Написание собственной криптовалюты в блокчейн сети Ethereum.

ЗАДАЧА:

В рамках лабы была поставлена задача познакомиться с технологией блокчейна. Посмотреть, какие API существуют для работы с блокчейном, а также написать приложение, в котором будет выпуск своих коинов, продажа коинов другим заинтересованным лицам, покупка чего-либо за эти коины и операции переводов.

Знакомство с блокчейном я начал сразу с того, что на слуху было с самого начала, и где эта технология была применена впервые, а именно криптовалюта Bitcoin, первый выпуск которой состоялся в начале 2009 года. В Bitcoin все транзакции происходят без посредников, они неабротимы, операции нельзя отменить. Одна из главных особенностей – система полностью децентрализована, копия базы хранится у каждого пользователя системы локально на компьютере, а программы клиенты соединяются друг с другом в сеть, каждый узел которой равноправен и самодостаточен. Комиссия за проведение операций назначается отправителем добровольно и ее размер влияет на приоритет при обработке транзакций. Объединение транзакций в блок и присоединение блока к блокчейну – этот процесс называется майнингом. В базе нет отдельных записей о текущем количестве биткоинов у какого-либо владельца. Лишь на основании цепочек транзакций становится понятно текущее количество биткоинов.

Биткоин по сути является частным случаем применения блокчейна. В 2015 году была запущена сеть Ethereum – это блокчейн, похожий на Bitcoin, платформа для создания децентрализованных онлайн-сервисов, реализуемых в виде смарт-контрактов. Как и Bitcoin, Ethereum имеет собственную криптовалюту называемую Эфиром (Eth).

Смарт-контракт - компьютерный алгоритм, предназначенный для формирования, контроля и предоставления информации о владении чем-либо. В более узком смысле под смарт-контрактом понимается набор функций и данных (текущее состояние), находящихся по определённому адресу в блокчейне. Когда идет обращение к смарт-контракту, вы либо читаете данные из его структуры данных(бесплатно, общедоступно), либо записываете их туда(платно, от имени своего кошелька). Поскольку операция записи будет происходить в сеть блокчейн Ethereum, вам необходимо будет заплатить за такую транзакцию майнерам. Чем больше вычислений внутри вашего смарт-контракта, тем больше необходимо будет заплатить. Умные контракты написаны на языке программирования Solidity, который похож на Javascript.

Например, новую криптовалюту(токен), которую можно будет использовать внутри сети Ethereum и покупать за Eth, возможно реализовать в виде смарт-контракта. Для того чтобы смарт контракт считался токеном он должен соответствовать стандарту, определяющему поведение этого токена. Наличие таких стандартов позволяет биржам и провайдерам кошельков создавать единую кодовую базу, взаимодействующую с любым смарт-контрактом поддерживающим этот стандарт. Вот некоторые из этих стандартов:

ERC-20: стандарт содержит 6 основных функций, которые должны быть реализованы смарт-контрактом:

* totalSupply() return (uint256) – получение общего количества токенов в сети.
* balanceOf(address \_owner) returns (uint256) – получение баланса токенов владельца аккаунта в сети
* transfer(address \_to, uint256 \_value) returns (bool) – перевод количество токенов на адрес \_to с адреса отправителя (отправителем в данном случае будет тот, кто вызывает метод)
* transferFrom(address \_from, address \_to, uint256 \_value) returns (bool) – перевод количества токенов с адреса \_from на адрес \_to
* approve(address \_spender, uint256 \_value) returns (bool) – разрешение адресу \_spender тратить какое-то количество токенов с кошелька отправителя.
* allowance(address \_owner, address \_spender) returns (uint256) – получение количество токенов, которое \_spender может тратить из кошелька \_owner.

События:

* Transfer(address indexed \_from, address indexed \_to, uint256 \_value) – срабатывает после отправки токенов
* Approval(address indexed \_owner, address indexed \_spender, uint256 \_value) – срабатывает когда вызвана функция approve(address \_spender, uint256 \_value)

ERC-721 – это стандарт токенов, который определяет не заменяемые токены в блокчейне Ethereum. Т.е., например, токен стандарта ERC-20 является заменяемым, т.к. любая единица токена может быть заменена другой и оба они будут иметь одинаковую ценность. Стандарт ERC-721 был введен, чтобы позволить каждому создавать токены, которые будут уникальными друг для друга. Не существует двух одинаковых токенов ERC-721, их можно считать уникальными предметами коллекционирования.

Функции, которые должны быть реализованы:

* balanceOf(address \_owner) returns (uint256) – получение баланса токенов владельца аккаунта в сети
* ownerOf(uint256 \_tokenId) returns (address) – получение адреса владельца конкретного токена
* safeTransferFrom(address \_from, address \_to, uint256 \_tokenId) – передача владения токеном
* safeTransferFrom(address \_from, address \_to, uint256 \_tokenId, bytes data) – передача владения токеном с возможностью передачи дополнительных данных получателю через параметр bytes
* transferFrom(address \_from, address \_to, uint256 \_tokenId) – передача владения токеном, отправитель ответственен за то, что получатель способен получить токен, иначе токены будут утрачены безвозвратно
* approve(address \_approved, uint256 \_tokenId) – разрешить адресу \_approved передавать токен
* setApprovalForAll(address \_operator, bool \_approved) – разрешить или запретить адресу \_operator передавать любые токены
* getApproved(uint256 \_tokenId) returns (address) – получения адреса, который может передавать токен
* isApprovedForAll(address \_owner, address \_operator) returns (bool) – определение, является ли \_operator полностью доверенным для передачи токенов с адреса \_owner

События:

* Transfer(address indexed \_from, address indexed \_to, uint256 indexed \_tokenId) – срабатывает после отправки токена
* Approval(address indexed \_owner, address indexed \_approved, uint256 indexed \_tokenId) – срабатывает после подтверждения передачи токена с другого адреса
* ApprovalForAll(address indexed \_owner, address indexed \_operator, bool\_approved) – срабатывает, когда оператору \_operator разрешается или запрещается передавать токены с адреса \_owner

Другие стандарты:

ERC-223 - стандарт, определяющий тип токена, аналогичный ERC-20, с дополнительными функциями. Если токены ERC-20 отправляются по умному контракту, не созданному для работы со стандартом ERC-20, эти токены будут недоступны навсегда. ERC-223 содержит метод tokenFallback, который обеспечивает отправку токенов только в контракты с соответствующей функциональностью.

ERC-777 - еще один стандарт, пытающийся улучшить токен ERC-20. Он определяет расширенные функции, включая операторов для отправки токенов от имени другого адреса, и предлагает пользователям больший контроль над их токенами.

ERC-1155 - интерфейс, который позволяет умным контрактам управлять несколькими типами токенов. Развернутый контракт может включать в себя любую комбинацию из заменяемых и незаменяемых токенов

ERC-1337 - стандарт токенов для повторяющихся подписок в блокчейне Ethereum. Этот стандарт фокусируется на функциональной совместимости, позволяя кошелькам заключать контракты на повторяющиеся платежи и предоставляя соответствующий пользовательский интерфейс для управления и отмены подписок.

Итак, если мы вернемся к поставленной задаче, то становится понятно, что “коины” в задаче можно считать “токенами” в блокчейне Ethereum. API – это инструменты для создания, управления, развертывания умных контрактов. “Выпуск своих токенов” – это развертывание умного контракта одного из стандартов (ERC-20, ERC-721 и т.д.) с инициализацией значения totalSupply – общего количества токенов в блокчейне Ethereum. “Операции перевода” – вызов функций transfer или transferFrom смарт-контракта токена.

Что же такое “продажа токенов” – кому и зачем они нужны? Один из популярных методов распространения токенов – это Initial Coin Offering (ICO), способ для компании привлечь капитал для своего бизнеса, создав свой токен, который может быть приобретен инвесторами за Eth. Механизм ICO также может быть реализован с помощью смарт-контракта в сети Ethereum.

В этой задаче будем использовать базовый стандарт токена ERC-20. Что бы участвовать в ICO, инвестор должен подключиться к блокчейну Ethereum со своей учетной записью. У этой учетной записи есть адрес кошелька, на котором можно хранить Eth, а также токены ERC-20, которые инвестор собирается приобретать. Когда инвестор покупает токен на сайте ICO, он отправляет Eth из своего кошелька в смарт-контракт, который перенаправляет Eth владельцу смарт-контракта ICO, а токены инвестору. Смарт-контракт устанавливает цену токена, а также определяет поведение ICO. Например, можно добавить white-листы, чтобы ограничить, какие инвесторы могут покупать токены.

Думаю, с теорией пока достаточно, пора бы что-то и написать.

Перед тем как приступить к кодированию, нам понадобятся следующие зависимости:

* Node Package Manager (NPM), который идет в комплекте с [Node.js](https://nodejs.org/en/). Правильность установки можно проверить, написав в терминале:

> node -v

* [Truffle Framework](https://www.trufflesuite.com/), позволяет создавать децентрализованные приложения на блокчейне Ethereum. Он предоставляет набор инструментов по написанию, компилированию, тестированию и развертыванию умных контрактов на языке программирования Solidity. Truffle можно установить с NPM в командной строке следующим образом:

> npm install -g truffle

* [Ganache](https://www.trufflesuite.com/ganache), предоставляет возможность развертывания своего собственного локального блокчейна на базе Ethereum в памяти. Вы можете установить Ganache по [ссылке с сайта Truffle Framework](https://www.trufflesuite.com/ganache). Он предоставляет 10 автоматически создаваемых аккаунтов в нашем локальном блокчейне. Каждый аккаунт содержит фейковых 100 Eth.
* [Metamask](https://chrome.google.com/webstore/detail/metamask/nkbihfbeogaeaoehlefnkodbefgpgknn?hl=en), расширение для Google Chrome. Перед тем как использовать блокчейн сеть, мы сперва должны подключиться к ней. Это именно то, для чего нам нужен Metamask. Мы сможем подключиться к нашему локальному блокчейну из браузера с помощью нашего личного аккаунта, а также взаимодействовать с нашими смарт-контрактами.
* [Visual Studio Code](https://code.visualstudio.com/) с [расширением](https://marketplace.visualstudio.com/items?itemName=juanblanco.solidity) для подсветки синтаксиса Solidity. Эта зависимость необязательна, можно использовать любой текстовый редактор.

Теперь, когда все зависимости установлены, давайте напишем наш смарт-контракт токена по стандарту ERC-20.

Для начала определим рабочий каталог:

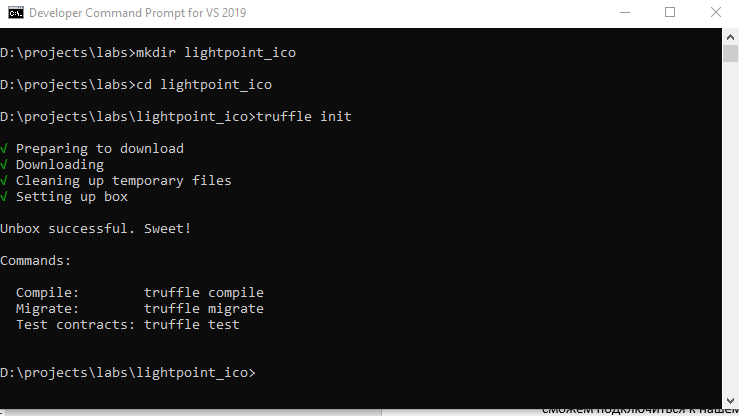
mkdir lightpoint\_ico

И с помощью truffle инициализируем проект.

cd lightpoint\_ico

truffle init

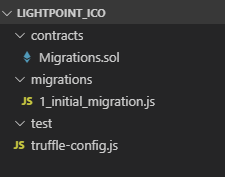
Должно получиться что-то вроде этого



Откроем проект в VS Code:

code .

Первоначальная структура проекта имеет следующий вид:



Migrations.sol – смарт-контракт для управления миграциями. Миграции это JavaScript файлы, которые помогают развертывать смарт-контракты в сети Ethereum. По мере развития вашего проекта вы будете создавать новые сценарии миграции для дальнейшей эволюции блокчейна. История ранее выполненных миграций записывается по цепочке через специальный смарт-контракт Migrations.sol.

Чтобы запустить миграции, нужно выполнить truffle migrate. Если ваши предыдущие миграции были выполнены успешно, truffle migrate начнет выполнение только с новых добавленных миграций. Если новых миграций нет truffle migrate не предпримет никаких действий. Вы можете использовать опцию –reset, чтобы запустить все миграции с начала.

truffle-config.js – конфигурационный файл, по умолчанию сконфигурирован на использование единственной development сети, запущенной на 127.0.0.1:8545.

module.exports = {

  networks: {

    development: {

      host: "127.0.0.1",

      port: 8545,

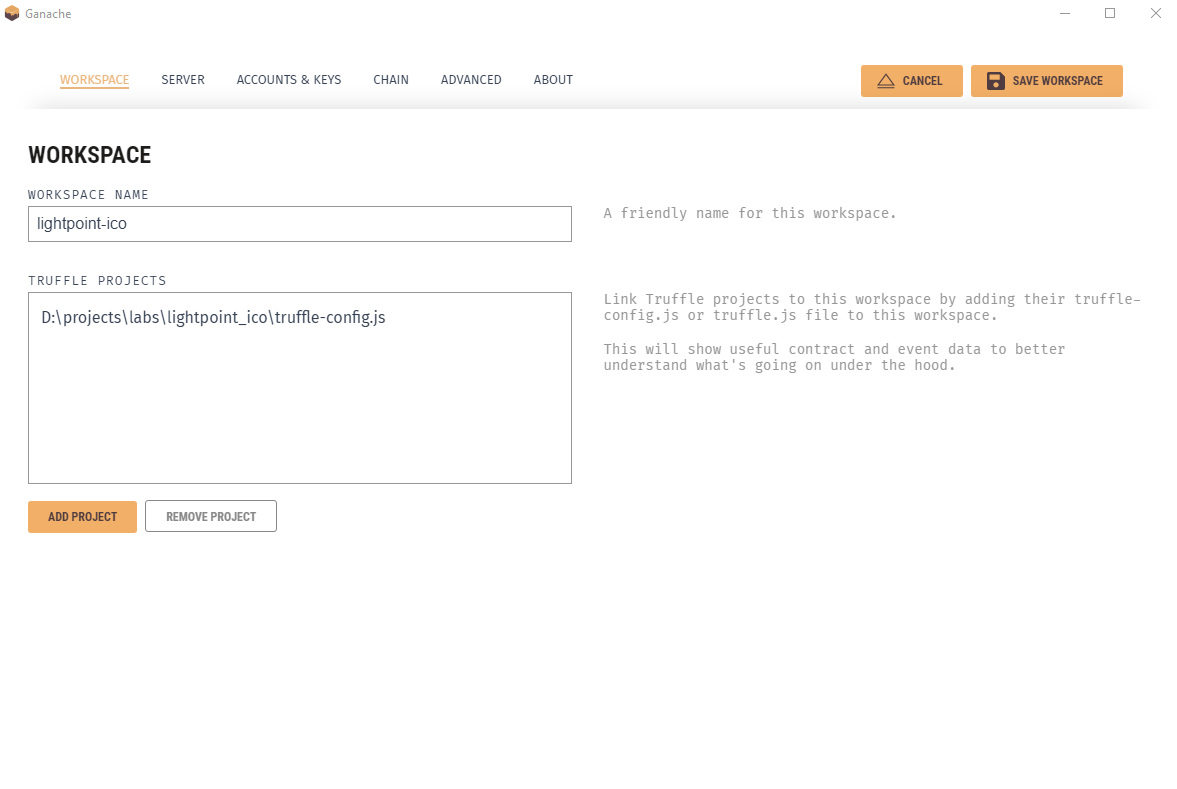
      network\_id: "\*",

     }

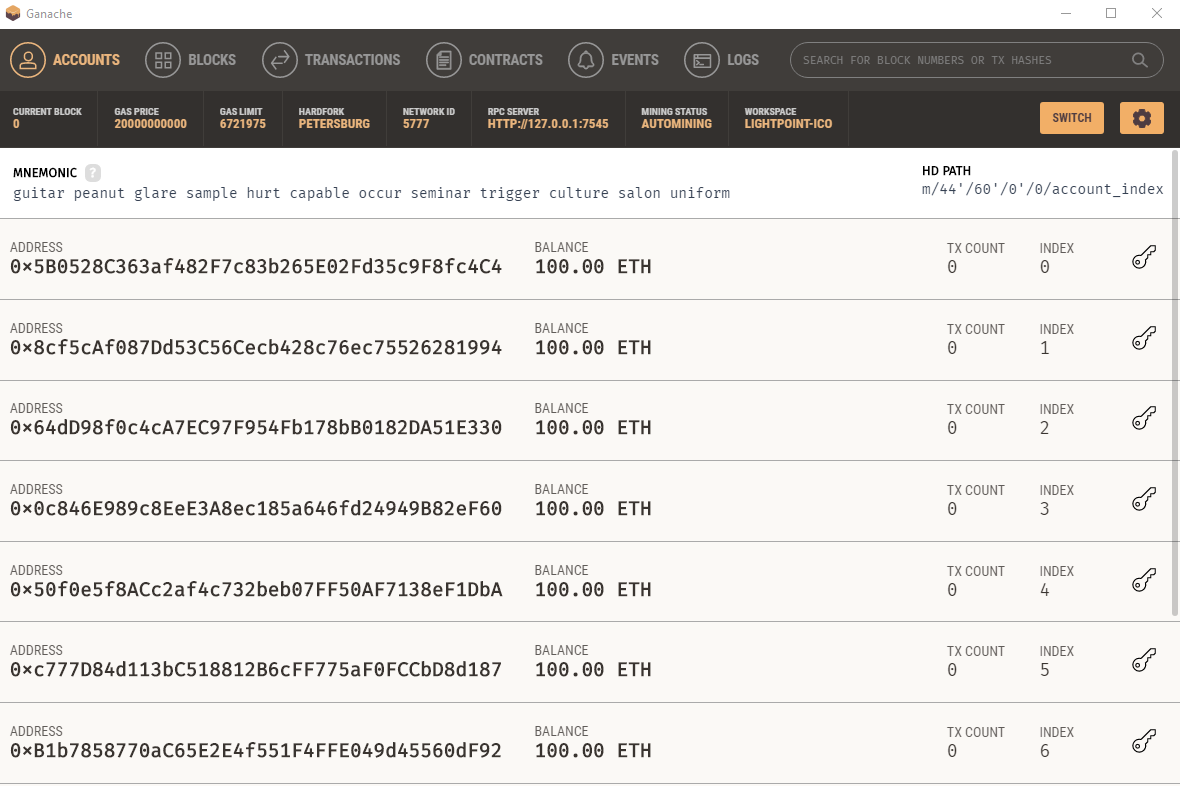
}

}

На этом этапе запустим свой локальный блокчейн, используя Ganache. Добавим новый workspace, и укажем путь до triffle-config.js файла.



Нажмем Save workspace.



Вуаля, локальный in-memory блокчейн запущен.

На этом этапе пока добавим в файл конфигурации две настройки contracts\_directory и contracts\_build\_directory, которые будут указывать место хранения исходных файлов контрактов и их abi (abstract binary interface), поместим контракт Migrations.sol в папку ‘./src/contracts’. Также изменим порт, для development сети на 7545, на котором запущен Ganache. Конфигурационный файл примет вид:

module.exports = {

  networks: {

    development: {

      host: "127.0.0.1",

      port: 7545,

      network\_id: "\*",

     }

    },

  contracts\_directory: './src/contracts/',

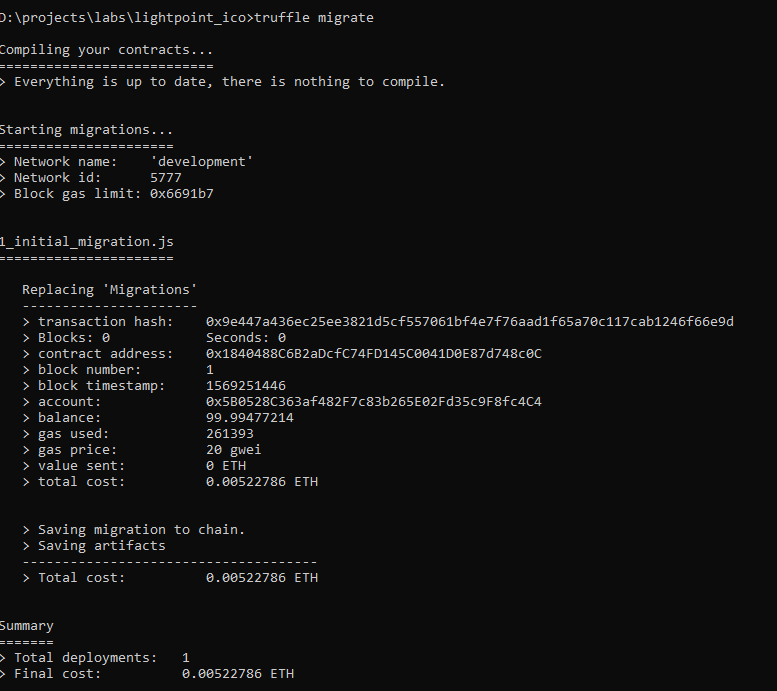
  contracts\_build\_directory: './src/abis/'

}

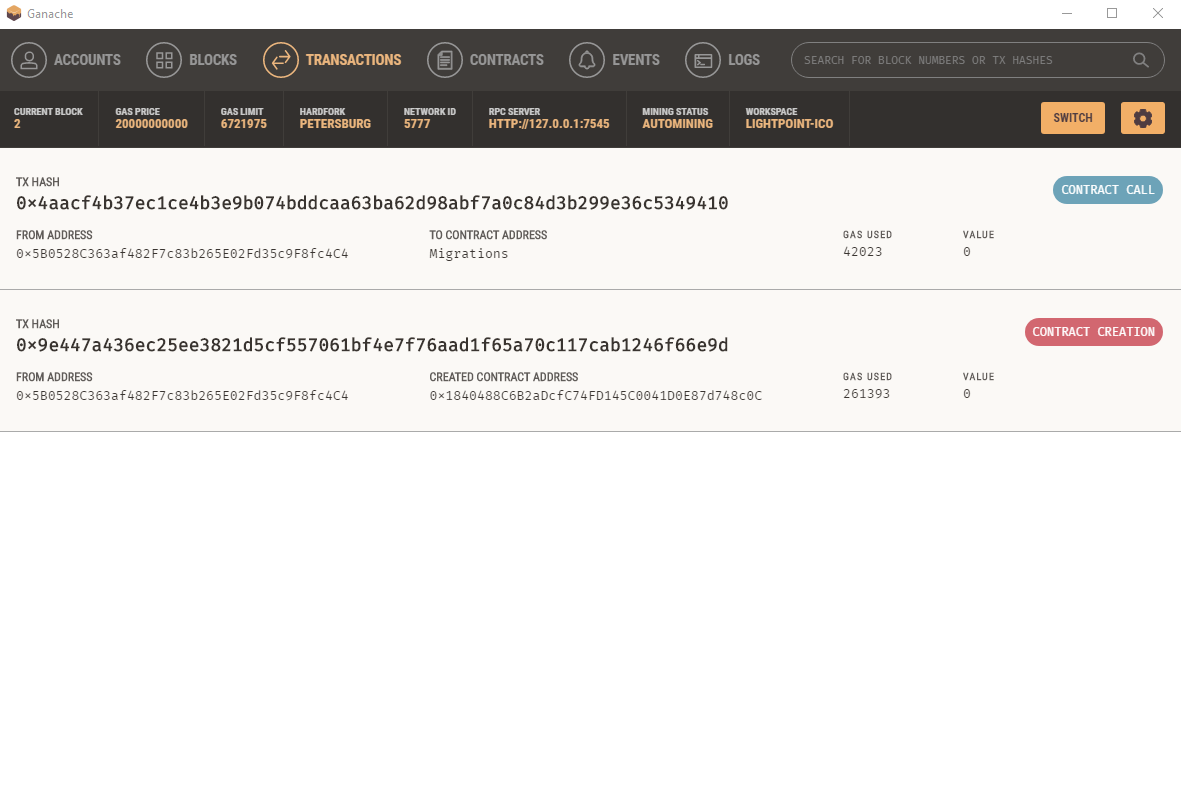
Теперь выполним следующую команду:

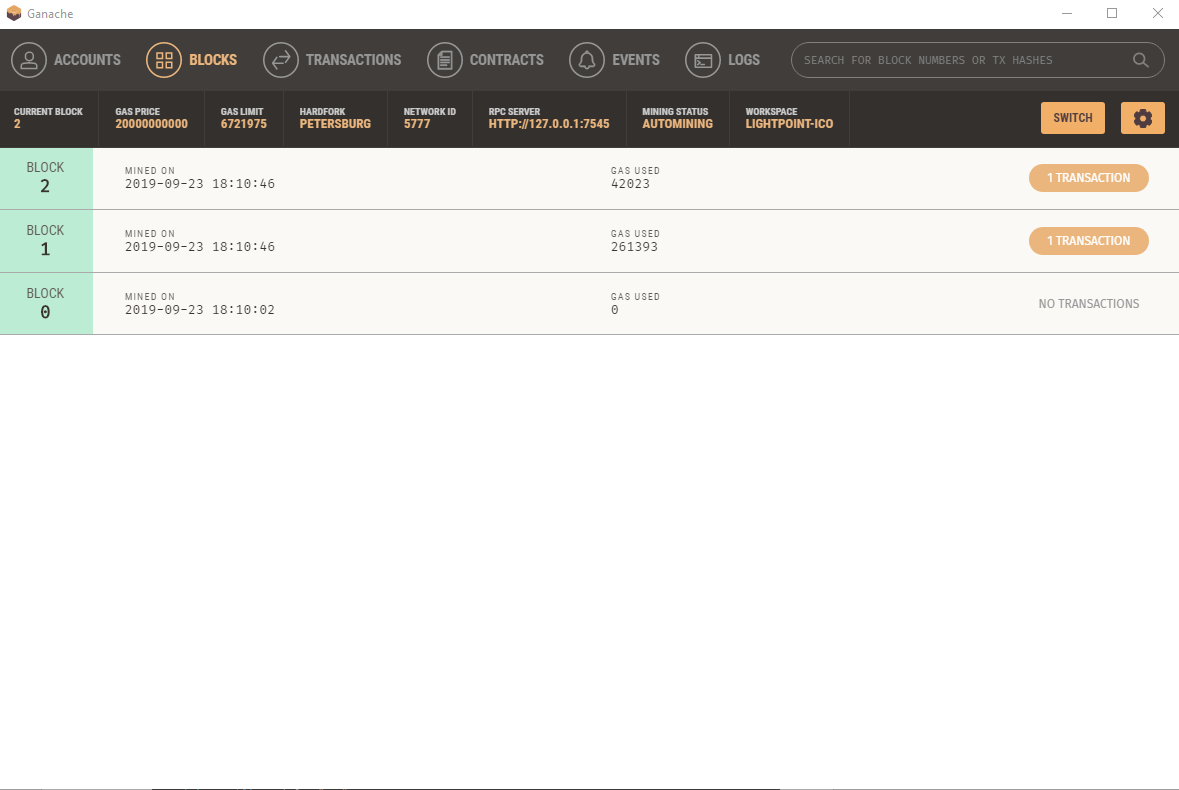
$ truffle migrate

Результатом должно быть что-то вроде этого:

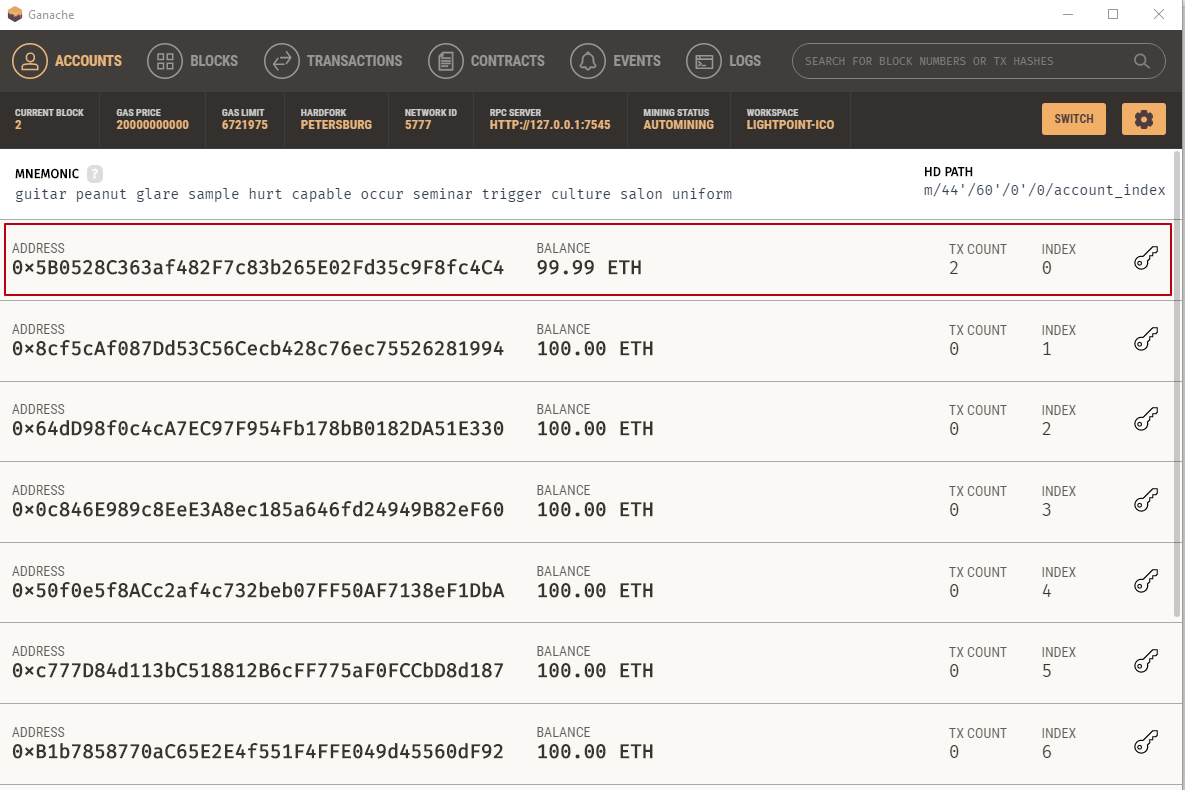


В Ganache мы увидим некоторую активность. Было добавлено 2 транзакции, одна по созданию смарт-контракта Migrations, вторая на вызов функции setCompleted этого контракта. Транзакции добавлены в блоки и присоединены к блокчейну. На создание и на вызов функции было затрачено некоторое кол-во «топлива» (награда майнеру на подключение блока к блокчейну), размер которого влияет на скорость деплоя.





По умолчанию, все контракты разворачиваются от имени первого аккаунта в списке, но это поведение можно настроить в атрибутах функции deploy.



Смарт-контракт LightpointToken

Теперь давайте напишем смарт контракт токенов. Добавим файл LightpointToken.sol в директорию с контрактами и напишем в него следующее:

pragma solidity ^0.5.0;

contract LightpointToken {

    uint256 public totalSupply;

    constructor (uint256 \_initialSupply) public {

        totalSupply = \_initialSupply;

    }

}

Создание контракта очень похоже на объявление класса в ООП (различие лишь в ключевом слове contract вместо class. Также есть конструктов контракта, в параметрах которого он принимает общее количество доступных токенов.

Ключевое слово pragma может использоваться для включения определенных функций или проверок компилятора. Оно также указывает, с какой версией компилятора исходный файл будет компилироваться (в данном случае указано ^0.5.0, это значит, что файл будет компилироваться на компиляторе версии не ниже 0.5.0, но не на 0.6.0 и выше).

Модификатором public помечаются свойства и методы, которые можно будет вызывать при обращении к смарт-контракту.

Теперь развернем наш новый контракт в тестовой сети. Для этого добавим новую миграцию 2\_deploy\_contracts.js с содержимым:

const LightpointToken = artifacts.require("LightpointToken");

module.exports = function(deployer) {

  deployer.deploy(LightpointToken, 1000000);

};

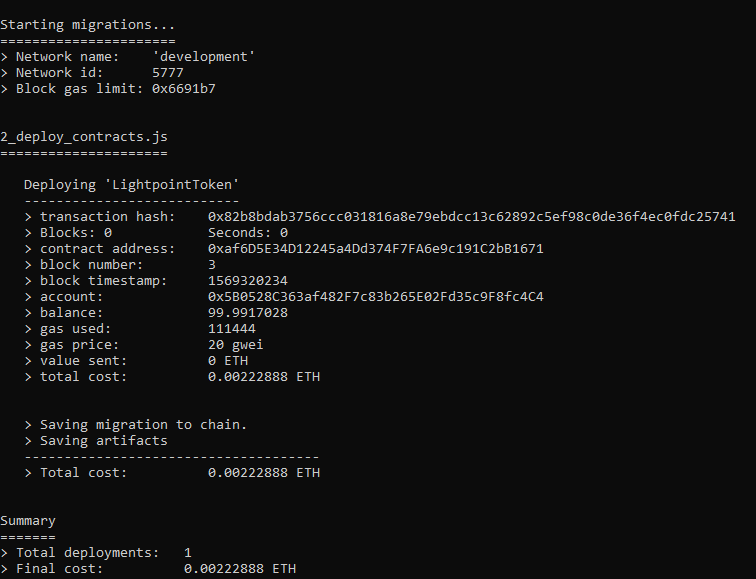
Заметьте, что имя файла должно начинаться с префикса номера, чтобы определить порядок успешного выполнения миграций. Также, обратите внимание, что параметры конструктора указываются через запятую после указания типа контракта, который мы хотим разворачивать. Подробнее про механизм деплоя и возможные варианты можно почитать по ссылке [https://www.trufflesuite.com/docs/truffle/getting-started/running-migrations#deployer-api](https://www.trufflesuite.com/docs/truffle/getting-started/running-migrations%23deployer-api).

В нашем случае инициализируем totalSupply значением 1 000 000 (т.е. в наличие у нас будет иметься 1 миллион токенов).

Теперь давайте попробуем запустить миграцию.

$ truffle migrate

Результат должен быть примерно следующий:



Теперь давайте проверим работу нашего смарт контракта. Сделать мы это можем в консоле truffle. Для этого введем команду:

$ truffle console

Получим экземпляр смарт-контракта в сети:

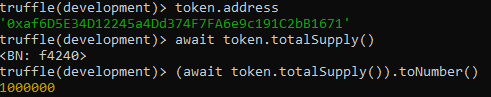
truffle(development)> LightpointToken.deployed().then(function(tokenInstance) { token = tokenInstance })

Теперь мы имеем доступ к объекту token. Мы также могли бы это сделать, используя механизмы async/await следующим образом:

truffle(development)> token = await LightpointToken.deployed()

Далее мы в большей степени будем использовать второй вариант вызова функций.

Ниже мы видим, что обращаясь к полю address мы можем получить адрес в сети, по которому расположен наш смарт-контракт, также мы уже можем получить значение totalSupply смарт-контракта.



Выйти из консоли мы можем командой:

truffle(development)> .exit

Либо дважды нажать комбинацию клавиш ctrl+c.

Теперь, рассмотрим как в truffle пишутся тесты. Для начала добавим тестовый файл в директорию ./test/ для тестирования нашего смарт контракта, в моем случае я назвал его ‘LightpointToken.test.js’. Несколько слов о тестировании: очень важно тестировать наши смарт-контракты, т.к. код смарт-контракта в блокчейне неизменяем, как и любые другие данные, попадаемые в блокчейн, должны быть неизменяемы, такова его концепция. Поэтому важно, чтобы смарт-контракты не содержали багов, т.к. если мы развернем его и найдем баг, то лучшее что мы сможем сделать – это выключить смарт контракт и развернуть его новую версию заново. Мы должны избегать этого, если это возможно.

const LightpointToken = artifacts.require("LightpointToken");

contract('LightpointToken', async function(accounts) {

    var tokenInstance;

    before(async() => {

        tokenInstance = await LightpointToken.deployed();

    });

    it('sets the total supply upon deployment', async function() {

        var totalSupply = await tokenInstance.totalSupply();

        assert.equal(totalSupply.toNumber(), 1000000, 'sets the total supply to 1,000,000');

    });

});

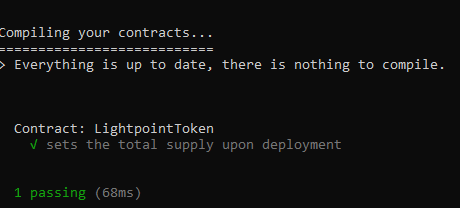
Также, как мы делали это в миграциях мы должны импортировать наш смарт-контракт из файла.

Truffle содержит в себе собственный test framework в комплекте с ним идет библиотека утверждений Chai. В данном примере до начала тестирования мы получаем экземпляр смарт-контракта в сети и проверяем, что инициализированное значение totalSupply должно равняться 1 000 000.

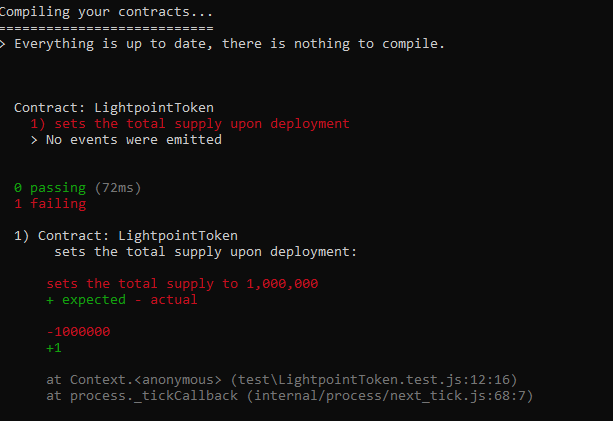
Запустим тест следующей командой:

$ truffle test

В результате мы должны увидеть следующее:



Если мы изменим ожидаемое значение, мы получим ошибку (в моем примере я заменил 1000000 на 1):



Теперь добавим необязательные поля для токена, но полезные и удобные для проведения операций обмена, а именно имя и символ:

pragma solidity ^0.5.0;

contract LightpointToken {

    string public name = "Lightpoint Token";

    string public symbol = "LPT";

    uint256 public totalSupply;

    constructor (uint256 \_initialSupply) public {

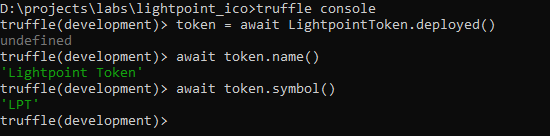
        totalSupply = \_initialSupply;

    }

}

В рамках этого документа: новые изменения, которые будут вноситься в код я буду выделять более большим шрифтом.

Выполним миграцию с флагом –reset, и проверим, что данные действительно появились в смарт-контракте:



Теперь добавим возможность отслеживать баланс для каждого аккаунта, который может в будущем владеть токенами.

В стандарте [ERC-20](https://eips.ethereum.org/EIPS/eip-20#balanceof) указано, что контракт должен содержать функцию balanceOf, которая будет возвращать баланс токенов для запрашиваемого адреса в сети. Для этого в Solidity есть специальная конструкция mapping(T1 => T2), которую можно описать как массив объектов ключ-значение.

pragma solidity ^0.5.0;

contract LightpointToken {

    string public name = "Lightpoint Token";

    string public symbol = "LPT";

    uint256 public totalSupply;

mapping(address => uint256) public balanceOf;

    constructor (uint256 \_initialSupply) public {

        balanceOf[msg.sender] = \_initialSupply;

        totalSupply = \_initialSupply;

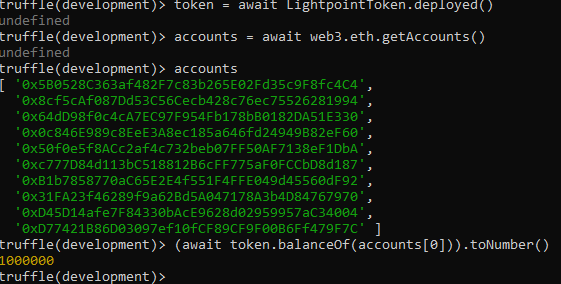
    }

}

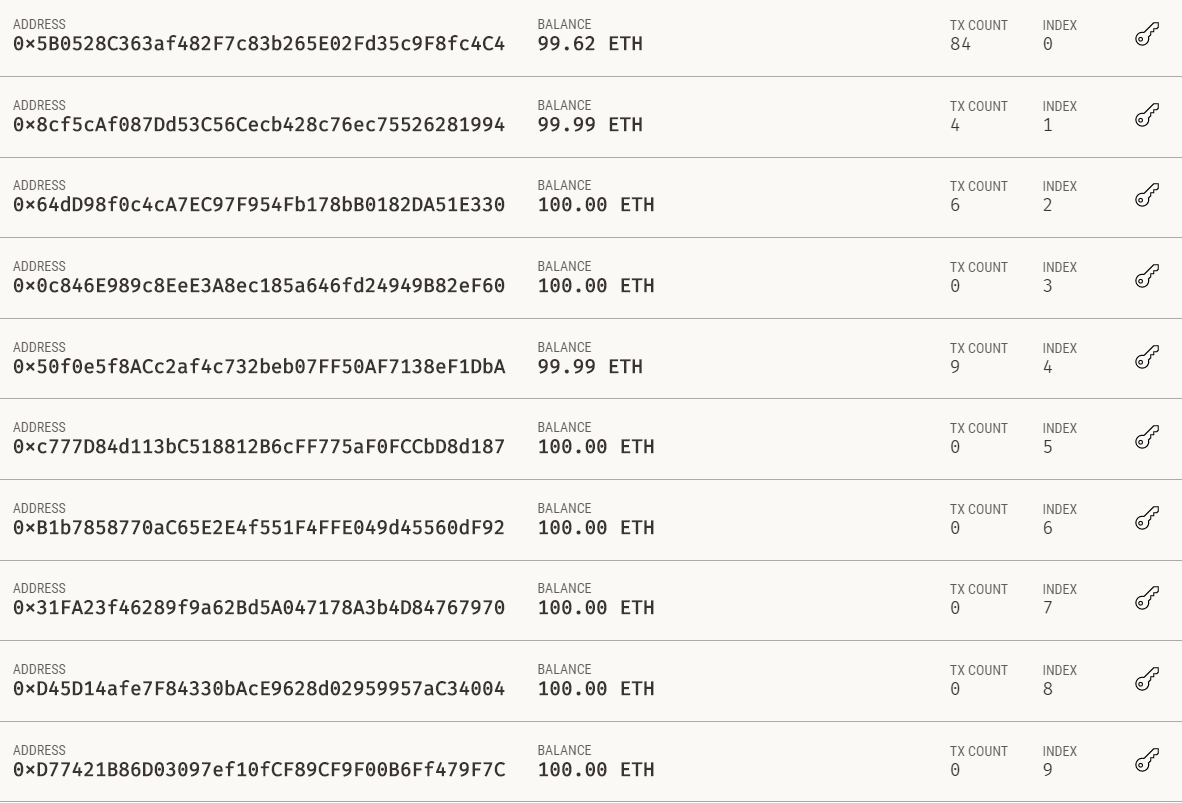
В данном случае конструкция mapping(address => uint256) public balanceOf; будет соответствовать сигнатуре функции balanceOf стандарта ERC-20. Первое, что мы сделали с этой конструкцией, это определили, кто будет владеть всем доступным количеством токенов, это будет адрес с которого развертывается контракт (будем называть его администратором). Конструктор смарт-контракта как раз таки вызывается единажды при развертывании контракта в сеть. Что такое msg.sender? В Solidity есть глобальная переменная msg, которая включает в себя несколько разных значений. Эта переменная по сути является метаданными, которые передаются в функцию при каждом ее вызове. В данном случае msg.sender является адресом того, кто вызывает функцию. Поэтому, мы установили баланс администратора, равным общему количеству доступных токенов в сети.

Проверим что это действительно работает:

$ truffle migrate –reset



Думаю, тут нужно объяснить, что происходит. web3 – это экземпляр класса Web3, который определен в библиотеке web3.js. Подробная документация доступна здесь <https://web3js.readthedocs.io/en/v1.2.1/>. Я только скажу, что эта библиотека содержит огромный функционал по работе с локальном и удаленными узлами блокчейн сети по JSON RPC протоколу. Truffle console в комплекте содержит web3.js библиотеку, поэтому ее не нужно устанавливать отдельно (отдельная установка: npm install web3). В данном случае в консоле truffle уже есть объект web3, который настроен на использование development узла, настроенного в truffle-config.js, в нашем случае это Ganache. Поэтому, при вызове функции getAccounts() мы получаем те самые 10 аккаунтов, которые мы можем видеть в Ganache.



Таким образом, проверяя баланс первого аккаунта в списке, от которого по умолчанию разворачиваются смарт-контракты (аккаунт администратора), мы можем увидеть баланс в размере 10000000 LPT токенов.

Следующая вещь, которую мы добавим – это функция transfer, которая даст нам возможность отправлять токены с одного адреса на другой.

Имеет она следующую сигнатуру:

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

Если посмотреть на описание этой функции в стандарте [ERC-20](https://eips.ethereum.org/EIPS/eip-20#transfer), то видно, что она должна инициировать событие Transfer, должна выбрасывать исключение, если на аккаунте, того, кто вызывает функцию не достаточно токенов. Также указано, что значение \_value = 0, это нормальная ситуация.

pragma solidity ^0.5.0;

contract LightpointToken {

    string public name = "LightpointToken";

    string public symbol = "LPT";

    uint256 public totalSupply;

mapping(address => uint256) public balanceOf;

    constructor (uint256 \_initialSupply) public {

        balanceOf[msg.sender] = \_initialSupply;

        totalSupply = \_initialSupply;

    }

event Transfer(

        address indexed from,

        address indexed to,

        uint256 value

    );

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

        require(balanceOf[msg.sender] >= \_value);

        balanceOf[msg.sender] -= \_value;

        balanceOf[\_to] += \_value;

        emit Transfer(msg.sender, \_to, \_value);

        return true;

    }

}

В реализации нет ничего сложного, по сути смарт-контракт выполняет математические действия по изменению баланса двух аккаунтов. Функция require зачастую нужна для валидации входных данных, проверки каких-либо условий перед выполнением бизнес логики, проверки результата выполнения другой функции. Если условие не выполняется, выбрасывается исключение. Ключевое слово emit инициирует событие. Таким образом, мы реализовали все требования для функции transfer, которые от нас требовал стандарт ERC-20. Атрибут indexed позволяет индексировать события, чтобы потом позволить искать события и использовать индексированные параметры как фильтры.

Добавим тесты на текущий функционал:

const LightpointToken = artifacts.require('LightpointToken');

contract('LightpointToken', function(accounts){

    var tokenInstance;

    var admin = accounts[0];

    var other = accounts[1];

    before(async() => {

        tokenInstance = await LightpointToken.deployed();

    })

    it('initializes the contract with the correct values', async function(){

        var name = await tokenInstance.name();

        assert.equal(name, "LightpointToken");

        var symbol = await tokenInstance.symbol();

        assert.equal(symbol, "LPT");

        var totalSupply = await tokenInstance.totalSupply();

        assert.equal(totalSupply.toNumber(), 1000000);

    })

    it('transfers token ownership', function() {

        return tokenInstance.transfer(other, 9999999999)

        .then(assert.fail).catch(async function(error){

            assert(error.message.indexOf('revert') >= 0, 'error message must contain revert');

            var receipt = await tokenInstance.transfer(other, 250000, {from:admin});

            assert.equal(receipt.logs.length, 1, 'triggers one event');

            var log = receipt.logs[0];

            assert.equal(log.event, 'Transfer', 'should be the "Transfer" event');

            assert.equal(log.args.from, admin, 'logs the account the tokens are trasferred from');

            assert.equal(log.args.to, other, 'logs the account the tokens are trasferred to');

            assert.equal(log.args.value, 250000, 'logs the transfer amount');

            var otherBalance = await tokenInstance.balanceOf(other);

            assert.equal(otherBalance.toNumber(), 250000, 'adds the amount to the receiving account');

            var adminBalance = await tokenInstance.balanceOf(admin);

            assert.equal(adminBalance.toNumber(), 750000, 'deducts the amount from the sending account');

        });

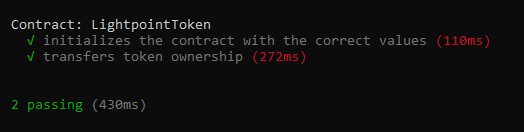
    })

});

Мы добавили проверки на наименование и символ токена. Затем при проверке правильности работы функции transfer мы последовательно проверили все условия:

* вначале пытаемся переслать количество токенов, превышающее текущий баланс, ожидаем ошибку, в которой ожидаем наличие слова ‘revert’
* передаем 250000 токенов с аккаунта администратора, на другой аккаунт, при этом ожидаем, что в свойстве logs объекта receipt будет видна информации об инициировании одного события Transfer
* после этого проверяем, что токены добавились на другой аккаунт и списались с аккаунта администратора.

Убедимся в том, что тесты выполняются:



Теперь, реализуем функцию [approve](https://eips.ethereum.org/EIPS/eip-20#approve) и [allowance](https://eips.ethereum.org/EIPS/eip-20#allowance). Исходя из документации стандарта, функция Approve позволяет аккаунту \_spender тратить токены с аккаунта, с которого мы вызываем данную функцию.

pragma solidity ^0.5.0;

contract LightpointToken {

    string public name = "LightpointToken";

    string public symbol = "LPT";

    uint256 public totalSupply;

    mapping(address => uint256) public balanceOf;

    mapping(address => mapping(address => uint256)) public allowance;

    constructor (uint256 \_initialSupply) public {

        balanceOf[msg.sender] = \_initialSupply;

        totalSupply = \_initialSupply;

    }

    event Transfer(

        address indexed from,

        address indexed to,

        uint256 value

    );

    event Approval(

        address indexed owner,

        address indexed spender,

        uint256 value

    );

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

        require(balanceOf[msg.sender] >= \_value);

        balanceOf[msg.sender] -= \_value;

        balanceOf[\_to] += \_value;

        emit Transfer(msg.sender, \_to, \_value);

        return true;

    }

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

        allowance[msg.sender][\_spender] = \_value;

        emit Approval(msg.sender, \_spender, \_value);

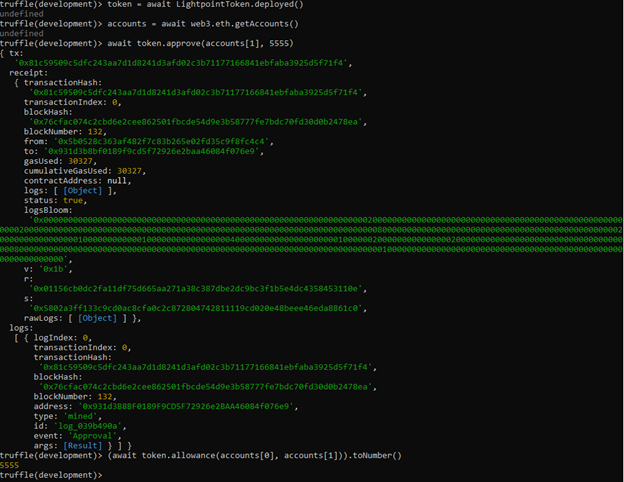
        return true;

    }

}

allowance – маппинг, обращаясь к нему через [msg.sender][\_spender], мы получаем количество токенов \_value, которое разрешено тратить аккаунту \_spender с аккаунта, с которого мы вызвали функцию approve.

Проверим как это работает:



Наконец, реализуем функцию transferFrom, которая нужна для делегированной передачи токенов.

pragma solidity ^0.5.0;

contract LightpointToken {

    string public name = "LightpointToken";

    string public symbol = "LPT";

    uint256 public totalSupply;

    mapping(address => uint256) public balanceOf;

    mapping(address => mapping(address => uint256)) public allowance;

    constructor (uint256 \_initialSupply) public {

        balanceOf[msg.sender] = \_initialSupply;

        totalSupply = \_initialSupply;

    }

    event Transfer(

        address indexed from,

        address indexed to,

        uint256 value

    );

    event Approval(

        address indexed owner,

        address indexed spender,

        uint256 value

    );

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

        require(balanceOf[msg.sender] >= \_value);

        balanceOf[msg.sender] -= \_value;

        balanceOf[\_to] += \_value;

        emit Transfer(msg.sender, \_to, \_value);

        return true;

    }

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

        allowance[msg.sender][\_spender] = \_value;

        emit Approval(msg.sender, \_spender, \_value);

        return true;

    }

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

        require(\_value <= balanceOf[\_from]);

        require(\_value <= allowance[\_from][msg.sender]);

        balanceOf[\_from] -= \_value;

        balanceOf[\_to] += \_value;

        allowance[\_from][msg.sender] -= \_value;

        emit Transfer(\_from, \_to, \_value);

        return true;

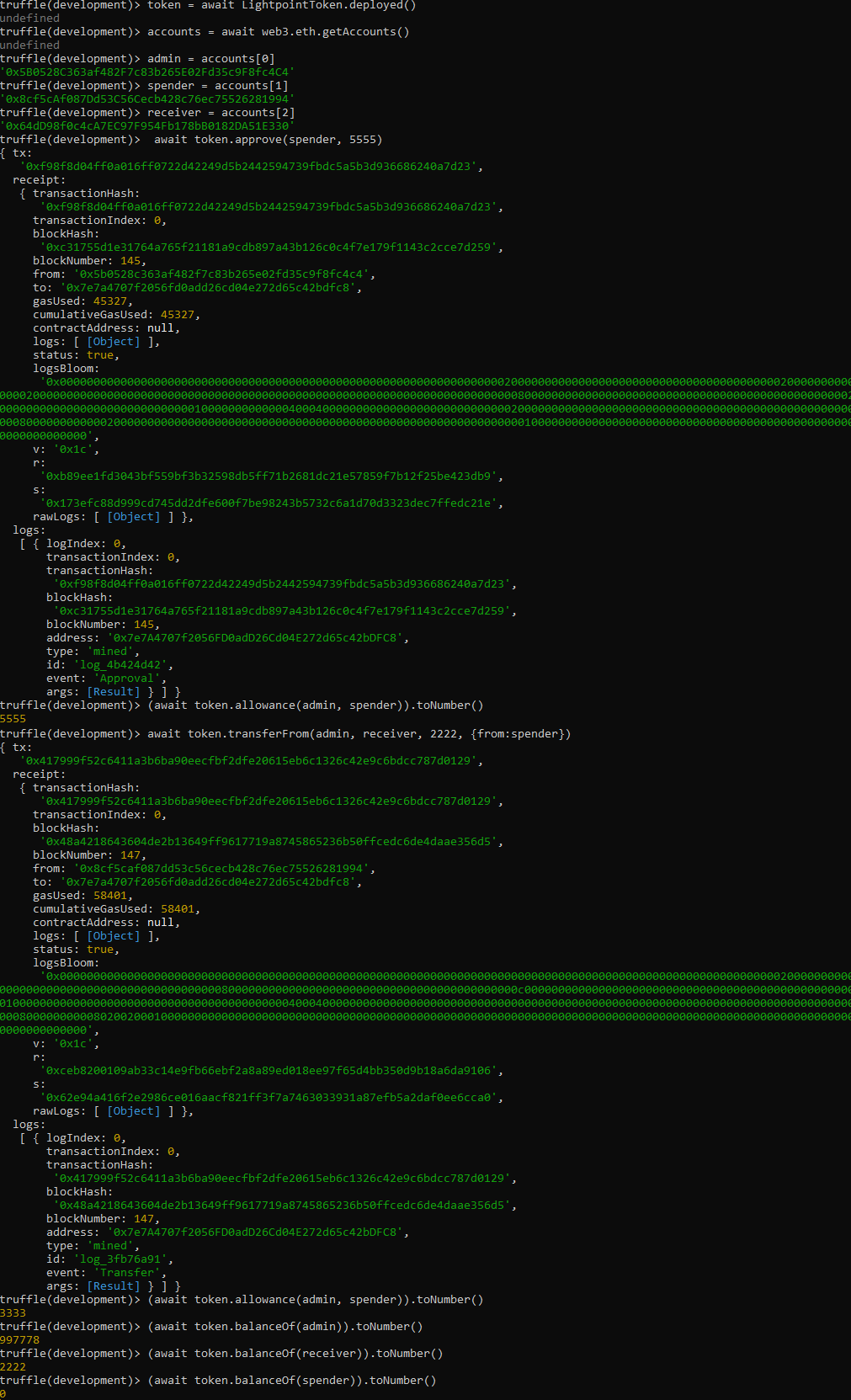
    }

}

Реализация следующая:

* Проверяем, есть ли на счете \_from требуемое количество токенов, если нет – выбрасываем исключение
* Проверяем, что количество токенов не превышает значение, которое \_from может тратить с аккаунта, с которого мы вызываем функцию (это значение содержится, в allowance[\_from][msg.sender]. Из этого следует, что для того, чтобы разрешить выполнять функцию transferFrom с какого-нибудь другого адреса, сначала должна быть вызвана функция Approve с нашего адреса, в которой мы можем разрешить тратить некоторое количество токенов какому-нибудь адресу.
* Далее изменяем балансы аккаунтов, а также уменьшаем значение allowance на переданное количество токенов
* Инициируем событие Transfer и возвращаем значение true

Проверим как это работает:



Мы разрешаем аккаунту spender тратить с аккаунта admin тратить 5555 токенов, вызываем функцию transferFrom с передачей метаданных {from:spender}, при этом msg.sender станет равен spender. После вызова функции изменились балансы admin и receiver, при этом уменьшилось значение allowance[admin][spender].

Добавим остальные тесты для нашего смарт-контракта:

const LightpointToken = artifacts.require('LightpointToken');

contract('LightpointToken', function(accounts){

    var tokenInstance;

    var admin = accounts[0];

    var other = accounts[1];

    before(async() => {

        tokenInstance = await LightpointToken.deployed();

    })

    it('initializes the contract with the correct values', async function(){

        var name = await tokenInstance.name();

        assert.equal(name, "LightpointToken");

        var symbol = await tokenInstance.symbol();

        assert.equal(symbol, "LPT");

        var totalSupply = await tokenInstance.totalSupply();

        assert.equal(totalSupply.toNumber(), 1000000);

    })

    it('transfers token ownership', function() {

        return tokenInstance.transfer(other, 9999999999)

        .then(assert.fail).catch(async function(error){

            assert(error.message.indexOf('revert') >= 0, 'error message must contain revert');

            var receipt = await tokenInstance.transfer(other, 250000, {from:admin});

            assert.equal(receipt.logs.length, 1, 'triggers one event');

            var log = receipt.logs[0];

            assert.equal(log.event, 'Transfer', 'should be the "Transfer" event');

            assert.equal(log.args.from, admin, 'logs the account the tokens are trasferred from');

            assert.equal(log.args.to, other, 'logs the account the tokens are trasferred to');

            assert.equal(log.args.value, 250000, 'logs the transfer amount');

            var otherBalance = await tokenInstance.balanceOf(other);

            assert.equal(otherBalance.toNumber(), 250000, 'adds the amount to the receiving account');

            var adminBalance = await tokenInstance.balanceOf(admin);

            assert.equal(adminBalance.toNumber(), 750000, 'deducts the amount from the sending account');

        });

    })

    it('approves tokens for delegated transfer', async function(){

        var receipt = await tokenInstance.approve(other, 100);

        assert.equal(receipt.logs.length, 1, 'triggers one event');

        var log = receipt.logs[0];

        assert.equal(log.event, 'Approval', 'should be the "Approval" event');

        assert.equal(log.args.owner, admin, 'logs the account the tokens are trasferred from');

        assert.equal(log.args.spender, other, 'logs the account the tokens are trasferred to');

        assert.equal(log.args.value, 100, 'logs the transfer amount');

        var allowance = await tokenInstance.allowance(admin, other);

        assert.equal(allowance, 100, 'stores the allowance for delegated transfer');

    })

    it('handles delegated token transfers', async function(){

        var fromAccount = accounts[2];

        var toAccount = accounts[3];

        var spendingAccount = accounts[4];

        return tokenInstance.transfer(fromAccount, 100, {from: admin})

        .then(function(receipt){

            return tokenInstance.approve(spendingAccount, 10, {from:fromAccount});

        })

        .then(function(receipt){

            return tokenInstance.transferFrom(fromAccount, toAccount, 9999, {from:spendingAccount});

        })

        .catch(function(error){

            assert(error.message.indexOf('revert') >= 0, 'cannot transfer value larger than balance');

            return tokenInstance.transferFrom(fromAccount, toAccount, 20, {from:spendingAccount})

        })

        .then(assert.fail).catch(async function(error){

            assert(error.message.indexOf('revert') >= 0, 'cannot transfer value larger than approved amount');

            var receipt = await tokenInstance.transferFrom(fromAccount, toAccount, 10, {from:spendingAccount})

            assert.equal(receipt.receipt.status, true);

            var log = receipt.logs[0];

            assert.equal(log.event, 'Transfer', 'should be the "Transfer" event');

            assert.equal(log.args.from, fromAccount, 'logs the account the tokens are trasferred from');

            assert.equal(log.args.to, toAccount, 'logs the account the tokens are trasferred to');

            assert.equal(log.args.value, 10, 'logs the transfer amount');

            var fromAccountBalance = await tokenInstance.balanceOf(fromAccount);

            assert.equal(fromAccountBalance.toNumber(), 90, 'deducts the amount from the sending account');

            var toAccountBalance = await tokenInstance.balanceOf(toAccount);

            assert.equal(toAccountBalance.toNumber(), 10, 'adds the amount from the receiving account');

            var allowance = await tokenInstance.allowance(fromAccount, spendingAccount);

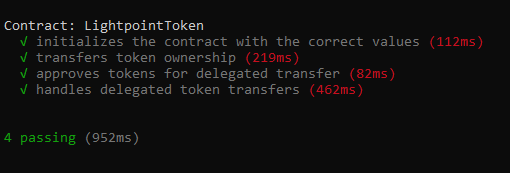
            assert.equal(allowance, 0, 'deducts the amount from the allowance');

        })

    })

});

Убедимся, что все тесты проходят:



В этом примере мы рассмотрели реализацию всех функций стандарта ERC-20 для смарт-контракта. Это мы сделали для понимания внутренней работы смарт-контракта. На самом деле, мы вполне можем использовать уже готовые контракты в качестве базовых, т.к. Solidity поддерживает наследование. Так, например, пакет @openzeppelin/contracts содержит базовые реализации некоторых стандартов ERC. Что бы использовать исходные файлы этого пакета, добавим файл package.json в корень нашего проекта с содержимым:

{

  "dependencies": {

    "@openzeppelin/contracts": "2.3.0"

  }

}

Затем выполним команду:

$ npm install

Теперь мы можем переписать наш смарт-контракт следующим образом:

pragma solidity ^0.5.0;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

import "@openzeppelin/contracts/token/ERC20/ERC20Detailed.sol";

contract LightpointToken is ERC20, ERC20Detailed {

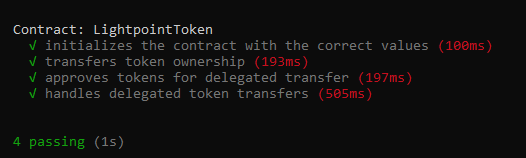
    constructor(uint256 initialSupply) ERC20Detailed("LightpointToken", "LPT", 0) public {

        \_mint(msg.sender, initialSupply);

    }

}

Проверим чтобы все тесты выполнялись:



Смарт-контракт LightpointTokenSale

Далее, реализуем смарт-контракт, который будет управлять процессом ICO. Добавим в папку с контрактами новый файл LightpointTokenSale.sol.

Мы познакомились частично с синтаксисом Solidity, поэтому для этого смарт-контракта я просто покажу реализацию, а позже пройдусь по всем методам и расскажу про них.

pragma solidity ^0.5.0;

import './LightpointToken.sol';

contract LightpointTokenSale {

    address payable public admin;

    LightpointToken public tokenContract;

    uint256 public tokenPrice;

    uint256 public tokensSold;

    bool public onSale;

    event Sell(address buyer, uint256 amount);

    event SaleChanged(bool onSale);

    modifier onlyAdmin() {

        require(msg.sender == admin);

        \_;

    }

    constructor(LightpointToken \_tokenContract, uint256 \_tokenPrice) public {

       admin = msg.sender;

       tokenContract = \_tokenContract;

       tokenPrice = \_tokenPrice;

    }

    function multiply(uint x, uint y) internal pure returns (uint z) {

        require(y == 0 || (z = x \* y) / y == x);

    }

    function buyTokens(uint256 \_numberOfTokens) public payable{

        require(onSale);

        require(msg.value == multiply(\_numberOfTokens, tokenPrice));

        require(tokenContract.balanceOf(address(this)) >= \_numberOfTokens);

        require(tokenContract.transfer(msg.sender, \_numberOfTokens));

        admin.transfer(msg.value);

        tokensSold += \_numberOfTokens;

        emit Sell(msg.sender, \_numberOfTokens);

    }

    function startSale() public onlyAdmin {

        require(!onSale);

        onSale = true;

        emit SaleChanged(onSale);

    }

    function endSale() public payable {

        pauseSale();

        require(tokenContract.transfer(admin, tokenContract.balanceOf(address(this))));

        admin.transfer(address(this).balance);

        tokensSold = 0;

    }

    function pauseSale() public onlyAdmin {

        require(onSale);

        onSale = false;

        emit SaleChanged(onSale);

    }

    function kill() public onlyAdmin {

        require(tokenContract.transfer(admin, tokenContract.balanceOf(address(this))));

        selfdestruct(admin);

    }

}

Конструктор принимает объект LightpointToken и цену одного токена в Eth. Функции startSale и pauseSale управляют значением onSale, которое определяет можно ли в данный момент покупать токены, также эти функции инициируют событие SaleChanged при изменении этого значения. Функция kill все токены, которые находятся на адресе смарт-контракта на адрес администратора, а также вызывает функцию selfdestruct, которая делает тоже самое, но уже с Eth, а также «убивает» смарт-контракт, удаляя хранимые значения и код из блокчейна. Эти 3 функции помечены кастомным модификатором onlyAdmin, который делает код более читабельным, вместо того чтобы в 3 местах делать проверку require(msg.sender == admin).

Также имеется функция buyTokens, которая и занимается продажей токенов.

* Проверяет, что onSale находится в значении true
* Проверяет, что передано значение msg.value равное количеству токенов, умноженных на цену одного токена
* Проверяет, что баланс токенов на адресе контракта не меньше количества покупаемых токенов
* Переводит токены покупателю
* Переводит Eth администратору
* Увеличивает значение проданных токенов tokenSold
* Инициирует Sell событие

К этой функции также применен модификатор payable. Им помечаются функции, в которых происходит обращение к msg.value с одного аккаунта на другой. Адреса, на которые происходит передача токенов, также помечаются модификатором payable, в данном случае admin.

В файле миграций изменим код по разворачиванию контрактов:

const LightpointToken = artifacts.require("LightpointToken");

const LightpointTokenSale = artifacts.require("LightpointTokenSale");

module.exports = function(deployer) {

  deployer.deploy(LightpointToken, 1000000).then(function(){

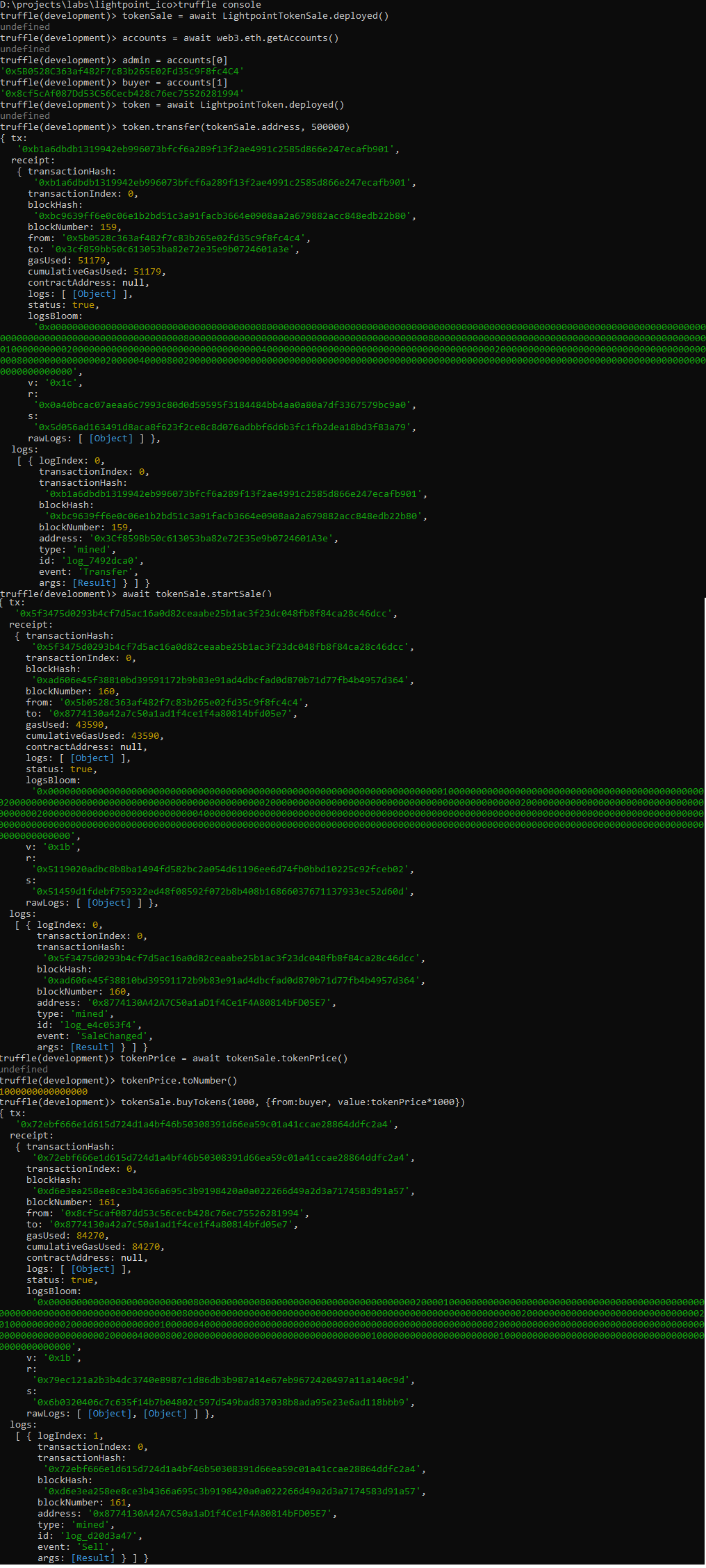
  return deployer.deploy(LightpointTokenSale, LightpointToken.address, 1000000000000000);

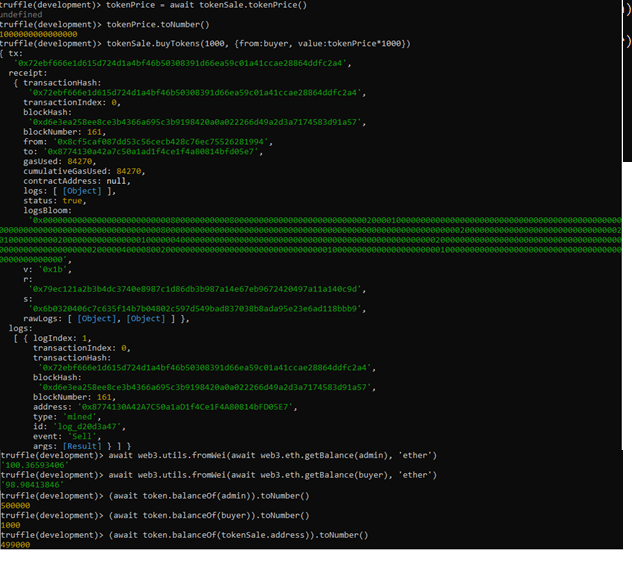
  })

};

Цену токенов мы здесь указываем в наименьших единицах wei (1 Eth = 1e18 wei). В итоге получается, что токен будет стоить 0.001 Eth

Проверим, чтобы все работало:





Перед началом продажи, на адрес смарт-контракта ICO передаем с адреса администратора 500,000 токенов. Начинаем продажи функцией startSale. Покупаем токены функцией buyTokens от имени покупателя. Балансы администратора, покупателя и смарт-контракта ICO изменяются.



Завершаем продажи функцией endSale, токены с адреса смарт-контракта ICO, переносятся на адрес администратора.

Добавим в папку test файл LightpointTokenSale.test.js, в котором будем тестировать поведение ICO контракта:

const LightpointToken = artifacts.require('LightpointToken');

const LightpointTokenSale = artifacts.require('LightpointTokenSale');

contract('LightpointTokenSale', function(accounts){

    var tokenInstance;

    var tokenSaleInstance;

    var \_tokenPrice = 1000000000000000;

    var tokensAvailable = 750000;

    var admin = accounts[0];

    var buyer = accounts[1];

    before(async() => {

        tokenInstance = await LightpointToken.deployed();

        tokenSaleInstance = await LightpointTokenSale.deployed();

    })

    it('initializes the contract with the correct values', async function(){

        assert.notEqual(tokenSaleInstance.address, 0x0, 'has contract address');

        var tokenContract = await tokenSaleInstance.tokenContract();

        assert.notEqual(tokenContract, 0x0, 'has token contract address');

        var tokenPrice = await tokenSaleInstance.tokenPrice();

        assert.equal(tokenPrice, \_tokenPrice, 'token price is correct');

    })

    it('facilitates token buying', async function (){

        // Provision 75% of all tokents to the token sale

        await tokenInstance.transfer(tokenSaleInstance.address, tokensAvailable, {from:admin})

        var numberOfTokens = 10;

        return tokenSaleInstance.buyTokens(numberOfTokens, { from:buyer, value: numberOfTokens\*\_tokenPrice })

        .then(assert.fail)

        .catch(async function(error){

            assert(error.message.indexOf('revert') >= 0, 'should be sale started');

            await tokenSaleInstance.startSale();

            var receipt = await tokenSaleInstance.buyTokens(numberOfTokens, { from:buyer, value: numberOfTokens\*\_tokenPrice })

            assert.equal(receipt.logs.length, 1, 'triggers one event');

            var log = receipt.logs[0];

            assert.equal(log.event, 'Sell', 'should be the "Sell" event');

            assert.equal(log.args.buyer, buyer, 'logs the account that purchased the tokens');

            assert.equal(log.args.amount, numberOfTokens, 'logs the number of tokens purchased');

            var tokenSold = await tokenSaleInstance.tokensSold();

            assert.equal(tokenSold.toNumber(), numberOfTokens, 'increments the number of tokens sold');

            var buyerBalance = await tokenInstance.balanceOf(buyer);

            assert.equal(buyerBalance.toNumber(), numberOfTokens);

            var saleBalance = await tokenInstance.balanceOf(tokenSaleInstance.address);

            assert.equal(saleBalance.toNumber(), tokensAvailable - numberOfTokens);

            return  tokenSaleInstance.buyTokens(numberOfTokens, { from:buyer, value: 1 })

            .then(assert.fail).catch(async function(error){

                assert(error.message.indexOf('revert') >= 0, 'msg.value must equal number of tokens in wei');

                return  tokenSaleInstance.buyTokens(800000, { from:buyer, value: 800000 \* \_tokenPrice })

                .then(assert.fail).catch(function(error){

                    assert(error.message.indexOf("sender doesn't have enough funds") >= 0, 'cannot purchase more tokens than available');

                });

            });

        })

    });

    it('ends token sale', function(){

       return tokenSaleInstance.endSale({from:buyer})

        .then(assert.fail).catch(async function(error){

            assert(error.message.indexOf('revert') >= 0, 'must be admin to end sale');

            await tokenSaleInstance.endSale({from: admin});

            var adminBalance = await tokenInstance.balanceOf(admin);

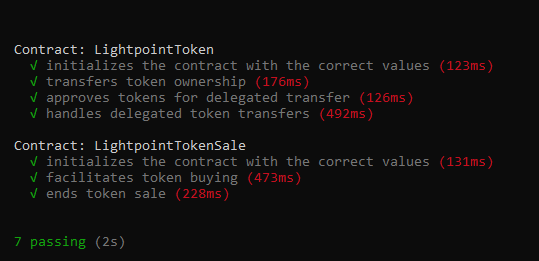
            assert.equal(adminBalance.toNumber(), 999990);

        });

    })

});

Проверим правильность выполнения тестов:



Смарт-контракт LightpointMarketplace

Реализуем контракт, с помощью которого мы сможем добавлять продукты в блокчейн, назначать им цену в LPT, покупать продукты.

pragma solidity ^0.5.0;

import './LightpointToken.sol';

contract LightpointMarketplace {

    string public name;

    uint public productCount = 0;

    mapping(uint => Product) public products;

    LightpointToken public tokenContract;

    struct Product {

        uint id;

        string name;

        uint price;

        address payable owner;

        bool purchased;

    }

    event ProductCreated(

        uint id,

        string name,

        uint price,

        address payable owner,

        bool purchased

    );

    event ProductPurchased(

        uint id,

        string name,

        uint price,

        address payable owner,

        bool purchased

    );

    constructor(LightpointToken \_tokenContract) public {

        name = "Lightpoint Marketplace";

        tokenContract = \_tokenContract;

    }

    function createProduct(string memory \_name, uint256 \_price) public {

        require(bytes(\_name).length > 0);

        require(\_price > 0);

        productCount++;

        products[productCount] = Product(productCount, \_name, \_price, msg.sender, false);

        emit ProductCreated(productCount, \_name, \_price, msg.sender, false);

    }

    function purchaseProduct(uint \_id) public payable {

        Product memory \_product = products[\_id];

        address payable \_seller = \_product.owner;

        require(\_product.id > 0 && \_product.id <= productCount);

        require(msg.value >= \_product.price);

        require(!\_product.purchased);

        require(\_seller != msg.sender);

        require(tokenContract.allowance(msg.sender, address(this)) >= msg.value);

        \_product.owner = msg.sender;

        \_product.purchased = true;

        products[\_id] = \_product;

        require(tokenContract.transferFrom(msg.sender, \_seller, msg.value));

        emit ProductPurchased(productCount, \_product.name, \_product.price, msg.sender, true);

    }

}

Здесь мы можем видеть новую конструкцию struct. Solidity позволяет создавать новые типы в форме структур. Которые можно будет сохранять в блокчейне. В данном случае структура продукта содержит идентификатор, наименование, цену, владельца и флаг, куплен ли продукт.

Функция createProduct занимается добавлением нового продукта в блокчейн:

* Она принимает в качестве параметров наименование продукта и его цену
* Проверяет чтобы длина наименования и цена были ненулевые
* Увеличивает количество продуктов
* Добавляет продукт в список ключей-значений
* Инициирует событие добавления нового продукта

Модификатором memory должны помечаться структуры и массивы, которые являются параметрами функции.

Функция покупки продукта purchaseProduct. Помечена модификатором payable, т.к. внутри имеется обращение к msg.value.

* Получает копию продукта из маппинга по ключу \_id
* Проверяет, что продукт существует, и его ключ не превышает общее количество продуктов
* Проверяет, что переданное значение msg.value не меньше цену продукта
* Проверяет, что продукт еще не куплен никем
* Проверяет, что покупатель не является владельцем
* Проверяет, что отправитель делегировал требуемое количество токенов на проведение операции передачи токенов
* Отправляет токены владельцу продукта
* Изменяет владельца продукта и флаг покупки
* Инициирует событие ProductPurchased

Здесь также имеется модификатор memory, но уже в теле функции. В данном случае он выделяет память на временную локальную переменную \_product, и туда купирует объект продукта. После изменения всех состояний по ключу \_id записывает в маппинг измененное состояние продукта.

Добавим тестовый файл LightpointMarketplace.test.js с содержимым:

const LightpointMarketplace = artifacts.require('LightpointMarketplace');

const LightpointToken = artifacts.require('LightpointToken');

require('chai')

.use(require('chai-as-promised'))

.should();

contract('LightpointMarketplace', ([deployer, seller, buyer]) => {

    let marketplace;

    let tokenInstance;

    before(async()=>{

        tokenInstance = await LightpointToken.deployed();

        marketplace = await LightpointMarketplace.deployed();

    })

    describe('deployment', async() => {

        it('deploys successfully', async()=>{

            const address = await marketplace.address;

            assert.notEqual(address, 0x0);

            assert.notEqual(address, '');

            assert.notEqual(address, null);

            assert.notEqual(address, undefined);

        })

        it('has a name', async ()=> {

            const name = await marketplace.name();

            assert.equal(name, 'Lightpoint Marketplace');

        })

    })

    describe('products', async() => {

        let result, productCount;

        before(async()=>{

            result = await marketplace.createProduct('iPhone X', 1000, {from:seller});

            productCount = await marketplace.productCount();

        })

        it('creates products', async ()=> {

            // SUCCESS

            assert.equal(productCount, 1);

            const event = result.logs[0].args;

            assert.equal(event.id.toNumber(), productCount.toNumber(), 'id is correct');

            assert.equal(event.name, 'iPhone X', 'name is correct');

            assert.equal(event.price, '1000', 'price is correct');

            assert.equal(event.owner, seller, 'owner is correct');

            assert.equal(event.purchased, false, 'purchased is correct');

            //FAILURE: Product must have a name

            marketplace.createProduct('', 1000, {from:seller}).should.be.rejected;

            //FAILURE: Product must have a price

            marketplace.createProduct('iPhone X', 0, {from:seller}).should.be.rejected;

        })

        it('lists products', async ()=> {

            // SUCCESS

            const product = await marketplace.products(productCount);

            assert.equal(product.id.toNumber(), productCount.toNumber(), 'id is correct');

            assert.equal(product.name, 'iPhone X', 'name is correct');

            assert.equal(product.price, '1000', 'price is correct');

            assert.equal(product.owner, seller, 'owner is correct');

            assert.equal(product.purchased, false, 'purchased is correct');

        })

        it('sells products', async ()=> {

            await tokenInstance.transfer(buyer, 10000, {from:deployer})

            // Track the seller balance before purchase

            let oldSellerBalance = await tokenInstance.balanceOf(seller);

            await tokenInstance.approve(marketplace.address, 1000, {from:buyer});

            const result = await marketplace.purchaseProduct(productCount,{from:buyer, value: 1000})

            //Check logs

            const event = result.logs[0].args;

            assert.equal(event.id.toNumber(), productCount.toNumber(), 'id is correct');

            assert.equal(event.name, 'iPhone X', 'name is correct');

            assert.equal(event.price, '1000', 'price is correct');

            assert.equal(event.owner, buyer, 'owner is correct');

            assert.equal(event.purchased, true, 'purchased is correct');

            // Check that seller received funds

            let newSellerBalance = await tokenInstance.balanceOf(seller);

            const expectedBalance = oldSellerBalance.toNumber() + 1000;

            assert.equal(newSellerBalance.toNumber(), expectedBalance);

            // FAILURE: Tries to but product that does not exists, i.e., product must have valid id

            marketplace.purchaseProduct(99, {from:buyer, value: 1000}).should.be.rejected;

            // FAILURE: Buyer tries to buy without enought LPT

            marketplace.purchaseProduct(productCount, {from:buyer, value: 500}).should.be.rejected;

            // FAILURE: Deployer tries to but the product, i.e., product can't be purchased twice

            marketplace.purchaseProduct(productCount, {from:deployer, value: 1000}).should.be.rejected;

            // FAILURE: Buyer tries to buy again, i.e., buyer can't be the seller

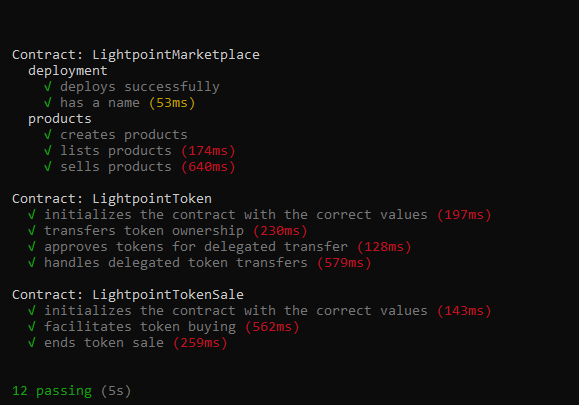
            marketplace.purchaseProduct(productCount, {from:buyer, value: 1000}).should.be.rejected;

        })

    })

})

Проверим, что бы все выполнялось:



Клиентская часть

Полный код проекта можно найти здесь <http://git.lp/vokasss/labs_ethereum_ico_and_marketplace>

В этом разделе я просто покажу, как происходит подключение и работа с блокчейн сетью, а также как вызывать методы наших смарт-контрактов.

Клиентская часть написана на react. В файле App.js при загрузке компонента вызывается метод componentWillMount(), который в свою очередь инициализирует объект web3.

  async loadWeb3(){

    // Modern dapp browsers...

    if(window.ethereum) {

      this.web3Provider = window.ethereum;

      window.web3 = new Web3(window.ethereum);

      await window.ethereum.enable();

    }

    // Legacy dapp browsers...

    else if (window.web3) {

      this.web3Provider = window.web3.currentProvider;

      window.web3 = new Web3(window.web3.currentProvider);

    } else {

      // Specify default instance if no web3 instance provided

      this.web3Provider = new Web3.providers.HttpProvider('http://localhost:7545');

      window.web3 = new Web3(this.web3Provider);

    }

    await this.initContracts();

  }

Нам понадобятся пакеты web3 и @truffle/contract

$ npm install web3

$ npm install @truffle/contract

[Infura](https://infura.io/) – предоставляет api для подключения к узлу и выполнения операций в какой-либо Ethereum сети.

[MetaMask](https://metamask.io/) – это расширение в браузере, которое по сути, является мостом между UI и Infura. С помощью него можно подтверждать транзакции, просматривать баланс своего аккаунта, создавать новые аккаунты и др. полезные функции. Если оно установлено, то в современных браузерах провайдер сети Ethereum будет доступен через window.ethereum, в старых браузерах через web3.currentProvider. Если же расширение не установлено, то с помощью web3 подключаемся к локальному узлу через HttpProvider.

Перед тем, как инициализировать объекты контрактов, нужно подключить файлы ABI.

import LightpointToken from '../abis/LightpointToken.json';

import LightpointTokenSale from '../abis/LightpointTokenSale.json';

import LightpointMarketplace from '../abis/LightpointMarketplace.json';

Функция initContracts, которая запускается после инициализации объекта web3 выглядит следующим образом:

async initContracts() {

    var \_this = this;

    var account = (await window.web3.eth.getAccounts())[0];

    \_this.setState({ account });

    this.contracts = [];

    this.contracts.LightpointToken = TruffleContract(LightpointToken);

    this.contracts.LightpointToken.setProvider(this.web3Provider);

    this.contracts.LightpointToken.deployed().then(function(lightpointToken) {

      \_this.LightpointTokenInstance = lightpointToken;

      console.log("Lightpoint Token Address:", lightpointToken.address);

      \_this.contracts.LightpointTokenSale = TruffleContract(LightpointTokenSale);

      \_this.contracts.LightpointTokenSale.setProvider(\_this.web3Provider);

      \_this.contracts.LightpointTokenSale.deployed().then(function(lightpointTokenSale) {

        \_this.LightpointTokenSaleInstance = lightpointTokenSale;

        console.log("Lightpoint Token Sale Address:", lightpointTokenSale.address);

        \_this.contracts.LightpointMarkeplace = TruffleContract(LightpointMarketplace);

        \_this.contracts.LightpointMarkeplace.setProvider(\_this.web3Provider);

        \_this.contracts.LightpointMarkeplace.deployed().then(async function(lightpointMarkeplace) {

            \_this.LightpointMarketplaceInstance = lightpointMarkeplace;

            console.log("Lightpoint Markeplace Address:", lightpointMarkeplace.address);

            await \_this.listenForEvents();

            await \_this.loadTokenSaleData();

            await \_this.loadProductsData();

        });

      });

    });

  }

После этого можно вызывать методы контрактов. Типичный вызов метода выглядит уже знакомо, как мы это делали в truffle console:

const tokenPrice = await this.LightpointTokenSaleInstance.tokenPrice();