Contributing to BigchainDB

BigchainDB Contributors

Dec 02, 2018

Contents

There are many ways you can contribute to BigchainDB. It includes several sub-projects.

- BigchainDB Server
- BigchainDB Python Driver
- BigchainDB JavaScript Driver
- BigchainDB Java Driver
- cryptoconditions (a Python package by us)
- py-abci (a Python package we use)
- BigchainDB Enhancement Proposals (BEPs)

CHAPTER 1

Contents

1.1 Ways to Contribute

1.1.1 Report a Bug

To report a bug, go to the relevant GitHub repository, click on the **Issues** tab, click on the **New issue** button, and read the instructions.

1.1.2 Write an Issue

To write an issue, go to the relevant GitHub repository, click on the **Issues** tab, click on the **New issue** button, and read the instructions.

1.1.3 Make a Feature Request or Proposal

To make a feature request or proposal, write a BigchainDB Enhancement Proposal (BEP).

1.1.4 Write a BigchainDB Enhancement Proposal (BEP)

If you have an idea for a new feature or enhancement, and you want some feedback before you write a full BigchainDB Enhancement Proposal (BEP), then feel free to:

- ask in the bigchaindb/bigchaindb Gitter chat room or
- open a new issue in the bigchaindb/BEPs repo and give it the label **BEP idea**.

If you want to discuss an existing BEP, then open a new issue in the bigchaindb/BEPs repo and give it the label **discuss existing BEP**.

Steps to Write a New BEP

1. Look at the structure of existing BEPs in the bigchaindb/BEPs repo. Note the section headings. BEP-2 (our variant of the consensus-oriented specification system [COSS]) says more about the expected structure and process.
2. Write a first draft of your BEP. It doesn't have to be long or perfect.
3. Push your BEP draft to the bigchaindb/BEPs repo and make a pull request. BEP-1 (our variant of C4) outlines the process we use to handle all pull requests. In particular, we try to merge all pull requests quickly.
4. Your BEP can be revised by pushing more pull requests.

1.1.5 Write Docs

If you're writing code, you should also update any related docs. However, you might want to write docs only, such as:

- General explainers
- Tutorials
- Courses
- Code explanations
- How BigchainDB relates to other blockchain things
- News from recent events

You can certainly do that!

- The docs for BigchainDB Server live under `bigchaindb/docs/` in the `bigchaindb/bigchaindb` repo.
- There are docs for the Python driver under `bigchaindb-driver/docs/` in the `bigchaindb/bigchaindb-driver` repo.
- There are docs for the JavaScript driver under `bigchaindb/js-bigchaindb-driver` in the `bigchaindb/js-bigchaindb-driver` repo.
- The source code for the BigchainDB website is in a private repo, but we can give you access if you ask.

The BigchainDB Transactions Specs (one for each spec version) are in the `bigchaindb/BEPs` repo.

You can write the docs using Markdown (MD) or RestructuredText (RST). Sphinx can understand both. RST is more powerful.

ReadTheDocs will automatically rebuild the docs whenever a commit happens on the `master` branch, or on one of the other branches that it is monitoring.

1.1.6 Answer Questions

People ask questions about BigchainDB in the following places:

- Gitter
  - bigchaindb/bigchaindb
  - bigchaindb/js-bigchaindb-driver
- Twitter
- Stack Overflow "bigchaindb"

Feel free to hang out and answer some questions. People will be thankful.

1.2 Developer Setup, Coding & Contribution Process

1.2.1 Write Code

Know What You Want to Write Code to Do

Do you want to write code to resolve an open issue (bug)? Which one?

Do you want to implement a BigchainDB Enhancement Proposal (BEP)? Which one?

You should know why you want to write code before you go any farther.

Refresh Yourself about the C4 Process

C4 is the Collective Code Construction Contract. It's quite short: re-reading it will only take a few minutes.

Set Up Your Local Machine. Here's How.

- Make sure you have Git installed.
- Get a text editor. Internally, we like:
  - Vim
  - PyCharm
  - Visual Studio Code
  - Atom
  - GNU Emacs (Trent is crazy)
  - GNU nano (Troy has lost his mind)
- If you plan to do JavaScript coding, get the latest JavaScript stuff (npm etc.).
- If you plan to do Python coding, get the latest Python, and get the latest `pip`.

Warning: Don't use apt or apt-get to get the latest `pip`. It won't work properly. Use `get-pip.py` from the pip website.

- Use the latest `pip` to get the latest `virtualenv`:

```
$ pip install virtualenv
```

- Create a Python Virttual Environment (virtualenv) for doing BigchainDB Server development. There are many ways to do that. Google around and pick one. An old-fashioned but fine way is:

```
$ virtualenv -p $(which python3.6) NEW_ENV_NAME
$ . NEW_ENV_NAME/bin/activate
```

  Be sure to use Python 3.6.x as the Python version for your virtualenv. The virtualenv creation process will actually get the latest `pip`, `wheel` and `setuptools` and put them inside the new virtualenv.

Start Writing Code

Use the Git Fork and Pull Request Workflow. Tip: You could print that page for reference.

Your Python code should follow our Python Style Guide. Similarly for JavaScript.

Make sure pre-commit actually checks commits. Do:

```
$ pip install pre-commit  # might not do anything if it is already installed,
↪ which is okay
$ pre-commit install
```

That will load the pre-commit settings in the file `.pre-commit-config.yaml`. Now every time you do `git commit <stuff>`, pre-commit will run all those checks.

To install BigchainDB Server from the local code, and to keep it up to date with the latest code in there, use:

```
$ pip install -e .[dev]
```

The `-e` tells it to use the latest code. The `.` means use the current directory, which should be the one containing `setup.py`. The `[dev]` tells it to install some extra Python packages. Which ones? Open `setup.py` and look for `dev` in the `extras_require` section.

Remember to Write Tests

We like to test everything, if possible. Unit tests and also integration tests. We use the pytest framework to write Python tests. Read all about it.

Most tests are in the `tests/` folder. Take a look around.

Running a Local Node/Network for Dev and Test

This is tricky and personal. Different people do it different ways. We documented some of those on separate pages:

- Dev node setup and running all tests with Docker Compose
- Dev node setup and running all tests as processes
- Dev network setup with stack.sh
- Dev network setup with Ansible
- More to come?

Create the PR on GitHub

Git push your branch to GitHub so as to create a pull request against the branch where the code you want to change *lives*.

Travis and Codecov will run and might complain. Look into the complaints and fix them before merging. Travis gets its configuration and setup from the files:

- Some environment variables, if they are used. See https://docs.travis-ci.com/user/environment-variables/
- `.travis.yml`
- `tox.ini` - What is tox? See tox.readthedocs.io
- `.ci/` (as in Travis CI = Continuous Integration)

  - `travis-after-success.sh`
  - `travis-before-install.sh`
  - `travis-before-script.sh`
  - `travis-install.sh`
  - `travis_script.sh`

Read about the Travis CI build lifecycle to understand when those scripts run and what they do. You can have even more scripts!

Codecov gets its configuration from the file codeocov.yaml which is also documented at docs.codecov.io. Codecov might also use `setup.cfg`.

First-Time Pull Requests from External Users

First-time pull requests from external users who haven't contributed before will get blocked by the requirement to agree to the BigchainDB Contributor License Agreement (CLA). It doesn't take long to agree to it. Go to https://www.bigchaindb.com/cla/ and:

- Select the CLA you want to agree to (for individuals or for a whole company)
- Fill in the form and submit it
- Wait for an email from us with the next step. There is only one: copying a special block of text to GitHub.

Merge!

Ideally, we like your PR and merge it right away. We don't want to keep you waiting.

If we want to make changes, we'll do them in a follow-up PR.

You are awesome. Do you want a job? Apply! Berlin is great. If you got this far, we'd be happy to consider you joining our team. Look at these Unsplash photos of Berlin. So nice.

1.2.2 Notes on Running a Local Dev Node with Docker Compose

Setting up a single node development environment with `docker-compose`

Using the BigchainDB 2.0 developer toolbox

We grouped all useful commands under a simple `Makefile`.

Run a BigchainDB node in the foreground:

```
$ make run
```

There are also other commands you can execute:

- `make start`: Run BigchainDB from source and daemonize it (stop it with `make stop`).
- `make stop`: Stop BigchainDB.
- `make logs`: Attach to the logs.
- `make test`: Run all unit and acceptance tests.

- `make test-unit-watch`: Run all tests and wait. Every time you change code, tests will be run again.
- `make cov`: Check code coverage and open the result in the browser.
- `make doc`: Generate HTML documentation and open it in the browser.
- `make clean`: Remove all build, test, coverage and Python artifacts.
- `make reset`: Stop and REMOVE all containers. WARNING: you will LOSE all data stored in BigchainDB.

Using `docker-compose` directly

The BigchainDB `Makefile` is a wrapper around some `docker-compose` commands we use frequently. If you need a finer granularity to manage the containers, you can still use `docker-compose` directly. This part of the documentation explains how to do that.

```
$ docker-compose build bigchaindb
$ docker-compose up -d bdb
```

The above command will launch all 3 main required services/processes:

- `mongodb`
- `tendermint`
- `bigchaindb`

To follow the logs of the `tendermint` service:

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

To follow the logs of the `bigchaindb` service:

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

To follow the logs of the `mongodb` service:

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

Simple health check:

```
$ docker-compose up curl-client
```

Post and retrieve a transaction – copy/paste a driver basic example of a `CREATE` transaction:

```
$ docker-compose -f docker-compose.yml run --rm bdb-driver ipython
```

TODO: A python script to post and retrieve a transaction(s).

Running Tests

Run all the tests using:

```
$ docker-compose run --rm --no-deps bigchaindb pytest -v
```

Run tests from a file:

```
$ docker-compose run --rm --no-deps bigchaindb pytest /path/to/file -v
```

Run specific tests:

```
$ docker-compose run --rm --no-deps bigchaindb pytest /path/to/file -k "<test_name>" -
↪v
```

Building Docs

You can also develop and build the BigchainDB docs using `docker-compose`:

```
$ docker-compose build bdocs
$ docker-compose up -d bdocs
```

The docs will be hosted on port **33333**, and can be accessed over localhost, 127.0.0.1 OR http:/HOST_IP:33333.

1.2.3 Notes on Running a Local Dev Node as Processes

The following doc describes how to run a local node for developing BigchainDB Tendermint version.

There are two crucial dependencies required to start a local node:

- MongoDB
- Tendermint

and of course you also need to install BigchainDB Sever from the local code you just developed.

Install and Run MongoDB

MongoDB can be easily installed, just refer to their installation documentation for your distro. We know MongoDB 3.4 and 3.6 work with BigchainDB. After the installation of MongoDB is complete, run MongoDB using `sudo mongod`

Install and Run Tendermint

Installing a Tendermint Executable

The version of BigchainDB Server described in these docs only works well with Tendermint 0.22.8 (not a higher version number). Install that:

```
$ sudo apt install -y unzip
$ wget https://github.com/tendermint/tendermint/releases/download/v0.22.8/tendermint_
↪0.22.8_linux_amd64.zip
$ unzip tendermint_0.22.8_linux_amd64.zip
$ rm tendermint_0.22.8_linux_amd64.zip
$ sudo mv tendermint /usr/local/bin
```

Installing Tendermint Using Docker

Tendermint can be run directly using the docker image. Refer here for more details.

Installing Tendermint from Source

Before we can begin installing Tendermint one should ensure that the Golang is installed on system and `$GOPATH` should be set in the `.bashrc` or `.zshrc`. An example setup is shown below

```
$ echo $GOPATH
/home/user/Documents/go
$ go -h
Go is a tool for managing Go source code.

Usage:

    go command [arguments]

The commands are:

    build       compile packages and dependencies
    clean       remove object files

...
```

- We can drop `GOPATH` in `PATH` so that installed Golang packages are directly available in the shell. To do that add the following to your `.bashrc`

```
export PATH=${PATH}:${GOPATH}/bin
```

Follow the Tendermint docs to install Tendermint from source.

If the installation is successful then Tendermint is installed at `$GOPATH/bin`. To ensure Tendermint's installed fine execute the following command,

```
$ tendermint -h
Tendermint Core (BFT Consensus) in Go

Usage:
  tendermint [command]

Available Commands:
  gen_validator               Generate new validator keypair
  help                        Help about any command
  init                        Initialize Tendermint
...
```

Running Tendermint

- We can initialize and run tendermint as follows,

```
$ tendermint init
...

$ tendermint node --consensus.create_empty_blocks=false
```

The argument `--consensus.create_empty_blocks=false` specifies that Tendermint should not commit empty blocks.

- To reset all the data stored in Tendermint execute the following command,

```
$ tendermint unsafe_reset_all
```

Install BigchainDB

To install BigchainDB from source (for dev), clone the repo and execute the following command, (it is better that you create a virtual env for this)

```
$ git clone https://github.com/bigchaindb/bigchaindb.git
$ cd bigchaindb
$ pip install -e .[dev]  #  or  pip install -e '.[dev]'  # for zsh
```

Running All Tests

To execute tests when developing a feature or fixing a bug one could use the following command,

```
$ pytest -v
```

NOTE: MongoDB and Tendermint should be running as discussed above.

One could mark a specific test and execute the same by appending `-m my_mark` to the above command.

Although the above should prove sufficient in most cases but in case tests are failing on Travis CI then the following command can be used to possibly replicate the failure locally,

```
$ docker-compose run --rm --no-deps bdb pytest -v --cov=bigchaindb
```

NOTE: before executing the above command the user must ensure that they reset the Tendermint container by executing `tendermint usafe_reset_all` command in the Tendermint container.

1.2.4 Run a BigchainDB network

NOT for Production Use

You can use the following instructions to deploy a single or multi node BigchainDB network for dev/test using the extensible `stack` script(s).

Currently, this workflow is only supported for the following Operating systems:

- Ubuntu >= 16.04
- CentOS >= 7
- Fedora >= 24
- MacOSX

Minimum Requirements

Minimum resource requirements for a single node BigchainDB dev setup. **The more the better**:

- Memory >= 512MB
- VCPUs >= 1

Download the scripts

Note: If you're working on BigchainDB Server code, on a branch based on recent code, then you already have local recent versions of *stack.sh* and *unstack.sh* in your bigchaindb/pkg/scripts/ directory. Otherwise you can get them using:

```
$ export GIT_BRANCH=master
$ curl -fOL https://raw.githubusercontent.com/bigchaindb/bigchaindb/${GIT_BRANCH}/pkg/
↪scripts/stack.sh

# Optional
$ curl -fOL https://raw.githubusercontent.com/bigchaindb/bigchaindb/${GIT_BRANCH}/pkg/
↪scripts/unstack.sh
```

Quick Start

If you run `stack.sh` out of the box i.e. without any configuration changes, you will be able to deploy a 4 node BigchainDB network with Docker containers, created from `master` branch of `bigchaindb/bigchaindb` repo and Tendermint version `0.22.8`.

Note: Run `stack.sh` with either root or non-root user with sudo enabled.

```
$ bash stack.sh
...Logs..
.........
.........
Finished stacking!
```

Configure the BigchainDB network

The `stack.sh` script has multiple deployment methods and parameters and they can be explored using: `bash stack.sh -h`

```
$ bash stack.sh -h

    Usage: $ bash stack.sh [-h]

    Deploys the BigchainDB network.

    ENV[STACK_SIZE]
        Set STACK_SIZE environment variable to the size of the network you desire.
        Network mimics a production network environment with single or multiple BDB
        nodes. (default: 4).

    ENV[STACK_TYPE]
        Set STACK_TYPE environment variable to the type of deployment you desire.
        You can set it one of the following: ["docker", "local", "cloud"].
        (default: docker)

    ENV[STACK_TYPE_PROVIDER]
        Set only when STACK_TYPE="cloud". Only "azure" is supported.
        (default: )

    ENV[STACK_VM_MEMORY]
        (Optional) Set only when STACK_TYPE="local". This sets the memory
```

(continues on next page)

(continued from previous page)

```
        of the instance(s) spawned. (default: 2048)

    ENV[STACK_VM_CPUS]
        (Optional) Set only when STACK_TYPE="local". This sets the number of VCPUs
        of the instance(s) spawned. (default: 2)

    ENV[STACK_BOX_NAME]
        (Optional) Set only when STACK_TYPE="local". This sets the box Vagrant box␣
↪name
        of the instance(s) spawned. (default: ubuntu/xenial64)

    ENV[STACK_REPO]
        (Optional) To configure bigchaindb repo to use, set STACK_REPO environment
        variable. (default: bigchaindb/bigchaindb)

    ENV[STACK_BRANCH]
        (Optional) To configure bigchaindb repo branch to use set STACK_BRANCH␣
↪environment
        variable. (default: master)

    ENV[TM_VERSION]
        (Optional) Tendermint version to use for the setup. (default: 0.22.8)

    ENV[MONGO_VERSION]
        (Optional) MongoDB version to use with the setup. (default: 3.6)

    ENV[AZURE_CLIENT_ID]
        Only required when STACK_TYPE="cloud" and STACK_TYPE_PROVIDER="azure". Steps␣
↪to generate:
        https://github.com/Azure/vagrant-azure#create-an-azure-active-directory-aad-
↪application

    ENV[AZURE_TENANT_ID]
        Only required when STACK_TYPE="cloud" and STACK_TYPE_PROVIDER="azure". Steps␣
↪to generate:
        https://github.com/Azure/vagrant-azure#create-an-azure-active-directory-aad-
↪application

    ENV[AZURE_SUBSCRIPTION_ID]
        Only required when STACK_TYPE="cloud" and STACK_TYPE_PROVIDER="azure". Steps␣
↪to generate:
        https://github.com/Azure/vagrant-azure#create-an-azure-active-directory-aad-
↪application

    ENV[AZURE_CLIENT_SECRET]
        Only required when STACK_TYPE="cloud" and STACK_TYPE_PROVIDER="azure". Steps␣
↪to generate:
        https://github.com/Azure/vagrant-azure#create-an-azure-active-directory-aad-
↪application

    ENV[AZURE_REGION]
        (Optional) Only applicable, when STACK_TYPE="cloud" and STACK_TYPE_PROVIDER=
↪"azure".
        Azure region for the BigchainDB instance. Get list of regions using Azure CLI.
        e.g. az account list-locations. (default: westeurope)

    ENV[AZURE_IMAGE_URN]
```

(continues on next page)

(continued from previous page)

```
        (Optional) Only applicable, when STACK_TYPE="cloud" and STACK_TYPE_PROVIDER=
↪"azure".
        Azure image to use. Get list of available images using Azure CLI.
        e.g. az vm image list --output table. (default: Canonical:UbuntuServer:16.04-
↪LTS:latest)

    ENV[AZURE_RESOURCE_GROUP]
        (Optional) Only applicable, when STACK_TYPE="cloud" and STACK_TYPE_PROVIDER=
↪"azure".
        Name of Azure resource group for the instance.
        (default: bdb-vagrant-rg-2018-05-30)

    ENV[AZURE_DNS_PREFIX]
        (Optional) Only applicable, when STACK_TYPE="cloud" and STACK_TYPE_PROVIDER=
↪"azure".
        DNS Prefix of the instance. (default: bdb-instance-2018-05-30)

    ENV[AZURE_ADMIN_USERNAME]
        (Optional) Only applicable, when STACK_TYPE="cloud" and STACK_TYPE_PROVIDER=
↪"azure".
        Admin username of the the instance. (default: vagrant)

    ENV[AZURE_VM_SIZE]
        (Optional) Only applicable, when STACK_TYPE="cloud" and STACK_TYPE_PROVIDER=
↪"azure".
        Azure VM size. (default: Standard_D2_v2)

    ENV[SSH_PRIVATE_KEY_PATH]
        Only required when STACK_TYPE="cloud" and STACK_TYPE_PROVIDER="azure".␣
↪Absolute path of
        SSH keypair required to log into the Azure instance.

    -h
        Show this help and exit.
```

The parameter that differentiates between the deployment type is `STACK_TYPE` which currently, supports an opinionated deployment of BigchainDB on `docker`, `local` and `cloud`.

STACK_TYPE: docker

This configuration deploys a docker based BigchainDB network on the dev/test machine that you are running `stack.sh` on. This is also the default `STACK_TYPE` config for `stack.sh`.

Example

Deploy a 4 node docker based BigchainDB network on your host.

```
#Optional, since 4 is the default size.
$ export STACK_SIZE=4

#Optional, since docker is the default type.
$ export STACK_TYPE=docker

```

(continues on next page)

(continued from previous page)

```
#Optional, repo to use for the network deployment
# Default: bigchaindb/bigchaindb
$ export STACK_REPO=bigchaindb/bigchaindb

#Optional, codebase to use for the network deployment
# Default: master
$ export STACK_BRANCH=master

#Optional, since 0.22.8 is the default tendermint version.
$ export TM_VERSION=0.22.8

#Optional, since 3.6 is the default MongoDB version.
$ export MONGO_VERSION=3.6

$ bash stack.sh
```

Note: For MacOSX users, the script will not install `Docker for Mac`, it only detects if `docker` is installed on the system, if not make sure to install Docker for Mac. Also make sure Docker API Version > 1.25. To check Docker API Version:

`docker version --format '{{.Server.APIVersion}}'`

STACK_TYPE: local

This configuration deploys a VM based BigchainDB network on your host/dev. All the services are running as processes on the VMs. For `local` deployments the following dependencies must be installed i.e.

- Vagrant
  - Vagrant plugins.
    - vagrant-cachier
    - vagrant-vbguest
    - vagrant-hosts
    - vagrant-azure
      - `vagrant plugin install vagrant-cachier vagrant-vbguest vagrant-hosts vagrant-azure`
- Virtualbox

Example

Deploy a 4 node VM based BigchainDB network.

```
$ export STACK_TYPE=local

# Optional, since 4 is the default size.
$ export STACK_SIZE=4

# Optional, default is 2048
$ export STACK_VM_MEMORY=2048

#Optional, default is 1
```

(continues on next page)

(continued from previous page)

```
$ export STACK_VM_CPUS=1

#Optional, default is ubuntu/xenial64. Supported/tested images: bento/centos-7,␣
↪fedora/25-cloud-base
$ export STACK_BOX_NAME=ubuntu/xenial64

#Optional, repo to use for the network deployment
# Default: bigchaindb/bigchaindb
$ export STACK_REPO=bigchaindb/bigchaindb

#Optional, codebase to use for the network deployment
# Default: master
$ export STACK_BRANCH=master

#Optional, since 0.22.8 is the default tendermint version
$ export TM_VERSION=0.22.8

#Optional, since 3.6 is the default MongoDB version.
$ export MONGO_VERSION=3.6

$ bash stack.sh
```

STACK_TYPE: cloud

This configuration deploys a docker based BigchainDB network on a cloud instance. Currently, only Azure is supported. For `cloud` deployments the following dependencies must be installed i.e.

- Vagrant
  - Vagrant plugins.
    - vagrant-cachier
    - vagrant-vbguest
    - vagrant-hosts
    - vagrant-azure
      - `vagrant plugin install vagrant-cachier vagrant-vbguest vagrant-hosts vagrant-azure`

Example: stack

Deploy a 4 node docker based BigchainDB network on an Azure instance.

- Create an Azure Active Directory(AAD) Application
- Generate or specify an SSH keypair to login to the Azure instance.
  - **Example:**

```
$ ssh-keygen -t rsa -C "<name>" -f /path/to/key/<name>
```

- Configure parameters for `stack.sh`

```
# After creating the AAD application with access to Azure Resource
# Group Manager for your subscription, it will return a JSON object

$ export AZURE_CLIENT_ID=<value from azure.appId>

$ export AZURE_TENANT_ID=<value from azure.tenant>

# Can be retrieved via
# az account list --query "[?isDefault].id" -o tsv
$ export AZURE_SUBSCRIPTION_ID=<your Azure subscription ID>

$ export AZURE_CLIENT_SECRET=<value from azure.password>

$ export STACK_TYPE=cloud

# Currently on azure is supported
$ export STACK_TYPE_PROVIDER=azure

$ export SSH_PRIVATE_KEY_PATH=</path/to/private/key>

# Optional, Azure region of the instance. Default: westeurope
$ export AZURE_REGION=westeurope

# Optional, Azure image urn of the instance. Default: Canonical:UbuntuServer:16.04-
↪LTS:latest
$ export AZURE_IMAGE_URN=Canonical:UbuntuServer:16.04-LTS:latest

# Optional, Azure resource group. Default: bdb-vagrant-rg-yyyy-mm-dd(current date)
$ export AZURE_RESOURCE_GROUP=bdb-vagrant-rg-2018-01-01

# Optional, DNS prefix of the Azure instance. Default: bdb-instance-yyyy-mm-
↪dd(current date)
$ export AZURE_DNS_PREFIX=bdb-instance-2018-01-01

# Optional, Admin username of the Azure instance. Default: vagrant
$ export AZURE_ADMIN_USERNAME=vagrant

# Optional, Azure instance size. Default: Standard_D2_v2
$ export AZURE_VM_SIZE=Standard_D2_v2

$ bash stack.sh
```

Delete/Unstack a BigchainDB network

Export all the variables exported for the corresponding `stack.sh` script and run `unstack.sh` to delete/remove/unstack the BigchainDB network/stack.

```
$ bash unstack.sh

OR

# -s implies soft unstack. i.e. Only applicable for local and cloud based
# networks. Only deletes/stops the docker(s)/process(es) and does not
# delete the instances created via Vagrant or on Cloud. Default: hard
$ bash unstack.sh -s
```

1.2.5 Run a BigchainDB network with Ansible

NOT for Production Use

You can use the following instructions to deploy a single or multi node BigchainDB network for dev/test using Ansible. Ansible will configure the BigchainDB node(s).

Currently, this workflow is only supported for the following distributions:

- Ubuntu >= 16.04
- CentOS >= 7
- Fedora >= 24
- MacOSX

Minimum Requirements

Minimum resource requirements for a single node BigchainDB dev setup. **The more the better**:

- Memory >= 512MB
- VCPUs >= 1

Clone the BigchainDB repository

```
$ git clone https://github.com/bigchaindb/bigchaindb.git
```

Install dependencies

- Ansible

You can also install `ansible` and other dependencies, if any, using the `boostrap.sh` script inside the BigchainDB repository. Navigate to `bigchaindb/pkg/scripts` and run the `bootstrap.sh` script to install the dependencies for your OS. The script also checks if the OS you are running is compatible with the supported versions.

Note: `bootstrap.sh` only supports Ubuntu >= 16.04, CentOS >= 7 and Fedora >=24 and MacOSX.

```
$ cd bigchaindb/pkg/scripts/
$ bash bootstrap.sh --operation install
```

BigchainDB Setup Configuration(s)

Local Setup

You can run the Ansible playbook `bigchaindb-start.yml` on your local dev machine and set up the BigchainDB node where BigchainDB can be run as a process or inside a Docker container(s) depending on your configuration.

Before, running the playbook locally, you need to update the `hosts` and `stack-config.yml` configuration, which will notify Ansible that we need to run the play locally.

Update Hosts

Navigate to `bigchaindb/pkg/configuration/hosts` inside the BigchainDB repository.

```
$ cd bigchaindb/pkg/configuration/hosts
```

Edit `all` configuration file:

```
# Delete any existing configuration in this file and insert
# Hostname of dev machine
<HOSTNAME> ansible_connection=local
```

Update Configuration

Navigate to `bigchaindb/pkg/configuration/vars` inside the BigchainDB repository.

```
$ cd bigchaindb/pkg/configuration/vars/stack-config.yml
```

Edit `bdb-config.yml` configuration file as per your requirements, sample configuration file(s):

```
---
stack_type: "docker"
stack_size: "4"


OR

---
stack_type: "local"
stack_type: "1"
```

BigchainDB Setup

Now, You can safely run the `bigchaindb-start.yml` playbook and everything will be taken care of by `Ansible`. To run the playbook please navigate to the `bigchaindb/pkg/configuration` directory inside the BigchainDB repository and run the `bigchaindb-start.yml` playbook.

```
$ cd bigchaindb/pkg/configuration/

$ ansible-playbook bigchaindb-start.yml -i hosts/all --extra-vars "operation=start␣
↪home_path=$(pwd)"
```

After successful execution of the playbook, you can verify that BigchainDB docker(s)/process(es) is(are) running.

Verify BigchainDB process(es):

```
$ ps -ef | grep bigchaindb
```

OR

Verify BigchainDB Docker(s):

```
$ docker ps | grep bigchaindb
```

You can now send transactions and verify the functionality of your BigchainDB node. See the BigchainDB Python Driver documentation for details on how to use it.

Note: The `bdb_root_url` can be be one of the following:

```
# BigchainDB is running as a process
bdb_root_url = http://<HOST-IP>:9984

OR

# BigchainDB is running inside a docker container
bdb_root_url = http://<HOST-IP>:<DOCKER-PUBLISHED-PORT>
```

Note: BigchainDB has other drivers as well.

Experimental: Running Ansible a Remote Dev/Host

Remote Setup

You can also run the Ansible playbook `bigchaindb-start.yml` on remote machine(s) and set up the BigchainDB node where BigchainDB can run as a process or inside a Docker container(s) depending on your configuration.

Before, running the playbook on a remote host, you need to update the `hosts` and `stack-config.yml` configuration, which will notify Ansible that we need to run the play on a remote host.

Update Hosts

Navigate to `bigchaindb/pkg/configuration/hosts` inside the BigchainDB repository.

```
$ cd bigchaindb/pkg/configuration/hosts
```

Edit `all` configuration file:

```
# Delete any existing configuration in this file and insert
<Remote_Host_IP/Hostname> ansible_ssh_user=<USERNAME> ansible_sudo_pass=<PASSWORD>
```

Note: You can add multiple hosts to the `all` configuration file. Non-root user with sudo enabled password is needed because ansible will run some tasks that require those permissions.

Note: You can also use other methods to get inside the remote machines instead of password based SSH. For other methods please consult Ansible Documentation.

Update Configuration

Navigate to `bigchaindb/pkg/configuration/vars` inside the BigchainDB repository.

```
$ cd bigchaindb/pkg/configuration/vars/stack-config.yml
```

Edit `stack-config.yml` configuration file as per your requirements, sample configuration file(s):

```
---
stack_type: "docker"
stack_size: "4"

```

(continues on next page)

(continued from previous page)

```
OR

---
stack_type: "local"
stack_type: "1"
```

After, the configuration of remote hosts, *run the Ansible playbook and verify your deployment*.

1.3 Policies

1.3.1 Contributor Code of Conduct

As contributors and maintainers of this project, and in the interest of fostering an open and welcoming community, we pledge to respect all people who contribute to the project.

We are committed to making participation in this project a harassment-free experience for everyone, regardless of level of experience, gender, gender identity and expression, sexual orientation, disability, personal appearance, body size, race, ethnicity, age, religion, nationality, or species–no picking on Wrigley for being a buffalo!

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery
- Personal attacks
- Trolling or insulting/derogatory comments
- Public or private harassment
- Publishing other's private information, such as physical or electronic addresses, without explicit permission
- Deliberate intimidation
- Other unethical or unprofessional conduct

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

By adopting this Code of Conduct, project maintainers commit themselves to fairly and consistently applying these principles to every aspect of managing this project. Project maintainers who do not follow or enforce the Code of Conduct may be permanently removed from the project team.

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community.

Instances of abusive, harassing, or otherwise unacceptable behavior directed at yourself or another community member may be reported by contacting a project maintainer at conduct@bigchaindb.com. All complaints will be reviewed and investigated and will result in a response that is appropriate to the circumstances. Maintainers are obligated to maintain confidentiality with regard to the reporter of an incident.

This Code of Conduct is adapted from the Contributor Covenant, version 1.3.0, available at http://contributor-covenant.org/version/1/3/0/

```
shortname: BEP-6
name: Shared Workspace Protocol
type: Meta
```

(continues on next page)

(continued from previous page)

```
status: Draft
editor: Alberto Granzotto <alberto@bigchaindb.com>
```

1.3.2 Abstract

The "Shared Workspace Protocol" provides a simple set of rules for people working in the same workspace.

1.3.3 Motivation

Focusing in a shared workspace (or in the office in general) can be a really challenging task. Working from home solves the problem, but it is not an ideal solution if forced by the difficulty to focus in the office.

1.3.4 Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

1.3.5 Rationale

SWP provides a reusable collaboration model for people working in the same workspace. It has these specific goals:

1. To maximize focus and productivity, by reducing the sources of distraction.
2. To reduce the burden to ask people to "behave" or "leave the room", by having an explicit protocol people agree before entering the room.

1.3.6 Implementation

1. When implemented, this protocol MUST be visible at the entrance of the shared workspace.
2. A person SHOULD write a message to another person before speaking with them.
3. A person SHOULD NOT be loud.
4. A person SHOULD NOT talk to more than one person at a time.
5. One-on-one conversations MUST be short and occasional.
6. A person MUST NOT have calls unless everyone in the room is involved.

1.3.7 Copyright Waiver

1.3.8 Python Style Guide

This guide starts out with our general Python coding style guidelines and ends with a section on how we write & run (Python) tests.

General Python Coding Style Guidelines

Our starting point is PEP8, the standard "Style Guide for Python Code." Many Python IDEs will check your code against PEP8. (Note that PEP8 isn't frozen; it actually changes over time, but slowly.)

BigchainDB uses Python 3.5+, so you can ignore all PEP8 guidelines specific to Python 2.

We use pre-commit to check some of the rules below before every commit but not everything is realized yet. The hooks we use can be found in the .pre-commit-config.yaml file.

Python Docstrings

PEP8 says some things about docstrings, but not what to put in them or how to structure them. PEP257 was one proposal for docstring conventions, but we prefer Google-style docstrings instead: they're easier to read and the napoleon extension for Sphinx lets us turn them into nice-looking documentation. Here are some references on Google-style docstrings:

- Google's docs on Google-style docstrings
- napoleon's docs include an overview of Google-style docstrings
- Example Google-style docstrings (from napoleon's docs)

Maximum Line Length

PEP8 has some maximum line length guidelines, starting with "Limit all lines to a maximum of 79 characters" but "for flowing long blocks of text with fewer structural restrictions (docstrings or comments), the line length should be limited to 72 characters."

We discussed this at length, and it seems that the consensus is: *try* to keep line lengths less than 79/72 characters, unless you have a special situation where longer lines would improve readability. (The basic reason is that 79/72 works for everyone, and BigchainDB is an open source project.) As a hard limit, keep all lines less than 119 characters (which is the width of GitHub code review).

Single or Double Quotes?

Python lets you use single or double quotes. PEP8 says you can use either, as long as you're consistent. We try to stick to using single quotes, except in cases where using double quotes is more readable. For example:

```
print('This doesn\'t look so nice.')
print("Doesn't this look nicer?")
```

Breaking Strings Across Multiple Lines

Should we use parentheses or slashes (\) to break strings across multiple lines, i.e.

```
my_string = ('This is a very long string, so long that it will not fit into just one␣
↪line '
             'so it must be split across multiple lines.')
# or
my_string = 'This is a very long string, so long that it will not fit into just one␣
↪line ' \
            'so it must be split across multiple lines.'
```

It seems the preference is for slashes, but using parentheses is okay too. (There are good arguments either way. Arguing about it seems like a waste of time.)

How to Format Long import Statements

If you need to `import` lots of names from a module or package, and they won't all fit in one line (without making the line too long), then use parentheses to spread the names across multiple lines, like so:

```
from Tkinter import (
    Tk, Frame, Button, Entry, Canvas, Text,
    LEFT, DISABLED, NORMAL, RIDGE, END,
)

# Or

from Tkinter import (Tk, Frame, Button, Entry, Canvas, Text,
                     LEFT, DISABLED, NORMAL, RIDGE, END)
```

For the rationale, see PEP 328.

Using the % operator or `format()` to Format Strings

Given the choice:

```
x = 'name: %s; score: %d' % (name, n)
# or
x = 'name: {}; score: {}'.format(name, n)
```

we use the `format()` version. The official Python documentation says, "This method of string formatting is the new standard in Python 3, and should be preferred to the % formatting described in String Formatting Operations in new code."

Running the Flake8 Style Checker

We use Flake8 to check our Python code style. Once you have it installed, you can run it using:

```
flake8 --max-line-length 119 bigchaindb/
```

Writing and Running (Python) Tests

The content of this section was moved to `bigchaindb/tests/README.md`.

Note: We automatically run all tests on all pull requests (using Travis CI), so you should definitely run all tests locally before you submit a pull request. See the above-linked README file for instructions.

1.3.9 BigchainDB JavaScript Style Guide

For consistent JavaScript across BigchainDB-related repos.

Introduction

At ascribe we write a lot of JavaScript and value quality code. Since all of us liked Airbnb's JavaScript Style Guide, we figured that we can just fork it and change it to our needs.

- *JavaScript Style Guide (this document)*
- React Style Guide

Usage

Use the provided ESlint packages under `packages/` and refer to their documentation for detailed usage:

- eslint-config-ascribe
- eslint-config-ascribe-react

Table of Contents

Types

- *1.1* **Primitives**: When you access a primitive type you work directly on its value.
  - `string`
  - `number`
  - `boolean`
  - `null`
  - `undefined`

```
const foo = 1;
let bar = foo;

bar = 9;

console.log(foo, bar); // => 1, 9
```

- *1.2* **Complex**: When you access a complex type you work on a reference to its value.
  - `object`
  - `array`
  - `function`

```
const foo = [1, 2];
const bar = foo;

bar[0] = 9;

console.log(foo[0], bar[0]); // => 9, 9
```

back to top

References

- *2.1* Use `const` for all of your references; avoid using `var`.

  Why? This ensures that you can't reassign your references (mutation), which can lead to bugs and difficult to comprehend code.

```
// bad
var a = 1;
var b = 2;

// good
const a = 1;
const b = 2;
```

- *2.2* If you must mutate references, use `let` instead of `var`.

  Why? `let` is block-scoped rather than function-scoped like `var`.

```
// bad
var count = 1;
if (true) {
    count += 1;
}

// good, use the let.
let count = 1;
if (true) {
    count += 1;
}
```

- *2.3* Note that both `let` and `const` are block-scoped.

```
// const and let only exist in the blocks they are defined in.
{
    let a = 1;
    const b = 1;
}
console.log(a); // ReferenceError
console.log(b); // ReferenceError
```

back to top

Objects

- *3.1* Use the literal syntax for object creation.

```
// bad
const item = new Object();

// good
const item = {};
```

- *3.2* If your code will be executed in browsers in script context, don't use reserved words as keys. It won't work in IE8. More info. It's OK to use them in ES6 modules and server-side code.

```
// bad
const superman = {
    default: { clark: 'kent' },
    private: true,
};

// good
```

(continues on next page)

(continued from previous page)

```
const superman = {
    defaults: { clark: 'kent' },
    hidden: true,
};
```

- *3.3* Use readable synonyms in place of reserved words.

```
// bad
const superman = {
    class: 'alien',
};

// bad
const superman = {
    klass: 'alien',
};

// good
const superman = {
    type: 'alien',
};
```

- *3.4* Use computed property names when creating objects with dynamic property names.

  Why? They allow you to define all the properties of an object in one place.

```
function getKey(k) {
    return `a key named ${k}`;
}

// bad
const obj = {
    id: 5,
    name: 'Berlin',
};
obj[getKey('enabled')] = true;

// good
const obj = {
    id: 5,
    name: 'Berlin',
    [getKey('enabled')]: true,
};
```

- *3.5* Use object method shorthand.

```
// bad
const atom = {
    value: 1,

    addValue: function (value) {
        return atom.value + value;
    },
};

// good
const atom = {
```

(continues on next page)

(continued from previous page)

```
    value: 1,

    addValue(value) {
        return atom.value + value;
    },
};
```

- *3.6* Use property value shorthand.

  Why? It is shorter to write and descriptive.

```
const lukeSkywalker = 'Luke Skywalker';

// bad
const obj = {
    lukeSkywalker: lukeSkywalker,
};

// good
const obj = {
    lukeSkywalker,
};
```

- *3.7* Group your shorthand properties at the beginning of your object declaration.

  Why? It's easier to tell which properties are using the shorthand.

```
const anakinSkywalker = 'Anakin Skywalker';
const lukeSkywalker = 'Luke Skywalker';

// bad
const obj = {
    episodeOne: 1,
    twoJediWalkIntoACantina: 2,
    lukeSkywalker,
    episodeThree: 3,
    mayTheFourth: 4,
    anakinSkywalker,
};

// good
const obj = {
    lukeSkywalker,
    anakinSkywalker,
    episodeOne: 1,
    twoJediWalkIntoACantina: 2,
    episodeThree: 3,
    mayTheFourth: 4,
};
```

- *3.8* Prefer quoting only properties that are invalid identifiers, but always ensure that all properties are consistently quoted.

  Why? In general we consider it subjectively easier to read. It improves syntax highlighting, and is also more easily optimized by many javascript engines.

```
// bad
const bad = {
    foo: 3,
    bar: 4,
    'data-blah': 5
};

// good
const good = {
    'foo': 3,
    'bar': 4,
    'data-blah': 5
};

// better
const better = {
    foo: 3,
    bar: 4,
    dataBlah: 5
};
```

back to top

Arrays

- *4.1* Use the literal syntax for array creation.

```
// bad
const items = new Array();

// good
const items = [];
```

- *4.2* Use Array#push instead of direct assignment to add items to an array.

```
const someStack = [];

// bad
someStack[someStack.length] = 'abracadabra';

// good
someStack.push('abracadabra');
```

- *4.3* Use array spreads `...` to copy arrays.

```
// bad
const len = items.length;
const itemsCopy = [];
let i;

for (i = 0; i < len; i++) {
    itemsCopy[i] = items[i];
}

// good
const itemsCopy = [...items];
```

- *4.4* To convert an array-like object to an array, use Array#from.

```
const foo = document.querySelectorAll('.foo');
const nodes = Array.from(foo);
```

back to top

Destructuring

- *5.1* Use object destructuring when accessing and using multiple properties of an object.

  Why? Destructuring saves you from creating temporary references for those properties.

```
// bad
function getFullName(user) {
    const firstName = user.firstName;
    const lastName = user.lastName;

    return `${firstName} ${lastName}`;
}

// good
function getFullName(obj) {
    const { firstName, lastName } = obj;
    return `${firstName} ${lastName}`;
}

// best
function getFullName({ firstName, lastName }) {
    return `${firstName} ${lastName}`;
}
```

- *5.2* When destructuring requires multiple lines, follow formatting rules for *objects*:

```
// bad
const { first: {
            nested
        },
        second } = obj;

// bad
const {
    first: {
        nested
    },
    second } = obj;

// good
const {
    first: {
        nested
    },
    second
} = obj;
```

- *5.3* Use array destructuring.

```
const arr = [1, 2, 3, 4];

// bad
const first = arr[0];
const second = arr[1];

// good
const [first, second] = arr;
```

- *5.4* Use object destructuring for multiple return values, not array destructuring.

  Why? You can add new properties over time or change the order of things without breaking call sites.

```
// bad
function processInput(input) {
    // then a miracle occurs
    return [left, right, top, bottom];
}

// the caller needs to think about the order of return data
const [left, __, top] = processInput(input);

// good
function processInput(input) {
    // then a miracle occurs
    return { left, right, top, bottom };
}

// the caller selects only the data they need
const { left, right } = processInput(input);
```

- *5.5* You can use destructuring and an object spread operator to filter out specific properties while keeping the other properties in a new object.

```
// bad
const val = obj.value;
delete obj.value;

// good
const { value: val, ...otherObj } = obj;
// otherObj will hold all other properties of obj except for value
```

back to top

Strings

- *6.1* Use single quotes `''` for strings.

```
// bad
const name = "Capt. Janeway";

// good
const name = 'Capt. Janeway';
```

- *6.2* When using (single- or double) quotes in a string, use the other literal (`''` or `""`).

```
// bad
const name = "What a \"nice\" day!";

// bad
const name = 'Let\'s go to Rosi\'s!';

// good
const name = 'What a "nice" day!';

// good
const name = "Let's go to Rosi's!";
```

- *6.3* Strings longer than 100 characters should be written across multiple lines using string concatenation.

```
// bad
const errorMessage = 'This is a super long error that was thrown because of␣
↪Batman. When you stop to think about how Batman had anything to do with this,␣
↪you would get nowhere fast.';

// bad
const errorMessage = 'This is a super long error that was thrown because \
of Batman. When you stop to think about how Batman had anything to do \
with this, you would get nowhere \
fast.';

// good
const errorMessage = 'This is a super long error that was thrown because ' +
    'of Batman. When you stop to think about how Batman had anything to do ' +
    'with this, you would get nowhere fast.';
```

- *6.4* Note: If overused, long strings with concatenation could impact performance. jsPerf & Discussion.
- *6.5* When programmatically building up strings, use template strings instead of concatenation.

  Why? Template strings give you a readable, concise syntax with proper newlines and string interpolation features.

```
// bad
function sayHi(name) {
    return 'How are you, ' + name + '?';
}

// bad
function sayHi(name) {
    return ['How are you, ', name, '?'].join();
}

// good
function sayHi(name) {
    return `How are you, ${name}?`;
}
```

- *6.6* **NEVER** use eval() on a string, it opens too many vulnerabilities.

back to top

Functions

- *7.1* Use function declarations instead of function expressions.

  > Why? Function declarations are named, so they're easier to identify in call stacks. Also, the whole body of a function declaration is hoisted, whereas only the reference of a function expression is hoisted. This rule makes it possible to always use *Arrow Functions* in place of function expressions.

  ```
  // bad
  const foo = function () {
  };

  // good
  function foo() {
  }
  ```

- *7.2* Immediately-invoked function expressions should use arrow functions as opposed to traditional functions:

  ```
  // immediately-invoked function expression (IIFE)
  (() => {
      console.log('Welcome to the Internet. Please follow me.');
  })();
  ```

- *7.3* **NEVER** declare a function in a non-function block (if, while, etc). Assign the function to a variable instead. Browsers will allow you to do it, but they all interpret it differently, which is bad news bears.

- *7.4* **Note:** ECMA-262 defines a `block` as a list of statements. A function declaration is not a statement. Read ECMA-262's note on this issue.

  ```
  // bad
  if (currentUser) {
      function test() {
          console.log('Nope.');
      }
  }

  // good
  let test;
  if (currentUser) {
      test = () => {
          console.log('Yup.');
      };
  }
  ```

- *7.5* **NEVER** name a parameter `arguments`. This will take precedence over the `arguments` object that is given to every function scope.

  ```
  // bad
  function nope(name, options, arguments) {
      // ...stuff...
  }

  // good
  function yup(name, options, args) {
      // ...stuff...
  }
  ```

- *7.6* **NEVER** use `arguments`, opt to use rest syntax `...` instead.

Why? `...` is explicit about which arguments you want pulled. Plus rest arguments are a real Array and not Array-like like `arguments`.

```
// bad
function concatenateAll() {
    const args = Array.prototype.slice.call(arguments);
    return args.join('');
}

// good
function concatenateAll(...args) {
    return args.join('');
}
```

- *7.7* Use default parameter syntax rather than mutating function arguments.

```
// really bad
function handleThings(opts) {
    // No! We shouldn't mutate function arguments.
    // Double bad: if opts is falsy it'll be set to an object which may
    // be what you want but it can introduce subtle bugs.
    opts = opts || {};
    // ...
}

// still bad
function handleThings(opts) {
    if (opts === void 0) {
        opts = {};
    }
    // ...
}

// good
function handleThings(opts = {}) {
    // ...
}
```

- *7.8* Avoid side effects with default parameters.

  Why? They are confusing to reason about.

```
var b = 1;
// bad
function count(a = b++) {
  console.log(a);
}
count();  // 1
count();  // 2
count(3); // 3
count();  // 3
```

- *7.9* Always put default parameters last.

```
// bad
function handleThings(opts = {}, name) {
    // ...
}
```

(continues on next page)

(continued from previous page)

```
// good
function handleThings(name, opts = {}) {
    // ...
}
```

- *7.10* **NEVER** use the Function constructor to create a new function.

  Why? Creating a function in this way evaluates a string similarly to eval(), which opens vulnerabilities.

```
// bad
var add = new Function('a', 'b', 'return a + b');

// still bad
var subtract = Function('a', 'b', 'return a - b');
```

back to top

Arrow Functions

- *8.1* When you must use function expressions (as when passing an anonymous function), use arrow function notation.

  Why? It creates a version of the function that executes in the context of `this`, which is usually what you want, and is a more concise syntax.

  Why not? If you have a fairly complicated function, you might move that logic out into its own function declaration.

```
// bad
[1, 2, 3].map(function (x) {
    const y = x + 1;
    return x * y;
});

// good
[1, 2, 3].map((x) => {
    const y = x + 1;
    return x * y;
});
```

- *8.2* If the function body consists of a single expression, feel free to omit the braces and use the implicit return. Otherwise use a `return` statement.

  Why? Syntactic sugar. It reads well when multiple functions are chained together.

  Why not? If you plan on returning an object.

```
// good
[1, 2, 3].map(number => `A string containing the ${number}.`);

// bad
[1, 2, 3].map(number => {
    const nextNumber = number + 1;
    `A string containing the ${nextNumber}.`;
});
```

(continues on next page)

(continued from previous page)

```
// good
[1, 2, 3].map(number => {
    const nextNumber = number + 1;
    return `A string containing the ${nextNumber}.`;
});
```

- *8.3* In case the expression spans over multiple lines, wrap it in parentheses for better readability.

  Why? It shows clearly where the function starts and ends.

```
// bad
[1, 2, 3].map(number => 'As time went by, the string containing the ' +
    `${number} became much longer. So we needed to break it over multiple ` +
    'lines.'
);

// good
[1, 2, 3].map(number => (
    `As time went by, the string containing the ${number} became much ` +
    'longer. So we needed to break it over multiple lines.'
));
```

- *8.4* If your function only takes a single argument, feel free to omit the parentheses.

  Why? Less visual clutter.

```
// good
[1, 2, 3].map(x => x * x);

// good
[1, 2, 3].reduce((y, x) => x + y);
```

back to top

Constructors

- *9.1* Always use `class`. Avoid manipulating `prototype` directly.

  Why? `class` syntax is more concise and easier to reason about.

```
// bad
function Queue(contents = []) {
    this._queue = [...contents];
}
Queue.prototype.pop = function() {
    const value = this._queue[0];
    this._queue.splice(0, 1);
    return value;
}


// good
class Queue {
    constructor(contents = []) {
        this._queue = [...contents];
```

(continues on next page)

(continued from previous page)

```
    }
    pop() {
        const value = this._queue[0];
        this._queue.splice(0, 1);
        return value;
    }
}
```

- *9.2* Use `extends` for inheritance.

  Why? It is a built-in way to inherit prototype functionality without breaking `instanceof`.

```
// bad
const inherits = require('inherits');
function PeekableQueue(contents) {
    Queue.apply(this, contents);
}
inherits(PeekableQueue, Queue);
PeekableQueue.prototype.peek = function() {
    return this._queue[0];
}

// good
class PeekableQueue extends Queue {
    peek() {
        return this._queue[0];
    }
}
```

- *9.3* Methods can return `this` to help with method chaining.

```
// bad
Jedi.prototype.jump = function() {
    this.jumping = true;
    return true;
};

Jedi.prototype.setHeight = function(height) {
    this.height = height;
};

const luke = new Jedi();
luke.jump(); // => true
luke.setHeight(20); // => undefined

// good
class Jedi {
    jump() {
        this.jumping = true;
        return this;
    }

    setHeight(height) {
        this.height = height;
        return this;
    }
}
```

(continues on next page)

(continued from previous page)

```
const luke = new Jedi();

luke.jump()
    .setHeight(20);
```

- *9.4* It's okay to write a custom toString() method, just make sure it works successfully and causes no side effects.

```
class Jedi {
    constructor({ name = 'no name' } = {}) {
        this.name = name;
    }

    getName() {
        return this.name;
    }

    toString() {
        return `Jedi - ${this.getName()}`;
    }
}
```

back to top

Modules

- *10.1* Always use modules (`import`/`export`) over a non-standard module system. You can always transpile to your preferred module system.

  Why? Modules are the future, let's start using the future now.

```
// bad
const AirbnbStyleGuide = require('./AirbnbStyleGuide');
module.exports = AirbnbStyleGuide.es6;

// ok
import AirbnbStyleGuide from './AirbnbStyleGuide';
export default AirbnbStyleGuide.es6;

// best
import { es6 } from './AirbnbStyleGuide';
export default es6;
```

- *10.2* Do not use wildcard imports.

  Why? This makes sure you have a single default export.

```
// bad
import * as AirbnbStyleGuide from './AirbnbStyleGuide';

// good
import AirbnbStyleGuide from './AirbnbStyleGuide';
```

- *10.3* And do not export directly from an import.

  Why? Although the one-liner is concise, having one clear way to import and one clear way to export makes things consistent.

```
// bad
// filename es6.js
export { es6 as default } from './airbnbStyleGuide';

// good
// filename es6.js
import { es6 } from './AirbnbStyleGuide';
export default es6;
```

back to top

Iterators and Generators

- *11.1* Prefer JavaScript's higher-order functions like `map()` and `reduce()` instead of loops like `for-of` unless there is a substantial performance disadvantage by doing so.

  Why? This enforces our immutable rule. Dealing with pure functions that return values is easier to reason about than side-effects.

```
const numbers = [1, 2, 3, 4, 5];

// bad
let sum = 0;
for (let num of numbers) {
    sum += num;
}

sum === 15;

// good
let sum = 0;
numbers.forEach((num) => sum += num);
sum === 15;

// best (use the functional force, Luke)
const sum = numbers.reduce((total, num) => total + num, 0);
sum === 15;
```

- *11.2* Only use `for-in` if you know exactly what you're doing. If unsure, prefer the options given in *11.1*.
- *11.3* Don't use generators for now.

  Why? They don't transpile well to ES5.

back to top

Properties

- *12.1* Use dot notation when accessing properties.

```
const luke = {
    jedi: true,
    age: 28,
};

// bad
```

(continues on next page)

(continued from previous page)

```
const isJedi = luke['jedi'];

// good
const isJedi = luke.jedi;
```

- *12.2* Use subscript notation `[]` when accessing properties with a variable.

```
const luke = {
    jedi: true,
    age: 28,
};

function getProp(prop) {
    return luke[prop];
}

const isJedi = getProp('jedi');
```

back to top

Variables

- *13.1* Always use `const` or `let` to declare variables. Not doing so will result in global variables. We want to avoid polluting the global namespace. Captain Planet warned us of that.

```
// bad
superPower = new SuperPower();

// good
const superPower = new SuperPower();
```

- *13.2* Use one `const` or `let` declaration per variable.

  Why? It's easier to add new variable declarations this way, and you never have to worry about swapping out a `;` for a `,` or introducing punctuation-only diffs.

```
// bad
const items = getItems(),
      goSportsTeam = true,
      dragonball = 'z';

// bad
// (compare to above, and try to spot the mistake)
const items = getItems(),
      goSportsTeam = true;
      dragonball = 'z';

// good
const items = getItems();
const goSportsTeam = true;
const dragonball = 'z';
```

- *13.3* Group all your `const`s and then group all your `let`s.

  Why? This is helpful when later on you might need to assign a variable depending on one of the previous assigned variables.

```
// bad
let i, len, dragonball,
    items = getItems(),
    goSportsTeam = true;

// bad
let i;
const items = getItems();
let dragonball;
const goSportsTeam = true;
let len;

// good
const goSportsTeam = true;
const items = getItems();
let dragonball;
let i;
let length;
```

- *13.4* Assign variables where you need them, but place them in a reasonable place.

  Why? `let` and `const` are block scoped and not function scoped.

```
// good
function() {
    test();
    console.log('doing stuff..');

    //..other stuff..

    const name = getName();

    if (name === 'test') {
        return false;
    }

    return name;
}

// bad - unnecessary function call
function(hasName) {
    const name = getName();

    if (!hasName) {
        return false;
    }

    this.setFirstName(name);

    return true;
}

// good
function(hasName) {
    if (!hasName) {
        return false;
    }
```

(continues on next page)

(continued from previous page)

```
    const name = getName();
    this.setFirstName(name);

    return true;
}
```

Note that referencing a variable declared by `let` or `const` before they are set results in a reference error, including typeof (see Why `typeof` is no longer "safe")

```
if (condition) {
    console.log(typeof value);     // ReferenceError!
    let value = "blue";
}
```

- *13.5* Avoid declaring unused variables, however the cases where it can be convenient (such as filtering some properties out of an object or destructuring an array, for example), prefix the variable name with `ignored`:

```
// bad
const {
    first, // ignored
    second, // ignored
    third
} = winners;

// good
const {
    first: ignoredFirst, // ignored
    second: ignoredSecond, // ignored
    third
} = winners;
```

Note that our ESLint configuration is set up to error on any unused variable unless it is prefixed by `ignored`. An exception to this is argument names; any arguments listed before the first one used is OK:

```
// bad -- `second` is unused
function (first, second) {
    return first;
}

// good -- `first` is listed before the used `second` argument
function (first, second) {
    return second;
}
```

back to top

Hoisting

- *14.1* `var` declarations get hoisted to the top of their scope, their assignment does not. `const` and `let` declarations are blessed with a new concept called Temporal Dead Zones (TDZ). It's important to know why typeof is no longer safe.

```
// we know this wouldn't work (assuming there
// is no notDefined global variable)
function example() {
```

(continues on next page)

(continued from previous page)

```
    console.log(notDefined); // => throws a ReferenceError
}

// creating a variable declaration after you
// reference the variable will work due to
// variable hoisting. Note: the assignment
// value of `true` is not hoisted.
function example() {
    console.log(declaredButNotAssigned); // => undefined
    var declaredButNotAssigned = true;
}

// The interpreter is hoisting the variable
// declaration to the top of the scope,
// which means our example could be rewritten as:
function example() {
    let declaredButNotAssigned;
    console.log(declaredButNotAssigned); // => undefined
    declaredButNotAssigned = true;
}

// using const and let
function example() {
    console.log(declaredButNotAssigned); // => throws a ReferenceError
    console.log(typeof declaredButNotAssigned); // => throws a ReferenceError
    const declaredButNotAssigned = true;
}
```

- *14.2* Anonymous function expressions hoist their variable name, but not the function assignment.

```
function example() {
    console.log(anonymous); // => undefined

    anonymous(); // => TypeError anonymous is not a function

    var anonymous = function() {
        console.log('anonymous function expression');
    };
}
```

- *14.3* Named function expressions hoist the variable name, not the function name or the function body.

```
function example() {
    console.log(named); // => undefined

    named(); // => TypeError named is not a function

    superPower(); // => ReferenceError superPower is not defined

    var named = function superPower() {
        console.log('Flying');
    };
}

// the same is true when the function name
// is the same as the variable name.
function example() {
```

(continues on next page)

(continued from previous page)

```
    console.log(named); // => undefined

    named(); // => TypeError named is not a function

    var named = function named() {
        console.log('named');
    }
}
```

- *14.4* Function declarations hoist their name and the function body.

```
function example() {
    superPower(); // => Flying

    function superPower() {
        console.log('Flying');
    }
}
```

- *14.5* ES6 `imports` are hoisted to the beginning of their module while modules imported through `requires` (ie. CommonJS modules) are not.

```
// This works
foo();

import { foo } from 'my_module';

// This will import 'imported_module' before 'required_module'
require('required_module');

import 'imported_module';
```

- For more information refer to JavaScript Scoping & Hoisting by Ben Cherry.

back to top

Comparison Operators & Equality

- *15.1* Use `===` and `!==` over `==` and `!=`. Avoid `==` and `!=` because they are 'loose' equality comparisons, only evaluating equality after coercing both values following confusing and difficult to remember rules (see MDN).
- *15.2* Conditional statements such as the `if` statement evaluate their expression using coercion with the `ToBoolean` abstract method and always follow these simple rules:
  - **Objects** evaluate to **true**
  - **Undefined** evaluates to **false**
  - **Null** evaluates to **false**
  - **Booleans** evaluate to **the value of the boolean**
  - **Numbers** evaluate to **false** if **+0, -0, or NaN**, otherwise **true**
  - **Strings** evaluate to **false** if an empty string `''`, otherwise **true**

```
if ([]) {
    // true
    // An array is an object, objects evaluate to true
}
```

- *15.3* Use shortcuts.

```
// bad
if (name !== '') {
    // ...stuff...
}

// good
if (name) {
    // ...stuff...
}

// bad
if (collection.length > 0) {
    // ...stuff...
}

// good
if (collection.length) {
    // ...stuff...
}
```

- *15.4* For more information see Truth Equality and JavaScript by Angus Croll.
- *15.5* Use braces to create blocks in `case` and `default` clauses that contain lexical declarations (e.g. `let`, `const`, `function`, and `class`).

  Why? Lexical declarations are visible in the entire `switch` block but only get initialized when assigned, which only happens when its `case` is reached. This causes problems when multiple `case` clauses attempt to define the same thing.

```
// bad
switch (foo) {
    case 1:
        let x = 1;
        break;
    case 2:
        const y = 2;
        break;
    case 3:
        function f() {}
        break;
    default:
        class C {}
}

// good
switch (foo) {
    case 1: {
        let x = 1;
        break;
    }
    case 2: {
```

(continues on next page)

(continued from previous page)

```
        const y = 2;
        break;
    }
    case 3: {
        function f() {}
        break;
    }
    case 4:
        bar();
        break;
    default: {
        class C {}
    }
}
```

- *15.6* Indent one full level for case statements.

```
// bad
switch (foo) {
case 1:
    break;
default:
    break;
}

// bad
switch (foo) {
  case 1:
    break;
  default:
    break;
}

// good
switch (foo) {
    case 1:
        break;
    default:
        break;
}
```

- *15.7* Ternaries should not be nested and generally be single line expressions.

```
// bad
const foo = maybe1 > maybe2
    ? "bar"
    : value1 > value2 ? "baz" : null;

// better
const maybeNull = value1 > value2 ? 'baz'
                                  : null;

const foo = maybe1 > maybe2
    ? 'bar'
    : maybeNull;

// best
```

(continues on next page)

(continued from previous page)

```
const maybeNull = value1 > value2 ? 'baz' : null;

const foo = maybe1 > maybe2 ? 'bar' : maybeNull;
```

- *15.8* Avoid unneeded ternary statements.

```
// bad
const foo = a ? a : b;
const bar = c ? true : false;
const baz = c ? false : true;

// good
const foo = a || b;
const bar = !!c;
const baz = !c;
```

- *15.9* Use any of the following styles for multi-line ternary statements:

```
// good
const foo = thisisasuperlongexpression ? value
                                       : otherValue;

// good
const foo = thisisasuperlongexpression
    ? value : otherValue;

// good
const foo = thisisasuperlongexpression
    ? value
    : otherValue;
```

back to top

Blocks

- *16.1* Use braces with all multi-line blocks.

```
// bad
if (test)
    return false;

// good
if (test) return false;

// good
if (test) {
    return false;
}

// bad
function() { return false; }

// good
function() {
    return false;
}
```

- *16.2* If you're using multi-line blocks with `if` and `else`, put `else` on the same line as your `if` block's closing brace.

```
// bad
if (test) {
    thing1();
    thing2();
}
else {
    thing3();
}

// good
if (test) {
    thing1();
    thing2();
} else {
    thing3();
}
```

back to top

Comments

- *17.1* Use `/** ... */` for multi-line comments. Include a description, specify types and values for all parameters and return values by using JSDoc.

```
// bad
// make() returns a new element
// based on the passed in tag name
//
// @param {String} tag
// @return {Element} element
function make(tag) {

    // ...stuff...

    return element;
}

// good
/**
 * make() returns a new element
 * based on the passed in tag name
 *
 * @param {String} tag
 * @return {Element} element
 */
function make(tag) {

    // ...stuff...

    return element;
}
```

- *17.2* Use `//` for single line comments. Place single line comments on a newline above the subject of the comment. Put an empty line before the comment.

```
// bad
const active = true;  // is current tab

// good
// is current tab
const active = true;

// bad
function getType() {
    console.log('fetching type...');
    // set the default type to 'no type'
    const type = this._type || 'no type';

    return type;
}

// good
function getType() {
    console.log('fetching type...');

    // set the default type to 'no type'
    const type = this._type || 'no type';

    return type;
}
```

- *17.3* Always put a single space between where your comment starts (ie. `/*`, `/**`, or `//`) and the comment.
- *17.4* Prefixing your comments with `FIXME` or `TODO` helps other developers quickly understand if you're pointing out a problem that needs to be revisited, or if you're suggesting a solution to the problem that needs to be implemented. These are different than regular comments because they are actionable. The actions are `FIXME -- need to figure this out` or `TODO -- need to implement`.
- *17.5* Use `// FIXME:` to annotate problems.

```
class Calculator extends Abacus {
    constructor() {
        super();

        // FIXME: shouldn't use a global here
        total = 0;
    }
}
```

- *17.6* Use `// TODO:` to annotate solutions to problems.

```
class Calculator extends Abacus {
    constructor() {
        super();

        // TODO: total should be configurable by an options param
        this.total = 0;
    }
}
```

back to top

Whitespace

- *18.1* Use soft tabs set to 4 spaces.

```
// good
function() {
const name;
}

// bad
function() {
const name;
}

// bad
function() {
const name;
}
```

- *18.2* Place 1 space before the leading brace.

```
// bad
function test(){
    console.log('test');
}

// good
function test() {
    console.log('test');
}

// bad
dog.set('attr',{
    age: '1 year',
    breed: 'Bernese Mountain Dog',
});

// good
dog.set('attr', {
    age: '1 year',
    breed: 'Bernese Mountain Dog',
});
```

- *18.3* Place 1 space before the opening parenthesis in control statements (`if`, `while` etc.) and anonymous function declarations. Place no space before the argument list in function calls and named declarations.

```
// bad
if(isJedi) {
    fight ();
}

// good
if (isJedi) {
    fight();
}

// bad
function() {
```

(continues on next page)

(continued from previous page)

```
    console.log('Anonymous');
}

// good -- easier to tell this is a function decarlation rather than function call
function () {
    console.log('Anonymous');
}

// bad
function fight () {
    console.log ('Swooosh!');
}

// good
function fight() {
    console.log('Swooosh!');
}
```

- *18.4* Set off operators with spaces.

```
// bad
const x=y+5;

// good
const x = y + 5;
```

- *18.5* End files with a single newline character.

```
// bad
(function(global) {
    // ...stuff...
})(this);
```

```
// bad
(function(global) {
    // ...stuff...
})(this);

```

```
// good
(function(global) {
    // ...stuff...
})(this);
```

- *18.6* Use indentation when making long method chains. Use a leading dot, which emphasizes that the line is a method call, not a new statement.

```
// bad
$('#items').find('.selected').highlight().end().find('.open').updateCount();

// bad
$('#items').
    find('.selected').
        highlight().
        end().
    find('.open').
```

(continues on next page)

(continued from previous page)

```
        updateCount();

// good
$('#items')
    .find('.selected')
        .highlight()
        .end()
    .find('.open')
        .updateCount();

// bad
const request = fetch('/users').then(...).catch(...).finally(...);

// good
const request = fetch('/users')
    .then(...)
    .catch(...)
    .finally(...);
```

- *18.7* Leave a blank line after blocks and before the next statement.

```
// bad
if (foo) {
    return bar;
}
return baz;

// good
if (foo) {
    return bar;
}

return baz;

// bad
const obj = {
    foo() {
    },
    bar() {
    },
};
return obj;

// good
const obj = {
    foo() {
    },

    bar() {
    },
};

return obj;

// bad
const arr = [
    function foo() {
```

(continues on next page)

(continued from previous page)

```
    },
    function bar() {
    },
];
return arr;

// good
const arr = [
    function foo() {
    },

    function bar() {
    },
];

return arr;
```

- *18.8* Break long logical operations into multiple lines, leaving operators at the end of the line and intenting the later lines to the first line's first operand.

```
// bad
if (aReallyReallyLongExpr && anotherSuperLongExpr && wowSoManyExpr &&␣
↪longExprToCheckTheWorldIsOk) {
    ...
}

// good
if (aReallyReallyLongExpr &&
    anotherSuperLongExpr &&
    wowSoManyExpr &&
    longExprToCheckTheWorldIsOk) {
    ...
}

// good
while (aReallyReallyLongExpr &&
       anotherSuperLongExpr &&
       wowSoManyExpr &&
       longExprToCheckTheWorldIsOk) {
    ...
}
```

- *18.9* Do not pad your blocks with blank lines.

```
// bad
function bar() {

    console.log(foo);

}

// also bad
if (baz) {

    console.log(qux);
} else {
```

(continues on next page)

(continued from previous page)

```
    console.log(foo);

}

// good
function bar() {
    console.log(foo);
}

// good
if (baz) {
    console.log(qux);
} else {
    console.log(foo);
}
```

- *18.10* Do not add spaces inside parentheses.

```
// bad
function bar( foo ) {
    return foo;
}

// good
function bar(foo) {
    return foo;
}

// bad
if ( foo ) {
    console.log(foo);
}

// good
if (foo) {
    console.log(foo);
}
```

- *18.11* Do not add spaces inside brackets.

```
// bad
const foo = [ 1, 2, 3 ];
console.log(foo[ 0 ]);

// good
const foo = [1, 2, 3];
console.log(foo[0]);
```

- *18.12* Add spaces inside curly braces.

```
// bad
const foo = {clark: 'kent'};

// good
const foo = { clark: 'kent' };
```

- *18.13* Avoid having lines of code that are longer than 100 characters (including whitespace).

Why? This ensures readability and maintainability.

```
// bad
const foo = 'Whatever national crop flips the window. The cartoon reverts within␣
↪the screw. Whatever wizard constrains a helpful ally. The counterpart ascends!';

// bad
$.ajax({ method: 'POST', url: 'https://airbnb.com/', data: { name: 'John' } }).
↪done(() => console.log('Congratulations!')).fail(() => console.log('You have␣
↪failed this city.'));

// good
const foo = 'Whatever national crop flips the window. The cartoon reverts within␣
↪the screw. ' +
            'Whatever wizard constrains a helpful ally. The counterpart ascends!';

// good
$.ajax({
    method: 'POST',
    url: 'https://airbnb.com/',
    data: { name: 'John' },
})
    .done(() => console.log('Congratulations!'))
    .fail(() => console.log('You have failed this city.'));
```

In some cases, you can go slightly over the limit (urls, code that's *just* slightly over), but our ESLint configuration is set up to warn on code lines that are over 105 characters.

- *18.14* When a function call needs to be broken up into multiple lines, put arguments on a separate line, indented four spaces:

```
// bad
const foo = funcCall(this, is, a, really,
                     realllllyyyyyyy, long,
                     function, call);

// good
const foo = funcCall(
    this, is, a, really,
    realllllyyyyyyy, long,
    function,c all
);
```

back to top

Commas

- *19.1* Leading commas: **Nope.**

```
// bad
const story = [
      once
    , upon
    , aTime
];

// good
```

(continues on next page)

(continued from previous page)

```
const story = [
    once,
    upon,
    aTime,
];

// bad
const hero = {
      firstName: 'Ada'
    , lastName: 'Lovelace'
    , birthYear: 1815
    , superPower: 'computers'
};

// good
const hero = {
    firstName: 'Ada',
    lastName: 'Lovelace',
    birthYear: 1815,
    superPower: 'computers',
};
```

- *19.2* Additional trailing comma: **Yup.**

  Why? This leads to cleaner git diffs. Also, transpilers like Babel will remove the additional trailing comma in the transpiled code which means you don't have to worry about the trailing comma problem in legacy browsers.

```
// bad - git diff without trailing comma
const hero = {
       firstName: 'Florence',
-      lastName: 'Nightingale'
+      lastName: 'Nightingale',
+      inventorOf: ['coxcomb graph', 'modern nursing']
};

// good - git diff with trailing comma
const hero = {
       firstName: 'Florence',
       lastName: 'Nightingale',
+      inventorOf: ['coxcomb chart', 'modern nursing'],
};

// bad
const hero = {
    firstName: 'Dana',
    lastName: 'Scully'
};

const heroes = [
    'Batman',
    'Superman'
];

// good
const hero = {
    firstName: 'Dana',
```

(continues on next page)

(continued from previous page)

```
    lastName: 'Scully',
};

const heroes = [
    'Batman',
    'Superman',
];
```

back to top

Semicolons

- *20.1* **Nope.**

```
// bad
(function() {
    const name = 'Skywalker';
    return name;
})();

// good
(() => {
    const name = 'Skywalker'
    return name
})()
```

back to top

Type Casting & Coercion

- *21.1* Perform type coercion at the beginning of the statement.
- *21.2* Strings:

```
//  => this.reviewScore = 9;

// bad
const totalScore = this.reviewScore + '';

// good
const totalScore = String(this.reviewScore);
```

- *21.3* Numbers: Use `Number` for type casting and `parseInt` always with a radix.

```
const inputValue = '4';

// bad
const val = new Number(inputValue);

// bad
const val = +inputValue;

// bad
const val = inputValue >> 0;
```

(continues on next page)

(continued from previous page)

```
// bad
const val = parseInt(inputValue);

// good
const val = Number(inputValue);

// good
const val = parseInt(inputValue, 10);
```

- *21.4* If for whatever reason you are doing something wild and `parseInt` is your bottleneck and need to use Bitshift for performance reasons, leave a comment explaining why and what you're doing.

```
// good
/**
 * parseInt was the reason my code was slow.
 * Bitshifting the String to coerce it to a
 * Number made it a lot faster.
 */
const val = inputValue >> 0;
```

- *21.5* **Note:** Be careful when using bitshift operations. Numbers are represented as 64-bit values, but Bitshift operations always return a 32-bit integer (source). Bitshift can lead to unexpected behavior for integer values larger than 32 bits. Discussion. Largest signed 32-bit Int is 2,147,483,647:

```
2147483647 >> 0 //=> 2147483647
2147483648 >> 0 //=> -2147483648
2147483649 >> 0 //=> -2147483647
```

- *21.6* Booleans:

```
const age = 0;

// bad
const hasAge = new Boolean(age);

// good
const hasAge = Boolean(age);

// good
const hasAge = !!age;
```

back to top

Naming Conventions

- *22.1* Avoid single letter names. Be descriptive with your naming.

```
// bad
function q() {
    // ...stuff...
}

// good
function query() {
    // ..stuff..
}
```

- *22.2* Use camelCase when naming objects, functions, and instances.

```
// bad
const OBJEcttssssss = {};
const this_is_my_object = {};
function c() {}

// good
const thisIsMyObject = {};
function thisIsMyFunction() {}
```

- *22.3* Use PascalCase when naming constructors or classes.

```
// bad
function user(options) {
    this.name = options.name;
}

const bad = new user({
    name: 'nope',
});

// good
class UserPascalCase {
    constructor(options) {
        this.name = options.name;
    }
}

const good = new UserPascalCase({
    name: 'yup',
});
```

- *22.4* Use a leading underscore _ when naming private properties.

```
// bad
this.__firstName__ = 'Panda';
this.firstName_ = 'Panda';

// good
this._firstName = 'Panda';
```

- *22.5* Don't save references to `this`. Use arrow functions or Function#bind.

```
// bad
function foo() {
    const self = this;
    return function() {
        console.log(self);
    };
}

// bad
function foo() {
    const that = this;
    return function() {
        console.log(that);
    };
```

(continues on next page)

(continued from previous page)

```
}

// good
function foo() {
    return () => {
        console.log(this);
    };
}

// good
function foo() {
    return (function() {
        console.log(this);
    }).bind(this);
}
```

- *22.6* If your file exports a single class, your filename should be exactly the name of the class, converted from PascalCase to snake_case.

```
// file contents
class CheckBox {
    // ...
}
export default CheckBox;

// in some other file
// bad
import CheckBox from './checkBox';

// bad
import CheckBox from './CheckBox';

// good
import CheckBox from './check_box';
```

- *22.7* Use camelCase when you export-default a function. Your filename should be identical to your function's name.

```
function makeStyleGuide() {
}

export default makeStyleGuide;
```

- *22.8* Use PascalCase when you export a singleton / function library / bare object.

```
const AirbnbStyleGuide = {
    es6: {
    }
};

export default AirbnbStyleGuide;
```

back to top

Accessors

- *23.1* Accessor functions for properties are not required.
- *23.2* If you do make accessor functions use getVal() and setVal('hello').

```
// bad
dragon.age();

// good
dragon.getAge();

// bad
dragon.age(25);

// good
dragon.setAge(25);
```

- *23.3* If the property is a `boolean`, use `isVal()` or `hasVal()`.

```
// bad
if (!dragon.age()) {
    return false;
}

// good
if (!dragon.hasAge()) {
    return false;
}
```

- *23.4* It's okay to create get() and set() functions, but be consistent.

```
class Jedi {
    constructor(options = {}) {
        const lightsaber = options.lightsaber || 'blue';
        this.set('lightsaber', lightsaber);
    }

    set(key, val) {
        this[key] = val;
    }

    get(key) {
        return this[key];
    }
}
```

back to top

Events

- *24.1* When attaching data payloads to events (whether DOM events or something more proprietary like Backbone events), pass a hash instead of a raw value. This allows a subsequent contributor to add more data to the event payload without finding and updating every handler for the event. For example, instead of:

```
// bad
$(this).trigger('listingUpdated', listing.id);
```

(continues on next page)

(continued from previous page)

```
...

$(this).on('listingUpdated', function(e, listingId) {
    // do something with listingId
});
```

prefer:

```
// good
$(this).trigger('listingUpdated', { listingId: listing.id });

...

$(this).on('listingUpdated', function(e, data) {
    // do something with data.listingId
});
```

back to top

jQuery

- *25.1* Prefix jQuery object variables with a `$`.

```
// bad
const sidebar = $('.sidebar');

// good
const $sidebar = $('.sidebar');

// good
const $sidebarBtn = $('.sidebar-btn');
```

- *25.2* Cache jQuery lookups.

```
// bad
function setSidebar() {
    $('.sidebar').hide();

    // ...stuff...

    $('.sidebar').css({
        'background-color': 'pink'
    });
}

// good
function setSidebar() {
    const $sidebar = $('.sidebar');
    $sidebar.hide();

    // ...stuff...

    $sidebar.css({
        'background-color': 'pink'
```

(continues on next page)

(continued from previous page)

```
    });
}
```

- *25.3* For DOM queries use Cascading `$('.sidebar ul')` or parent > child `$('.sidebar > ul')`. jsPerf
- *25.4* Use `find` with scoped jQuery object queries.

```
// bad
$('ul', '.sidebar').hide();

// bad
$('.sidebar').find('ul').hide();

// good
$('.sidebar ul').hide();

// good
$('.sidebar > ul').hide();

// good
$sidebar.find('ul').hide();
```

back to top

ECMAScript 5 Compatibility

- *26.1* Refer to Kangax's ES5 compatibility table.

back to top

ECMAScript 6 Styles

- *27.1* This is a collection of links to the various es6 features.
  1. *Arrow Functions*
  2. *Classes*
  3. *Object Shorthand*
  4. *Object Concise*
  5. *Object Computed Properties*
  6. *Template Strings*
  7. *Destructuring*
  8. *Default Parameters*
  9. *Rest*
  10. *Array Spreads*
  11. *Let and Const*
  12. *Iterators and Generators*
  13. *Modules*

- *27.2* Khan Academy has a nice section in their Javascript styleguide that discusses various ways to accomplish tasks in ES6 rather than using underscore/lodash.

back to top

Testing

- *28.1* **Yup.**

```
function() {
  return true;
}
```

back to top

Performance

- On Layout & Web Performance
- String vs Array Concat
- Try/Catch Cost In a Loop
- Bang Function
- jQuery Find vs Context, Selector
- innerHTML vs textContent for script text
- Long String Concatenation
- Loading…

back to top

Resources

Learning ES6

- Draft ECMA 2015 (ES6) Spec
- ExploringJS
- ES6 Compatibility Table
- Comprehensive Overview of ES6 Features

Read This

- Standard ECMA-262

Tools

- Code Style Linters
  - ESlint - Airbnb Style .eslintrc
  - JSHint - Airbnb Style .jshintrc
  - JSCS - Airbnb Style Preset

Other Style Guides

- Google JavaScript Style Guide
- jQuery Core Style Guidelines
- Principles of Writing Consistent, Idiomatic JavaScript

Other Styles

- Naming this in nested functions - Christian Johansen
- Conditional Callbacks - Ross Allen
- Popular JavaScript Coding Conventions on Github - JeongHoon Byun
- Multiple var statements in JavaScript, not superfluous - Ben Alman

Further Reading

- Understanding JavaScript Closures - Angus Croll
- Basic JavaScript for the impatient programmer - Dr. Axel Rauschmayer
- You Might Not Need jQuery - Zack Bloom & Adam Schwartz
- ES6 Features - Luke Hoban
- Frontend Guidelines - Benjamin De Cock

Books

- JavaScript: The Good Parts - Douglas Crockford
- JavaScript Patterns - Stoyan Stefanov
- Pro JavaScript Design Patterns - Ross Harmes and Dustin Diaz
- High Performance Web Sites: Essential Knowledge for Front-End Engineers - Steve Souders
- Maintainable JavaScript - Nicholas C. Zakas
- JavaScript Web Applications - Alex MacCaw
- Pro JavaScript Techniques - John Resig
- Smashing Node.js: JavaScript Everywhere - Guillermo Rauch
- Secrets of the JavaScript Ninja - John Resig and Bear Bibeault
- Human JavaScript - Henrik Joreteg
- Superhero.js - Kim Joar Bekkelund, Mads Mobæk, & Olav Bjorkoy
- JSBooks - Julien Bouquillon
- Third Party JavaScript - Ben Vinegar and Anton Kovalyov
- Effective JavaScript: 68 Specific Ways to Harness the Power of JavaScript - David Herman
- Eloquent JavaScript - Marijn Haverbeke
- You Don't Know JS: ES6 & Beyond - Kyle Simpson

Blogs

- DailyJS
- JavaScript Weekly
- JavaScript, JavaScript…
- Bocoup Weblog

- Adequately Good
- NCZOnline
- Perfection Kills
- Ben Alman
- Dmitry Baranovskiy
- Dustin Diaz
- nettuts

Podcasts

- JavaScript Jabber

back to top

In the Wild

This is a list of organizations that are using this style guide. Send us a pull request and we'll add you to the list.

- **Aan Zee**: AanZee/javascript
- **Adult Swim**: adult-swim/javascript
- **Airbnb**: airbnb/javascript
- **Apartmint**: apartmint/javascript
- **Ascribe**: You're reading it!
- **Avalara**: avalara/javascript
- **Billabong**: billabong/javascript
- **Blendle**: blendle/javascript
- **ComparaOnline**: comparaonline/javascript
- **Compass Learning**: compasslearning/javascript-style-guide
- **DailyMotion**: dailymotion/javascript
- **Digitpaint** digitpaint/javascript
- **Ecosia**: ecosia/javascript
- **Evernote**: evernote/javascript-style-guide
- **ExactTarget**: ExactTarget/javascript
- **Expensify** Expensify/Style-Guide
- **Flexberry**: Flexberry/javascript-style-guide
- **Gawker Media**: gawkermedia/javascript
- **General Electric**: GeneralElectric/javascript
- **GoodData**: gooddata/gdc-js-style
- **Grooveshark**: grooveshark/javascript
- **How About We**: howaboutwe/javascript
- **Huballin**: huballin/javascript

- **HubSpot**: HubSpot/javascript
- **Hyper**: hyperoslo/javascript-playbook
- **InfoJobs**: InfoJobs/JavaScript-Style-Guide
- **Intent Media**: intentmedia/javascript
- **Jam3**: Jam3/Javascript-Code-Conventions
- **JSSolutions**: JSSolutions/javascript
- **Kinetica Solutions**: kinetica/javascript
- **Mighty Spring**: mightyspring/javascript
- **MinnPost**: MinnPost/javascript
- **MitocGroup**: MitocGroup/javascript
- **ModCloth**: modcloth/javascript
- **Money Advice Service**: moneyadviceservice/javascript
- **Muber**: muber/javascript
- **National Geographic**: natgeo/javascript
- **National Park Service**: nationalparkservice/javascript
- **Nimbl3**: nimbl3/javascript
- **Orion Health**: orionhealth/javascript
- **Peerby**: Peerby/javascript
- **Razorfish**: razorfish/javascript-style-guide
- **reddit**: reddit/styleguide/javascript
- **REI**: reidev/js-style-guide
- **Ripple**: ripple/javascript-style-guide
- **SeekingAlpha**: seekingalpha/javascript-style-guide
- **Shutterfly**: shutterfly/javascript
- **Springload**: springload/javascript
- **StudentSphere**: studentsphere/javascript
- **Target**: target/javascript
- **TheLadders**: TheLadders/javascript
- **T4R Technology**: T4R-Technology/javascript
- **VoxFeed**: VoxFeed/javascript-style-guide
- **Weggo**: Weggo/javascript
- **Zillow**: zillow/javascript
- **ZocDoc**: ZocDoc/javascript

back to top

Contributors

- View Contributors

License

(The MIT License)

Copyright (c) 2014 Airbnb

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the 'Software'), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED 'AS IS', WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

back to top

Amendments

We encourage you to fork this guide and change the rules to fit your team's style guide. Below, you may list some amendments to the style guide. This allows you to periodically update your style guide without having to deal with merge conflicts.

1.3.10 Our Release Process

Notes

BigchainDB follows the Python form of Semantic Versioning (i.e. MAJOR.MINOR.PATCH), which is almost identical to regular semantic versioning, but there's no hyphen, e.g.

- `0.9.0` for a typical final release
- `4.5.2a1` not `4.5.2-a1` for the first Alpha release
- `3.4.5rc2` not `3.4.5-rc2` for Release Candidate 2

Note 1: For Git tags (which are used to identify releases on GitHub), we append a `v` in front. For example, the Git tag for version `2.0.0a1` was `v2.0.0a1`.

Note 2: For Docker image tags (e.g. on Docker Hub), we use longer version names for Alpha, Beta and Release Candidate releases. For example, the Docker image tag for version `2.0.0a2` was `2.0.0-alpha2`.

We use `0.9` and `0.9.0` as example version and short-version values below. You should replace those with the correct values for your new version.

We follow BEP-1, which is our variant of C4, the Collective Code Construction Contract, so a release is just a tagged commit on the `master` branch, i.e. a label for a particular Git commit.

The following steps are what we do to release a new version of *BigchainDB Server*. The steps to release the Python Driver are similar but not the same.

Steps

1. Create a pull request where you make the following changes:
   - Update `CHANGELOG.md`
   - Update all Docker image tags in all Kubernetes YAML files (in the `k8s/` directory). For example, in the files:
     - `k8s/bigchaindb/bigchaindb-ss.yaml` and
     - `k8s/dev-setup/bigchaindb.yaml`

     find the line of the form `image: bigchaindb/bigchaindb:0.8.1` and change the version number to the new version number, e.g. `0.9.0`. (This is the Docker image that Kubernetes should pull from Docker Hub.) Keep in mind that this is a *Docker image tag* so our naming convention is a bit different; see Note 2 in the **Notes** section above.
   - In `bigchaindb/version.py`:
     - update `__version__` to e.g. `0.9.0` (with no `.dev` on the end)
     - update `__short_version__` to e.g. `0.9` (with no `.dev` on the end)
   - In the docs about installing BigchainDB (and Tendermint), and in the associated scripts, recommend/install a version of Tendermint that *actually works* with the soon-to-be-released version of BigchainDB. You can find all such references by doing a search for the previously-recommended version number, such as `0.22.8`.
   - In `setup.py`, *maybe* update the development status item in the `classifiers` list. For example, one allowed value is `"Development Status :: 5 - Production/Stable"`. The allowed values are listed at pypi.python.org.
2. **Wait for all the tests to pass!**
3. Merge the pull request into the `master` branch.
4. Go to the bigchaindb/bigchaindb Releases page on GitHub and click the "Draft a new release" button.
5. Fill in the details:
   - **Tag version:** version number preceded by `v`, e.g. `v0.9.1`
   - **Target:** the last commit that was just merged. In other words, that commit will get a Git tag with the value given for tag version above.
   - **Title:** Same as tag version above, e.g `v0.9.1`
   - **Description:** The body of the changelog entry (Added, Changed, etc.)
6. Click "Publish release" to publish the release on GitHub.
7. On your local computer, make sure you're on the `master` branch and that it's up-to-date with the `master` branch in the bigchaindb/bigchaindb repository (e.g. `git pull upstream master`). We're going to use that to push a new `bigchaindb` package to PyPI.
8. Make sure you have a `~/.pypirc` file containing credentials for PyPI.
9. Do `make release` to build and publish the new `bigchaindb` package on PyPI. For this step you need to have `twine` installed. If you get an error like `Makefile:135: recipe for target 'clean-pyc' failed` then try doing

```
sudo chown -R $(whoami):$(whoami) .
```

10. Log in to readthedocs.org and go to the **BigchainDB Server** project, then:
    - Click on "Builds", select "latest" from the drop-down menu, then click the "Build Version:" button.
    - Wait for the build of "latest" to finish. This can take a few minutes.
    - Go to Admin –> Advanced Settings and make sure that "Default branch:" (i.e. what "latest" points to) is set to the new release's tag, e.g. `v0.9.1`. (It won't be an option if you didn't wait for the build of "latest" to finish.) Then scroll to the bottom and click "Save".
    - Go to Admin –> Versions and under **Choose Active Versions**, do these things:
      (a) Make sure that the new version's tag is "Active" and "Public"
      (b) Make sure the **stable** branch is *not* active.
      (c) Scroll to the bottom of the page and click "Save".
11. Go to Docker Hub and sign in, then:
    - Click on "Organizations"
    - Click on "bigchaindb"
    - Click on "bigchaindb/bigchaindb"
    - Click on "Build Settings"
    - Find the row where "Docker Tag Name" equals `latest` and change the value of "Name" to the name (Git tag) of the new release, e.g. `v0.9.0`.
    - If the release is an Alpha, Beta or Release Candidate release, then a new row must be added. You can do that by clicking the green "+" (plus) icon. The contents of the new row should be similar to the existing rows of previous releases like that.
    - Click on "Tags"
    - Delete the "latest" tag (so we can rebuild it)
    - Click on "Build Settings" again
    - Click on the "Trigger" button for the "latest" tag and make sure it worked by clicking on "Tags" again
    - If the release is an Alpha, Beta or Release Candidate release, then click on the "Trigger" button for that tag as well.

Congratulations, you have released a new version of BigchainDB Server!