

Лабораторная работа 2. Создание смарт-контракта и взаимодействие с ним

Цель

Познакомиться с языком программирования смарт-контрактов Solidity, познакомиться с методами взаимодействия со смарт-контрактами из dApp.

Шаги:

1. Создать смарт-контракт
2. Развернуть его на блокчейне
3. Создать пользовательский интерфейс для взаимодействия со смарт-контрактом

Замечание об окружении

Ниже подразумевается, что вы работаете в UNIX-подобной системе: Linux, FreeBSD, MacOS, и т.п. Если вы работаете с Microsoft Windows, вам потребуется дополнительно установить [Windows Subsystem for Linux](#) или использовать виртуальную машину с Linux через [Vagrant](#), [VirtualBox](#) и др.

Создание смарт-контракта

Для выполнения лабораторной работы предлагается использовать набор инструментов разработки Truffle. Для его выполнения требуется установить на компьютер последнюю LTS версию Node.js и NPM (поставляется вместе с Node.js) с официального сайта: <https://nodejs.org/ru/>

Когда Node.js установлена, следует установить truffle:

```
npm install -g truffle
```

Truffle создаёт проект в текущей папке. Создадим новую папку для контракта, и перейдём в неё:

```
mkdir poster-contract  
cd poster-contract
```

Воспользуемся уже [существующим шаблоном \(или Truffle Box\)](#) для создания контрактов на сети Polygon:

```
truffle unbox polygon
```

Команда создаст в текущей папке минимальный набор файлов для начала работы:

- `contracts` - содержит код смарт-контрактов
 - `ethereum` - код смарт-контрактов для Ethereum (нам не потребуется)
 - `polygon` - код смарт-контрактов для сети Polygon
- `migrations` - набор файлов для разворачивания и/или изменения разрабатываемых смарт-контрактов

- `test` - содержит тесты.

Чтобы проверить корректность работы, запустим тесты:

```
npm run test
```

Они упадут с ошибкой:

```
poster-contract -- zsh -- 80x52
[ukstv@slab poster-contract % npm run test

> polygon-box@1.0.0 test
> truffle test

> Something went wrong while attempting to connect to the network at http://127.0.0.1:8545. Check your network configuration.
ProviderError:
Could not connect to your Ethereum client with the following parameters:
  - host      > 127.0.0.1
  - port      > 8545
  - network_id > *
Please check that your Ethereum client:
  - is running
  - is accepting RPC connections (i.e., "--rpc" or "--http" option is used in geth)
  - is accessible over the network
  - is properly configured in your Truffle configuration file (truffle-config.js)

      at /Users/ukstv/Library/Application Support/fnm/node-versions/v16.14.0/installation/lib/node_modules/truffle/build/webpack:/packages/provider/wrapper.js:76:1
      at /Users/ukstv/Library/Application Support/fnm/node-versions/v16.14.0/installation/lib/node_modules/truffle/build/webpack:/packages/provider/wrapper.js:114:1
      at XMLHttpRequest.request.onreadystatechange (/Users/ukstv/Library/Application Support/fnm/node-versions/v16.14.0/installation/lib/node_modules/truffle/build/webpack:/node_modules/web3-providers-http/lib/index.js:98:1)
      at XMLHttpRequestEventTarget.dispatchEvent (/Users/ukstv/Library/Application Support/fnm/node-versions/v16.14.0/installation/lib/node_modules/truffle/build/webpack:/node_modules/xhr2-cookies/dist/xml-http-request-event-target.js:34:1)
      at XMLHttpRequest.exports.modules.996763.XMLHttpRequest._setReadyState (/Users/ukstv/Library/Application Support/fnm/node-versions/v16.14.0/installation/lib/node_modules/truffle/build/webpack:/node_modules/xhr2-cookies/dist/xml-http-request.js:208:1)
      at XMLHttpRequest.exports.modules.996763.XMLHttpRequest._onHttpRequestError (/Users/ukstv/Library/Application Support/fnm/node-versions/v16.14.0/installation/lib/node_modules/truffle/build/webpack:/node_modules/xhr2-cookies/dist/xml-http-request.js:349:1)
      at ClientRequest.<anonymous> (/Users/ukstv/Library/Application Support/fnm/node-versions/v16.14.0/installation/lib/node_modules/truffle/build/webpack:/node_modules/xhr2-cookies/dist/xml-http-request.js:252:48)
      at ClientRequest.emit (node:events:520:28)
      at Socket.socketErrorListener (node:_http_client:442:9)
      at Socket.emit (node:events:520:28)
      at emitErrorNT (node:internal/streams/destroy:157:8)
      at emitErrorCloseNT (node:internal/streams/destroy:122:3)
      at processTicksAndRejections (node:internal/process/task_queues:83:21)
Truffle v5.6.0 (core: 5.6.0)
Node v16.14.0
ukstv@slab poster-contract %
```

Ошибка говорит, что Truffle для тестирования попытался связаться с локальным узлом блокчайна, а тот оказался недоступен. В самом деле, мы не запускали локальный узел блокчайна. Для его запуска нам потребуется ещё один пакет [Ganache](#):

```
npm install -g ganache
```

Теперь в соседнем терминале, в той же папке `poster-contract` запустим локальный узел Ganache:

```
ganache
```

В консоль Ganache выведет диагностическую информацию:

```
[ukstv@slab poster-contract % ganache
ganache v7.4.4 (@ganache/cli: 0.5.4, @ganache/core: 0.5.4)
Starting RPC server

Available Accounts
=====
(0) 0x7cAEb8D64C3819b3ce0ED16Eb10702D679347f65 (1000 ETH)
(1) 0x68c4c28C6746A979927684931C759f6a53cdC0D4 (1000 ETH)
(2) 0x6BFc994Fd15939e60c61200946a0aC13698a3499 (1000 ETH)
(3) 0x99F816C87F8979317862433a13697e7Df15Ceb27 (1000 ETH)
(4) 0x6CdbbF278851349Cbf2bB6D684b40bf06A909b92 (1000 ETH)
(5) 0x268e0046503447bA5866745963a80e2316eDB249 (1000 ETH)
(6) 0xa05969bc36B21b274B48636a30D7E39A713e49CE (1000 ETH)
(7) 0xEf3B47D9B790b6fEd4B03854711d2f8b2a842bBf (1000 ETH)
(8) 0x8FbB9f9ABCBCAb3B66B9d1E9979f30f960991c8b (1000 ETH)
(9) 0x33a3C70C2a0eB816C547FC04179C82b433Ed13B6 (1000 ETH)

Private Keys
=====
(0) 0x9f5b80e1a950991ac8e618b62b8648cba8917e11d86a110ca23678827d80a38c
(1) 0xaf52d067fc70c42a160dc48263e9048ea9f3e2029c17112f6eee9c2e786d85da
(2) 0x4169161f42854ffffaea8ec85350a8a7dd2bec273c619e985c059dac2409e8a45
(3) 0x7b3bec0714852594951983f4a305b57c6e191aab075dc079a220d1777bbc4f2b
(4) 0xa91fa857d04950b12ab8aace34ab191c973e8c36d85d2ce0fb7a81b0fae30a06
(5) 0x761ae121765ae9aad927d053ac4f25a1f71463c05c4432d54d0a4cb0e3ad9afc
(6) 0x81b32c11bda0167e6c5fa373804d35cc395f8deb3de6e17e5cbcfc2f62670f0d
(7) 0x10f0ddf2eeeeae80243fe000289ae2ae06d5d0bfe4131226c269a5960560ea521
(8) 0x6b025e5d0ba69acdd9512d6ccc895018eb673f09d7aebae9ffa7f3ba1c213f48
(9) 0x26706a34b386fe7624a9796d7ff240ec32a54e74d7ab8ef309ebdc52876f5995

HD Wallet
=====
Mnemonic: joy census abstract code control beyond shock lion pond ripple rose fluid
Base HD Path: m/44'/60'/0'/0/{account_index}

Default Gas Price
=====
2000000000

BlockGas Limit
=====
30000000

Call Gas Limit
=====
50000000

Chain Id
=====
1337

RPC Listening on 127.0.0.1:8545
```

Теперь мы можем вернуться в первый терминал. Запустим тесты ещё раз: `npm run test`. В терминале должен появиться вывод с сообщением об успешных тестах:

```
lukstv@slab poster-contract % npm run test

> polygon-box@1.0.0 test
> truffle test

Compiling your contracts...
=====
> Compiling ./contracts/ethereum/SimpleStorage.sol
> Artifacts written to /var/folders/3x/0gdz5cvn1_l6nls9vn00qrvc0000gn/T/test--58
662-WlCUASPgAdrH
> Compiled successfully using:
  - solc: 0.5.16+commit.9c3226ce.Emscripten clang

Contract: SimpleStorage
  ✓ ...should store the value 89.

1 passing (30ms)

lukstv@slab poster-contract %
```

Теперь время создавать свой смарт-контракт. К несчастью, используемый нами шаблон содержит некоторое количества мусора. Давайте его выкинем. Для этого, перенесите имеющийся контракт `SimpleStorage` в папку `contracts` и удалите ненужные папки внутри `contracts`:

```
mv ./contracts/polygon/SimpleStorage.sol ./contracts
rm -rf ./contracts/ethereum ./contracts/polygon
```

Удалите `truffle-config.js` и переименуйте `truffle-config.polygon.js` на `truffle-config.js`:

```
rm ./truffle-config.js
mv ./truffle-config.polygon.js ./truffle-config.js
```

Внутри конфигурационного файла `./truffle-config.js` вы можете увидеть, что он использует ссылки на Polygon-специфичные папки. Давайте заменим их на значения по умолчанию, удалив поля:

- `contracts_build_directory`: значение по умолчанию `./build/contracts`,
- `contracts_directory`: значение по умолчанию `./contracts`.

Кроме того, имеет смысл переименовать названия сетей:

- `polygon_infura_testnet` → `polygon_mumbai`,
- `polygon_infura_mainnet` → `polygon_matic`.

Дополнительно следует убрать ссылки на Polygon-специфичный конфигурационный файл из `package.json`. Уберите все строки вида `--config=truffle-config.polygon.js` и переименуйте элементы в поле `scripts` так, чтобы они не ссылались на Polygon. Должно получиться такое содержание поля `scripts`:

```
{  
  "test": "truffle test --network=$npm_config_network",  
  "compile": "truffle compile",  
  "migrate": "truffle migrate --network=$npm_config_network"  
}
```

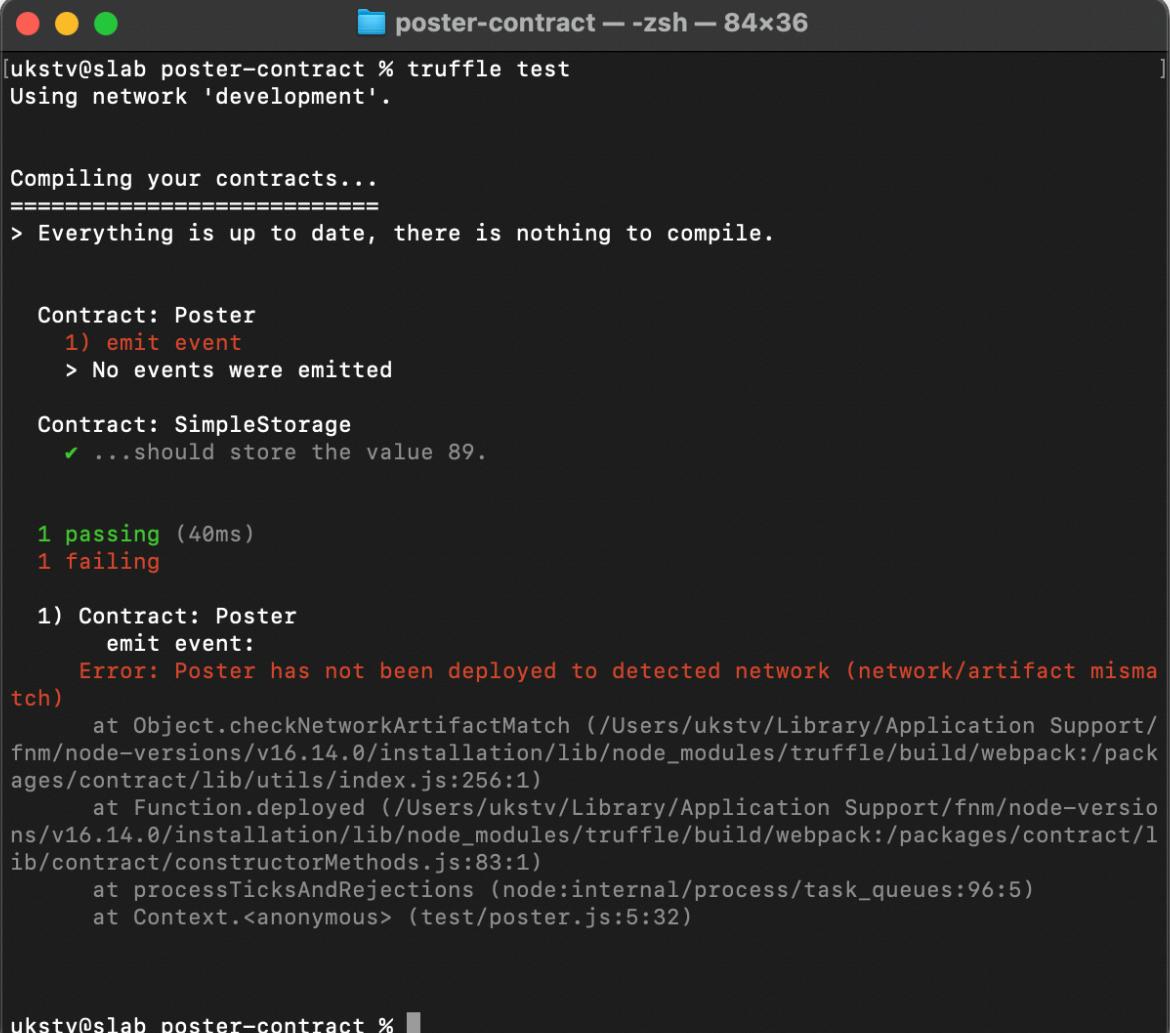
Теперь время создать свой контракт гостевой книги Poster. Создайте файл `Poster.sol` в папке `contracts`. Он должен иметь вид:

```
// SPDX-License-Identifier: MIT  
pragma solidity >=0.4.21 <0.9.0;  
  
contract Poster {  
    event NewPost(address indexed user, string content, string indexed tag);  
  
    function post(string memory content, string memory tag) public {  
        emit NewPost(msg.sender, content, tag);  
    }  
}
```

Единственная функция, которая здесь выполняется, это `post`. Её задача - принять пользовательские параметры и положить как событие на блокчейн. Создадим тест для этой функции в файле `test/poster.js` по аналогии с существующим тестом `test/simplestorage.js`:

```
const Poster = artifacts.require("./Poster.sol");  
  
contract("Poster", accounts => {  
  it("emit event", async () => {  
    const posterInstance = await Poster.deployed();  
    const eventsBefore = await posterInstance.getPastEvents('NewPost')  
    assert.deepEqual(eventsBefore, [])  
    const content = "Hello, world!"  
    const tag = "hello"  
    await posterInstance.post(content, tag, {from: accounts[0]})  
    const eventsAfter = await posterInstance.getPastEvents('NewPost')  
    assert.equal(eventsAfter.length, 1)  
    const postedEvent = eventsAfter[0]  
    assert.equal(postedEvent.args.user, accounts[0])  
    assert.equal(postedEvent.args.content, content)  
    assert.equal(postedEvent.args.tag, web3.utils.keccak256(tag))  
  });  
});
```

Теперь запустим тест через `truffle test`. Как вы можете обнаружить, тест для нашего контракта не запускается: Poster has not been deployed to detected network.



```
[ukstv@slab poster-contract % truffle test
Using network 'development'.

Compiling your contracts...
=====
> Everything is up to date, there is nothing to compile.

Contract: Poster
1) emit event
  > No events were emitted

Contract: SimpleStorage
✓ ...should store the value 89.

1 passing (40ms)
1 failing

1) Contract: Poster
   emit event:
     Error: Poster has not been deployed to detected network (network/artifact mismatch)
      at Object.checkNetworkArtifactMatch (/Users/ukstv/Library/Application Support/fnm/node-versions/v16.14.0/installation/lib/node_modules/truffle/build/webpack:/packages/contract/lib/utils/index.js:256:1)
      at Function.deployed (/Users/ukstv/Library/Application Support/fnm/node-versions/v16.14.0/installation/lib/node_modules/truffle/build/webpack:/packages/contract/lib/contract/constructorMethods.js:83:1)
      at processTicksAndRejections (node:internal/process/task_queues:96:5)
      at Context.<anonymous> (test/poster.js:5:32)

ukstv@slab poster-contract %
```

Truffle перед запуском тестов разворачивает все контракты на локальную тестовую сеть (Ganache). Порядок разворачивания определяется файлами миграции в `migrations`. На текущий момент эта папка содержит единственную миграцию `1_deploy_simple_storage.js`. Чтобы Truffle знал о контракте Poster из `artifacts.require('./Poster.sol')`, нужно создать ещё одну миграцию.

Создадим файл `migrations/2_poster.js` со следующим содержанием:

```
const Poster = artifacts.require("./Poster.sol");

module.exports = function (deployer) {
  deployer.deploy(Poster);
};
```

Теперь запустим тесты через `truffle test`. Команда выдаст положительный результат тестирования:

```
[ukstv@slab poster-contract % truffle test
Using network 'development'.

Compiling your contracts...
=====
> Everything is up to date, there is nothing to compile.

Contract: Poster
✓ emit event

Contract: SimpleStorage
✓ ...should store the value 89.

2 passing (51ms)

ukstv@slab poster-contract %
```

Теперь, когда мы имеем протестированный смарт-контракт, имеет смысл его развернуть на блокчейн.

2. Развёртывание контракта на блокчейн

Для развёртывания контракта на блокчейне нужны следующие элементы:

- байткод контракта,
- доступ к сети, реализуемый через Ethereum JSON RPC URL,
- Эфириум (или Matic, или другой нативный токен), чтобы заплатить за транзакцию создания контракта на блокчейне,

Байт-код контракта Truffle создаёт по команде `truffle compile`. После выполнения этой команды вы можете заметить появление новых файлов в папке `build/contracts`. Каждый из файлов там содержит помимо прочего байткод контракта и его ABI.

Для доступа к сети нам потребуется Ethereum JSON RPC URL. Мы можем использовать тот же URL, который мы использовали для подключения Метамаршала к Polygon Mumbai. Например, мы можем взять его с [ChainList](#). Мы используем этот URL в конфигурационном файле `truffle-config.js`. Там найдите секцию `polygon_mumbai`. Внутри секции задайте параметр `providerOrUrl` у конструктора `HDWalletProvider` значением вашего JSON RPC URL. Секция `polygon_mumbai` будет выглядеть примерно так:

```

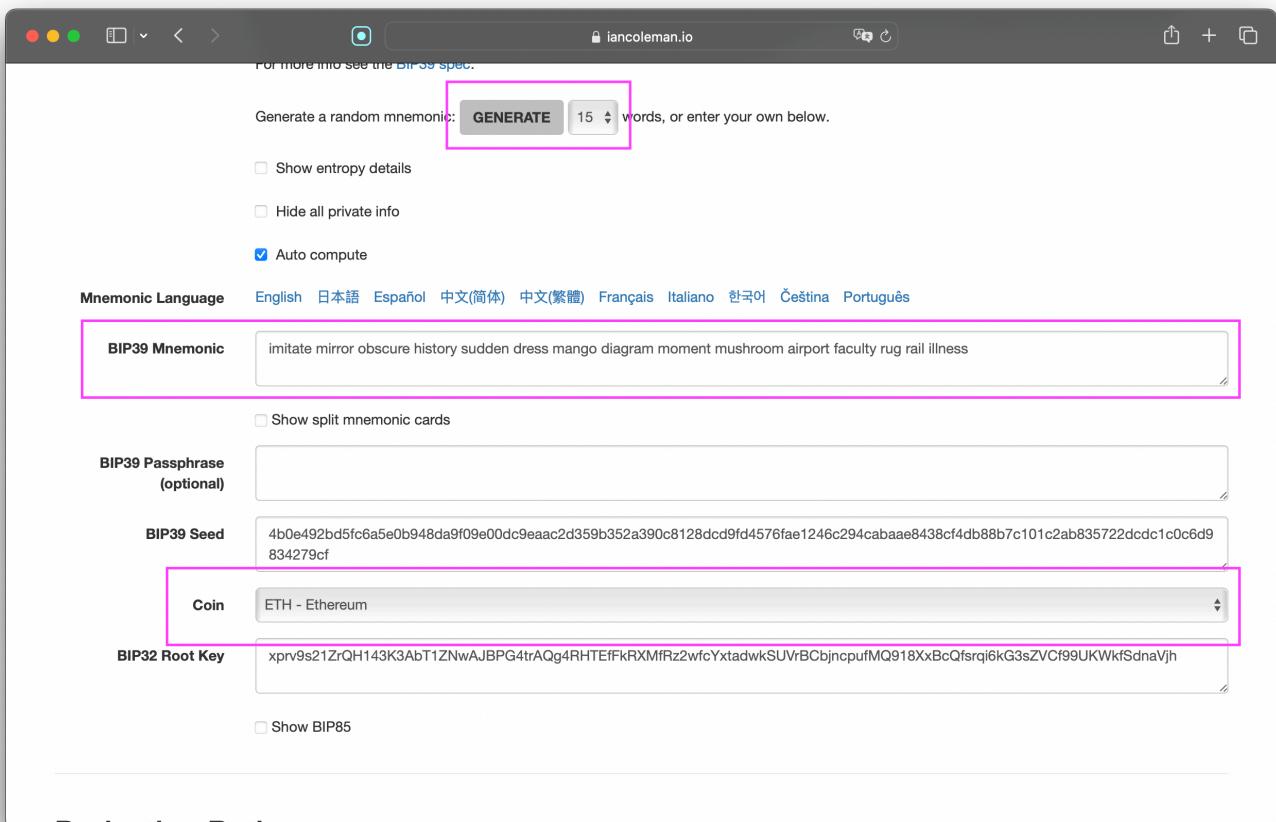
polygon_mumbai: {
  provider: () => new HDWalletProvider({
    mnemonic: {
      phrase: mnemonic
    },
    providerOrUrl: "https://matic-mumbai.chainstacklabs.com"
  }),
  network_id: 80001,
  confirmations: 2,
  timeoutBlocks: 200,
  skipDryRun: true,
  chainId: 80001
}

```

Здесь вы можете обратить внимание на переменную `mnemonic`. Мнемоника - другое название секретной фразы или сид-фразы.

Для оплаты транзакции создания контракта на блокчейне следует передать мнемонику как переменную окружения при запуске `truffle deploy`. Вы можете использовать ту же секретную фразу, которую вы использовали для МетаМаска из лабораторной 1. Вы также можете создать новую мнемонику и перевести туда достаточное количество нативных токенов с вашего счёта в МетаМаске.

Так, для данной работы можно воспользоваться инструментом Mnemonic Code Converter <https://iancoleman.io/bip39/> для создания мнемоники.



Пролистав страницу этого инструмента вниз вы можете увидеть адреса, выведенные из этой мнемоники. Truffle будет использовать первый из выведенных адресов. Именно сюда вы можете перевести часть нативных токенов с вашего Метамаска.

The screenshot shows a web application interface for generating derived addresses. At the top, there are two checkboxes: 'Encrypt private keys using BIP38 and this password:' and 'Use hardened addresses'. Below these are two tabs: 'Table' (selected) and 'CSV'. A table lists 11 derived addresses, each with a path, address, public key, and private key. The first address's row is highlighted with a pink border. The columns are labeled: Path, Address, Public Key, and Private Key.

Path	Address	Public Key	Private Key
m/44'/60'/0'/0/0	0x1f181B9cE6f98525f0bD152425E6B004269c7671	0x0268781aa32b7b4db53f675bcf1510031850471ce69e637bac2d68ed89e4c6d4c5	0x13bb13
m/44'/60'/0'/0/1	0xA9FA19eFE981b3873671A8f4d5dAe91aC25F135B	0x03a77c36c187fc80e75504939c0ec51dd2b967b7f2e6f577f93607715043c6fd11	0x3711f9
m/44'/60'/0'/0/2	0x237d704D2b54dFc3eb59CA3e16Dd1c889eD51FF0	0x0213c45189ce32c7217ad54694ab8d67ad0cd66fe78dc60b99b487cccb7e27a5f7	0xc8c1d3
m/44'/60'/0'/0/3	0x4ad77b729D43A2426238eBC258B8fd673Fd9E858	0x02ffa5bbdbb31489af7a3987fb6e97af37a51919679f6bdf8584b4cd4128d34a88	0xb9ba3
m/44'/60'/0'/0/4	0xCbDC3902C021DF73791b307522641d0D62bd2d97	0x02bb6c84c01eeaf8533c8687e2f1024325a6d9fd1ad9abef650bc6c5ced33cb100	0x7799cc
m/44'/60'/0'/0/5	0xE7dcc35659b94A672805683cF00EBA7711770f8a	0x0321bb3189628d2b631bfe3f968ff6c8b57178fdbd75b1710c468c7921ff7da7a	0x364b3b
m/44'/60'/0'/0/6	0x80EA18f4B5c920EB8B259eb7CE1d0ecD19b3f062	0x033c88948139cd70e72bdf1a67467c218785a3d584a49f5cedbe1b7e960d24e9d	0x01fba0
m/44'/60'/0'/0/7	0xE547F6132AB1369A6C3754460C9856a0B2Bcd5a1	0x0389ed5044beabad70aa29f26bbdef1fcfc3a41bded18ad2dbc74fa7bfade6f59	0xf54f24
m/44'/60'/0'/0/8	0xc86f7068868467BB4738e5593731ca9be9cdf55a	0x02c6a0b795540fdc2394b34364179ed556d2be1110b8896f413a9b60d3de7c2de6	0x95b399
m/44'/60'/0'/0/9	0x62D0753E828B181756876a8f7D2D692e813c0A03	0x03f5b08ff601ffbad73aac600590fc8d1c004d7840b939728a44821c768601f9e	0xf6e51a
m/44'/60'/0'/0/10	0x9a62D23398a508D2044DA3062E57172aC0B245aB	0x02e3f062b1574406e2a05382c61d1003ece2fbfbfffe5310b878f586ebfba034c6	0xbbf043

Следует убедиться, что следующие утверждения справедливы:

- вы можете скомпилировать код контракта в байт-код без ошибок через `truffle compile`,
- вы передаёте в `truffle-config.js` корректный Ethereum JSON RPC URL,
- вы имеете в своём распоряжении корректную секретную фразу,
- соответствующий адрес имеет достаточное TODO количество нативных токенов.

После этого можно запускать команду развертывания контрактов:

```
MNEMONIC="here goes your mnemonic phrase" truffle deploy --network=polygon_mumbai
```

В результате вы должны иметь в консоли что-то подобное:

The terminal window title is 'poster-contract — zsh — 84x75'. The command entered is 'MNEMONIC="century [REDACTED] mail" truffle migrate --network=polygon_mumbai'. The output shows the compilation of contracts and a message indicating that everything is up to date.

```
poster-contract — zsh — 84x75
ukstv@slab poster-contract % MNEMONIC="century [REDACTED] mail" truffle migrate --network=polygon_mumbai

Compiling your contracts...
=====
> Everything is up to date, there is nothing to compile.
```

```
Starting migrations...
=====
> Network name:      'polygon_mumbai'
> Network id:       80001
> Block gas limit: 20958951 (0x13fce7)

1_deploy_simple_storage.js
=====

Deploying 'SimpleStorage'

> transaction hash: 0xf353[REDACTED]745a9536f42
2a809de3
> Blocks: 1           Seconds: 8
> contract address: 0x4db[REDACTED]9Ac72e
> block number:      2[REDACTED]0
> block timestamp:   16[REDACTED]3
> account:           0x64AB[REDACTED]8870
> balance:            0.196597797863905694
> gas used:          96189 (0x177bd)
> gas price:          35.369970954 gwei
> value sent:         0 ETH
> total cost:         0.003402202136094306 ETH

Pausing for 2 confirmations...

=====
> confirmation number: 1 (block: 2[REDACTED]1)
> confirmation number: 2 (block: 2[REDACTED]2)
> Saving artifacts
-----
> Total cost:        0.003402202136094306 ETH

2_deploy_poster.js
=====

Deploying 'Poster'

> transaction hash: 0xe296[REDACTED]cb06c5457
39e3e617
> Blocks: 2           Seconds: 8
> contract address: 0x738[REDACTED]45b
> block number:      2[REDACTED]5
> block timestamp:   1[REDACTED]3
> account:           0x64[REDACTED]370
> balance:            0.18927234588929391
> gas used:          208442 (0x32e3a)
> gas price:          35.143838452 gwei
> value sent:         0 ETH
> total cost:         0.007325451974611784 ETH

Pausing for 2 confirmations...

=====
> confirmation number: 1 (block: 2[REDACTED]6)
> confirmation number: 2 (block: 2[REDACTED]7)
> Saving artifacts
-----
> Total cost:        0.007325451974611784 ETH

Summary
=====
> Total deployments:  2
> Final cost:        0.01072765411070609 ETH
```

Вы можете открыть полученный адрес контракта на блокчейн-эксплорере и убедиться, что он в самом деле доступен в сети.

The screenshot shows the PolygonScan (Mumbai) interface for a specific Ethereum contract. The top navigation bar includes links for Home, Blockchain, Tokens, Misc, and Testnet. The main content area displays the Contract Overview, which shows a balance of 0 MATIC and the My Name Tag field as Not Available. The More Info section lists the Contract Creator as 0x64ab... at txn 0xe29... and 0xe29... . Below this, the Transactions tab is selected, showing one transaction from block 2155, timestamped 14 mins ago, originating from 0x64ab... and ending with a Contract Creation event. The transaction hash is 0xe29... . A CSV export option is available for download. The footer of the page includes a Powered by Polygon Chain logo and links for Add Mumbai Network, Preferences, and a dark mode switch.

Если вы перейдёте на вкладку "Contract", вы можете просмотреть байткод контракта.

The screenshot shows the PolygonScan interface for the Mumbai network. The main header includes the logo, the network name "Mumbai", and navigation links for Home, Blockchain, Tokens, Misc, and Testnet. The top right features a search bar and filter options. The central content area displays a "Contract Overview" card for a specific contract. This card includes a "More Info" section with details like "My Name Tag: Not Available" and "Contract Creator: 0x64ab... at txn 0xe29...". Below the card, tabs for Transactions, ERC-20 Token Txns, Contract (which is selected), and Events are visible. A note encourages users to verify their contract source code. At the bottom, three buttons are shown: "Decompile ByteCode" (with a dropdown arrow), "Switch to Opcodes View", and "Similar Contracts". A large, scrollable text area contains the raw bytecode of the contract, starting with 0x608060405234801561001057600080fd5b...

Правила хорошего тона предписывают раскрытие исходного кода контракта для блокчейн-эксплорера. Это называют "верификацией" смарт-контракта, не путать с формальной верификацией.

Для верификации вам потребуется API-ключ PolygonScan. Создайте аккаунт на <https://polygonscan.com/register> и затем создайте API ключ на <https://polygonscan.com/myapikey>

Теперь вернёмся в папку `poster-contract`. Добавим плагин `truffle-plugin-verify` как зависимость:

```
npm add truffle-plugin-verify -D
```

Добавим плагин в конфигурационный файл `./truffle-config.js`:

```
module.exports = {
  /* ... предыдущее содержимое */
  plugins: ['truffle-plugin-verify']
}
```

И добавим туда полученный API-ключ PolygonScan:

```
module.exports = {
  /* ... предыдущее содержимое */
  api_keys: {
    polygonscan: 'API_KEY'
  }
}
```

Теперь вызовем команду верификации. Она тоже требует секретную фразу в переменной MNEMONIC

```
MNEMONIC="here goes your mnemonic phrase" truffle run verify Poster --network polygon_mumbai
```

По завершении процесса откройте страницу контракта на экспортере. Заметьте, как изменилось содержимое вкладки "Contract". Теперь можно вызывать функции контракта!

Пользовательский интерфейс для взаимодействия со смарт-контрактом

Теперь наша задача - создать пользовательский интерфейс для работы с контрактом. Можно считать, что это простейший dApp.

Для построения dApp мы будем использовать классический Next.js и React. В родительской для `poster-contract` папке вызовите команду

```
npx create-next-app@latest
```

Это подготовит каркас приложения для пользовательского интерфейса. Команда спросит вас об имени приложения. Выберите `poster-ui` как имя приложения. По завершении работы команды перейдите в только что созданную папку `poster-ui` и установите зависимости:

```
npm install
```

Теперь можно запускать dev-сервер (`npm run dev`) и открывать основной файл, с которым будем работать: `pages/index.js`.

Он содержит много лишнего, приведем его до состояния продуктивной пустоты:

```
import Head from 'next/head'
import Image from 'next/image'

export default function Home() {
  return (
    <div>
    </div>
  )
}
```

Во-первых, нам необходимо отсюда вызвать функцию `post`. Как указывает лекция, для вызова функции контракта достаточно адреса и ABI контракта. Эти сведения можно узнать или из Truffle artifact в `poster-contract/build/contracts/Poster.json`, либо из блокчейн-эксплорера.

Мы можем использовать библиотеку web3.js для взаимодействия с контрактом. Её нужно установить как зависимость:

```
npm add web3
```

Одна из сложных часть взаимодействия с контрактами, как это ни удивительно, это аутентификация пользователя в приложении. Вы можете использовать код ниже как стартовую точку. Он запрашивает у пользователя его адрес и позволяет оперировать с ним из приложения.

```
import Head from 'next/head'
import Image from 'next/image'
import {useEffect, useState} from "react";
import Web3 from "web3";

export default function Home() {
  const [web3, setWeb3] = useState(undefined)
  const [userAddress, setUserAddress] = useState(undefined)

  const handleConnect = async () => {
    const web3 = new Web3(window.ethereum)
    const [address] = await window.ethereum.enable()
    setUserAddress(address)
    setWeb3(web3)
  }

  return (
    <div>
      <button onClick={handleConnect}>Connect</button>
      <address>{userAddress}</address>
    </div>
  )
}
```

Для того, чтобы считать, что вы выполнили лабораторную работу, от вас потребуется достроить этот dApp. Необходимо предоставить вашему пользователю:

1. Возможность постить текст в контракт через функцию контракта `post`.
2. Возможность смотреть текст, теги и адрес пользователя для всех, кто запостил сообщение в контракт.
3. Возможность фильтровать посты по тегам.

Формат предоставления отчёта

В результате работы у вас должно получиться совсем небольшое, но законченное приложение. Код приложения - контракты и пользовательский интерфейс - должен быть доступен в системе контроля версий: GitHub, GitLab, Radicle и т.д.

По завершении кодирования попросите своих одногруппников воспользоваться вашим приложением.

Отчёт должен содержать:

- ссылку на верифицированный контракт на блокчейн-эксплорере,
- ссылку на репозиторий,
- описание вашего процесса работы над приложением,

- описание приложения как краткое руководство пользователя.