Exercise 9

This is an extra exercise sheet that goes beyond the lecture. Programming JavaScript is not relevant for the exam but is relevant if you want to do real-world blockchain engineering.

1. In this exercise, we create a simple data analysis program using NodeJS, Web3, and Infura.

   (a) What is Web3.js?

   > **Solution:**
   > **Web3.js** is the official Ethereum JavaScript API that provides a wrapper for the JSON RPC interface of a full node to interact directly with the blockchain. It can be used to deploy smart contracts, read the blockchain, send transactions, and more.

   (b) What is Infura? Create an account at `https://infura.io` and create a new project in order to obtain an API key.

   > **Solution:**
   > **Infura** is a full node provider for Ethereum. After registering a free account and obtaining an API key, one can make use of the JSON RPC interface to get information from the blockchain.

   (c) Create a new NodeJS project and install Web3.

   > **Solution:**
   > Use `npm init` to create the new project. Install Web3 with `npm install web3`. The `package.js` file should look something like this:
   >
   > ```
   > {
   >   "name": "first_contract",
   >   "version": "1.0.0",
   >   "description": "",
   >   "main": "index.js",
   >   "scripts": {
   >     "test": "echo \"Error: no test specified\" && exit 1"
   >   },
   >   "author": "BBSE",
   >   "license": "MIT",
   >   "dependencies": {
   >     "web3": "^1.0.0-beta.55"
   >   }
   > }
   > ```

   (d) Write a JavaScript program to determine the number of the block that contains the very first contract-creation transaction, i.e. a transaction with recipient `null`. Use the endpoint of your Infura project from sub-task (b) as the Web3 provider.

   > **Solution:**

```
const Web3 = require("web3");
const web3 = new Web3(new Web3.providers.HttpProvider("https://mainnet.infura.io/v3/API_KEY"));

function iterate(blockNr) {
  web3.eth.getBlock(blockNr, true, function(error, block) {
    for(const tx of block.transactions) {
      if(tx.to == null) {
        console.log("Block number with first contract-creation:", blockNr);
        return;
      }
    }
    iterate(blockNr + 1);
  });
}

iterate(0);
```

To execute the program, run `node index.js`. The first contract-creation transaction is in block 46402.

(e) What could be done to speed up the execution?

Solution:
Infura is quite slow. To speed up the execution, we should host our own Ethereum full node.

2. Truffle is a popular development environment and testing framework for Ethereum smart contracts. In this exercise, we use it to develop and test an ERC20 token.

   (a) Install Truffle (`https://truffleframework.com/`) and create a new project with `truffle init`. Which folders does Truffle create and what are they for?

   Solution:

   - **contracts** – This folder holds all Solidity smart contracts we write.
   - **migrations** – This folder is for all deployment scripts.
   - **test** – This is a folder for all test scripts.

   (b) The `truffle-config.js` file is used to configure Truffle. Set the deployment network to the pre-configured `development` network and set the compiler version to 0.5.1. Also, enable the optimizer by uncommenting the appropriate lines in the config file.

   Solution:

```
module.exports = {
  networks: {
    development: {
     host: "127.0.0.1",
     port: 8545,
     network_id: "*",
    },
  },
  mocha: {
  },
  compilers: {
    solc: {
      version: "0.5.1",
      settings: {
       optimizer: {
         enabled: true,
         runs: 200
       },
      }
    }
  }
}
```

(c) We want to implement a token for the BBSE lecture according to the ERC20 standard (`https://eips.ethereum.org/EIPS/eip-20`). Create a new interface `ERC20.sol` containing only the function and event definitions from the standard.

Solution:

```
pragma solidity >=0.5.1 <0.6.0;

interface ERC20 {
  function totalSupply() external view returns (uint256);
  function balanceOf(address _owner) external view returns (uint256 balance);
  function transfer(address _to, uint256 _value) external returns (bool success);
  function transferFrom(address _from, address _to, uint256 _value) external returns (bool success);
  function approve(address _spender, uint256 _value) external returns (bool success);
  function allowance(address _owner, address _spender) external view returns (uint256 remaining);

  event Transfer(address indexed _from, address indexed _to, uint256 _value);
  event Approval(address indexed _owner, address indexed _spender, uint256 _value);
}
```

(d) Create another contract `BBSECoin.sol`, which implements the interface. Use the directive `import "./ERC20.sol";` to import your other file. Add global variables for the token name and symbol. Also, set the decimals to 0 and the total supply to 1000. Then, add the following two global variables and explain what they are used for:

```
1 mapping (address => uint256) public balances;
2 mapping (address => mapping (address => uint256)) public allowed;
```

Solution:

```
pragma solidity >=0.5.1 <0.6.0;

import "./ERC20.sol";

contract BBSECoin is ERC20 {

  string public name = "BBSECoin";
  string public symbol = "BBSE";
  uint8 public decimals = 0;
  uint256 private _totalSupply = 1000;
  mapping (address => uint256) public balances;
  mapping (address => mapping (address => uint256)) public allowed;
}
```

The first mapping is used for keeping track of how many tokens each account has. The second mapping is used to keep track of which account has allowed which other account to transfer tokens on their behalf. It also sets a limit of how many tokens each account can transfer in total for them.

(e) Now implement the token functions. See the standard for the exact function definitions and their purpose. Initially, all tokens should belong to the contract creator. Make sure to emit the proper events at the right places.
Use the command `truffle compile` to compile your contract and check for compilation errors.

Solution:

```
  constructor() public {
    balances[msg.sender] = _totalSupply;
  }

  function totalSupply() public view returns (uint256){
    return _totalSupply;
  }

  function balanceOf(address _owner) public view returns (uint256 balance){
    return balances[_owner];
  }

  function transfer(address _to, uint256 _value) public returns (bool success){
    require(balances[msg.sender] >= _value);
    balances[msg.sender] -= _value;
    balances[_to] += _value;
    emit Transfer(msg.sender, _to, _value);
    return true;
  }

  function transferFrom(address _from, address _to, uint256 _value) public returns (bool success){
    require(allowed[_from][msg.sender] >= _value && balances[_from] >= _value);
    allowed[_from][msg.sender] -= _value;
    balances[_from] -= _value;
    balances[_to] += _value;
    emit Transfer(_from, _to, _value);
    return true;
  }

  function approve(address _spender, uint256 _value) public returns (bool success){
    allowed[msg.sender][_spender] = _value;
    emit Approval(msg.sender, _spender, _value);
    return true;
  }

  function allowance(address _owner, address _spender) public view returns (uint256 remaining){
    return allowed[_owner][_spender];
  }
```

(f) Write a migration script `2_bbse_coin.js` that deploys your `BBSECoin` contract to the blockchain.

Refer to the other already existing migration script in your project to see what it should look like.

Solution:

```
const BBSECoin = artifacts.require("BBSECoin");

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

(g) Now use `truffle develop` to start the local development environment. Type `migrate` to deploy your contract. Explain what Truffle does in these two steps.

Solution:

Truffle creates ten test accounts on your local private Ethereum chain with 100 ETH each. The first account then deploys the `BBSECoin` contract onto the blockchain.

(h) Truffle supports the JavaScript test framework mocha.js. The following script tests whether the token name is correct.

```
1 const BBSECoin = artifacts.require("BBSECoin");
2
3 contract("BBSECoin", async accounts => {
4   it("should set the token name correctly", async () => {
5     let bbseCoinInstance = await BBSECoin.deployed();
6     assert.equal(await bbseCoinInstance.name(), "BBSECoin");
7   });
8 });
```

See `https://truffleframework.com/docs/truffle/testing/writing-tests-in-javascript` for some more information on how to reference accounts and send transactions from specific accounts. Then add the following test cases to the script:

- The initial token balance of the creator account is equal to the total token supply
- Tokens can be transferred using the `transfer()` function
- The allowance can be set and read
- Accounts can transfer tokens on behalf of other accounts

Type `test` to run the test cases.

Solution:

```
it("should initialize the owner balance", async () => {
  let bbseCoinInstance = await BBSECoin.deployed();
  let balance0 = await bbseCoinInstance.balanceOf(accounts[0]);
  let totalSupply = await bbseCoinInstance.totalSupply();

  assert.equal(balance0.toNumber(), totalSupply.toNumber());
});

it("should be able to transfer tokens", async () => {
  const amount = 50;
  let bbseCoinInstance = await BBSECoin.deployed();
  let balance0 = await bbseCoinInstance.balanceOf(accounts[0]);
  let balance1 = await bbseCoinInstance.balanceOf(accounts[1]);
  await bbseCoinInstance.transfer(accounts[1], amount, { from: accounts[0] });

  assert.equal(await bbseCoinInstance.balanceOf(accounts[0]), balance0.toNumber() - amount);
  assert.equal(await bbseCoinInstance.balanceOf(accounts[1]), balance1.toNumber() + amount);
});

const allowanceAmount = 20;

it("should be able to set and read allowance", async () => {
  let bbseCoinInstance = await BBSECoin.deployed();
  await bbseCoinInstance.approve(accounts[2], allowanceAmount, { from: accounts[0] });

  assert.equal(await bbseCoinInstance.allowance(accounts[0], accounts[2]), allowanceAmount);
  assert.equal(await bbseCoinInstance.allowance(accounts[0], accounts[1]), 0);
});

it("should be able to transfer on the behalf of other accounts", async () => {
  let bbseCoinInstance = await BBSECoin.deployed();
  let balance0 = await bbseCoinInstance.balanceOf(accounts[0]);
  let balance3 = await bbseCoinInstance.balanceOf(accounts[3]);
  await bbseCoinInstance.transferFrom(accounts[0], accounts[3], allowanceAmount, { from: accounts[2] });

  assert.equal(await bbseCoinInstance.allowance(accounts[0], accounts[2]), 0);
  assert.equal(await bbseCoinInstance.balanceOf(accounts[0]), balance0.toNumber() - allowanceAmount);
  assert.equal(await bbseCoinInstance.balanceOf(accounts[3]), balance3.toNumber() + allowanceAmount);
});
```

(i) In order to implement some advanced test cases, the npm library `truffle-assertions` is useful. It allows to check if events occurred, if a transaction reverted, and more. See the documentation here for more details: `https://www.npmjs.com/package/truffle-assertions`. Install the library, then implement and run the following test cases:

- An insufficient balance throws an error when trying to transfer tokens
- Transferring from an account that has not explicitly authorized the transfer should revert the transaction
- The `transfer()` and `transferFrom()` functions must fire the `Transfer` event (even for 0 value transfers)
- The `approve()` function must fire the `Approval` event

Note: Of course, this list of test cases is far from complete. Many more useful test cases can be thought of.

Solution:

```
it("should throw if balance is not enough when transferring tokens using transfer()", async () => {
  const amount = 5000;
  let bbseCoinInstance = await BBSECoin.deployed();
  let balance0 = await bbseCoinInstance.balanceOf(accounts[0]);
  let balance1 = await bbseCoinInstance.balanceOf(accounts[1]);
  await truffleAssert.reverts(bbseCoinInstance.transfer(accounts[1], amount, { from: accounts[0] }));

  assert.equal(await bbseCoinInstance.balanceOf(accounts[0]), balance0.toNumber());
  assert.equal(await bbseCoinInstance.balanceOf(accounts[1]), balance1.toNumber());
});

it("should throw if an unauthorized transferFrom happens", async () => {
  const amount = 10;
  let bbseCoinInstance = await BBSECoin.deployed();
  let balance0 = await bbseCoinInstance.balanceOf(accounts[0]);
  let balance1 = await bbseCoinInstance.balanceOf(accounts[1]);
  await truffleAssert.reverts(bbseCoinInstance.transferFrom(accounts[1], accounts[2], amount,
    { from: accounts[2] }));

  assert.equal(await bbseCoinInstance.balanceOf(accounts[0]), balance0.toNumber());
  assert.equal(await bbseCoinInstance.balanceOf(accounts[1]), balance1.toNumber());
});

it("should fire Transfer event for transfer() (even for 0 transfers)", async () => {
  const amount = 10;
  let bbseCoinInstance = await BBSECoin.deployed();
  let tx = await bbseCoinInstance.transfer(accounts[1], amount, { from: accounts[0] });
  truffleAssert.eventEmitted(tx, 'Transfer', (ev) => {
    return ev._from === accounts[0] && ev._to === accounts[1] && ev._value.toNumber() === amount;
  });

  let tx2 = await bbseCoinInstance.transfer(accounts[1], 0, { from: accounts[0] });
  truffleAssert.eventEmitted(tx2, 'Transfer', (ev) => {
    return ev._from === accounts[0] && ev._to === accounts[1] && ev._value.toNumber() === 0;
  });
});

it("should fire Transfer event for transferFrom() (even for 0 transfers)", async () => {
  const amount = 10;
  let bbseCoinInstance = await BBSECoin.deployed();
  await bbseCoinInstance.approve(accounts[1], amount, { from: accounts[0] });

  let tx = await bbseCoinInstance.transferFrom(accounts[0], accounts[2], amount, { from: accounts[1] });
  truffleAssert.eventEmitted(tx, 'Transfer', (ev) => {
    return ev._from === accounts[0] && ev._to === accounts[2] && ev._value.toNumber() === amount;
  });

  let tx2 = await bbseCoinInstance.transferFrom(accounts[0], accounts[3], 0, { from: accounts[4] });
  truffleAssert.eventEmitted(tx2, 'Transfer', (ev) => {
    return ev._from === accounts[0] && ev._to === accounts[3] && ev._value.toNumber() === 0;
  });
});

it("should fire Approval event when setting an allowance", async () => {
  const amount = 10;
  let bbseCoinInstance = await BBSECoin.deployed();
  let tx = await bbseCoinInstance.approve(accounts[1], amount, { from: accounts[0] });

  truffleAssert.eventEmitted(tx, 'Approval', (ev) => {
    return ev._owner === accounts[0] && ev._spender === accounts[1] && ev._value.toNumber() === amount;
  });
});
```

Don't forget to add the line `const truffleAssert = require('truffle-assertions');` at the beginning of the test script.