ICO smart contracts Documentation

Release 0.1

Mikko Ohtamaa

Aug 22, 2017

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

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

Links

Github issue tracker and source code

Documentation

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

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

- Civic
- Storj
- Monaco
- DENT
- Bitquence
- InsureX
- ... and many more!

Quick token sale walkthrough

Features and design goals

- **Best practices**: Smart contracts are written with the modern best practices of Ethereum community
- **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
- **Migration**: 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

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.

CHAPTER 2

Contracts

Introduction

This chapter describers Ethereum crowdsale smart contracts.

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

TODO

CHAPTER 3

Installation

Preface

Instructions are written in OSX and Linux in mind.

Experience needed

- Basic command line usage
- Basic Github usage

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 .
```

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 # ...
cd Smart-Contracts
python3.5 -m venv venv
source venv/bin/activate
pip install wheel
pip install -r requirements.txt
pip install -e .
```

CHAPTER 4

Command line commands

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.

deploy-contracts

Scripted deployment of multiple related Ethereum smart contracts.

Example YAML deployment scripts

- **‘allocated-token-sale <https://github.com/TokenMarketNet/ico/blob/master/crowdsales/allocated-**token-sale-example.yml>‘_ (based on DENT)
- dummy mintable token saale 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:

      deploy-contracts --deployment-file=crowdsales/example.yml
      --deployment-name=kovan--
      address=0x001fc7d7e506866aeab82c11da515e9dd6d02c25

  Example files:

  * 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  YAML section name we are deploying. Usual options
                          include "mainnet" or "kovan"  [required]
  --deployment-file TEXT  YAML file definiting the crowdsale  [required]
  --address TEXT          Deployment address that pays the gas for the
                          deployment cost. This account must exist on Ethereum
                          node you are connected to.  [required]
  --help                  Show this message and exit.
```

deploy-token

Deploy a single token contract.

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.
```

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):

      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.
```

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

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.
```

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:
  --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.
```

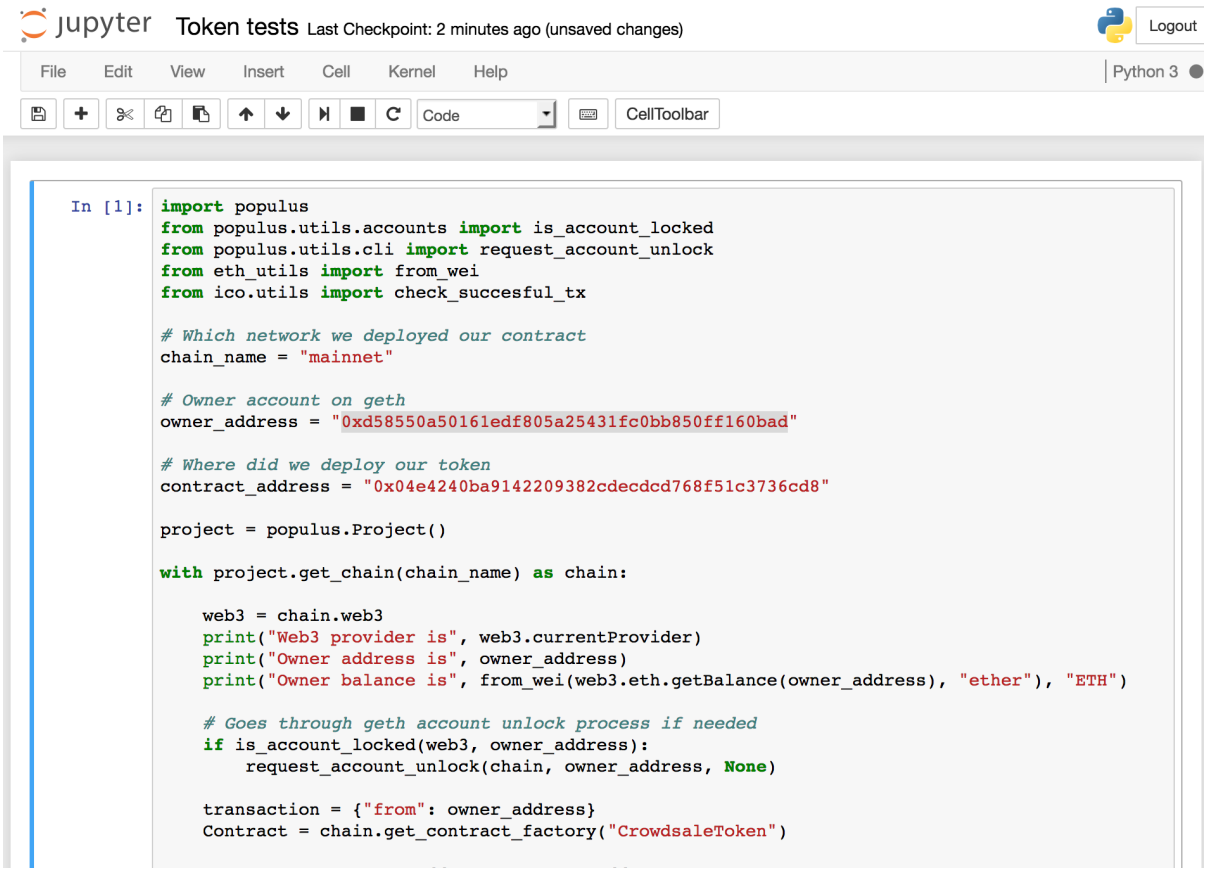
CHAPTER 5

Interacting with deployed smart contracts

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`.

Getting Jupyter Notebook

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

```
pip install jupyter
```

Then start Jupyter Notebook:

```
jupyter notebook
```

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

# 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 = chain.contract_factories.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)
```

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

# 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 = chain.get_contract_factory("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")
```

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
from ico.utils import check_succesful_tx

# 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 = chain.get_contract_factory("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}
```

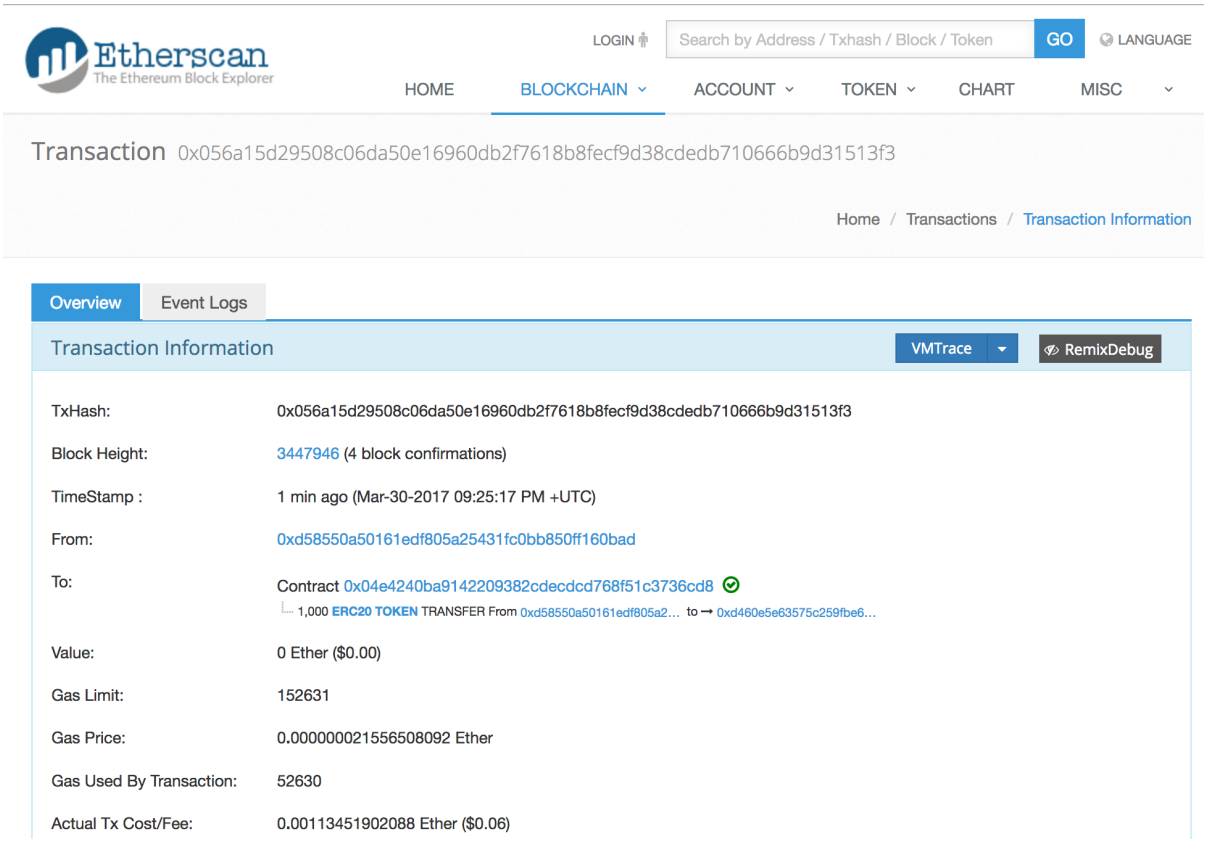
```
    print("Attempting to transfer some tokens to our MyEtherWallet account")
    txid = contract.transact(transaction).transfer(buddy_address, amount)
    check_succesful_tx(web3, txid)
    print("Transfered", amount, "tokens to", buddy_address, "in transaction https://
↪ropsten.etherscan.io/tx/{}".format(txid))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

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/
↪0x5460742a4f40dd573aeadedde95fc57fff6de800dde9494520c4f7852d7a956d
```

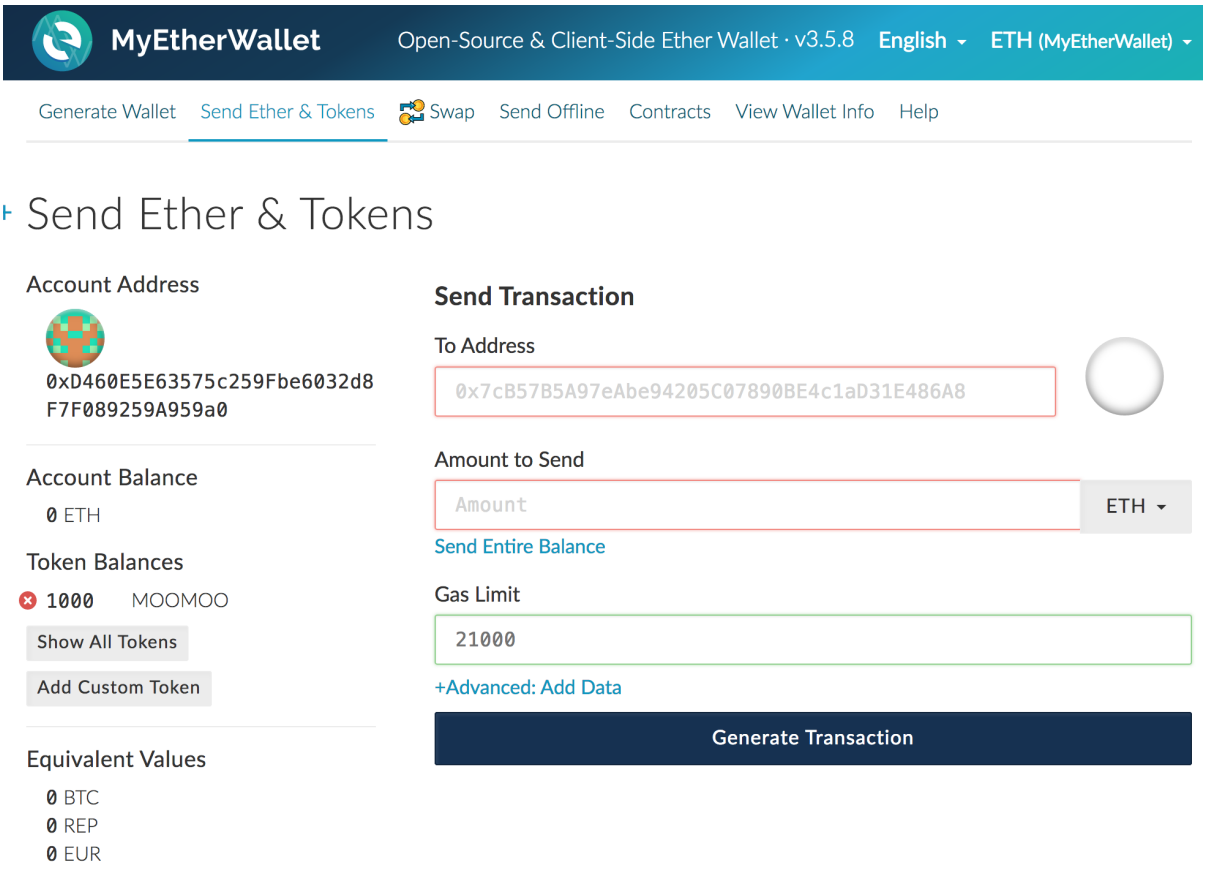
Etherscan transfer confirmation

We can see the transaction in the blockchain explorer:



MyEtherWallet transfer confirmation

And then finally we see tokens in our MyEtherWallet:



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
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 = getattr(chain.contract_factories, "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
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 = getattr(chain.contract_factories, "PresaleFundCollector")
    contract = Contract(address="0x858759541633d5142855b27f16f5f67ea78654bf")

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

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

Whitelisting crowdsale participants

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

```
from ico.utils import check_succesful_tx
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 = getattr(chain.contract_factories, "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")
```

Change pricing strategy

To mix fat finger errors:

```
from ico.utils import check_succesful_tx
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 = getattr(chain.contract_factories, "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")
```

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
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 = getattr(chain.contract_factories, "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)
    check_succesful_tx(web3, txid)
    print("OK")
```

Halt payment forwarder

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

```
from ico.utils import check_succesful_tx
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 = getattr(chain.contract_factories, "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")
```

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
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 = getattr(chain.contract_factories, "PreICOProxyBuyer")
    # contract = Contract(address="0x")

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

Set early participant pricing

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

```
from ico.utils import check_succesful_tx
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")),
]

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 = chain.contract_factories.Crowdsale
    crowdsale = Crowdsale(address=crowdsale_address)
    assert crowdsale.call().pricingStrategy() == pricing_strategy_address

    # Get owner access to pricing
    MilestonePricing = chain.contract_factories.MilestonePricing
    pricing_strategy = MilestonePricing(address=pricing_strategy_address)

    PresaleFundCollector = chain.contract_factories.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)
```

Move early participant funds to crowdsale

Move early participant funds from PresaleFundCollector to crowdsale.

Example:

```
from ico.utils import check_succesful_tx
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",
    "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 = chain.contract_factories.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"))
```

Triggering presale proxy buy contract

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

```
from ico.utils import check_succesful_tx
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 = chain.contract_factories.Crowdsale
    crowdsale = Crowdsale(address=crowdsale_address)

    # Make sure we are getting special price
    EthTranchePricing = chain.contract_factories.EthTranchePricing
    pricing_strategy = EthTranchePricing(address=crowdsale.call().pricingStrategy())
    assert crowdsale.call().earlyParticipantWhitelist(proxy_buy_address) == True
    assert pricing_strategy.call().preicoAddresses(proxy_buy_address) > 0

    # Get owner access to pricing
    PreICOProxyBuyer = chain.contract_factories.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)
```

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

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 = chain.contract_factories.Crowdsale
    crowdsale = Crowdsale(address=crowdsale_address)
    txid = crowdsale.transact({"from": deploy_address}).setEndsAt(closing_time)
    print(crowdsale.call().getState())
```

Finalizing a crowdsale

Example:

```
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

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 = chain.contract_factories.Crowdsale
    crowdsale = Crowdsale(address=crowdsale_address)

    BonusFinalizeAgent = chain.contract_factories.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())
```

Send ends at

Example:

```
from ico.utils import check_succesful_tx
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 = getattr(chain.contract_factories, "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")
```

Approving tokens

Example:

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

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

with p.get_chain("kovan") as chain:
    web3 = chain.web3
    Token = getattr(chain.contract_factories, "CentrallyIssuedToken")
    token = Token(address="")

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

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

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
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 = getattr(chain.contract_factories, "CrowdsaleToken")
    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")
```

Set token name

Update info of a token.

Example:

```
from ico.utils import check_succesful_tx
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

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

Read crowdsale variables

Read a crowdsale contract variable.

Example:

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

p = populus.Project()

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

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

Participating presale

You can test presale proxy buy participation.

Example:

```
from ico.utils import check_succesful_tx
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 = getattr(chain.contract_factories, "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)
```

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

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

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.

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
```

Give approve() for the issuer contract

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

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
```

CHAPTER 6

Contract source code verification

Verifying contracts on EtherScan

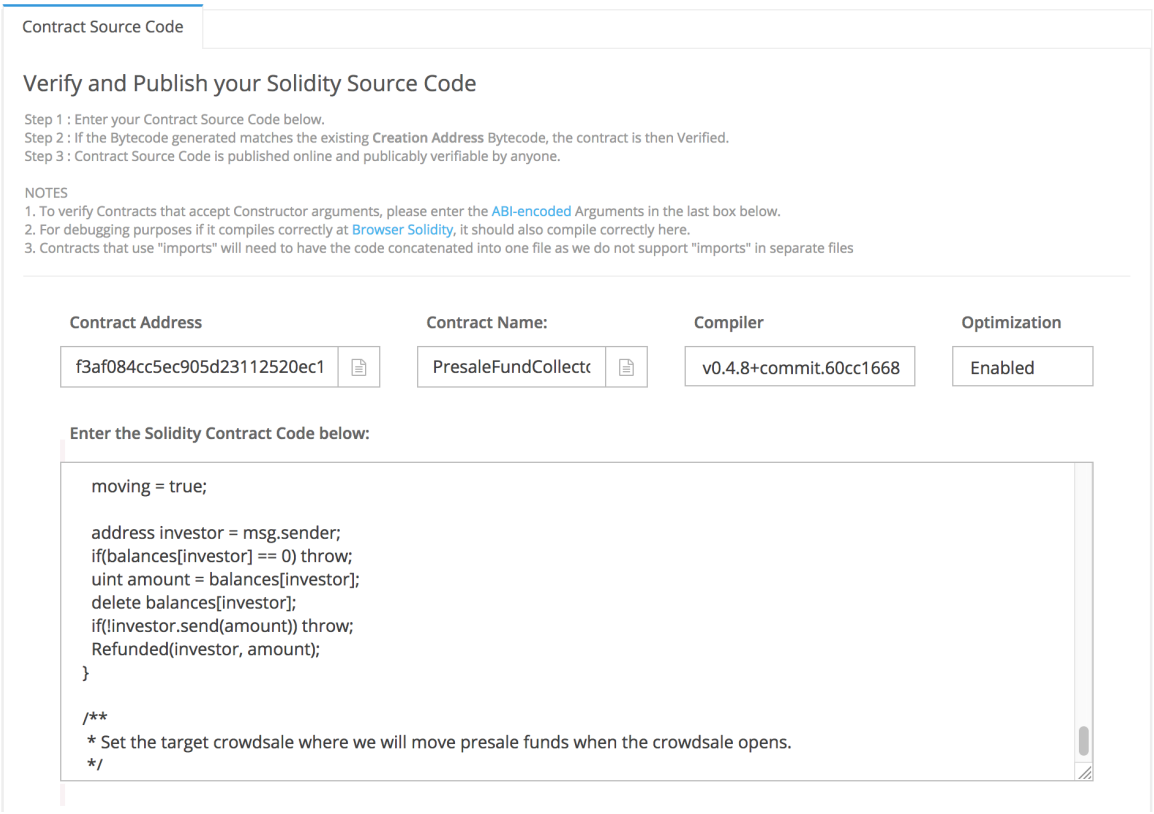
ICO package has a semi-automated process to verify deployed contracts on EtherScan verification service.

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

How verification works

- You need to have Firefox installed with necessary Selenium drivers
- Give *–verify* option to a *deployment script*
- After the command line script has deployed the contract a browser will open
- The script autofills the verification page details (source code, construction arguments, linked libraries)


Contract Source Code
Verify and Publish your Solidity Source Code
Step 1 : Enter your Contract Source Code below.
Step 2 : If the Bytecode generated matches the existing Creation Address Bytecode, the contract is then Verified.
Step 3 : Contract Source Code is published online and publicably verifiable by anyone.
NOTES
1. To verify Contracts that accept Constructor arguments, please enter the ABI-encoded Arguments in the last box below.
2. For debugging purposes if it compiles correctly at Browser Solidity, it should also compile correctly here.
3. Contracts that use "imports" will need to have the code concatenated into one file as we do not support "imports" in separate files
Contract Address
f3af084cc5ec905d23112520ec1
Contract Name:
PresaleFundCollecto
Compiler
v0.4.8+commit.60cc1668
Optimization
Enabled
Enter the Solidity Contract Code below:
moving = true;
address investor = msg.sender;
if(balances[investor] == 0) throw;
uint amount = balances[investor];
delete balances[investor];
if(!investor.send(amount)) throw;
Refunded(investor, amount);
}
/**
* Set the target crowdsale where we will move presale funds when the crowdsale opens.
*/

CHAPTER 7

Test suite

Introduction

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

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

Running tests

Running tests:

```
py.test tests
```

Run a specific test:

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

CHAPTER 8

Chain configuration

Introduction

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

Supported backend and nodes include

- Go Ethereum (geth)
- Parity
- Ethereum mainnet
- Ethereum Ropsten test network
- Ethreum Kovan test network
- ... or basically anything that responds to JSON RPC

Default configuration

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

- `http://127.0.0.1:8545` is mainnet JSON-RPC
- `http://127.0.0.1:8546` is Kovan JSON-RPC

CHAPTER 9

Design choices

Introduction

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

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.

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.

Background information

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

CHAPTER 10

Other

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.

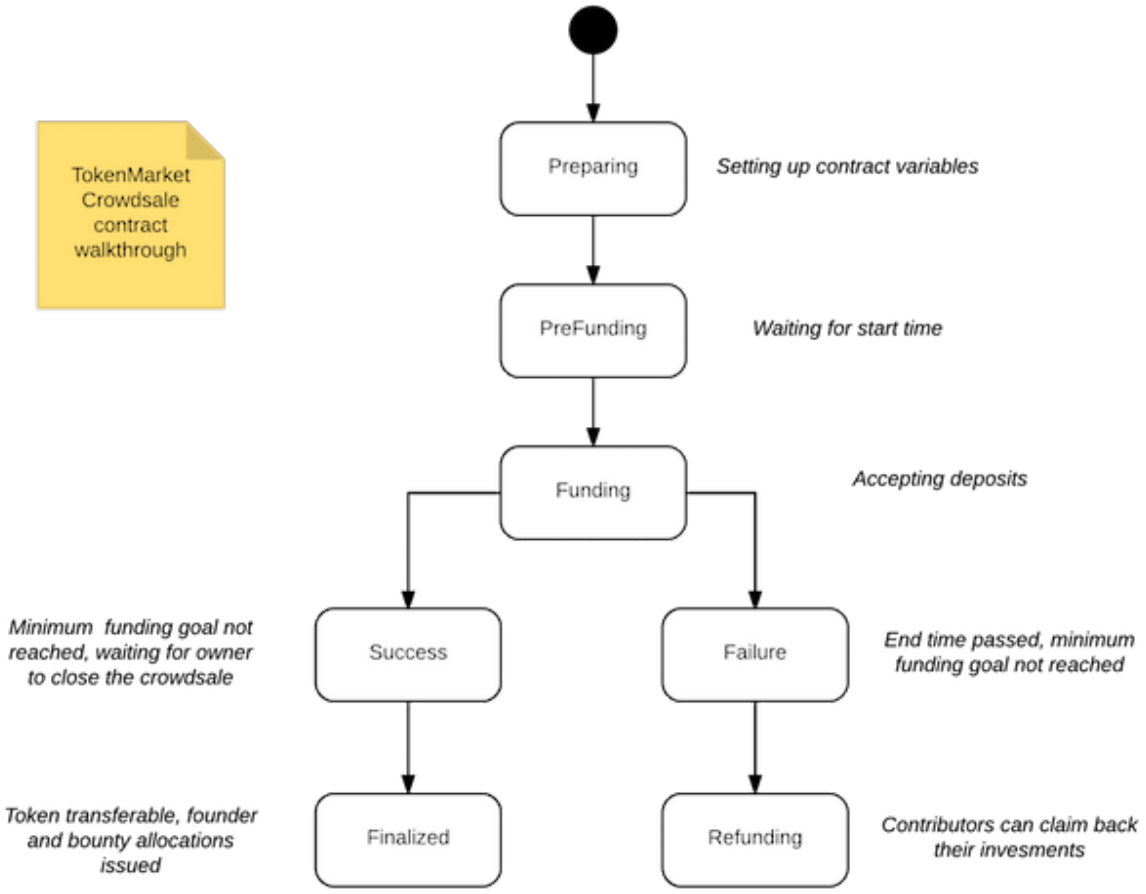
More information

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

CHAPTER 11

Commercial support

Contact TokenMarket for launching your ICO or crowdsale



CHAPTER 12

Links

Github issue tracker and source code

Documentation