ICO smart contracts Documentation

Release 0.1

Mikko Ohtamaa

Feb 01, 2019

Contents:

This is a documentation for ICO package providing Ethereum smart contracts and Python based command line tools for launching your ICO crowdsale or token offering.

ICO stands for a token or cryptocurrency initial offering crowdsale. It is a common method in blockchain space, decentralized applications and in-game tokens for bootstrap funding of your project.

This project aims to provide standard, secure smart contracts and tools to create crowdsales for Ethereum blockchain.

CHAPTER 1

Introduction

1.1 About

This package contains Ethereum smart contracts and command line toolchain for launching and managing token sales.

1.2 Links

STO - security token tool chain - please note that this project is being phased out in the favor of the upgraded STO tool

TokenMarket website

Github issue tracker and source code

Documentation

1.3 About the project

ICO stands for a token or cryptocurrency initial offering crowdsale. It is a common method in blockchain space, decentralized applications and in-game tokens for bootstrap funding of your project.

This project aims to provide standard, secure smart contracts and tools to create crowdsales for Ethereum blockchain.

As the writing of this, Ethereum smart contract ICO business has been booming almost a year. The industry and development teams are still figuring out the best practices. A lot of similar smart contracts get written over and over again. This project aims to tackle this problem by providing reusable ICO codebase, so that developers can focus on their own project specific value adding feature instead of rebuilding core crowdfunding logic. Having one well maintained codebase with best practice and security audits benefits the community as a whole.

This package provides

- Crowdsale contracts: token, ICO, uncapped ICO, pricing, transfer lock ups, token upgrade in Solidity smart contract programming language
- Automated test suite in Python
- Deployment tools and scripts

1.4 Token sales

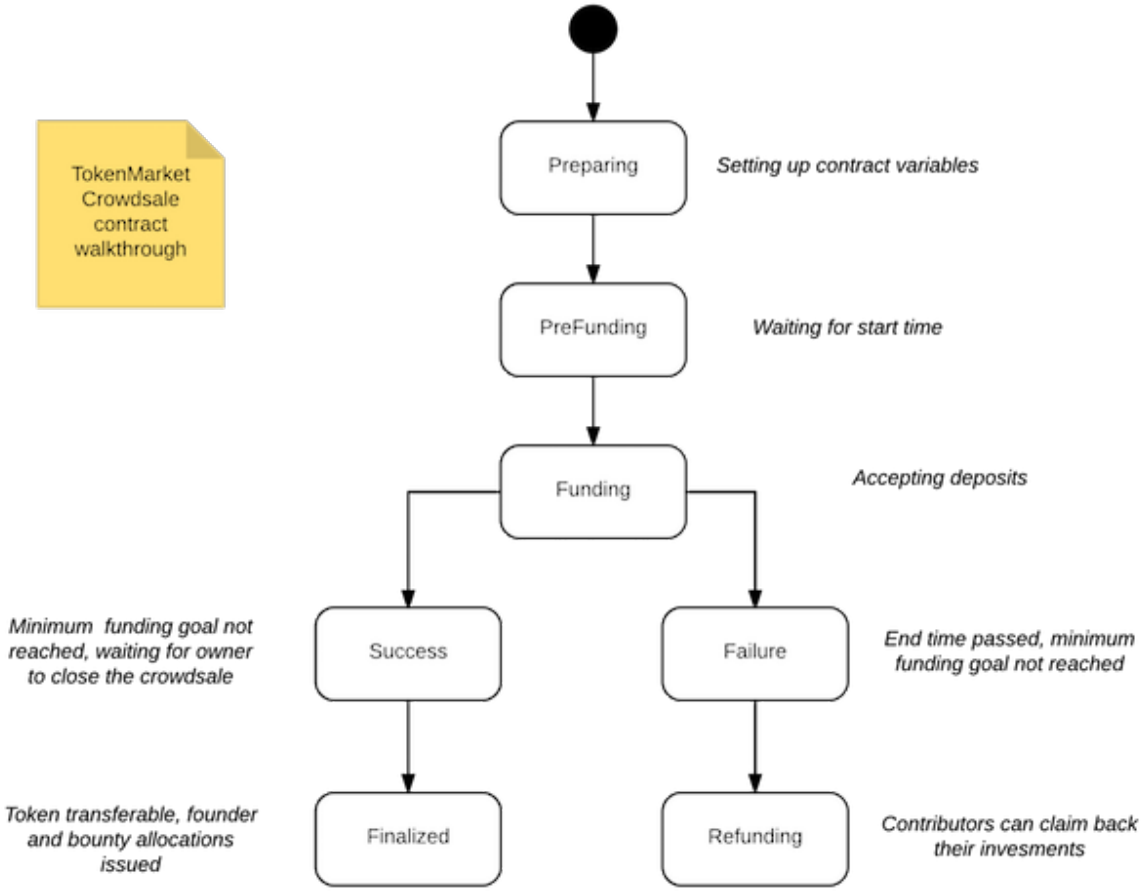
These contracts have been tested, audited and used by several projects. Below are some notable token sales that we have used these contracts

- AppCoins
- Civic
- Storj
- Monaco
- DENT
- Ethos
- ixLedger
- . . . and many more!

We also have third party token sales using these smart contracts

- Dala

1.5 Quick token sale walkthrough



1.6 Features and design goals

- **Best practices**: Smart contracts are written with the modern best practices of Ethereum community
- **KYC**: Know your customer processes are support enabled to minimize legal risks associated with anonymous payments - see KYCCrowdsale
- **AML**: Anti-money laundering processes are supported through offloaded chain analysis - often a requirement to open a bank account - see AMLToken
- **Separation of concerns**: Crowdsale, token and other logic lies in separate contracts that can be assembled together like lego bricks
- **Testable**: We aim for 100% branch code coverage by automated test suite
- **Auditable**: Our tool chain supports verifiable EtherScan.io contract builds
- **Reusable**: The contract code is modularized and reusable across different projects, all variables are parametrized and there are no hardcoded values or magic numbers
- **Refund**: Built-in refund and minimum funding goal protect investors

- **Token upgrade**: Token holders can opt in to a new version of the token contract in the case the token owner wants to add more functionality to their token
- **Reissuance**: There can be multiple crowdsales for the same token (pre-ICO, ICO, etc.)
- **Emergency stop**: To try to save the situation in the case we found an issue in the contract post-deploy
- **Build upon a foundation**: Instead of building everything from the scratch, use OpenZeppelin contracts as much as possible as they are the gold standard of Solidity development

1.7 Support

TokenMarket can be a launch and hosting partner for your token sale. We offer advisory, legal, technical and marketing services. For more information see TokenMarket ICO services. TokenMarket requires everyone to have at least business plan or whitepaper draft ready before engaging into any discussions.

Community support is available on the best effort basis - your mileage may vary. To get the most of the community support we expect you to be on a senior level of Solidity, Python and open source development. Meet us at the Gitter support chat.

1.8 Running tests locally

Quick tutorial (see docs for more information):

```
export SOLC_BINARY=$(pwd)/dockerized-solc.sh
export SOLC_VERSION=0.4.24
tox
```

1.9 Audit reports

Some public audit reports available for some revisions of this codebase:

- For Atonomi by LevelK, May 2018
- For Dala by Iosiro, October 2017
- For Civic by Zeppelin, June 2017

More audit reports available on a request.

CHAPTER 2

Contracts

2.1 Introduction

This chapter describers Ethereum crowdsale smart contracts.

2.2 Preface

- You must understand Ethereum blockchain and Solidity smart contract programming basics
- You must have a running Ethereum full node with JSON-RPC interface enabld

2.3 TODO

CHAPTER 3

Installation

3.1 Preface

Instructions are written in OSX and Linux in mind.

Experience needed

- Basic command line usage
- Basic Github usage

3.2 Setting up - OSX

Packages needed

- Populus native dependencies

Get Solidity compiler. Use version 0.4.12+. For OSX:

```
brew install solidity
```

Clone this repository from Github using submodules:

```
git clone --recursive git@github.com:TokenMarketNet/ico.git
```

Python 3.5+ required. See installing Python.

```
python3.5 --version
Python 3.5.2
```

Create virtualenv for Python package management in the project root folder (same as where `setup.py` is):

```
python3.5 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
pip install -e .
```

3.3 Setting up - Ubuntu Linux 16.04

Install dependencies:

```
sudo apt install -y git build-essential libssl-dev python3 python3-venv python3-
↪setuptools python3-dev cmake libboost-all-dev
```

Python 3.5+ required. Make sure you have a compatible version:

```
python3.5 --version
Python 3.5.2
```

Install Solidity solc compiler:

```
sudo apt install software-properties-common
sudo add-apt-repository -y ppa:ethereum/ethereum
sudo apt update
sudo apt install -y ethereum solc
```

Then install `ico` Python package and its dependencies:

```
git clone --recursive git@github.com:TokenMarketNet/ico.git
cd ico
python3.5 -m venv venv
source venv/bin/activate
pip install wheel
pip install -r requirements.txt
pip install -e .
```

3.4 Installing Ethereum node (geth or parity)

You need to have Go Ethereum (geth), Parity or some other mean to communicate with Ethereum blockchain.

The default set up assumes you run JSON-RPC in *http://localhost:8545* for mainnnet and *http://localhost:8547* for Kovan testnet.

For more information see *chain configuration*.

3.5 Using your desired Solidity version

We recommend using Docker and official Ethereum Solidity docker builds as the static binary like installation for the compiler.

Example:

```
# This is a supplied shell script wrapper that is honoured by Populus and py-solc
↪packages when they look for solc binary
export SOLC_BINARY=`pwd`/dockerized-solc.sh

# Give the Solidity version we want to use for our Docker wrapper scripts
export SOLC_VERSION=0.4.18

# Populus now uses Dockerized solc. Any missing version is automatically downloaded
↪and cached.
populus compile
```

Note: Docker volume mounts do not support symbolic links and thus this kind of *solc* alias behavior might be differnt from having natively installed solc.

3.6 Docker Ganache image

TokenMarket contracts can optionally be built, run, and tested using Docker (https://www.docker.com/). To be able to TokenMarket development environment inside Docker, install Docker and docker-compose (https://docs.docker.com/compose/) first. Then run in ico folder:

```
docker-compose up
```

If everything is ok, you will see something like below:

```
MacBook-Pro-mac:docs mac$ docker-compose up

WARNING: The Docker Engine you're using is running in swarm mode.

Compose does not use swarm mode to deploy services to multiple nodes in a swarm. All
↪containers will be scheduled on the current node.

To deploy your application across the swarm, use `docker stack deploy`.

Starting ganache-cli ... done
Starting tkn ... done
Attaching to ganache-cli, tkn
ganache-cli | Ganache CLI v6.1.0-beta.1 (ganache-core: 2.1.0-beta.1)
ganache-cli |
ganache-cli | Available Accounts
ganache-cli | ==================
ganache-cli | (0) 0xab2d52942a9875143e94e9fe09a548a45dceb1e8
ganache-cli | (1) 0xdc4b3cc214b77407ef77f3fa38108a2de48d0cf7
ganache-cli | (2) 0xaf98b165c2dcadc8e17a717b795ee6dcacf0d306
ganache-cli | (3) 0xeeb5e1c68201d2fc58e07a2c3619377ea742d0ad
ganache-cli | (4) 0xa050538c2203055a82bdfc18004c872095283362
ganache-cli | (5) 0x7b3fe777be5e6b49b3580657ad3792d55e31d0f7
```

(continues on next page)

(continued from previous page)

```
ganache-cli | (6) 0xc198cf10296d1ed5df408f94890fd57dbad4750c
ganache-cli | (7) 0xf2dc5b1b4ba8465aac47484ae9dd0ff09844cc27
ganache-cli | (8) 0xe8431646004065981552516548 7d436f047fad78
ganache-cli | (9) 0x1be235ca98cd4a56be34218e8b3265be11bd3f0a
ganache-cli |
ganache-cli | Private Keys
ganache-cli | ==================
ganache-cli | (0) 29b65e26c903d588f5706d7850cf125f78bef030a993b2a36db859e9f1a4ac3e
ganache-cli | (1) c7b0146725f16d0e261289e1183304e2f829990bafd695d444b93af995e5c7d7
ganache-cli | (2) 2dfb4b4e054cc9881ee1170ce5278c65b52e9a5e2afa1f2882376adcd4a339af
ganache-cli | (3) 00e9470ce3c13cbdbc60e4f2a6c284245ff47a3595d139bef6e04ab3007097e3
ganache-cli | (4) 613d14fb4045ee80a30649bee4c75d82b7478dab2e834e544e8d4eda8da0915c
ganache-cli | (5) 8705cfda49b76911fb74ce2b1c704f172070b95e75e4c467e08b99142d531c06
ganache-cli | (6) 0acaf2b8a74aac3a38406e6a4bc4f6229c2130d1d9e526c7f7a56d5b35e93244
ganache-cli | (7) b3d28e482d9e1aa3ae696b7f20261200bc077f4771bdb4e202278256b3e94575
ganache-cli | (8) 3e89a5e223e0919b2b0b61c71590af0f6e96fb0a1c82e0e3ec7a390314b7ded3
ganache-cli | (9) 6bc7b7209dd5a06cf89876efece6dfd6524f49df039d822d15beaac91afb4d37
ganache-cli |
ganache-cli | HD Wallet
ganache-cli | ==================
ganache-cli | Mnemonic:      great lunch cushion melt remind harvest taxi prosper␣
↪hawk ahead split reopen
ganache-cli | Base HD Path:  m/44'/60'/0'/0/{account_index}
ganache-cli |
ganache-cli | Listening on localhost:8545
ganache-cli | eth_getBalance
```

To login into dockerized TokenMarket environment:

```
docker exec -it tkn /bin/bash
```

To deploy contract from inside dockerized ico environment (example for Ganache chain address 0xab2d52942a9875143e94e9fe09a548a45dceb1e8):

```
python3 ico/cmd/deploycontracts.py --deployment-file crowdsales/crowdsale-token-
↪example-ganache.yml --deployment-name local-token --address␣
↪0xab2d52942a9875143e94e9fe09a548a45dceb1e8
```

The following folders & filles are mapped as volumes so you can edit them from outside Docker and compile/run tests inside Docker:

```
contracts
crowdsales
zeppelin
ico
populus.json
```

CHAPTER 4

Command line commands

4.1 Introduction

`ico` package provides tooling around deploying and managing token sales and related tasks.

Here are listed some of the available command line commands. For full list see setup.py `[console-scripts]` section.

All commands read *populus.json* file for the chain configuration from the current working directory. The chain configuration should set up a Web3 HTTP provider how command line command talks to an Ethereum node. The Ethereum node must have an address with ETH balance for the operations. For more information see *Chain configuration*.

The most important command is *deploy-contracts* that allows scripted and orchestrated deployment of multiple related Ethereum smart contracts.

4.2 deploy-contracts

Scripted deployment of multiple related Ethereum smart contracts.

- Deploy contracts

- Automatically verify contracts on EtherScan
- Link contracts together
- Set common parameters
- Verify contracts have been deployed correctly through assert mechanism

See also *Contract source code verification*.

Example YAML deployment scripts

- allocated-token-sale (based on DENT)
- dummy mintable token sale example

Help:

```
Usage: deploy-contracts [OPTIONS]

  Makes a scripted multiple contracts deployed based on a YAML file.

  Reads the chain configuration information from populus.json. The resulting
  deployed contracts can be automatically verified on etherscan.io.

  Example files:

  * https://github.com/TokenMarketNet/ico/blob/master/crowdsales/crowdsale-
  token-example.yml

  * https://github.com/TokenMarketNet/ico/blob/master/crowdsales/allocated-
  token-sale-example.yml

  * https://github.com/TokenMarketNet/ico/blob/master/crowdsales/example.yml

Options:
  --deployment-name TEXT  Project section id inside the YAML file. The topmost
                          YAML key. Example YAML files use "mainnet" or
                          "kovan".  [required]
  --deployment-file TEXT  Deployment script YAML .yml file to process
                          [required]
  --address TEXT          Your Ethereum account that is the owner of
                          deployment and pays the gas cost. This account must
                          exist on Ethereum node we connect to. Connection
                          parameteres, port and IP, are defined in
                          populus.json.  [required]
  --help                  Show this message and exit.
```

4.3 deploy-token

Deploy a single token contract.

Warning: This command is depracated. Instead, use deploy-contracts command. See example here.

Example usage:

```
deploy-token --help
Usage: deploy-token [OPTIONS]

  Deploy a single crowdsale token contract.

  Examples:

  deploy-token --chain=ropsten
  --address=0x3c2d4e5eae8c4a31ccc56075b5fd81307b1627c6 --name="MikkoToken
  2.0" --symbol=MOO --release-
  agent=0x3c2d4e5eae8c4a31ccc56075b5fd81307b1627c6  --supply=100000

  deploy-token --chain=kovan --contract-name="CentrallyIssuedToken"
  --address=0x001FC7d7E506866aEAB82C11dA515E9DD6D02c25  --name="TestToken"
  --symbol=MOO --supply=916 --decimals=0 --verify --verify-
  filename=CentrallyIssuedToken.sol

Options:
  --chain TEXT            On which chain to deploy - see populus.json
  --address TEXT          Address to deploy from and who becomes as a owner
                          (must exist on geth)  [required]
  --contract-name TEXT    Name of the token contract
  --release-agent TEXT    Address that acts as a release agent (can be same as
                          owner)
  --minting-agent TEXT    Address that acts as a minting agent (can be same as
                          owner)
  --name TEXT             Token name  [required]
  --symbol TEXT           Token symbol  [required]
  --supply INTEGER        Initial token supply (multipled with decimals)
  --decimals INTEGER      How many decimal points the token has
  --verify / --no-verify  Verify contract on EtherScan.io
  --verify-filename TEXT  Solidity source file of the token contract for
                          verification
  --master-address TEXT   Move tokens and upgrade master to this account
  --help                  Show this message and exit.
```

4.4 distribute-tokens

Help:

```
Usage: distribute-tokens [OPTIONS]

  Distribute tokens to centrally issued crowdsale participant or bounty
  program participants.

  Reads in distribution data as CSV. Then uses Issuer contract to distribute
  tokens. All token counts are multiplied by token contract decimal
  specifier. E.g. if CSV has amount 15.5, token has 2 decimal places, we
  will issue out 1550 raw token amount.

  To speed up the issuance, transactions are verified in batches. Each batch
  is 16 transactions at a time.

  Example (first run):
```

(continues on next page)

(continued from previous page)

```
      distribute-tokens --chain=kovan
      --address=0x001FC7d7E506866aEAB82C11dA515E9DD6D02c25
      --token=0x1644a421ae0a0869bac127fa4cce8513bd666705 --master-
      address=0x9a60ad6de185c4ea95058601beaf16f63742782a --csv-
      file=input.csv --allow-zero --address-column="Ethereum address"
      --amount-column="Token amount"

  Example (second run, continue after first run was interrupted):

      distribute-tokens --chain=kovan
      --address=0x001FC7d7E506866aEAB82C11dA515E9DD6D02c25
      --token=0x1644a421ae0a0869bac127fa4cce8513bd666705 --csv-
      file=input.csv --allow-zero --address-column="Ethereum address"
      --amount-column="Token amount" --issuer-
      address=0x2c9877534f62c8b40aebcd08ec9f54d20cb0a945

Options:
  --chain TEXT                    On which chain to deploy - see populus.json
  --address TEXT                  The account that deploys the issuer
                                  contract, controls the contract and pays for
                                  the gas fees  [required]
  --token TEXT                    Token contract address  [required]
  --csv-file TEXT                 CSV file containing distribution data
                                  [required]
  --address-column TEXT           Name of CSV column containing Ethereum
                                  addresses
  --amount-column TEXT            Name of CSV column containing decimal token
                                  amounts
  --limit INTEGER                 How many items to import in this batch
  --start-from INTEGER            First row to import (zero based)
  --issuer-address TEXT           The address of the issuer contract - leave
                                  out for the first run to deploy a new issuer
                                  contract
  --master-address TEXT           The team multisig wallet address that does
                                  StandardToken.approve() for the issuer
                                  contract
  --allow-zero / --no-allow-zero  Stops the script if a zero amount row is
                                  encountered
  --help                          Show this message and exit.
```

4.5 token-vault

Help:

```
token-vault --help
Usage: token-vault [OPTIONS]

  TokenVault control script.

  1) Deploys a token vault contract

  2) Reads in distribution data as CSV

  3) Locks vault
```

(continues on next page)

(continued from previous page)

```
Options:
  --action TEXT                   One of: deploy, load, lock
  --chain TEXT                    On which chain to deploy - see populus.json
  --address TEXT                  The account that deploys the vault contract,
                                  controls the contract and pays for the gas
                                  fees  [required]
  --token-address TEXT            Token contract address  [required]
  --csv-file TEXT                 CSV file containing distribution data
  --address-column TEXT           Name of CSV column containing Ethereum
                                  addresses
  --amount-column TEXT            Name of CSV column containing decimal token
                                  amounts
  --limit INTEGER                 How many items to import in this batch
  --start-from INTEGER            First row to import (zero based)
  --vault-address TEXT            The address of the vault contract - leave
                                  out for the first run to deploy a new issuer
                                  contract
  --freeze-ends-at INTEGER        UNIX timestamp when vault freeze ends for
                                  deployment
  --tokens-to-be-allocated INTEGER
                                  Manually verified count of tokens to be set
                                  in the vault
  --help                          Show this message and exit.
```

4.6 combine-csvs

Help:

```
combine-csvs --help
Usage: combine-csvs [OPTIONS]

  Combine multiple token distribution CSV files to a single CSV file good
  for an Issuer contract.

  - Input is a CSV file having columns Ethereum address, number of tokens

  - Round all tokens to the same decimal precision

  - Combine multiple transactions to a single address to one transaction

  Example of cleaning up one file:

      combine-csvs --input-file=csvs/bounties-unclean.csv --output-
      file=combine.csv --decimals=8 --address-column="address" --amount-
      column="amount"

  Another example - combine all CSV files in a folder using zsh shell:

      combine-csvs csvs/*.csv(P:--input-file:) --output-file=combined.csv
      --decimals=8 --address-column="Ethereum address" --amount-
      column="Total reward"

Options:
```

(continues on next page)

(continued from previous page)

```
  --input-file TEXT      CSV file to read and combine. It should be given
                         multiple times for different files.  [required]
  --output-file TEXT     A CSV file to write the output  [required]
  --decimals INTEGER     A number of decimal points to use  [required]
  --address-column TEXT  Name of CSV column containing Ethereum addresses
  --amount-column TEXT   Name of CSV column containing decimal token amounts
  --help                 Show this message and exit.
```

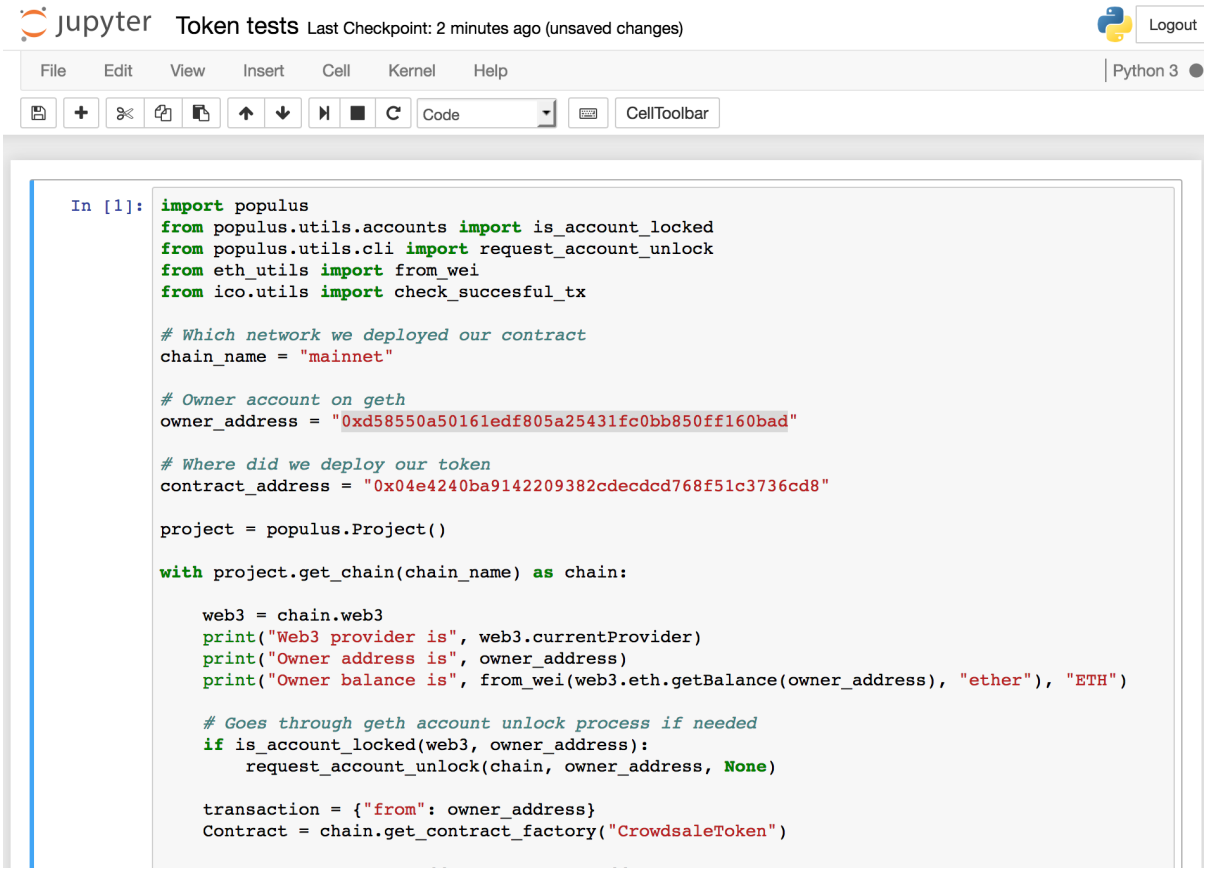
CHAPTER 5

Interacting with deployed smart contracts

5.1 Introduction

This chapter shows how one can interact with deployed smart contracts.

Interaction is easiest through a Jupyter Notebook console where you can edit and run script snippets.



All snippets will connect to Ethereum node through a JSON RPC provider that has been configured in `populus.json`.

5.1.1 Getting Jupyter Notebook

Install it with *pip* in the activated Python virtual environment:

```
pip install jupyter
```

Then start Jupyter Notebook:

```
jupyter notebook
```

5.2 Transferring tokens

Example:

```
from decimal import Decimal
import populus
from populus.utils.accounts import is_account_locked
from populus.utils.cli import request_account_unlock
from eth_utils import from_wei
from ico.utils import check_succesful_tx
from ico.utils import get_contract_by_name
```

(continues on next page)

(continued from previous page)

```
# Which network we deployed our contract
chain_name = "mainnet"

# Owner account on geth
owner_address = "0x"

# Where did we deploy our token
contract_address = "0x"

receiver = "0x"

amount = Decimal("1.0")

project = populus.Project()

with project.get_chain(chain_name) as chain:

    web3 = chain.web3
    print("Web3 provider is", web3.currentProvider)
    print("Owner address is", owner_address)
    print("Owner balance is", from_wei(web3.eth.getBalance(owner_address), "ether"),
↪"ETH")

    # Goes through geth account unlock process if needed
    if is_account_locked(web3, owner_address):
        request_account_unlock(chain, owner_address, None)

    transaction = {"from": owner_address}
    FractionalERC20 = get_contract_by_name(chain, "FractionalERC20")

    token = FractionalERC20(address=contract_address)
    decimals = token.call().decimals()
    decimal_multiplier = 10 ** decimals

    print("Token has", decimals, "decimals")
    print("Owner token balance is", token.call().balanceOf(owner_address) / decimal_
↪multiplier)

    # Use lowest denominator amount
    normalized_amount = int(amount * decimal_multiplier)

    # Transfer the tokens
    txid = token.transact({"from": owner_address}).transfer(receiver, normalized_
↪amount)
    print("TXID is", txid)
    check_succesful_tx(web3, txid)
```

5.3 Releasing a token

See *deploy-contracts* example how to deploy crowdsale token contracts that have a transfer lock up. The crowdsale tokens cannot be transferred until the release agent makes the token transferable. As we set our owner address as the release agent we can do this from Python console.

Then copy and edit the following snippet with your address information:

```
import populus
from populus.utils.accounts import is_account_locked
from populus.utils.cli import request_account_unlock
from eth_utils import from_wei
from ico.utils import check_succesful_tx
from ico.utils import get_contract_by_name

# Which network we deployed our contract
chain_name = "ropsten"

# Owner account on geth
owner_address = "0x3c2d4e5eae8c4a31ccc56075b5fd81307b1627c6"

# Where did we deploy our token
contract_address = "0x513a7437d355293ac92d6912d9a8b257a343fb36"

project = populus.Project()

with project.get_chain(chain_name) as chain:

    web3 = chain.web3
    print("Web3 provider is", web3.currentProvider)
    print("Owner address is", owner_address)
    print("Owner balance is", from_wei(web3.eth.getBalance(owner_address), "ether"),
↪"ETH")

    # Goes through geth account unlock process if needed
    if is_account_locked(web3, owner_address):
        request_account_unlock(chain, owner_address, None)

    transaction = {"from": owner_address}
    Contract = get_contract_by_name(chain, "CrowdsaleToken")

    contract = Contract(address=contract_address)
    print("Attempting to release the token transfer")
    txid = contract.transact(transaction).releaseTokenTransfer()
    print("TXID", txid)
    check_succesful_tx(web3, txid)
    print("Token released")
```

5.4 Transfering tokens

We have deployed a crowdsale token and made it transferable as above. Now let's transfer some tokens to our friend in Ropsten testnet.

- We create a Ropsten testnet wallet on MyEtherWallet.com - in this example our MyEtherWallet address is 0x47FcAB60823D13B73F372b689faA9D3e8b0C48b5
- We include our deployed token contract there through *Add Custom Token* button
- Now let's transfer some tokens into this wallet through IPython console from our owner account

```
import populus
from populus.utils.accounts import is_account_locked
from populus.utils.cli import request_account_unlock
from eth_utils import from_wei
```

(continues on next page)

(continued from previous page)

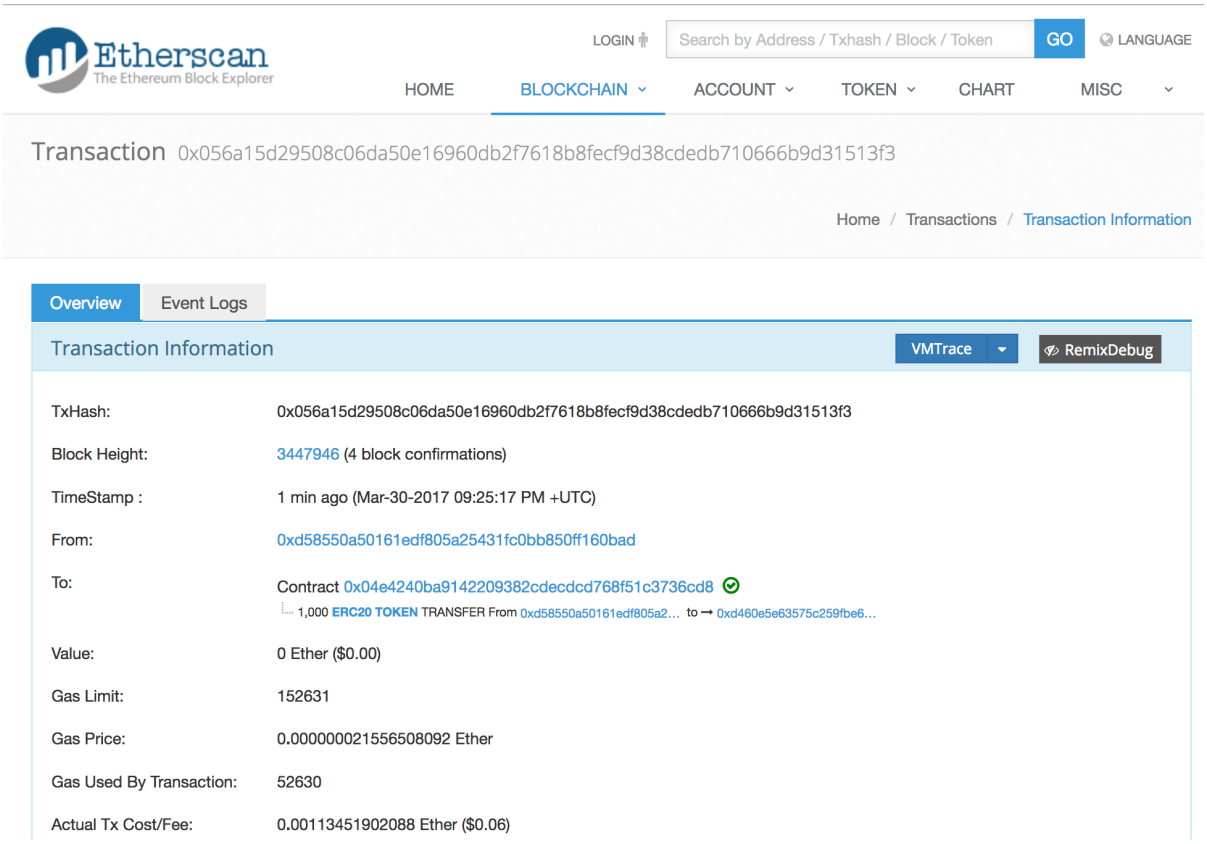
```
from ico.utils import check_succesful_tx
from ico.utils import get_contract_by_name

# Which network we deployed our contract
chain_name = "ropsten"

# Owner account on geth
owner_address = "0x3c2d4e5eae8c4a31ccc56075b5fd81307b1627c6"

# Where did we deploy our token
contract_address = "0x513a7437d355293ac92d6912d9a8b257a343fb36"

# The address where we are transfering tokens into
buddy_address = "0x47FcAB60823D13B73F372b689faA9D3e8b0C48b5"

# How many tokens we transfer
amount = 1000

project = populus.Project()

with project.get_chain(chain_name) as chain:

    Contract = get_contract_by_name(chain, "CrowdsaleToken")
    contract = Contract(address=contract_address)

    web3 = chain.web3
    print("Web3 provider is", web3.currentProvider)
    print("Owner address is", owner_address)
    print("Owner balance is", from_wei(web3.eth.getBalance(owner_address), "ether"),
↪"ETH")
    print("Owner token balance is", contract.call().balanceOf(owner_address))

    # Goes through geth account unlock process if needed
    if is_account_locked(web3, owner_address):
        request_account_unlock(chain, owner_address, None)

    transaction = {"from": owner_address}

    print("Attempting to transfer some tokens to our MyEtherWallet account")
    txid = contract.transact(transaction).transfer(buddy_address, amount)
    check_succesful_tx(web3, txid)
    print("Transfered", amount, "tokens to", buddy_address, "in transaction https://
↪ropsten.etherscan.io/tx/{}".format(txid))
```

We get output like:

```
Web3 provider is RPC connection http://127.0.0.1:8546
Owner address is 0x3c2d4e5eae8c4a31ccc56075b5fd81307b1627c6
Owner balance is 1512.397773239968990885 ETH
Owner token balance is 99000
Attempting to transfer some tokens to our MyEtherWallet account
Transfered 1000 tokens to 0x47FcAB60823D13B73F372b689faA9D3e8b0C48b5 in transaction␣
↪https://ropsten.etherscan.io/tx/
↪0x5460742a4f40dd573aeaddede95fc57fff6de800dde9494520c4f7852d7a956d
```

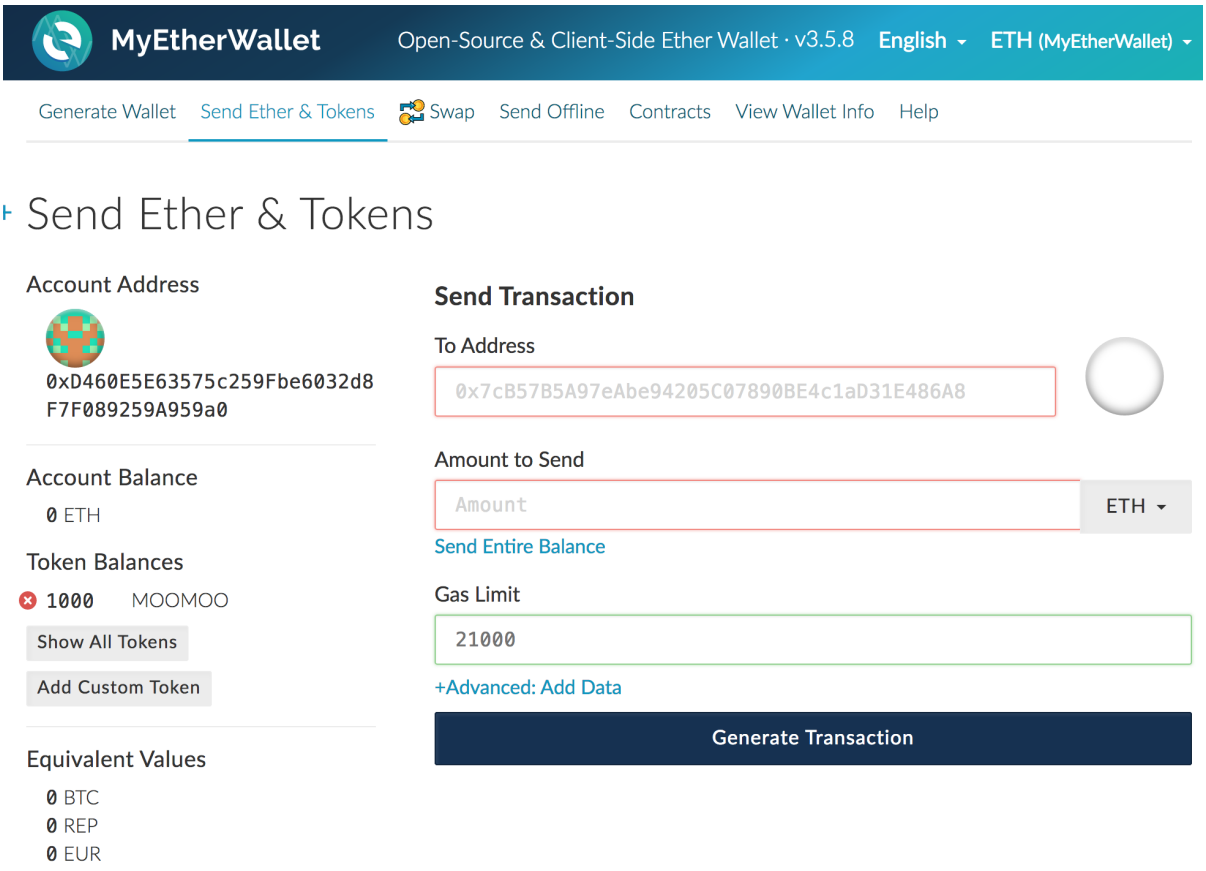
5.4.1 Etherscan transfer confirmation

We can see the transaction in the blockchain explorer:



5.4.2 MyEtherWallet transfer confirmation

And then finally we see tokens in our MyEtherWallet:



5.5 Setting the actual ICO contract for a pre-ICO contract

Example setting the ICO contract for a presale:

```
from ico.utils import check_succesful_tx
from ico.utils import get_contract_by_name
import populus
from populus.utils.cli import request_account_unlock
from populus.utils.accounts import is_account_locked

p = populus.Project()
account = "0xd58550a50161edf805a25431fc0bb850ff160bad"

with p.get_chain("mainnet") as chain:
    web3 = chain.web3
    Contract = get_contract_by_name(chain, "PresaleFundCollector")
    contract = Contract(address="0x858759541633d5142855b27f16f5f67ea78654bf")

    if is_account_locked(web3, account):
        request_account_unlock(chain, account, None)

    txid = contract.transact({"from": account}).setCrowdsale(
↪"0xb57d88c2f70150cb688da7b1d749f1b1b4d72f4c")
    print("TXID is", txid)
    check_succesful_tx(web3, txid)
    print("OK")
```

Example triggering the funds transfer to ICO:

```
from ico.utils import check_succesful_tx
from ico.utils import get_contract_by_name
import populus
from populus.utils.cli import request_account_unlock
from populus.utils.accounts import is_account_locked

p = populus.Project()
account = "0xd58550a50161edf805a25431fc0bb850ff160bad"

with p.get_chain("mainnet") as chain:
    web3 = chain.web3
    Contract = get_contract_by_name(chain, "PresaleFundCollector")
    contract = Contract(address="0x858759541633d5142855b27f16f5f67ea78654bf")

    if is_account_locked(web3, account):
        request_account_unlock(chain, account, None)

    txid = contract.transact({"from": account}).participateCrowdsaleAll()
    print("TXID is", txid)
    check_succesful_tx(web3, txid)
    print("OK")
```

5.6 Whitelisting crowdsale participants

Here is an example how to whitelist ICO participants before the ICO beings:

```
from ico.utils import check_succesful_tx
from ico.utils import get_contract_by_name
import populus
from populus.utils.cli import request_account_unlock
from populus.utils.accounts import is_account_locked

p = populus.Project()
account = "0x001FC7d7E506866aEAB82C11dA515E9DD6D02c25"  # Our controller account on␣
↪Kovan

with p.get_chain("kovan") as chain:
    web3 = chain.web3
    Contract = get_contract_by_name(chain, "Crowdsale")
    contract = Contract(address="0x06829437859594e19276f87df601436ef55af4f2")

    if is_account_locked(web3, account):
        request_account_unlock(chain, account, None)

    txid = contract.transact({"from": account}).setEarlyParicipantWhitelist(
↪"0x65cbd9a48c366f66958196b0a2af81fc73987ba3", True)
    print("TXID is", txid)
    check_succesful_tx(web3, txid)
    print("OK")
```

5.7 Change pricing strategy

To mix fat finger errors:

```
from ico.utils import check_succesful_tx
from ico.utils import get_contract_by_name
import populus
from populus.utils.cli import request_account_unlock
from populus.utils.accounts import is_account_locked

p = populus.Project()
account = "0x"  # Our controller account on Kovan

with p.get_chain("mainnet") as chain:
    web3 = chain.web3
    Contract = get_contract_by_name(chain, "Crowdsale")
    contract = Contract(address="0x")

    if is_account_locked(web3, account):
        request_account_unlock(chain, account, None)

    txid = contract.transact({"from": account}).setPricingStrategy("0x")
    print("TXID is", txid)
    check_succesful_tx(web3, txid)
    print("OK")
```

5.8 Test buy token

Try to buy from a whitelisted address or on a testnet with a generated customer id:

```
from ico.utils import check_succesful_tx
from ico.utils import get_contract_by_name
import populus
from populus.utils.cli import request_account_unlock
from populus.utils.accounts import is_account_locked
from eth_utils import to_wei

import uuid

p = populus.Project()
account = "0x"  # Our controller account on Kovan

with p.get_chain("kovan") as chain:
    web3 = chain.web3
    Contract = get_contract_by_name(chain, "Crowdsale")
    contract = Contract(address="0x")

    if is_account_locked(web3, account):
        request_account_unlock(chain, account, None)

    customer_id = int(uuid.uuid4().hex, 16)  # Customer ids are 128-bit UUID v4

    txid = contract.transact({"from": account, "value": to_wei(2, "ether")}).buy()
    print("TXID is", txid)
```

(continues on next page)

(continued from previous page)

```
    check_succesful_tx(web3, txid)
    print("OK")
```

5.9 Halt payment forwarder

After a token sale is ended, stop ETH payment forwarder.

```
from ico.utils import check_succesful_tx
from ico.utils import get_contract_by_name
import populus
from populus.utils.cli import request_account_unlock
from populus.utils.accounts import is_account_locked
from eth_utils import to_wei

import uuid

p = populus.Project()
account = "0x"  # Our controller account on Kovan

with p.get_chain("mainnet") as chain:
    web3 = chain.web3
    Contract = get_contract_by_name(chain, "PaymentForwarder")
    contract = Contract(address="0x")

    if is_account_locked(web3, account):
        request_account_unlock(chain, account, None)

    initial_gas_price = web3.eth.gasPrice
    txid = contract.transact({"from": account, "gasPrice": initial_gas_price*5}).
↪halt()
    print("TXID is", txid)
    check_succesful_tx(web3, txid)
    print("OK")
```

5.10 Getting data field value for a function call

You can get the function signature (data field payload for a tranaction) for any smart contract function using the following:

```
from ico.utils import check_succesful_tx
from ico.utils import get_contract_by_name
import populus
from populus.utils.cli import request_account_unlock
from populus.utils.accounts import is_account_locked
from eth_utils import to_wei

import uuid

p = populus.Project()
account = "0x"  # Our controller account on Kovan
```

(continues on next page)

(continued from previous page)

```
with p.get_chain("kovan") as chain:
    web3 = chain.web3
    Contract = get_contract_by_name(chain, "PreICOProxyBuyer")
    # contract = Contract(address="0x")

    sig_data = Contract._prepare_transaction("claimAll")
    print("Data payload is", sig_data["data"])
```

5.11 Set early participant pricing

Set pricing data for early investors using PresaleFundCollector + MilestonePricing contracts.

```
from ico.utils import check_succesful_tx
from ico.utils import get_contract_by_name
import populus
from populus.utils.cli import request_account_unlock
from populus.utils.accounts import is_account_locked
from eth_utils import to_wei, from_wei

# The base price for which we are giving discount %
RETAIL_PRICE = 0.0005909090909090909

# contract, price tuples
PREICO_TIERS = [
    # 40% bonus tier
    ("0x78c6b7f1f5259406be3bc73eca1eaa859471b9f3", to_wei(RETAIL_PRICE * 1/1.4, "ether
↪")),

    # 35% tier A
    ("0x6022c6c5de7c4ab22b070c36c3d5763669777f68", to_wei(RETAIL_PRICE * 1/1.35,
↪"ether")),

    # 35% tier B
    ("0xd3fa03c67cfba062325cb6f4f4b5c1e642f1cffe", to_wei(RETAIL_PRICE * 1/1.35,
↪"ether")),

    # 35% tier C
    ("0x9259b4e90c5980ad2cb16d685254c859f5eddde5", to_wei(RETAIL_PRICE * 1/1.35,
↪"ether")),

    # 25% tier
    ("0xee3dfe33e53deb5256f31f63a59cffd14c94019d", to_wei(RETAIL_PRICE * 1/1.25,
↪"ether")),

    # 25% tier B
    ("0x2d3a6cf3172f967834b59709a12d8b415465bb4c", to_wei(RETAIL_PRICE * 1/1.25,
↪"ether")),

    # 25% tier C
    ("0x70b0505c0653e0fed13d2f0924ad63cdf39edefe", to_wei(RETAIL_PRICE * 1/1.25,
↪"ether")),

    # 25% tier D
    ("0x7cfe55c0084bac03170ddf5da070aa455ca1b97d", to_wei(RETAIL_PRICE * 1/1.25,
↪"ether")),
```

(continues on next page)

(continued from previous page)

```
]

p = populus.Project()
deploy_address = "0xe6b645a707005bb4086fa1e366fb82d59256f225"  # Our controller␣
↪account on mainnet
pricing_strategy_address  = "0x9321a0297cde2f181926e9e6ac5c4f1d97c8f9d0"
crowdsale_address = "0xaa817e98ef1afd4946894c4476c1d01382c154e1"

with p.get_chain("mainnet") as chain:
    web3 = chain.web3

    # Safety check that Crodsale is using our pricing strategy
    Crowdsale = get_contract_by_name(chain, "Crowdsale")
    crowdsale = Crowdsale(address=crowdsale_address)
    assert crowdsale.call().pricingStrategy() == pricing_strategy_address

    # Get owner access to pricing
    MilestonePricing = get_contract_by_name(chain, "MilestonePricing")
    pricing_strategy = MilestonePricing(address=pricing_strategy_address)

    PresaleFundCollector = get_contract_by_name(chain, "PresaleFundCollector")
    for preico_address, price_wei_per_token in PREICO_TIERS:

        eth_price = from_wei(price_wei_per_token, "ether")
        tokens_per_eth = 1 / eth_price
        print("Tier", preico_address, "price per token", eth_price, "tokens per eth",␣
↪round(tokens_per_eth, 2))

        # Check presale contract is valid
        presale = PresaleFundCollector(address=preico_address)
        assert presale.call().investorCount() > 0, "No investors on contract {}".
↪format(preico_address)

        txid = pricing_strategy.transact({"from": deploy_address}).
↪setPreicoAddress(preico_address, price_wei_per_token)
        print("TX is", txid)
        check_succesful_tx(web3, txid)
```

5.12 Move early participant funds to crowdsale

Move early participant funds from PresaleFundCollector to crowdsale.

Example:

```
from ico.utils import check_succesful_tx
from ico.utils import get_contract_by_name
import populus
from populus.utils.cli import request_account_unlock
from populus.utils.accounts import is_account_locked
from eth_utils import to_wei, from_wei
from ico.earlypresale import participate_early

presale_addresses = [
    "0x78c6b7f1f5259406be3bc73eca1eaa859471b9f3",
```

(continues on next page)

(continued from previous page)

```
    "0x6022c6c5de7c4ab22b070c36c3d5763669777f68",
    "0xd3fa03c67cfba062325cb6f4f4b5c1e642f1cffe",
    "0x9259b4e90c5980ad2cb16d685254c859f5eddde5",
    "0xee3dfe33e53deb5256f31f63a59cffd14c94019d",
    "0x2d3a6cf3172f967834b59709a12d8b415465bb4c",
    "0x70b0505c0653e0fed13d2f0924ad63cdf39edefe",
    "0x7cfe55c0084bac03170ddf5da070aa455ca1b97d",
]

p = populus.Project()
deploy_address = "0x"  # Our controller account on mainnet
pricing_strategy_address = "0x"
crowdsale_address = "0x"

with p.get_chain("mainnet") as chain:
    web3 = chain.web3

    Crowdsale = get_contract_by_name(chain, "Crowdsale")
    crowdsale = Crowdsale(address=crowdsale_address)

    for presale_address in presale_addresses:
        print("Processing contract", presale_address)
        participate_early(chain, web3, presale_address, crowdsale_address, deploy_
↪address, timeout=3600)
        print("Crowdsale collected", crowdsale.call().weiRaised() / 10**18, "tokens␣
↪sold", crowdsale.call().tokensSold() / 10**8, "money left", from_wei(web3.eth.
↪getBalance(deploy_address), "ether"))
```

5.13 Triggering presale proxy buy contract

Move funds from the proxy buy contract to the actual crowdsale.

```
from ico.utils import check_succesful_tx
from ico.utils import get_contract_by_name
import populus
from populus.utils.cli import request_account_unlock
from populus.utils.accounts import is_account_locked
from eth_utils import to_wei, from_wei

p = populus.Project()
deploy_address = "0x"  # Our controller account on mainnet
proxy_buy_address  = "0x"
crowdsale_address = "0x"

with p.get_chain("mainnet") as chain:
    web3 = chain.web3

    # Safety check that Crodsale is using our pricing strategy
    Crowdsale = get_contract_by_name(chain, "Crowdsale")
    crowdsale = Crowdsale(address=crowdsale_address)

    # Make sure we are getting special price
    EthTranchePricing = get_contract_by_name(chain, "EthTranchePricing")
    pricing_strategy = EthTranchePricing(address=crowdsale.call().pricingStrategy())
```

(continues on next page)

(continued from previous page)

```
    assert crowdsale.call().earlyParticipantWhitelist(proxy_buy_address) == True
    assert pricing_strategy.call().preicoAddresses(proxy_buy_address) > 0

    # Get owner access to pricing
    PreICOProxyBuyer = get_contract_by_name(chain, "PreICOProxyBuyer")
    proxy_buy = PreICOProxyBuyer(address=proxy_buy_address)
    # txid = proxy_buy.transact({"from": deploy_address}).setCrowdsale(crowdsale.
↪address)
    # print("TXID", txid)

    txid = proxy_buy.transact({"from": deploy_address}).buyForEverybody()
    print("Buy txid", txid)
```

5.14 Resetting token sale end time

The token sale owner might want to reset the end date. This can happen in the case the crowdsale has ended and tokens could not be fully sold, because of fractions. Alternatively, a manual soft cap is invoked because no more money is coming in and it makes sense to close the token sale.

```
import populus
from populus.utils.cli import request_account_unlock
from populus.utils.accounts import is_account_locked
from eth_utils import to_wei, from_wei
from ico.utils import check_succesful_tx
from ico.utils import get_contract_by_name

p = populus.Project()
deploy_address = "0x"  # Our controller account on mainnet
crowdsale_address = "0x"

with p.get_chain("mainnet") as chain:
    web3 = chain.web3

    block = web3.eth.getBlock('latest')
    timestamp = block["timestamp"]

    # 15 minutes in the future
    closing_time = int(timestamp + 15*60)

    # Safety check that Crodsale is using our pricing strategy
    Crowdsale = get_contract_by_name(chain, "Crowdsale")
    crowdsale = Crowdsale(address=crowdsale_address)
    txid = crowdsale.transact({"from": deploy_address}).setEndsAt(closing_time)
    print(crowdsale.call().getState())
```

5.15 Finalizing a crowdsale

Example:

```
import populus
from populus.utils.cli import request_account_unlock
```

(continues on next page)

(continued from previous page)

```
from populus.utils.accounts import is_account_locked
from eth_utils import to_wei, from_wei
from ico.utils import check_succesful_tx
from ico.utils import get_contract_by_name

p = populus.Project()
deploy_address = "0x"  # Our controller account on mainnet
crowdsale_address = "0x"
team_multisig = "0x"

with p.get_chain("mainnet") as chain:
    web3 = chain.web3

    Crowdsale = get_contract_by_name(chain, "Crowdsale")
    crowdsale = Crowdsale(address=crowdsale_address)

    BonusFinalizeAgent = get_contract_by_name(chain, "BonusFinalizeAgent")
    finalize_agent = BonusFinalizeAgent(address=crowdsale.call().finalizeAgent())
    assert finalize_agent.call().teamMultisig() == team_multisig
    assert finalize_agent.call().bonusBasePoints() > 1000

    # Safety check that Crodsale is using our pricing strategy
    txid = crowdsale.transact({"from": deploy_address}).finalize()
    print("Finalize txid is", txid)
    check_succesful_tx(web3, txid)
    print(crowdsale.call().getState())
```

5.16 Send ends at

Example:

```
from ico.utils import check_succesful_tx
from ico.utils import get_contract_by_name
import populus
from populus.utils.cli import request_account_unlock
from populus.utils.accounts import is_account_locked

p = populus.Project()
account = "0x4af893ee43a0aa328090bcf164dfa535a1619c3a"  # Our controller account on␣
↪Kovan

with p.get_chain("mainnet") as chain:
    web3 = chain.web3
    Contract = get_contract_by_name(chain, "Crowdsale")
    contract = Contract(address="0x0FB81a518dCa5495986C5c2ec29e989390e0E406")

    if is_account_locked(web3, account):
        request_account_unlock(chain, account, None)

    txid = contract.transact({"from": account}).setEndsAt(1498631400)
    print("TXID is", txid)
    check_succesful_tx(web3, txid)
    print("OK")
```

5.17 Approving tokens for issuer

Usually you need to approve() tokens for a bounty distribution or similar distribution contract (Issuer.sol). Here is an example.

Example:

```
import populus
from populus.utils.cli import request_account_unlock
from populus.utils.accounts import is_account_locked

from ico.utils import check_succesful_tx
from ico.utils import get_contract_by_name

p = populus.Project()
account = "0x"  # Our controller account
issuer_contract = "0x"  # Issuer contract who needs tokens
normalized_amount = int("123000000000000")  # Amount of tokens, decimal points␣
↪unrolled
token_address = "0x"  # The token contract whose tokens we are dealing with

with p.get_chain("mainnet") as chain:
    web3 = chain.web3
    Token = get_contract_by_name(chain, "CrowdsaleToken")
    token = Token(address=token_address)

    if is_account_locked(web3, account):
        request_account_unlock(chain, account, None)

    print("Approving ", normalized_amount, "raw tokens")

    txid = token.transact({"from": account}).approve(issuer_contract, normalized_
↪amount)
    print("TXID is", txid)
    check_succesful_tx(web3, txid)
    print("OK")
```

5.18 Whitelisting transfer agent

Token owner sets extra transfer agents to allow test tranfers for a locked up token.

Example:

```
from ico.utils import check_succesful_tx
from ico.utils import get_contract_by_name
import populus
from populus.utils.cli import request_account_unlock
from populus.utils.accounts import is_account_locked

p = populus.Project()
account = "0x51b9311eb6ec8beb049dafeafe389ee2818b1b20"  # Our controller account

with p.get_chain("mainnet") as chain:
    web3 = chain.web3
    Token = get_contract_by_name(chain, "CrowdsaleToken")
```

(continues on next page)

(continued from previous page)

```
    token = Token(address="0x")

    if is_account_locked(web3, account):
        request_account_unlock(chain, account, None)

    txid = token.transact({"from": account}).setTransferAgent("0x", True)
    print("TXID is", txid)
    check_succesful_tx(web3, txid)
    print("OK")
```

5.19 Reset token name and symbol

Update name and symbol info of a token. There are several reasons why this information might not be immutable, like trademark rules.

Example:

```
import populus
from populus.utils.cli import request_account_unlock
from populus.utils.accounts import is_account_locked
from ico.utils import check_succesful_tx
from ico.utils import get_contract_by_name

p = populus.Project()
account = "0x"  # Our controller account

with p.get_chain("mainnet") as chain:
    web3 = chain.web3
    Token = get_contract_by_name(chain, "CrowdsaleToken")
    token = Token(address="0x")

    if is_account_locked(web3, account):
        request_account_unlock(chain, account, None)

    txid = token.transact({"from": account}).setTokenInformation("Tokenizer", "TOKE")
    print("TXID is", txid)
    check_succesful_tx(web3, txid)
    print("OK")
```

5.20 Read crowdsale variables

Read a crowdsale contract variable.

Example:

```
from ico.utils import check_succesful_tx
from ico.utils import get_contract_by_name
import populus
from populus.utils.cli  import request_account_unlock
from populus.utils.accounts import is_account_locked

p = populus.Project()
```

(continues on next page)

(continued from previous page)

```
with p.get_chain("mainnet") as chain:
    web3 = chain.web3
    Crowdsale = get_contract_by_name(chain, "Crowdsale")
    crowdsale = Crowdsale(address="0x")

    print(crowdsale.call().weiRaised() / (10**18))
```

5.21 Reset token name and symbol

Update name and symbol info of a token. There are several reasons why this information might not be immutable, like trademark rules.

Example:

```
import populus
from populus.utils.cli import request_account_unlock
from populus.utils.accounts import is_account_locked
from ico.utils import check_succesful_tx
from ico.utils import get_contract_by_name

p = populus.Project()
account = "0x"  # Our controller account

with p.get_chain("mainnet") as chain:
    web3 = chain.web3
    Token = get_contract_by_name(chain, "CrowdsaleToken")
    token = Token(address="0x")

    if is_account_locked(web3, account):
        request_account_unlock(chain, account, None)

    txid = token.transact({"from": account}).setTokenInformation("Tokenizer", "TOKE")
    print("TXID is", txid)
    check_succesful_tx(web3, txid)
    print("OK")
```

5.22 Reset upgrade master

`upgradeMaster` is the address who is allowed to set the upgrade path for the token. Originally it may be the deployment account, but you must likely want to move it to be the team multisig wallet.

Example:

```
import populus
from populus.utils.cli import request_account_unlock
from populus.utils.accounts import is_account_locked
from ico.utils import check_succesful_tx
from ico.utils import get_contract_by_name

p = populus.Project()
```

(continues on next page)

(continued from previous page)

```
account = "0x"  # Our deployment account

team_multisig  = "0x"  # Gnosis wallet address

token_address = "0x"  # Token contract address

with p.get_chain("mainnet") as chain:
    web3 = chain.web3
    Token = get_contract_by_name(chain, "CrowdsaleToken")
    token = Token(address=token_address)

    if is_account_locked(web3, account):
        request_account_unlock(chain, account, None)

    txid = token.transact({"from": account}).setUpgradeMaster(team_multisig)
    print("TXID is", txid)
    check_succesful_tx(web3, txid)
    print("OK")
```

5.23 Participating presale

You can test presale proxy buy participation.

Example:

```
from ico.utils import check_succesful_tx
from ico.utils import get_contract_by_name
import populus
from populus.utils.cli  import request_account_unlock
from populus.utils.accounts import is_account_locked
from eth_utils import to_wei

p = populus.Project()

with p.get_chain("kovan") as chain:
    web3 = chain.web3

    PreICOProxyBuyer = get_contract_by_name(chain, "PreICOProxyBuyer")
    presale = PreICOProxyBuyer(address="0x4fe8b625118a212e56d301e0f748505504d41377")

    print("Presale owner is", presale.call().owner())
    print("Presale state is", presale.call().getState())

    # Make sure minimum buy in threshold is exceeeded in the value
    txid = presale.transact({"from": "0x001fc7d7e506866aeab82c11da515e9dd6d02c25",
↪"value": to_wei(40, "ether")}).invest()
    print("TXID", txid)
    check_succesful_tx(web3, txid)
```

5.24 Distributing bounties

There are two commands to support token bounty distribution

- `combine-csvs` allows to merge externally managed bountry distribution sheets to one combined CSV distribution file
- `distribute-tokens` deploys an issuer contract and handles the token transfers

5.24.1 Prerequisites

- An account with gas money
- A token contract address
- CSV files for the token distribution (Twitter, Facebook, Youtube, translations, etc.)
- A multisig wallet holding the source tokens

5.24.2 Merge any CSV files

Merge any or a single CSV files using `combine-csvs`. This command will validate input Ethereum addresses and merge any duplicate transactions to a single address to one transaction.

5.24.3 Deploy issuer contract

Example:

```
distribute-tokens --chain=mainnet --
↪address=0x1e10231145c0b670e9ee5a7f5b47172afa3b6186 --
↪token=0x5af2be193a6abca9c8817001f45744777db30756 --csv-file=combined.csv --address-
↪column="Ethereum address" --amount-column="Total reward" --master-
↪address=0x9a60ad6de185c4ea95058601beaf16f63742782a
```

5.24.4 Give approve() for the issuer contract

Use the multisig wallet to approve() the token distribution.

5.24.5 Run the issuance

Example:

```
distribute-tokens --chain=mainnet --
↪address=0x1e10231145c0b670e9ee5a7f5b47172afa3b6186 --
↪token=0x5af2be193a6abca9c8817001f45744777db30756 --csv-file=combined-bqx.csv --
↪address-column="Ethereum address" --amount-column="Total reward" --master-
↪address=0x9a60ad6de185c4ea95058601beaf16f63742782a --issuer-
↪address=0x78d30c42a5f9fb19df60768e4c867b697e24b615
```

5.25 Extracting Ethereum transaction data payload from a function signature

This allows you to see what goes into an Ethereum transaction data field payload, when you call a smart contract function in a transaction.

Example:

```
import populus
from ico.utils import get_contract_by_name

p = populus.Project()

with p.get_chain("kovan") as chain:

    contract = get_contract_by_name(chain, "PreICOProxyBuyer")

    # With arguments
    # contract._prepare_transaction("refund", fn_kwargs={"customerId": raw_id})


    function = "refund"
    # Without arguments
    # Get a Dayta payload for calling a contract function refund()
    sig_data = contract._prepare_transaction(function)
    print("Data payload for {}() is {}".format(function, sig_data["data"]))
```

5.26 Splitting a payment

Call `PaymentSplitter` contract to split the money amount the participants.

Example:

```
import populus
import binascii
from ico.utils import check_succesful_tx
from ico.utils import get_contract_by_name

p = populus.Project()

with p.get_chain("mainnet") as chain:

    PaymentSplitter = get_contract_by_name(chain, "PaymentSplitter")
    web3 = chain.web3

    splitter = PaymentSplitter(address="...")
    txid = splitter.transact({"from": "..."}).split()
    print("TXID", binascii.hexlify(txid))
    check_succesful_tx(web3, txid)
```

CHAPTER 6

Contract source code verification

6.1 Verifying contracts on EtherScan

ICO package has a semi-automated process to verify deployed contracts on EtherScan verification service.

6.2 Benefits of verification

- You can see the state of your contract variables real time on EtherScan block explorer
- You prove that there are deterministic and verifiable builds for your deployed smart contracts

6.3 Prerequisites

- You need to have Chrome and chromedriver installed for the browser automation
- You need to have Splinter Python package installed:

```
pip install Splinter
```

6.4 How automatic verification works

You need to specify the verification settings in your YAML deployment script for *deploy-contracts* command.

You need to make sure that you have your Solidity version and optimization parameters correctly.

Example how to get Solidity version:

```
solc --version
```

Here is an example YAML section:

```
# Use automated Chrome to verify all contracts on etherscan.io
verify_on_etherscan: yes
browser_driver: chrome

solc:

    # This is the Solidity version tag we verify on EtherScan.
    # For available versions see
    # https://kovan.etherscan.io/verifyContract2
    #
    # See values in Compiler drop down.
    # You can also get the local compiler version with:
    #
    #     solc --version
    #
    # Note that for EtherScan you need to add letter "v" at the front of the version
    #
    # Note: You need to have correct optmization settings for the compiler
    # in populus.json that matches what EtherScan is expecting.
    #
    version: v0.4.14+commit.c2215d46

    #
    # We supply these to EtherScan as the solc settings we used to compile the␣
↪contract.
    # They must match values in populus.json compilication / backends section.
    # These are the defaults supplied with the default populus.json.
    #
    optimizations:
        optimizer: true
        runs: 500
```

When you run *deploy-contracts* and *verify_on_etherscan* is turned *on*, a Chrome browser will automatically open after a contract has been deployed. It goes to Verify page on EtherScan and automatically submits all verification information, including libraries.

In the case there is a problem with the verification, *deploy-contracts* will stop and ask you to continue. During this time, you can check what is the actual error from EtherScan on the opened Chrome browser.

CHAPTER 7

Test suite

7.1 Introduction

ICO package comes with extensive automated test suite for smart contracts.

7.2 About Populus

Populus is a tool for the Ethereum blockchain and smart contract management. The project uses Populus internally. Populus is a Python based suite for

- Running arbitrary Ethereum chains (mainnet, testnet, private testnet)
- Running test suites against Solidity smart contracts

7.3 Running tests

Install first as given in the instructions.

Running tests using tox

```
export SOLC_BINARY=$(pwd)/script/travis-dockerized-solc.sh export SOLC_VERSION=0.4.18 tox
```

If `solc` fails, create a local virtual environment and test populus command locally:

```
populus compile
```

Reasons could include: Docker not running.

Running tests in the current virtual environemtn:

```
py.test tests
```

Run a specific test:

```
py.test tests -k test_get_price_tiers
```

7.4 Troubleshooting

Seeing how it looks like inside Dockerized solc environment:

```
docker run -it -v `pwd`:`pwd` -v `pwd`/zeppelin:`pwd`/zeppelin -w `pwd` --entrypoint /
↪bin/sh ethereum/solc:$SOLC_VERSION
```

This lands you to in shell in Docker mounted volume.

CHAPTER 8

Chain configuration

8.1 Introduction

ico package uses underlying Populus framework to configure different Ethereum backends.

Supported backend and nodes include

- Go Ethereum (geth)
- Parity
- Infura (Ethereum node as a service)
- Quicknode (Ethereum node as a service)
- Ethereum mainnet
- Ethereum Ropsten test network
- Ethreum Kovan test network
- ... or basically anything that responds to JSON RPC

8.2 Default configuration

The default configuration set in the packge distribution is in populus.json file.

Edit this file for your own node IP addresses and ports.

The default configuration is

- `http://127.0.0.1:8545` is mainnet JSON-RPC, *populus.json* network sa *mainnet*
- `http://127.0.0.1:8546` is Kovan JSON-RPC, *populus.json* network sa *kovan*
- `http://127.0.0.1:8547` is Kovan JSON-RPC, *populus.json* network sa *ropsten*

Ethereum node software (geth, parity) must be started beforehand and configured to allow JSON-RPC in the particular port.

For more information about *populus.json* file refer to Populus documentation.

8.3 Starting Ethereum node and creating deployment accounts

Below are two examples for Go Ethereum and Parity.

Note: We recommend using Kovan or Ropsten testnet for any testing and trials, because of faster transaction confirmation times. However, as the writing of this, Kovan testnet is only available for Parity and not for Go Ethereum. Go Ethereum and Parity have a different command line syntax and account unlocking mechanisms. It might take some effort to learn and start using both.

8.3.1 Account unlocking

When you make an Ethereum transaction, including deploying a contract, you need to have an Ethereum account with ETH balance on it. Furthermore this account must be unlocked. By default the accounts are available only in an encrypted file in the hard disk. When you unlock the account you can use it from the scripts for performing transactions.

8.3.2 Go Ethereun for mainnet

Example how to start Go Ethereum JSON-RPC for mainnet:

```
geth --fast --ipcdisable --rpc --rpcapi "db,eth,net,web3,personal" --verbosity 3 --
↪rpccorsdomain "*"  --cache 2048
```

You can create a new mainnet account which you will use a deployment account from geth console:

```
geth attach http://localhost:8545
```

Create a new private key from a seed phrase in geth console:

```
> web3.sha3("my super secret seed phrase")
0x000000...
```

Now import this 256-bit number as a geth account private key:

```
> personal.importRawKey("0x00000", "my account password")
```

You also need to unlock your deployment every time you do a deployment from *geth* console.

Example:

```
geth attach http://localhost:8545
```

Then unlock account for 1 hour in geth console:

```
personal.unlockAccount("0x00000000...", "my account password", 3600)
```

8.3.3 Parity with Kovan testnet

First start *parity –chain=kovan* to generate the chaindata files and such.

Connect to the Parity UI using your web browser.

Create a new Kovan testnet account. The account password will be stored in plain text, so do not use a strong password.

Create a file *password.txt* and store the password there.

Example how to start Parity JSON-RPC for Kovan testnet, unlocking your Kovan account for test transactions. It will permanently unlock your account using the password given in *password.txt* and listen to JSON-RPC in port *http://localhost:8547*.

```
parity --chain=kovan --unlock 0x001fc7d... --password password.txt --jsonrpc-apis
↪"web3,eth,net,parity,traces,rpc,personal" --jsonrpc-port 8547 --no-ipc --port 30306␣
↪--tracing on --allow-ips=public
```

8.3.4 Getting Kovan testnet ETH

Your options

- Kindly ask people to send you Kovan ETH (KETH) on the Kovan Gitter channel
- Use Parity provided SMS authentication to get KETH. in this case you need to start the Parity node in mainnet first, import in the same account and then get some real ETH balance for it.

CHAPTER 9

Design choices

9.1 Introduction

In this chapter we explain some design choices made in the smart contracts.

9.2 Timestamp vs. block number

The code uses block timestamps instead of block numbers for start and events. We work on the assumption that crowdsale periods are not so short or time sensitive there would be need for block number based timing. Furthermore if the network miners start to skew block timestamps we might have a larger problem with dishonest miners.

9.3 Crowdsale strategies and compound design pattern

Instead of cramming all the logic into a single contract through mixins and inheritance, we assemble our crowdsale from multiple components. Benefits include more elegant code, better reusability, separation of concern and testability.

Mainly, our crowdsales have the following major parts

- Crowdsale core: capped or uncapped
- Pricing strategy: how price changes during the crowdsale

- Finalizing strategy: What happens after a successful crowdsale: allow tokens to be transferable, give out extra tokens, etc.

9.4 Background information

- https://drive.google.com/file/d/0ByMtMw2hul0EN3NCaVFHSFdxRzA/view

CHAPTER 10

Other

10.1 Importing raw keys

You often need need to work with raw private keys. To import a raw private key to geth you can do from console:

```
web3.personal.importRawKey("<Private Key>","<New Password>")
```

Private key must be **without** 0x prefixed hex format.

More information

- http://ethereum.stackexchange.com/a/10020/620

10.2 Flattening source code for verification

Here is a snippet that will expand the source code of all contracts for the generated `build/contracts.json` file and embed the source inside the file. This will allow easier verification (reproducible builds) when using ABI data.

You can run from Python shell:

```
import populus
import json
from ico.importexpand import expand_contract_imports

p = populus.Project()
```

(continues on next page)

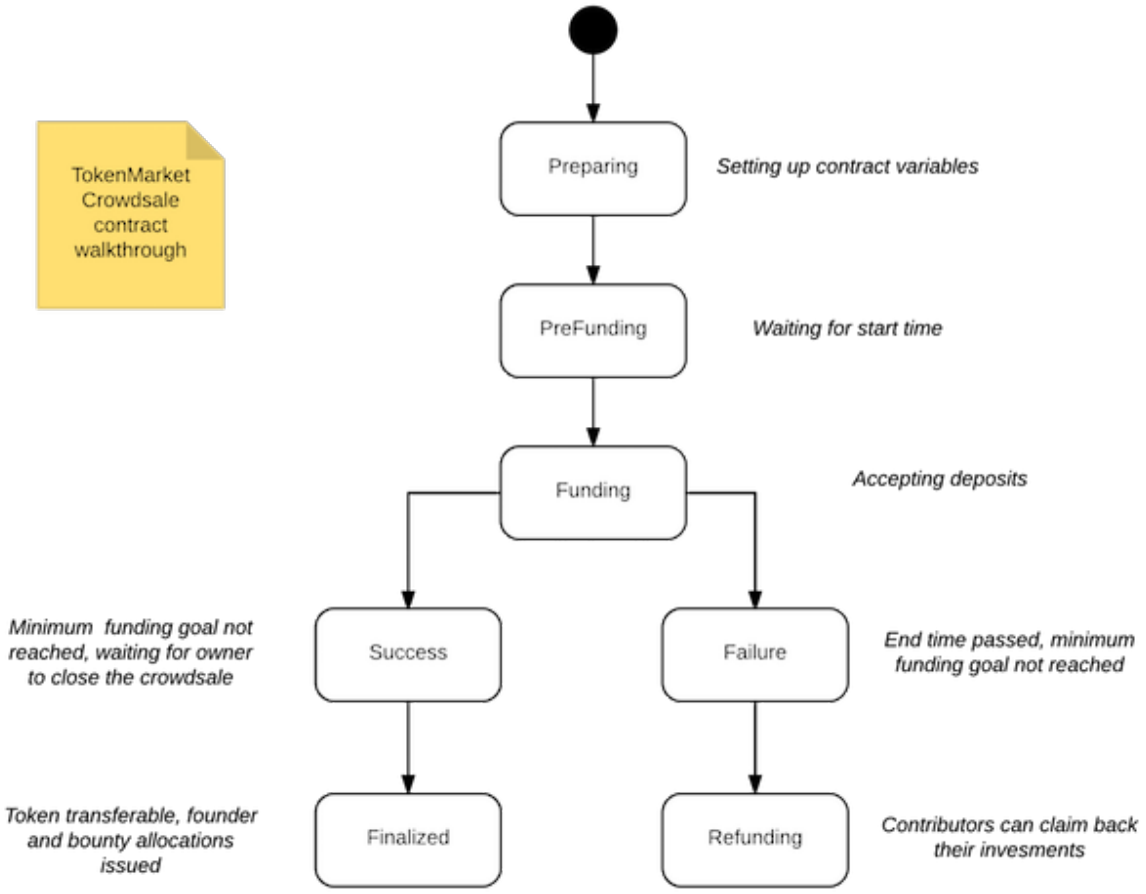
(continued from previous page)

```
data = json.load(open("build/contracts.json", "rt"))
for contract in data.values():

    # This was a source code file for an abstract contract
    if not contract["metadata"]:
        continue

    targets = contract["metadata"]["settings"]["compilationTarget"]

    contract_file = list(targets.keys())[0]  # "contracts/AMLToken.sol": "AMLToken"

    # Eliminate base path, as this will be set by expand_contract_imports
    if "zeppelin/" not in contract_file:
        contract_file = contract_file.replace("contracts/", "")
    else:
        pass
        # contract_file = contract_file.replace("zeppelin/", "zeppelin/contracts/")

    source, imports = expand_contract_imports(p, contract_file)
    contract["source"] = source

# Write out expanded ABI data
json.dump(data, open("build/contracts-flattened.json", "wt"))
```

CHAPTER 11

Commercial support

Contact TokenMarket for launching your ICO or crowdsale



CHAPTER 12

Links

Github issue tracker and source code

Documentation