Created by: Navraj

Reviewed by: Anthony

Abstract

Documentation for the loyalty program

BEYOND IMAGINATION TECHNOLOGIES

Loyalty Program

Contents

[Introduction 2](#_Toc103759401)

[How it works 2](#_Toc103759402)

[Smart Contract 3](#_Toc103759403)

[Flow 3](#_Toc103759404)

[Functions 3](#_Toc103759405)

[Events 4](#_Toc103759406)

[API 5](#_Toc103759407)

[Setting up Nodes for the Network 9](#_Toc103759408)

Loyalty Program

# Introduction

A lot of companies and organizations have a loyalty program for their customers. In this program, customers are rewarded for being loyal to the company.

There are many different types of such programs out there. But most of them follow the same pattern.

1. The customers register themselves as a customer of the company. (Sometimes this is done automatically.)
2. The customers get some reward every time they some product of the company. Often times this reward is of some fraction of the value of the products they purchase.
3. The reward for the customers accumulates with every purchase. Essentially the more loyal they are to the company, the more reward they earn.
4. Customers can cash-out this reward, usually in terms of some discount from the company.

Loyalty programs work because they make the customers feel recognized and special which further leads to retention, more referrals and profits.

# How it works

The entire data of the loyalty program is stored on blockchain with a smart contract. The smart contract is written in a generic way so that it can be used for any company without much interruption of the existing operations of the company. The smart contract can be accessed via the company’s web application (front-end or back-end).

A simple API is written to interact with the smart contract. The API uses “javascript” and “web3.js” library. It can be integrated with any “javascript” based application. The standard public/private key encryption of the blockchain is used for security and authorization.

# Smart Contract

The smart contract is written in a way that it can be used for any organization. The interface is kept simple and easy-to-use while considering gas optimization. Although for private chain architecture, the concept of gas is redundant, still the principles of gas optimization are applied to optimize resource utilization.

## Flow

Points can be assigned to any userId. The userIds are input as strings(mostly alpha numeric). The contract converts the userId into bytes32 data type via keccak256 algorithm. The UserId string is saved against the bytes32 hash value.

The points are saved against a userId by creating a unique couponId for every allotment. All allotment details like expiry date, ref\_via etc. are saved against the unique coupon id.

The points of a user are redeemed, expired or modified using userId and couponId.

## Functions

* // Set the max supply of the points during contract creation.

constructor (uint256 \_maxSupply) { }

* //Modifier to check if caller is the owner of this contract. Used on most state change functions.

modifier onlyOwner () {}

* // This function can transfer the ownership of the contract. Only existing owner can transfer it to the new owner.

function setOwner(address \_newOwner) public onlyOwner {}

* // earn points
* //The function creates a new coupon and assigns the input data to the coupon.

function allocatePoints(string memory \_userIdString, uint256 \_points, uint256 \_expiryDate, string memory \_refVia) public onlyOwner {}

* // Goes through all coupon Ids. Checks for the expired coupons and burns the points of the expired coupons. Filters the coupons with the user ID. Adds up all the points of a user's coupons and updates the 'userIdToPoints' mapping.

function updateUserPoints(bytes32 \_userId) private {}

* //gets points available to a user. calls the 'updateUserPoints' function first to check for expired coupons. returns the 'userIdToPoints' mapping.

function getUserPoints (string memory \_userIdString) public onlyOwner returns(uint256) {}

* //redeemUserPoints
* // The function will check the current points the user has. Goes through all the coupons ids in increasing order. Filters the coupons with user Id and availability of points. Subtracts points from coupon Ids untill the desired number of points have been extracted.

function redeemUserPoints (string memory \_userIdString, uint256 \_points) public onlyOwner {}

* // Modify the number of points in a coupon. Checks the coupon for expiry first. Then replaces the coupon points with new points.

function modifyCouponPoints (uint256 \_couponId, uint256 \_points ) public onlyOwner {}

## Events

Events are used mainly as return statements from function calls. Events are also useful to retrieve smart contract usage history.

* Event emitted whenever points are allocated to a user.

 event pointsAllocated(bytes32 indexed \_userId, uint256 indexed \_points, uint256 indexed \_couponId);

* Event emitted whenever a coupon expires and its corresponding points are removed from circulation.

   event pointsExpired(bytes32 indexed \_userId, uint256 indexed \_points, uint256 indexed \_couponId);

* Event emitted whenever a user’s points are redeemed and the points are removed from circulation.

event pointsRedeemed(bytes32 indexed \_userId, uint256 indexed \_points, uint256 indexed \_couponId);

* Event emitted whenever a user’s points are queried. The event is then caught by the application.

   event userPoints (bytes32 indexed \_userId, uint256 indexed \_points);

* Event emitted whenever the ownership of the smart contract/ data is changed to a new address.

   event ownerSet (address indexed \_newOwner);

* Event emitted whenever a coupon’s points are modified.

   event couponModified (bytes32 indexed \_userId, uint256 indexed \_points, uint256 indexed \_couponId);

# API

The API is written to be used by the company’s application in order to interact with the smart contract. It uses “web3.js” libraries to call the end points of a node.

* //\*\*\*\*FORMAT OF API CALLS TO THE BLOCKCHAIN. \*/
* //POPPULATED WITH SAMPLE VALUES.
* // Install web3.js library
* // npm install web3
* // OR
* // yarn add web3
* // documentation here : https://web3js.readthedocs.io

import Contract from "web3-eth-contract";

import Web3 from "web3";

* //Set the API endpoint like so...

const apiEndPoint = "HTTP://127.0.0.1:8545";

* // Set up multiple endpoints with router for production.

const web3 = new Web3(apiEndPoint);

Contract.setProvider(apiEndPoint);

* // Set up API calls

const ContractAddress = "0xc2edC08fd8bE4C0e327B19C8577edCF2bae22999";

const compiledContract = require("../compiledContract.json");

const ABI = compiledContract["abi"];

const myContract = new Contract(ABI, ContractAddress);

const hexToEvent = {

  "0x46aefb6c4c70c1593913b0d8f3ff25987bb344f5b361fef5a20cae35ff2bbe84":

    "pointsAllocated",

  "0x97ff9ddf4b40d9907f04692208f062caaffa52ef7354486e482ad5e0c5655c96":

    "userPoints",

  "0xe18bf1e48e24f90f6c01a701b2a85e9b7dbde20be612d81046eb0d3da3abe4ff":

    "pointsBurned",

};

* //Setup public-private key pair for encryption.

const myAccount = "0xc388C5e09964A06684C782C6E8090B5CF50c40EA"; // keys shall be provided. Public key can be shared.

const privateKey =

  "e23b2d073aa65b8a57d98525ed241c4767afdd5a4048ec064df4a04dff43730c"; // keys shall be provided. Private key should never ever be shared.

* // API request for Earning points. Allocate points to a user like so...
* // parameters include userId(String), points(integer), expiryDate(epoch time integer), refVia(string)
* // Returns object
* //   {status: Status code of response. 200 - Success, 500 - Failed.
* //   response: "Success",
* //   userId: User Id as a string,
* //   points: Points allocated to the user with this request. Integer,
* //   couponId: The unique Id of the coupon created,
* //   expiryDate: Expiry date of the created coupon in epoch time. Integer.
* //   refVia: The refVia input saved with the coupon. String.}

const allocatePoints = async (userId, points, expiryDate, refVia) => {};

* // API request for Burning points / redeeming points from a user like so...
* // parameters include userId(string), points(integer)
* // Returns object
* //   {status: Status code of response. 200 - Success, 500 - Failed.
* //   response: "Success"}

const redeemPoints = async (userId, points) => {};

* // API request for getting a user's coin balace...
* // parameters include userId(string)
* // Returns object
* //   {status: Status code of response. 200 - Success, 500 - Failed.
* //   response: "Success",
* //   userId: User Id as a string,
* //   points: Total Points available to the user. Integer}

const getPoints = async (userId) => {};

* // API request for getting a user's coin history...
* // parameters include userId(string)
* // Returns object
* //   {status: Status code of response. 200 - Success, 500 - Failed.
* //   response: "Success",
* //  transactionHistory: Array of all transactions of the user.
* // Array elements are objects with the following format:
* //   {userId: User Id as a string,
* //   type: Type of transaction. Earned, Burned, Redeemed or Modified.
* //   points: Points allocated to the user with this request. Integer,
* //   couponId: The unique Id of the coupon included in the transaction.}

const getHistory = async (userIdString) => {};

* // API request for getting the total number of coins that can ever be in circulation.
* // Max supply is specified during contract creation.
* // returns the max supply of coins(integer)
* // Returns object
* //   {status: Status code of response. 200 - Success, 500 - Failed.
* //   response: "Success",
* //   maxSupply: Max supply of coins. Integer}

const getMaxSupply = async () => {};

* // API request for getting the total number of coins in circulation at the time of api call...
* // returns the current supply of coins(integer)
* // Returns object
* //   {status: Status code of response. 200 - Success, 500 - Failed.
* //   response: "Success",
* //   currentSupply: Current supply of points.}

const getCurrentSupply = async () => {};

* // API request for all details of a coupon with using the unique coupon id.
* // parameters include couponId(integer)
* // Returns object
* //   {status: Status code of response. 200 - Success, 500 - Failed.
* //   response: "Success",
* //   userId: User Id as a string. The owner of this coupon,
* //   points: Points assigned in this coupon. Integer,
* //   couponId: The unique Id of the this coupon,
* //   expiry: Expiry date of the coupon in epoch time. Integer.
* //   refVia: The refVia input saved with the coupon. String.}

const couponDetails = async (couponId) => {};

1. // API request for modifying the points in a coupon using the coupon id..
2. // parameters include couponId(integer), points(integer).
3. // Returns object
4. //   {status: Status code of response. 200 - Success, 500 - Failed.
5. //   response: "Success",
6. //   userId: User Id as a string,
7. //   points: Modified points assigned in this coupon. Integer,
8. //   couponId: The unique Id of the coupon }

const modifyCoupon = async (couponId, points) => {};

# Setting up Nodes for the Network

The network uses Ethereum environment and comprises multiple nodes running the “geth” protocol.

The nodes can be set up in a windows OS by using the following steps.

1. Step1

Download and install Geth client from here: <https://geth.ethereum.org/downloads/>

1. Step2

Create a new directory for the node. Multiple directories can be created for running multiple nodes in the same machine. One directory can host only one node at a time. So, all nodes should be started in separate directories.

Create a new account for every node using a CLI by running this command.

Geth –datadir ./node1 account new

Change “node1” to the desired name of the node.

A new account will be created for the directory. The account address should be copied and saved at a secure location. The private key for the account can not be retrieved. The encrypted private key will be saved in the “keystore” folder inside the main directory. Geth uses this keystore folder to unlock the account.

1. Step3

Create the genesis block by using the “puppeth” tool which is installed automatically with “geth” client. It can be initiated by simply typing “puppet” in the CLI.

puppeth

Follow the instructions afterwards and export the genesis block files (e.g. testgeth.json)

1. Step4

Initiate the nodes by using the following command in the CLI

geth –datadir ./node1 init ../testgeth.json

It will initiate the node directories but won’t start the nodes.

1. Step5

If this is the initiation of the network, initiate a boot node first by using the following command

bootnode -genkey boot.key

bootnode -nodekey boot.key

It will start the boot node at default IP address (localhost) and will return the enode path of the boot node. e.g.

enode://120dec4a123587e97a11cda1c2335585e9800c505c8be1c5a08b9445f0e3089e9f5ba2e2c7bb41e0550517ee27dc33b193559bbfff3f938869a701a4870da7a2@127.0.0.1:0?discport=30301

The IP address and port of the bootnode can be specified by using the “-addr” flag like so…

bootnode -nodekey ./boot.key -addr “127.0.0.1:30301”

The enode should be saved securely.

1. Step6

Start the nodes by using the following command

geth --datadir ./node1 --port 30302 --http --http.port 8545 --http.api 'personal,db,eth,net,web3,txpool,miner' --bootnodes enode://120dec4a123587e97a11cda1c2335585e9800c505c8be1c5a08b9445f0e3089e9f5ba2e2c7bb41e0550517ee27dc33b193559bbfff3f938869a701a4870da7a2@127.0.0.1:0?discport=30301 --networkid 4466 -unlock 0xd0Bf995a6eC2D5320f47Dd64a38B8f5b6850ca1E --password pass.txt --mine console --miner.gasprice 0 --allow-insecure-unlock --http.corsdomain "\*"

This is starting the node named “node1” located at “./node1” at port 30302 using the enode the bootnode specified in the previous step. The port 8545 can be used by the API to connect to this node. At least two mining nodes are required to run the network.