

Experimento Blockchain

Fazendo o deploy de Smart Contracts em uma Rede Ethereum local.

Alunos Responsáveis:

Marcos de Souza Boger	9837570
Naiane Ayone Yanachi	9345189
Henrique Corrêa Vasconcellos	9837611

Objetivo

Este relatório tem como objetivo explicar de forma sucinta o passo a passo para criar uma rede Ethereum local, desenvolver um smart contract que será lançado nesta rede e subir um servidor capaz de interagir com o blockchain e as funções definidas no smart contract.. Será descrito como instalar todas as ferramentas necessárias para então fazer o *deploy* do *smart contract* e depois fazer algumas requisições para alterar o valor dos contratos, criando *tokens* na rede e fornecendo esses *tokens* para determinadas *accounts*. O relatório foi feito pensando na reprodutibilidade do experimento e não há pré-requisitos para sua realização. Conhecimento básico do funcionamento de NodeJS e do conceito de API facilita o desenvolvimento.

Introdução e Conceitos Teóricos

Blockchain é um banco de dados distribuído organizado por meio de uma lista de blocos ordenados. Tem como característica básica o fato de que uma vez posicionado um bloco na rede, esse bloco não pode ser mais alterado, garantindo assim a integridade de informações possibilitando ser um sistema distribuído sem uma figura centralizada na qual os próprios componentes da rede podem fazer a validação das transações.

Neste relatório foram utilizadas ferramentas para realizar o *deploy* de uma rede Ethereum local (baseada em blockchain), onde será feito o *deploy* de um *smart contract* nessa rede e poderão então ser realizadas transações e alguns testes utilizando esse contrato. O Ethereum tem três pilares básicos que são: contas, *smart contracts* e clientes. Contas são entidades que possuem um balanço em Ether (a criptomoeda da rede) e podem fazer transações de acordo com a permissão que possuem. Já os smart contracts são vistos como um tipo de conta, possuindo um balanço e podendo executar transações mas não possuem um usuário por trás, eles estão na rede como programas de fato. Por fim, clientes são programas rodando nos nós da rede e que fazem a checagem das diversas transações e mantêm a integridade da rede como um todo. Explicados os conceitos básicos por trás do Blockchain e Ethereum, pode-se passar ao experimento.

Roteiro do experimento

- 1- Se certificar que possui [Node](#) e NPM instalados na máquina (Windows ou Linux)
- 2- Criar repositório para salvar o projeto (Ex: `experimento-blockchain`)
- 3- Criar uma pasta dentro deste diretório chamada `contratos`
- 4- Instalar globalmente o truffle através do comando

```
npm install -g truffle
```

Caso esteja em Linux e dê problema de permissão, acrescentar `sudo` na frente do comando ou, se preferir, mudar o diretório onde o NPM instala os pacotes globais (mais detalhes em <http://npm.github.io/installation-setup-docs/installing/a-note-on-permissions.html>)

O Truffle é um *framework* que facilita o desenvolvimento de smart contracts. Nesse caso ele será usado para fazer a compilação e o *deploy* do *smart contract* na rede ethereum.

- 4.1- Verificar se o truffle está corretamente instalado e no PATH do sistema com o comando

```
truffle v
```

```
Truffle v5.4.0 (core: 5.4.0)
Solidity v0.5.16 (solc-js)
Node v16.3.0
Web3.js v1.4.0
```

- 5- instalar globalmente o ganache-cli através do comando

```
npm install -g ganache-cli
```

O ganache é um programa que será utilizado para subir uma rede Ethereum localmente. O ganache-cli seria a versão para terminal e permite criar uma rede Ethereum com um único comando.

- 6- Dentro do diretório `contratos`, abrir terminal e rodar o comando

```
npm init -y
```

O conteúdo do arquivo `package.json` deve ser parecido com esse:

```
{
  "name": "contratos",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "@openzeppelin/contracts": "^4.2.0"
  }
}
```

7- Neste mesmo terminal, rodar o comando

```
npm install @openzeppelin/contracts
```

O OpenZeppelin é uma biblioteca de código aberto, que possui *smart contracts* já definidos e testados pela comunidade. Eles são escritos na linguagem solidity, que permite a definição de smart contracts através de mecanismos de herança, possibilitando a definição de um smart contract próprio sem ter que desenvolver do zero.

8- No mesmo terminal, rodar o comando

```
truffle init
```

9- Na pasta contracts recém-criada, criar um arquivo `MyToken.sol` (não esquecer a extensão .sol)

10- Escrever o seguinte neste arquivo:

```
// contracts/MyToken.sol
// SPDX-License-Identifier: MIT
pragma solidity >=0.4.22 <0.9.0;
```

```
import
"@openzeppelin/contracts/token/ERC20/presets/ERC20PresetMinterPauser.sol";

contract MyToken is ERC20PresetMinterPauser {
    constructor(string memory _name, string memory _symbol, uint8 decimals)
        ERC20PresetMinterPauser(_name, _symbol) {
    }
}
```

O arquivo define o contrato que será feito o *deploy* na rede e que ele irá importar um tipo de contrato já definido na biblioteca OpenZeppelin com funções já implementadas, nesse caso o **ERC20PresetMinterPauser**

11- Na pasta migrations recém-criada, criar um arquivo `2_deploy_token.js`

12- Escrever o seguinte neste arquivo:

```
const MyToken = artifacts.require("./MyToken");

module.exports = function (deployer) {
    const name = "My Token";
    const symbol = "MT";
    const decimals = 10;
    deployer.deploy(MyToken, name, symbol, decimals);
};
```

O arquivo define que será realizado o *deploy* do *smart contract* definido no arquivo `MyToken.sol` anteriormente quando for utilizado o comando `truffle migrate`.

13- No arquivo `truffle-config.js`, descomentar as linhas relativas a `networks` {
`development` }

```
development: {
  host: "127.0.0.1",      // Localhost (default: none)
  port: 8545,            // Standard Ethereum port (default: none)
  network_id: "*",       // Any network (default: none)
},
```

14- No mesmo arquivo, descomentar as linhas de `compilers { solc }`, deixar apenas as linhas de `docker`, `settings` e `evmVersion` comentadas

15- Alterar a version do compiler `{ solc.version }` para 0.8.0

```
compilers: {
  solc: {
    version: "0.8.0",
    // docker: true,
    // settings: {
    //   optimizer: {
    //     enabled: false,
    //     runs: 200
    //   },
    //   evmVersion: "byzantium"
    // }
  }
},
```

16- Abrir um OUTRO terminal e rodar o comando

```
ganache-cli
```

O comando utiliza o ganache para subir uma rede local Ethereum já com algumas contas definidas para serem utilizadas e suas respectivas *private keys*. Será possível então realizar algumas transações entre essas contas de forma a mexer com a rede. Vale ressaltar que as contas fornecidas no display do ganache-cli são desbloqueadas, ou seja, é possível fazer transferências a partir destas contas sem realizar a assinatura destas transações. Caso se realize o deploy em uma rede remota, as contas não serão desbloqueadas e será necessário fazer a assinatura da transação ([mais detalhes](#)).

A maioria das criptomoedas utilizam criptografia assimétrica, que faz uso de chaves públicas e privadas. As chaves públicas são publicamente conhecidas e essenciais para identificação. As

chaves privadas são secretas e usadas para autenticação e encriptação. O endereço das contas é uma versão menor, mas representativa, da sua chave pública.

A chave privada garante a propriedade dos fundos em uma conta. Quando se faz uma transação, o software assina com sua chave privada (sem divulgá-la), o que indica que a rede toda sabe que você tem autoridade para transferir os fundos da conta referente à transação.

```
Ganache CLI v6.12.2 (ganache-core: 2.13.2)
```

Available Accounts

```
=====
```

```
(0) 0x324E5877D211a04b116EAe4C0755c6506ed2AA00 (100 ETH)
(1) 0x3a2b495d6Dc890A8D98F49575d699500bfa9236D (100 ETH)
(2) 0x4E838334cB4596ac2CdCFb1D50Ffa0564fC02135 (100 ETH)
(3) 0xC6291466711DEb5068C89103EA51623075a1f9a7 (100 ETH)
(4) 0x0E3cC434387ca65aaCb1683a1A61d2E2dF642083 (100 ETH)
(5) 0x4F983d4654F5d5d9B69461C0dbE5e19208962D2C (100 ETH)
(6) 0xe538b7DF9696A49062c51B62eA7Bf6a349378966 (100 ETH)
(7) 0x42052170974E180d89A83fdc4E44721Ad28E9C53 (100 ETH)
(8) 0x930c3e8bd551f1D3167ce8D57d368393C4d7280a (100 ETH)
(9) 0x6580dc90693011020f2e1FB4e61cd72B3cC630A9 (100 ETH)
```

Private Keys

```
=====
```

```
(0) 0xc55bb39e011c7955b0e99421c121993b55e34f16f7b04e316210aec6426e4aa0
(1) 0x0a26be95c6291e7a2a1a577c604d556cd4fdcc9cb4bf80dd2f44e585d03ff395
(2) 0x8551560c622f4d39373157c00073775214c99fbc1f0f803d3250bda1e7a7c880
(3) 0xbe894272a2f8bdd3611fbee1a5e5a8c5f9d906ed7fae771369704296020f8a15
(4) 0x8ae88cd65b923bfc226a8cc7e1575553986649f16176c14c670949229375088
(5) 0xff074e4b50fb282a6c21d5dddec756fe618d2e97ccb464e346eda2258d7669b28
(6) 0x9f8d48a6af6f444893c2e8e2fd5e45a74932fb8e6f844601c1bc3e230f428a0c
(7) 0xf4eb37134ec31714936ab5702bc5b41ded8f2a74a7600951574c371e529410d8
(8) 0x5b7514ca9fa58c8074c19fe408ff9c9e253779bedea8f9347d2c37fb9fd6b67a
(9) 0xd4356cfdae043377f56ba85bad24eee23aed20480455b6d56f99f0d1beb177dc
```

```
HD Wallet
=====
Mnemonic:      embody page common sell coach define desert random milk theme flag addict
Base HD Path:  m/44'/60'/0'/0/{account_index}

Gas Price
=====
20000000000

Gas Limit
=====
6721975

Call Gas Limit
=====
9007199254740991

Listening on 127.0.0.1:8545
█
```

Deixar esse terminal aberto até o fim da experiência

17- No terminal antigo, aberto no diretório contratos, rodar os seguintes comandos

17.1- Compilação

```
truffle compile
```



```

Compiling your contracts...
=====
✓ Fetching solc version list from solc-bin. Attempt #1
✓ Downloading compiler. Attempt #1.
> Compiling ./contracts/Migrations.sol
> Compiling ./contracts/MyToken.sol
> Compiling @openzeppelin/contracts/access/AccessControl.sol
> Compiling @openzeppelin/contracts/access/AccessControlEnumerable.sol
> Compiling @openzeppelin/contracts/security/Pausable.sol
> Compiling @openzeppelin/contracts/token/ERC20/ERC20.sol
> Compiling @openzeppelin/contracts/token/ERC20/IERC20.sol
> Compiling @openzeppelin/contracts/token/ERC20/extensions/ERC20Burnable.sol
> Compiling @openzeppelin/contracts/token/ERC20/extensions/ERC20Pausable.sol
> Compiling @openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol
> Compiling @openzeppelin/contracts/token/ERC20/presets/ERC20PresetMinterPauser.sol
> Compiling @openzeppelin/contracts/utils/Context.sol
> Compiling @openzeppelin/contracts/utils/Strings.sol
> Compiling @openzeppelin/contracts/utils/introspection/ERC165.sol
> Compiling @openzeppelin/contracts/utils/introspection/IERC165.sol
> Compiling @openzeppelin/contracts/utils/structs/EnumerableSet.sol
> Artifacts written to /mnt/d/POLI 4o MA/SegInfo/contratos/build/contracts
> Compiled successfully using:
  - solc: 0.8.0+commit.c7dfd78e.Emscripten.clang

```

Se houver problema com a versão do `solc`, rodar o comando a seguir para atualizar a versão:

```
npm install solc
```

```

@openzeppelin/contracts/token/ERC20/presets/ERC20PresetMinterPauser.sol:3:1: ParserError: Source
  file requires different compiler version (current compiler is 0.5.16+commit.9c3226ce.Emscripten
  .clang - note that nightly builds are considered to be strictly less than the released version
  pragma solidity ^0.8.0;
  ^-----^

Error: Truffle is currently using solc 0.5.16, but one or more of your contracts specify "pragma
  solidity ^0.8.0".
Please update your truffle config or pragma statement(s).
(See https://trufflesuite.com/docs/truffle/reference/configuration#compiler-configuration for in
  formation on
  configuring Truffle to use a specific solc compiler version.)

Compilation failed. See above.

```

O comando faz a compilação dos *smart contracts* definidos e indica se não há nenhum erro na definição deles.

17.2- Migração

```
truffle migrate
```

Summary

=====

```
> Total deployments: 2
> Final cost: 0.06879886 ETH
```

O comando faz o *deploy* dos *smart contracts* na rede Ethereum que está rodando pelo comando `ganache-cli` e passa alguns valores importantes como a *Account* associada com os *contracts* e o *contract address*.

Ao final do comando de `migrate`, o resumo mostra que foram feitas duas operações de *deploy*. Como na rede Ethereum cada transação tem um custo para ser executado, é mostrado esse custo final em Ethereum para as duas transações.

18- Anotar o `contract address` e `account`

2_deploy_token.js

=====

Deploying 'MyToken'

```
> transaction hash: 0xbdd48f0ce400a58c48ca48304b069522679f3c13edee5b6a0bd291ac0c74f1b7
> Blocks: 0 Seconds: 0
> contract address: 0x9a24120B1265E54732DC626e5C2dC034305Eb7D3
> block number: 3
> block timestamp: 1626045189
> account: 0x556139aaf426CEE323adc882553F0288027Eb575
> balance: 99.93035088
> gas used: 3195643 (0x30c2fb)
> gas price: 20 gwei
> value sent: 0 ETH
> total cost: 0.06391286 ETH
```

```
> Saving migration to chain.
```

```
> Saving artifacts
```

```
> Total cost: 0.06391286 ETH
```

Aqui tem-se algumas informações importantes relativas à transação de *deploy* do *Smart Contract*. O hash da transação identifica essa transação na rede e garante a integridade da transação, enquanto o *contract address* é o endereço do *smart contract* e deve ser utilizado para fazer operações usando esse contrato. O *block number* indica qual o número do bloco na blockchain, nesse caso é 3 porque antes dele vem o bloco padrão inicial e o bloco Migrations

que o truffle sobe por padrão também. O valor de *account* identifica qual foi a conta onde o smart contract está localizado, enquanto o *balance* indica o balanço em Ether desta conta. Por fim, o *gas price* indica qual foi o valor oferecido por unidade de gás na transação e *gas used* indica a quantidade de gás de fato utilizado para então calcular o custo total da transação. Para entender mais sobre o conceito de gás na rede Ethereum, vale assistir esse [vídeo](#) explicativo.

19- Criar um outro diretório no mesmo repositório do item **2** com o nome `blockchain`

20- Rodar o seguinte comando:

```
npm init -y
```

21- Escrever o seguinte no arquivo `package.json` recém gerado:

```
{
  "name": "app",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "nodemon index.js"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.17.1",
    "web3": "^1.3.6"
  },
  "devDependencies": {
    "nodemon": "^2.0.7"
  }
}
```

22- Rodar o comando

```
npm install
```

23- Criar um arquivo `index.js`

24- Escrever o seguinte neste arquivo:

```
const express = require('express');
const app = express();
const port = 3001;

const contract = require('./constants/contract');
const address = contract.address;
const ABI = contract.ABI;
const account = contract.account;

const Web3 = require('web3');
const rpcURL = "http://localhost:8545";
const web3 = new Web3(rpcURL);

const Token = new web3.eth.Contract(ABI, address)

app.get('/contract/totalSupply', (req, res) => {
  Token.methods.totalSupply().call()
    .then(supply => res.json(supply))
    .catch(err => res.json(err));
});

app.get('/contract/balance', (req, res) => {
  Token.methods.balanceOf(account).call()
    .then(balance => res.json(balance))
    .catch(err => res.json(err));
});

app.post('/contract/mint', (req, res) => {
  Token.methods.mint(account, 1000).send({from:account})
    .then(response => res.json(response))
    .catch(err => res.json(err));
});

app.get('/contract/getPastEvents', (req, res) => {
```

```
Token.getPastEvents("allEvents")
  .then(response => res.json(response))
  .catch(err => res.json(err));
});

app.listen(port, () => console.log('Server ready'));
```

O arquivo define como interagir com os contratos, chamando as funções do *smart contract* por meio de requisições em *endpoints*. Isso facilita com que, uma vez rodando o `index.js` por meio do NPM como um servidor local, possam ser feitas diversas requisições como por exemplo criando tokens na rede (pelo comando `mint`) ou checando o balanço de contas (pelo comando `balanceOf`).

25- Criar uma pasta chamada `constants`

26- Criar um arquivo chamado `contract.js`

27- O conteúdo deste arquivo deve ser o mesmo do [link](#)

28- Mude o `address` deste mesmo arquivo para o `address` que você anotou anteriormente (linha 685).

```
const address = '0x9a24120B1265E54732DC626e5C2dC034305Eb7D3';
```

29- Mude o `account` deste mesmo arquivo para o `account` que você anotou anteriormente (linha 687).

```
const account = '0x556139aaf426CEE323adc882553F0288027Eb575';
```

30- Rode o seguinte comando:

```
npm run start
```

No final, seu repositório deve estar parecido com [este](#).

Requisições HTTP

Com algum programa que faz requisições HTTP, como o [Postman](#) ou o [cURL](#), vamos então utilizar as rotas que criamos para entender o blockchain.

Se você já possuir o Postman instalado, esse é o [link](#) com a *collection* para as rotas utilizadas. Instruções para importá-la nesse [link](#).

GET	<code>http://localhost:3001/contract/totalSupply</code>
-----	---

Requisição na qual pode-se ver o número total de *Tokens* na rede. Inicialmente vai ter valor zero, mas conforme forem feitas operações de **mint** (criação de *tokens*), esse valor vai refletir o total de todas as contas da rede.

GET	<code>http://localhost:3001/contract/balance</code>
-----	---

Requisição na qual pode-se ver o balanço de uma conta em específico. Inicialmente terá valor zero para todas as contas, mas conforme forem feitas operações de **mint** para determinadas contas, o novo saldo pode ser visualizado por meio deste comando.

POST	<code>http://localhost:3001/contract/mint</code>
------	--

Requisição na qual pode-se criar tokens e associar esses *tokens* para uma dada *account*. Para realizar a operação, o chamador deve possuir a permissão de *minter*, que no nosso caso seria a *account* na qual o *smart contract* do Truffle foi feito o *deploy*. Assim, pode-se usar esse comando para inserir *tokens* na rede, alterando os valores de *Balance* e de *Total Supply* das requisições anteriores.

GET	<code>http://localhost:3001/contract/getPastEvents</code>
-----	---

Requisição na qual pode-se ver todos os eventos dessa rede. Na resposta alguns dos campos que se obtém são `transactionHash`, `blockHash` e `signature`, que correspondem ao hash da transação, o hash do bloco e a assinatura da transação.

Se fosse possível forjar uma transação, a sua *hash* também mudaria. Mas lembre que o bloco também tem uma *hash* própria, que contém as informações de todas as transações do bloco. Logo, se qualquer informação do bloco for alterada, a sua *hash* também ficaria diferente da original, afetando toda a cadeia.

É aí que entra o diferencial da Blockchain: como cada bloco contém a *hash* do bloco anterior, adulterar uma simples transação provocaria um efeito dominó em toda a cadeia daquele ponto em diante. Mas como milhões de cópias da Blockchain original estão espalhadas pelo mundo, as versões contaminadas pelo ataque seriam comparadas com as outras. Nesse momento, ocorre uma espécie de votação entre os computadores (consenso distribuído). Como a rede original é maior, a versão adulterada é descartada, eliminando o fraudador.

Desafio

Agora que você já sabe o que cada rota faz e utilizando a [documentação](#) da API de contratos para referência, **faça operações de *mint* para outras accounts e transfira *tokens* entre elas**. No final de suas transações, confira o `totalSupply` e os eventos da rede.

Para isso, você precisará editar o arquivo `blockchain/index.js` (criando novas rotas ou editando as existentes) e utilizar os endereços de outras contas (disponível na saída do comando `ganache-cli`).

Se necessário, utilize também a [documentação](#) do pacote `express`.