Blockchain-based Crowdfunding Platform

Le Hoang Tuan

Last updated: 4 June, 2019

Timetable task scheduler of project:

|  |  |  |
| --- | --- | --- |
| Time | Task | Progress |
| Sep 2018 – Dec 2018  (2 months) | * Complete description of project (issue, background, milestone, requirement) * Understand the knowledge related to the project. |  |
| Feb 2019  (1 month) | Design system (diagrams, flowchart) |  |
| Mar 2019  (1 month) | Design front-end |  |
| Apr 2019 – May 2019  (2 months) | Build back-end |  |
| Jun 2019 – Aug 2019  (3 months) | Testing and complete |  |

Table 1. Project’s plan

Summary of last week's work and reference details in the report:

|  |  |  |
| --- | --- | --- |
| Week | Work done | Detail in report |
| 04/03/2019 – 07/03/2019 | * Draw the use case diagram * Implemented a component in the system, this component will keep the amount of money that users send into the system for investment. | 3.3.1  See smart contract at 8.1  See UI design at 3.3.3.1 |
| 11/03/2019 – 14/03/2019 | Write smart contract for some functions |  |
| 18/03/2019 – 21/03/2019 | Continue task last week |  |

Contents

[1 Introduction 5](#_Toc10534473)

[1.1 Mission 5](#_Toc10534474)

[1.2 Importance 5](#_Toc10534475)

[1.3 Definition 6](#_Toc10534476)

[1.4 Business models of crowdfunding 6](#_Toc10534477)

[1.5 Crowdfunding’s process 6](#_Toc10534478)

[2 Current platforms and its problem 8](#_Toc10534479)

[2.1 Non-blockchain 8](#_Toc10534480)

[2.2 Blockchain 8](#_Toc10534481)

[3 Solution 11](#_Toc10534482)

[3.1 Requirement 11](#_Toc10534483)

[3.2 System Analysis 14](#_Toc10534484)

[3.2.1 Overview 14](#_Toc10534485)

[3.3 System design 15](#_Toc10534486)

[3.3.1 Use case diagram 15](#_Toc10534487)

[3.3.2 System architecture 16](#_Toc10534488)

[3.3.3 UI design 16](#_Toc10534489)

[3.4 Blockchain 17](#_Toc10534490)

[3.4.1 ERC standard 17](#_Toc10534491)

[3.4.2 Hybrid blockchain 20](#_Toc10534492)

[4 Installation 21](#_Toc10534493)

[4.1 Preparation 21](#_Toc10534494)

[4.2 Install 21](#_Toc10534495)

[5 Testing and result 21](#_Toc10534496)

[6 Conclusion 21](#_Toc10534497)

[7 Bibliography 21](#_Toc10534498)

[8 Appendix 22](#_Toc10534499)

[8.1 TokenSystem contract 22](#_Toc10534500)

[8.2 Campaigns contract 27](#_Toc10534501)

# Introduction

## Mission

The project’s mission is makes the system serves for the entrepreneurs and the investors. It makes entrepreneurship accessible to everyone, everywhere and trustable for investors.

## Importance

Crowdfunding has quickly become a popular avenue of funding for investment, seed money and start-up funding. The growth rates have been astounding over its short life span. In 2015, a worldwide estimate totaling over US$34 billion was raised by crowdfunding [1], more than double from the year before. Fred Wilson of Union Square Ventures predicted that crowdfunding could grow to US$300 billion.

Notable crowdfunding campaigns include the Oculus Rift, which effectively launched the modern virtual reality industry and sold to Facebook for US$2 billion after just two years, and the Pebble Smartwatch, which similarly jumpstarted the smartwatch industry and collectively raised over $30 million from backers.

The crowdfunding platform serves an immense need with important future economic and social potential. However, today’s centralized implementation of the model is failing to meet the changing needs of the global market. Therefore, there should be a decentralized system, reducing dependence on third parties. In one study, showed "Distributed ledger technologies are in path to disrupt economic interactions across markets. Originating from cryptocurrencies, blockchain technologies offer access to an infinite range of applications benefiting to sharing economies", "..inefficiencies of the crowdfunding market and address them by combining smart contracts with smart property in a trustless decentralized application build on Ethereum" [2].

## Definition

The concept of crowdfunding comes from the broader concept of crowdsourcing, which involves using the “crowd” to obtain ideas, feedback, and solutions to develop corporate activities. Crowdfunding is the practice of funding a project or venture by raising many small amounts of money from a large number of people, typically via the Internet.

## Business models of crowdfunding

There are four major models of crowdfunding divided by investor profitability [1]:

* **Equity-based -** investors will receive an equity portion of the fundraiser’s company.
* **Lending-based** - investors will receive an interest based on the amount invested by the project.
* **Reward-based** - investors may receive tangible or intangible rewards.
* **Donation-based** - The role of donation-based crowdfunding is to support humanitarian and artistic projects. Funders on donation-based crowdfunding can be seen as philanthropists. And they may not receive anything.

## Crowdfunding’s process

The crowdfunding process has three main actors [3]:

* **Platform** – is the central role of the intermediary, who serves as a matchmaker between promoters and funder.
* **Project creators** (also can be entrepreneurs and other) **–** is raising funds via a crowdfunding platform. These fundraisers use crowdfunding to get direct access to the market and to gather financial support from truly interested supporter.
* **Backers** - who decide to financially support projects of the fundraisers.

The sequence diagram of process overview is detailed in Figure 1.

A screenshot of a cell phone

Description automatically generated

Figure 1. The crowdfunding process overview

The existence of crowdfunding platforms allows fundraiser (startups) to present ideas (step **1** in diagram) for the wide public and ask for funding. Crowdfunding platforms will consider the ideas (step **1.1** in diagram) to be public and then create an investment possibility (step **1.2** in diagram) for investors – ordinary people, who most likely would not have a chance to invest using any other way.

Investors analyse proposed ideas and choose the ones they like and believe in to fund (step 2 in diagram). Also, as investors like and believe in the funded idea or project, and, moreover, desire for it to succeed, they tend to (if there is such a possibility) provide advice from their experience for the startup. When the deadline ends, if the goal is completed, all funds will be transferred to the fundraiser (step **2.2** in diagram); then fundraiser pay to the platform a fee (step **2.3** in diagram). Otherwise, the investment amount will be returned to the investor (step **2.1** in diagram).

Startups offer investors something in return for a reward (step **3** in diagram): a small gift, product/services, equity, a percentage of revenue or nothing.

# Current platforms and its problem

## Non-blockchain

In current, there are some largest crowdfunding platform as Kickstarter, Indiegogo,…

The main problem facing crowdfunding is an issue of trust. Backers are burned repeatedly by creators who fail to follow through on their promises. A lack of security mechanisms and transparency provides backers with little protection.

A Wharton University professor researching the leading crowdfunding platform, Kickstarter, found that up to 85% of projects delay while 14% ultimately fail to deliver what was promised. Notable projects that raised millions but stumbled in fulfillment include the Ouya (raised US$8.5M), Zano Drone (US$3.5M), Skully (US$2.4M), and the Coolest Cooler (US$13M). In each of these cases, the project creators were paid in full immediately after funding concluded. Backers, on the other hand, waited months (and sometimes years), to receive nothing but excuses for their money and trust.

## Blockchain

Benefits of blockchain:

* Blockchain technology is perfectly capable of managing value exchange under contract.
* The feature that will come handy when it comes to managing rewards based and equity-based crowdfunding campaigns.
* The whole process of assigning relevant rewards and equity against their contributions can be automated, using blockchain based smart contracts for crowdfunding.
* By programming a set of pre-defined crowdfunding conditions on smart contracts, the system can be auotmated to execute the smart contract to issue certain rewards or proof of ownership of a certain percentage of equity based on the amount contributed towards the campaign.

Following is a survey of some of the existing platforms based on their whitepaper:

**WeiFund**: is a toolkit for running crowdfunding campaigns on the Ethereum blockchain.

* Wallet: user can use a wallet in Metamask or Jaxx
* Support currency: ETH. Because ETH collects a small fee (<0.1 USD)
* Method to fund a campaign: Setup a wallet, then send ETH to lightweight wallet (in WeiFund). Funds are typically stored in the campaign smart contracts until the campaign ends. When goal is reached, send ETH to fundraiser.
* Claim a refund: not automate, investors have to click "Claim Refund Owed" in the page of campaign that has failed.
* Create a campaign: is a manual process that the WeiFund team performs for our clients.

**YouToken:** is a blockchain-based crowdfunding platform providing a service to entrepreneurs that converts their intelligence into a financial asset (token).

* Overview:
  + The total value of the YouToken platform is expressed in YTN token. YTN token could be purchased/sold/traded on external exchanges. The YTN token price depends on the quantity of entrepreneurs registered on YouToken and the quality of their projects.
  + In addition to other benefits, all YTN token holders receive dividends from all successful projects launched on the YouToken platform. The amount of dividends will directly depend on the number of creators registered on the platform and the quality/profitability of their projects.
  + In addition to the tokens of the entire platform (YTN token), investors can obtain an entrepreneur’s creator tokens (YTN\_creatorname). YTN\_creatorname is a unique financial asset representing the value of a specific entrepreneur (e.g. Elon Jobs → YTN\_elonjobs). YTN\_creatorname token is anticipated to be tradable on their internal exchange. The YTN\_creatorname price depends on the success/failure of the entrepreneur’s project(s) registered on the YouToken Marketplace.
  + Entrepreneurs will be allowed to raise money for their project via Crowdfunding offering. On the YouToken platform, investors will be able to invest in entrepreneurs by purchasing a stake in creators’ projects. For some offerings, investment limits will apply depending on investors’ income and net worth.
* About revenue share smart contract: YTN is automatically connected to a Revenue share smart contract. All funds collected from YouToken Platform Revenue Streams will be collected to the General Bucket. 30% of all platform Revenue will be proportionally shared with all YTN holders.
* It is not just a “MicroICO” platform. YouToken supports entrepreneurs during 100% of their “business life”, pushing their lifetime value via the most applicable forms and types of smart contracts.
* Fund will be received through many stages of project, not directly to the entrepreneur, it is stored in "Vault". Whether or not receive fund in next stage will depends on voting of token holders (has a decision-making mechanism specify).
* If creators want to receive fund for next stages, they made a request for investors to vote, Project smart contract will be wait for 72 hours for investor’s responses. Platform will send 3 emails/notifications with call to action to all investors.

**PledgeCamp:**

* To list a campaign, a creator must place a deposit on the blockchain with a native cryptocurrency called Pledge Coin (PLG).
* Once a campaign is successfully funded, the Backer Insurance amount is automatically held in a smart contract encoded with the agreed-upon milestones and voting days. When a vote approaches, creators are given the chance to demonstrate their progress and justify their use of funds. Backers are then able to vote either to continue the campaign, or else cancel the campaign with a sufficient vote of “no confidence.”
* Backer Insurance holds creators accountable to their promises and motivates them to be prepared and realistic while setting expectations.

# Solution

## Requirement

In this document, we list some options for fexible choose

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Opt. # | Funding Types | Support users | Type of project | Geography (country) | Funding goal | Receive investment |
| 1 | Donation, Reward, Lending, Equity | Variety users. Two main type:   * Fundraiser: who need money. * Funder: who have money | Variety of category project. | * Unlimited * Limited (only for equity) | Both flexible and fixed goal | Period time (not entire amount) |
| 2 | Donation and reward | * Fundraiser: startups, producers, patients,… * Funder: sponsor, consumers. | Technology, medical, education,… | Unlimited | Fixed goal | Entire amount |
| 3 | Donation | * Fundraiser: startups, patients,.. * Funder: sponsor. | Variety of category project. | Unlimited | Fixed goal | Entire or period amount (fundraiser set) |
| 4 | Lending and Equity | * Fundraiser: startups, entrepreneur. * Funder: investors. | Some category project | Limited | Fixed goal | Entire or period amount (fundraiser set) |

Table 2. Some options for requirement

Note:

* ***Fixed goal****: in a period of time or specific time; and with amount of money. If the expiration date does not meet the conditions, the fundraiser will not receive anything. => “All or nothing”*
* ***Flexible goal:*** *When the fundraising goal fails, the fundraiser can receive the money they raise. But that requires an agreement between the investor and the fundraiser, which means there should be a vote.*

**I will choose option #2 to implement system.** Because I look it useful and possible to implement.

In the donation-based, a person in need of help can seek support from the community, and helpers are relieved because their money is transferred to the right people. For example, if a patient needs money to treat cancer, they can find crowdfunding as a help to them.

In the reward-based, producers can find opportunities to find capital to produce. And supporters will also find unique and new products. And there are many promotion for them.

## System Analysis

### Overview

Overview of the system:

- Users in the system include two types of fundraisers and funder.

- Users participating in the system must provide their Ethereum Wallet address (for receiving investment if the user is fundraiser, and for investing in campaigns if they are funders). This project may use a identify process from external service.

- In order to invest for campaigns, investors must deposit into system to get amount of token.

- To create a fundraising campaign, fundraisers need to provide detailed description of the project, including description of text, images, videos, fundraising goals, time. The description of the project will be stored on a private blockchain.

- About the fundraising process, described in Figure 1. When the fundraising time is over, if the goal is reached, the entire fundraising amount will be transferred to the investor. Fundraisers can withdraw tokens to ETH for the project. Otherwise, if you don't complete the target, all tokens will be returned to the investors.

- Although the fundraising goal and fundraising time are set by the campaign creator, the system will set a MIN -> MAX limit for each type of project that the fundraiser has chosen.

- The system will consist of two blockchain networks, public (the main Ethereum network) and private (this network is in the system). Specifically, the tasks of these two networks are as follows:

* Main Ethereum Network: take on the role of deposit (from ETH to "tokens") and withdraw money (from "tokens" to ETH).
* Private blockchain: taking on the role of handling internal transactions in the system including: creating campaigns, buying tokens, … and other internal transactions in the system.

- Why use a private blockchain? -> the problem of transaction costs and the processing time of transactions on the main network is not really effective, so it is necessary to create a more optimal private network of transaction cost and time confirmation. This private network only allows members of the system to use.

## System design

### Use case diagram

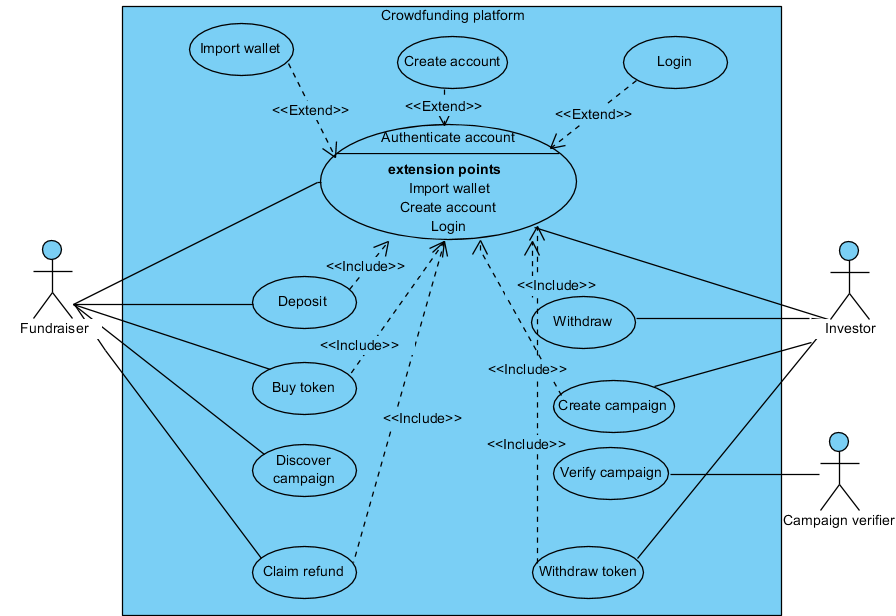


Figure 2. Use case diagram

*(above diagram is draft, I will complete)*

### System architecture

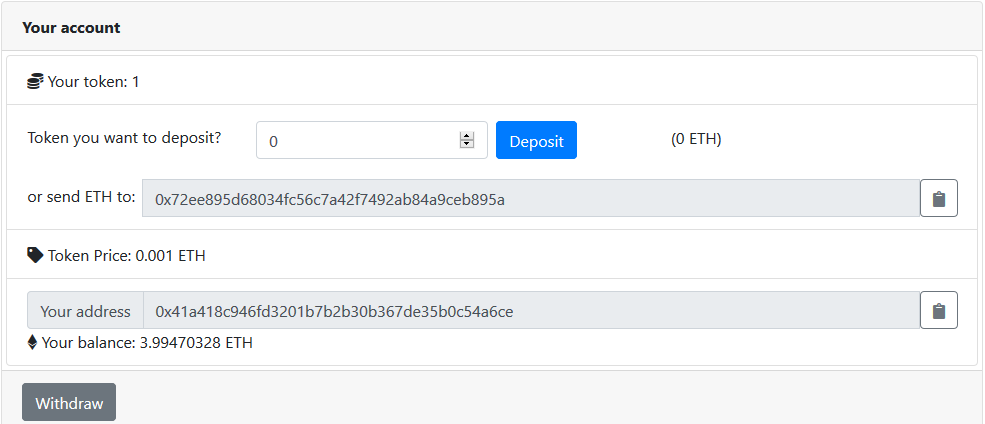
A close up of text on a white background

Description automatically generated

Figure 3. System architecture

### UI design

#### “Deposit and withdraw” view



## Blockchain

### ERC standard

ERC stands for Ethereum Request for Comments. An ERC is authored by Ethereum community developers in the form of a memorandum describing methods, behaviors, research, or innovations applicable to the working of the Ethereum ecosystem. It is submitted either for peer review or simply to convey new concepts or information. After core developers and community approval, the proposal becomes a standard.

There are many different ERC standards, however each ERC is designed for a specific field.

For example, ERC-137 is for Domain Name Service, or ERC-1167 is Minimal Proxy Contract.

One of the most basic and primitive standards is ERC-20.

Specifically, ERC-20 is a standard interface for tokens. This standard allows for the implementation of a standard API for tokens within smart contracts. This standard provides basic functionality to transfer tokens, as well as allow tokens to be approved so they can be spent by another on-chain third party. More detail at: <https://eips.ethereum.org/EIPS/eip-20>

**Specification of ERC-20:**

* Methods:
  + **name**: returns the name of the token - e.g. "MyToken".

function name() public view returns (string)

* + **symbol:** returns the symbol of the token. E.g. “HIX”.

function symbol() public view returns (string)

* + **decimals:** returns the number of decimals the token uses - e.g. 8, means to divide the token amount by 100000000 to get its user representation.

function decimals() public view returns (uint8)

* + **totalSupply:** returns the total token supply.

function totalSupply() public view returns (uint256)

* + **balanceOf**: returns the account balance of another account with address \_owner.

function balanceOf(address \_owner) public view returns (uint256 balance)

* + **transfer:** transfers \_value amount of tokens to address \_to, and MUST fire the Transfer event. The function SHOULD throw if the \_from account balance does not have enough tokens to spend.

function transfer(address \_to, uint256 \_value) public returns (bool success)

* + **transferFrom:** transfers \_value amount of tokens from address \_from to address \_to, and MUST fire the Transfer event. The transferFrom method is used for a withdraw workflow, allowing contracts to transfer tokens on your behalf. This can be used for example to allow a contract to transfer tokens on your behalf and/or to charge fees in sub-currencies. The function SHOULD throw unless the \_from account has deliberately authorized the sender of the message via some mechanism.

function transferFrom(address \_from, address \_to, uint256 \_value) public returns (bool success)

* + **approve:** allows \_spender to withdraw from your account multiple times, up to the \_value amount. If this function is called again it overwrites the current allowance with \_value.

function approve(address \_spender, uint256 \_value) public returns (bool success)

* + allowance: returns the amount which \_spender is still allowed to withdraw from \_owner.

function allowance(address \_owner, address \_spender) public view returns (uint256 remaining)

* Events:
  + **Transfer:** MUST trigger when tokens are transferred, including zero value transfers. A token contract which creates new tokens SHOULD trigger a Transfer event with the \_from address set to 0x0 when tokens are created.

event Transfer(address indexed \_from, address indexed \_to, uint256 \_value)

* + **Approval:** MUST trigger on any successful call to approve(address \_spender, uint256 \_value).

event Approval(address indexed \_owner, address indexed \_spender, uint256 \_value)

But ERC20 standard has the ambiguity between transfer and transferFrom function, making money loss possible.

So applying a new standard is necessary, I found that standard - **ERC777**.

This standard defines a new way to interact with a token contract while remaining backward compatible with ERC20.

It defines advanced features to interact with tokens. Namely, operators to send tokens on behalf of another address—contract or regular account—and send/receive hooks to offer token holders more control over their tokens.

It takes advantage of ERC820 to find out whether and where to notify contracts and regular addresses when they receive tokens as well as to allow compatibility with already-deployed contracts.

The main advantages of this standard are:

1. Uses the same philosophy as Ether in that tokens are sent with send(dest, value, data).
2. Both contracts and regular addresses can control and reject which token they send by registering a tokensToSend hook. (Rejection is done by reverting in the hook function.)
3. Both contracts and regular addresses can control and reject which token they receive by registering a tokensReceived hook. (Rejection is done by reverting in the hook function.)
4. The tokensReceived hook allows to send tokens to a contract and notify it in a single transaction, unlike ERC20 which require a double call (approve/transferFrom) to achieve this.
5. The token holder can “authorize” and “revoke” operators which can send tokens on their behalf. These operators are intended to be verified contracts such as an exchange, a cheque processor or an automatic charging system.
6. Every token transaction contains a data bytes field and a similar operatorData to be used freely to pass data to the recipient.
7. It is backward compatible with wallets that do not contain the tokensReceived hook function by deploying a proxy contract implementing the tokensReceived hook for the wallet.

More detailed at: <https://eips.ethereum.org/EIPS/eip-777>

### Hybrid blockchain

Hybrid Blockchains lie somewhere between private and public blockchains, depending on their architecture.

Each type of blockchain has its strengths and weaknesses. Public blockchains while being transparent and resistant to tampering are slow and expensive whereas, private blockchains are somewhat centralised but can deliver much higher throughput and speeds. As a logical step, hybrid blockchains combine the benefits of both of the blockchains while trying to limit the disadvantages.

Current Hybrid Blockchains: XinFin was launched in 2017, XinFin completed their ICO in March 2018 after successfully launching their first dApp, TradeFinex — a global trade and finance platform.

XinFin is the first hybrid blockchain and currently the only hybrid blockchain. It is built on both Ethereum, a public blockchain, and Quorum, a private blockchain. Quorum is an enterprise-focused version of Ethereum developed by J.P. Morgan of the Enterprise Ethereum Alliance.

So with this project, I decided to use hybrid blockchain to take advantage of the public network blockchain for transactions, and the private blockchain for internal system tasks.

# Installation

## Preparation

## Install

# Testing and result

# Conclusion

# Bibliography

|  |  |
| --- | --- |
| [1] | N. O. M. P. Paul Belleflamme, "The Economics of Crowdfunding Platforms," *Information Economics and Policy,* vol. 33, pp. 11-28, 2015. |
| [2] | O. Starkenmann, "Implementation of a Crowdfunding Decentralized Application on Ethereum," 2017. |
| [3] | A. a. B. A. Tomczak, "A conceptualized investment model of crowdfunding," *Venture Capital,* vol. 15, no. Taylor & Francis, pp. 335-359, 2013. |
| [4] | M. Swan, Blockchain: Blueprint for a new economy, O'Reilly Media, Inc., 2015. |
| [5] | L. a. S. J. Valančienė, "Crowdfunding for creating value: stakeholder approach.," *Procedia-Social and Behavioral Sciences,* vol. 156, pp. 599-604, 2014. |
| [6] | PledgeCamp, "PledgeCamp White paper," 24 11 2018. [Online]. Available: https://pledgecamp.com/\_\_pdf/whitepaper-pledgecamp-9-23.pdf. [Accessed 13 12 2018]. |
| [7] | Youtoken.co, "YouToken Whitepaper," 8 8 2018. [Online]. Available: https://ico.youtoken.io/whitepaper. [Accessed 13 12 2018]. |

# Appendix

## TokenSystem contract

pragma solidity ^0.5;

import {SafeMath} from "./SafeMath.sol";

import {Campaigns} from "./Campaigns.sol";

*/// @title This is contract for hold all token of users in system*

*/// @author tuanlh*

contract **TokenSystem** {

using SafeMath for uint;

struct **Investment** {

uint id;

uint amount;

}

Campaigns internal campaigns;

address internal admin;

uint internal mGranularity; *//Minium value of Wei*

mapping(address => uint) internal mBalances; *// wei*

uint internal mTotalBalances;

mapping(address => Investment[]) investment;

event **Deposit**(address from, uint amount);

event **Withdraw**(address to, uint amount);

*/\* -- Constructor -- \*/*

*//*

*/// @notice Constructor to create a TokenSystem*

*/// @dev This contract is deployed by system and only once deploy*

*/// set Granularity is the minimum transferable chunk*

*/// Or it is the minimum value of wei corresponds to a token*

*/// Or it is price of token*

*/// 1 token = mGranularity (wei)*

*/// Set admin = msg.sender (runner contract)*

constructor() public {

admin = msg.sender;

mGranularity = 10\*\*15; *// 1 ETH = 1000 tokens*

mTotalBalances = 0;

}

*/// @dev granularity can be understood as the price of a token. 1 token = granularity form as wei*

*/// @return the granularity of the token*

function **granularity**() public view returns (uint) { return mGranularity; }

*/// @dev Get token of user without campaigns*

*/// @param \_addr is address of user that you want check token*

*/// @return Number of token*

function **balances**(address \_addr) public view returns(uint) {

return mBalances[\_addr].**div**(mGranularity);

}

*/// @notice get my balance (form as Token) of msg.sender with campaign*

*/// @param \_addr is address of user*

*/// @return Result is number of token*

function **getBalance**(address \_addr) public view returns (uint) {

if (mBalances[msg.sender] == 0) {

return 0;

}

uint balance;

balance = mBalances[msg.sender].**div**(mGranularity);

balance = balance.**sub**(campaigns.**getTotalInvest**(\_addr));

return balance;

}

*/// @dev Call function getBalance() with params is msg.sender*

*/// @return Number of token of sender*

function **getMyBalance**() public view returns (uint) {

return **getBalance**(msg.sender);

}

*/// @notice Allow user transfer balances to this contract*

*/// @dev This function will receive balance that user send into contract and store in contract*

*/// value will be stored in the mBalances variable*

*/// Amount is msg.value form as Wei (1 ETH = 10^18 wei)*

function **deposit**() public payable {

require(

msg.value > 0,

"Amount to deposit MUST be greater zero"

);

**requireMultiple**(msg.value);

mBalances[msg.sender] = mBalances[msg.sender].**add**(msg.value);

mTotalBalances = mTotalBalances.**add**(msg.value);

assert(address(this).balance >= mTotalBalances);

emit **Deposit**(msg.sender, msg.value);

}

*/// @notice This function allow user withdraw balances in contract to ETH*

*/// @dev Withdraw token in system to ETH. (Wei = token \* mGranularity)*

*/// @param \_amountToken number of token that you want withdraw*

*/// amount of token MUST be multiple with mGranularity before send*

*/// @return `true` if withdraw process successful*

function **withdraw**(uint \_amountToken) public returns (bool) {

require(

\_amountToken > 0,

"Amount to deposit MUST be greater zero"

);

require(

\_amountToken <= **getMyBalance**(),

"You don't have enough token"

);

uint amount = \_amountToken.**mul**(mGranularity);

*// It is important to subtract amount before real transfer*

*// because the recipient can call this function again as part of the receiving call*

*// before `send` returns.*

mBalances[msg.sender] = mBalances[msg.sender].**sub**(amount);

mTotalBalances = mTotalBalances.**sub**(amount);

if (!msg.sender.**send**(amount)) {

mBalances[msg.sender] = mBalances[msg.sender].**add**(amount);

mTotalBalances = mTotalBalances.**add**(amount);

return false;

}

emit **Withdraw**(msg.sender, amount);

return true;

}

*/// @dev This function add address of campaign contract*

*/// @param \_campaign is an address*

function **updateCampaignAddr**(Campaigns \_campaign) public {

require(

msg.sender == admin,

"You MUST be owner of this contract");

campaigns = \_campaign;

}

*/// @notice Allow campaign owner (startups) can withdraw token from a succeed campaign*

*/// @dev This function MUST be run by a contract*

*/// @param \_i is index of campaign*

*/// @param \_owner is owner of campaign*

*/// @param \_tokenCollected total token was sold in campaign*

*/// @return `true` if withdraw process successful*

function **withdrawFromCampaign**(uint \_i, address \_owner, uint \_tokenCollected) public

returns(bool)

{

require(

**isContractAddress**(msg.sender),

"Run this contract MUST be a contract"

);

require(

address(campaigns) == msg.sender,

"Sender this function is invalid"

);

require(

\_tokenCollected > 0,

"Amount MUST be greater zero"

);

require(

campaigns.**getFinStatus**(\_i) >= Campaigns.FinStatus.accepted,

"Campaign MUST be accepted"

);

require(

campaigns.**getStatus**(\_i) >= Campaigns.Status.succeed,

"Campaign MUST be succeed"

);

uint \_amount = \_tokenCollected.**mul**(mGranularity);

mBalances[\_owner] = mBalances[\_owner].**add**(\_amount);

return true;

}

*/// @notice check wether is admin*

*/// @dev This function is called by a contract to verify address of sender*

*/// @param \_addr is address that you want to check*

*/// @return `true` if \_addr == admin*

function **isAdmin**(address \_addr) public view returns(bool) {

return admin == \_addr;

}

*/\* -- Helper Functions -- \*/*

*//*

*/// @notice Internal function that ensures `\_amount` is multiple of the granularity*

*/// @param \_amount The quantity that want's to be checked*

function **requireMultiple**(uint \_amount) internal view {

require(\_amount % mGranularity == 0, "Amount is not a multiple of granualrity");

}

*/// @notice Check whether an address is an address belong to contract or not.*

*/// @param \_addr Address of the contract that has to be checked*

*/// @return `true` if `\_addr` is a contract address*

function **isContractAddress**(address \_addr) internal view returns(bool) {

uint size;

assembly { size := **extcodesize**(\_addr) } *// solium-disable-line security/no-inline-assembly*

return size > 0;

}

**function** () external payable {**deposit**();}

}

Above, I used an additional library called SafeMath, I refer to it from the OpenZeppelin library, this is its source code:

pragma solidity ^0.5;

*/\*\**

*\* @title SafeMath*

*\* @dev Unsigned math operations with safety checks that revert on error*

*\*/*

library **SafeMath** {

*/\*\**

*\* @dev Multiplies two unsigned integers, reverts on overflow.*

*\*/*

function **mul**(uint256 a, uint256 b) internal pure returns (uint256) {

*// Gas optimization: this is cheaper than requiring 'a' not being zero, but the*

*// benefit is lost if 'b' is also tested.*

*// See: https://github.com/OpenZeppelin/openzeppelin-solidity/pull/522*

if (a == 0) {

return 0;

}

uint256 c = a \* b;

require(c / a == b);

return c;

}

*/\*\**

*\* @dev Integer division of two unsigned integers truncating the quotient, reverts on division by zero.*

*\*/*

function **div**(uint256 a, uint256 b) internal pure returns (uint256) {

*// Solidity only automatically asserts when dividing by 0*

require(b > 0);

uint256 c = a / b;

*// assert(a == b \* c + a % b); // There is no case in which this doesn't hold*

return c;

}

*/\*\**

*\* @dev Subtracts two unsigned integers, reverts on overflow (i.e. if subtrahend is greater than minuend).*

*\*/*

function **sub**(uint256 a, uint256 b) internal pure returns (uint256) {

require(b <= a);

uint256 c = a - b;

return c;

}

*/\*\**

*\* @dev Adds two unsigned integers, reverts on overflow.*

*\*/*

function **add**(uint256 a, uint256 b) internal pure returns (uint256) {

uint256 c = a + b;

require(c >= a);

return c;

}

*/\*\**

*\* @dev Divides two unsigned integers and returns the remainder (unsigned integer modulo),*

*\* reverts when dividing by zero.*

*\*/*

function **mod**(uint256 a, uint256 b) internal pure returns (uint256) {

require(b != 0);

return a % b;

}

}

## Campaigns contract

pragma solidity ^0.5;

import {SafeMath} from "./SafeMath.sol";

import {TokenSystem} from "./TokenSystem.sol";

*/// @title This contract store info about campaigns*

*/// @author tuanlh*

contract **Campaigns** {

TokenSystem token;

using SafeMath for uint;

*/\* Explaintation of campaign status*

*\* During: end date < now*

*\* Failed: end date >= now AND token collected < goal*

*\* Succeed: end date >= now AND token collected >= goal*

*\*/*

enum **Status** {during, failed, succeed}

*/\* Explaintation of campaign FINACIAL status*

*\* Pending: new campaign just added. NOT allow investor fund to campaign*

*\* Accepted: a campaign was verified => Allow investors fund to campaign*

*\* Paid: a campaign that owner withdraw token completed => end campaign*

*\*/*

enum **FinStatus** {pending, accepted, paid}

struct **CampaignInfo** {

string name;

*//string ipfsHash; //coming soon*

address owner;

uint startDate;

uint endDate;

uint goal;

uint collected;

FinStatus finstt;

mapping(address => uint) investment;

mapping(address => bool) isInvest;

}

CampaignInfo[] internal campaigns;

mapping(address => uint[]) internal investors; *//mapping investors to campaigns id*

event **Added**(uint id);

event **Accepted**(uint id);

event **Invested**(uint id, address invester, uint token);

event **Refund**(uint id, address investor, uint token);

event **Paid**(uint id, address ownerCampaign, uint token);

*/\* -- Constructor -- \*/*

*//*

*/// @notice Constructor to create a campaign contract*

*/// @dev This contract MUST be run after TokenSystem*

*/// @param \_tokenAddr is address of token contract*

constructor(TokenSystem \_tokenAddr) public {

token = \_tokenAddr;

}

function **getInfo**(uint \_index) public view

returns(

string memory name,

uint startDate,

uint endDate,

uint goal,

uint collected,

address owner,

FinStatus finStatus,

Status status

) {

name = campaigns[\_index].name;

startDate = campaigns[\_index].startDate;

endDate = campaigns[\_index].endDate;

goal = campaigns[\_index].goal;

collected = campaigns[\_index].collected;

owner = campaigns[\_index].owner;

finStatus = campaigns[\_index].finstt;

status = **getStatus**(\_index);

}

*/// @notice Create a campaign*

*/// @dev Add an element to variable campaigns array*

*/// @param \_name is campaign name*

*/// @param \_symbol is symbol character of campaign. Example: Token\_ABC*

*/// @param \_days is deadline for fundraising of a campaign. Min: 15 days*

*/// @param \_goal is goal of a campaign. Min-Max: 100.000-1.000.000.000*

function **createCampaign**(

string memory \_name,

uint \_days,

uint \_goal)

public {

require(

\_goal >= 100000 && \_goal <= 1000000000,

"The goal of campaign must be include range is from 100.000 to 1.000.000.000 tokens"

);

*// To testing, you can comment following lines*

*//require(*

*// \_days >= 15,*

*// "The minimum fundraising time for the campaign is 15 days."*

*//);*

*//In current Testing, default set Finacial Status is Accepted*

campaigns.**push**(**CampaignInfo**(

\_name,

msg.sender,

now,

now + \_days\*60, *// Test case in minutes*

\_goal,

0,

FinStatus.accepted

*// FinStatus.pending*

));

emit **Added**(campaigns.length - 1);

}

*/// @notice Accept a campaign is allow all investor can invest to that campaign*

*/// @param \_i is index of campaigns array*

function **acceptCampaign**(uint \_i) public {

require(

token.**isAdmin**(msg.sender),

"You MUST be verifier");

campaigns[\_i].finstt = FinStatus.accepted;

emit **Accepted**(\_i);

}

*/// @notice Allow user can buy token of campaign*

*/// @param \_i is index of campaigns array*

*/// @param \_token is amount of token that you buy*

*/// function will call token.fundToCampaign to verify*

function **invest**(uint \_i, uint \_token) public {

CampaignInfo memory campaign = campaigns[\_i];

require(

\_token > 0,

"amount of token must be greater than zero"

);

require(

now <= campaign.endDate,

"Campaign is ended"

);

require(

campaign.collected < campaign.goal,

"Campaign is reached goal"

);

require(

campaign.collected + \_token <= campaign.goal,

"Amount without goal of campaign"

);

require(

campaigns[\_i].finstt == FinStatus.accepted,

"This campaign MUST be accepted and NOT paid"

);

require(

\_token <= (token.**balances**(msg.sender) - **getTotalInvest**(msg.sender)),

"You don't have enough token");

campaigns[\_i].investment[msg.sender] = campaigns[\_i].investment[msg.sender].**add**(\_token);

if (!campaigns[\_i].isInvest[msg.sender]) {

investors[msg.sender].**push**(\_i);

campaigns[\_i].isInvest[msg.sender] = true;

}

campaigns[\_i].collected = campaigns[\_i].collected.**add**(\_token);

emit **Invested**(\_i, msg.sender, \_token);

}

*/// @notice Allow investor can claim refund when campaign during*

*/// when campaign failed, you don't need claim refund, because it is automatic proccess*

*/// @param \_i is index of campaigns array*

*/// @param \_token Amount investor want withdraw*

function **claimRefund**(uint \_i, uint \_token) public {

require(

\_token > 0,

"amount of token must be greater than zero"

);

require(

campaigns[\_i].investment[msg.sender] >= \_token,

"You don't have enough to claim refund"

);

require(

**getStatus**(\_i) == Status.during,

"You only can claim refund when campaigns during"

);

campaigns[\_i].investment[msg.sender] = campaigns[\_i].investment[msg.sender].**sub**(\_token);

campaigns[\_i].collected = campaigns[\_i].collected.**sub**(\_token);

emit **Refund**(\_i, msg.sender, \_token);

}

*/// @notice Handle after campaign. Only allow campaign owner run this function*

*/// @dev If campaign is succeed, fundraiser will receive funds*

*/// @param \_i is index of campaign array*

function **endCampaign**(uint \_i) public {

require(

msg.sender == campaigns[\_i].owner,

"This function MUST be run by owner"

);

require(

**getStatus**(\_i) == Status.succeed,

"Campaign MUST be succeed"

);

require(

campaigns[\_i].finstt != FinStatus.pending,

"Campaign MUST be verify");

require(

campaigns[\_i].finstt != FinStatus.paid,

"Campaign already paid"

);

*// Important: set status PAID before call external function to withdraw*

campaigns[\_i].finstt = FinStatus.paid;

if(!token.**withdrawFromCampaign**(\_i, msg.sender, campaigns[\_i].collected)) {

campaigns[\_i].finstt = FinStatus.accepted;

} else {

emit **Paid**(\_i, msg.sender, campaigns[\_i].collected);

}

}

*/// @notice Get token invested for a campaign of investor*

*/// @param \_i is index of campaigns*

*/// @param \_addr is address of investor that you want to check*

*/// @return Number of token that investor invested for a campaign*

function **getInvest**(uint \_i, address \_addr) public view returns(uint) {

return campaigns[\_i].investment[\_addr];

}

*/// @notice Get all amount of investor that invest to campaigns*

*/// @dev Get all amount of investor that invested and have checked campaign failed or succeed*

*/// If campaign is failed, NOT count that token*

*/// @param \_addr is address that you want check to amount of invest*

*/// @return Number of token that invested all campaigns*

function **getTotalInvest**(address \_addr) public view returns(uint) {

uint tokens = 0;

uint[] memory campaignsOf = investors[\_addr];

for (uint i=0; i < campaignsOf.length; i++) {

if (**getStatus**(i) != Status.failed) {

if (campaigns[i].investment[\_addr] > 0) {

tokens = tokens.**add**(campaigns[i].investment[\_addr]);

}

}

}

return tokens;

}

*/// @notice Get status of a campaign*

*/// @param \_i is index of campaigns array*

*/// @return {0 => during, 1 => failed, 2 => succeed, 3 => paid}*

function **getStatus**(uint \_i) public view returns(Status) {

if (now < campaigns[\_i].endDate) {

return Status.during;

} else {

if (campaigns[\_i].collected < campaigns[\_i].goal) {

return Status.failed;

} else {

return Status.succeed;

}

}

}

*/// @notice Get financial status of campaign*

*/// @param \_i is index of campaign*

*/// @return {0 => pending, 1 => accepted, 2 => paid}*

function **getFinStatus**(uint \_i) public view returns(FinStatus) {

return campaigns[\_i].finstt;

}

*/// @notice Get number of campaigns*

*/// @return Number of campaigns*

function **length**() public view returns(uint) {

return campaigns.length;

}

}