is that you are using Ubuntu 14.04 for host/container)

Let us create an LXC container (via LXD) with the following command:

`lxc launch ubuntu:14.04 bigchaindb`

(ubuntu:14.04 - this is the remote server the command fetches the image from) (bigchaindb - is the name of the container)

Below is the `install.sh` script you will need to install BigchainDB within your container.

Here is my `install.sh`:

```
#!/bin/bash
set -ex
export DEBIAN_FRONTEND=noninteractive
apt-get install -y wget
source /etc/lsb-release && echo "deb http://download.rethinkdb.com/apt $DISTRIB_
↪CODENAME main" | sudo tee /etc/apt/sources.list.d/rethinkdb.list
wget -qO- https://download.rethinkdb.com/apt/pubkey.gpg | sudo apt-key add -
apt-get update
apt-get install -y rethinkdb python3-pip
pip3 install --upgrade pip wheel setuptools
pip install ptpython bigchaindb
```

Copy/Paste the above `install.sh` into the directory/path you are going to execute your LXD commands from (ie. the host).

Make sure your container is running by typing:

```
lxc list
```

Now, from the host (and the correct directory) where you saved `install.sh`, run this command:

```
cat install.sh | lxc exec bigchaindb /bin/bash
```

If you followed the commands correctly, you will have successfully created an LXC container (using LXD) that can get you up and running with BigchainDB in <5 minutes (depending on how long it takes to download all the packages).

Python Module Index

HTTP Routing Table

/api

Index