Ethereum Smart Contracts: Writing and Testing Smart Contracts

1 Overview

In this exercise sheet, you are asked to complete the missing parts of two smart contracts and make them successfully pass the prepared test suite before deployment.

- The GitHub repository for the Truffle setup including the incomplete contracts, migration files, and test suites can be found here `https://github.com/sebischair/bbse-bank-student`.
- You can fork, clone, or just download the ZIP version of the repo. You don't need to upload your solution anywhere. It is sufficient to make the tests pass.

2 Requirements

BBSEBank is a decentralized application (dApp) that enables users to earn interest by depositing Ether. The interest is calculated based on a pre-defined yearly return rate. Although the users can withdraw their Ether back anytime they want, the longer their deposit stays in the contract, the more interest they earn. The interest is paid in the form of **BBSE token** which is an ERC20 token. The application is currently in the development stage of its on-chain logic.

Tokens are smart contracts that implement a standardized interface and are currently the main use case in the Ethereum ecosystem. Depending on the actual use case, different token standards exist. The most common standards are ERC20 and ERC721. The former defines a token class where tokens are indistinguishable and have exactly the same value (i.e. fungible tokens). The latter standard defines a token class where each token is uniquely identifiable (i.e. Non-fungible Tokens - NFTs).

The ERC20 standard was first proposed by Fabian Vogelsteller and Vitalik Buterin in November 2015. The specification defines an interface that a contract must implement to be ERC20 compliant. It does not specify the actual implementation of the functions. You can find the interface here `https://eips.ethereum.org/EIPS/eip-20`. Read through the documentation and try to understand the role of each function.

- The requirements for the *BBSEBank.sol* contract are:
  - A yearly return rate should be configurable initially.
  - The minimum deposit amount should be 1 Ether.
  - It should restrict each investor (i.e. account) to have one active deposit.
  - It should include a logic for calculating the interest per second based on the minimum allowed deposit amount.
  - A **deposit** and a **withdraw** function should be available.
    - **deposit**: Checks whether the deposit amount is greater or equal to 1 Ether and the investor has no active deposits. If both conditions are satisfied, accepts and starts the investment.
    - **withdraw**: Checks whether the investor has an active deposit. If the condition is satisfied, calculates the interest based on the deposited amount and deposit duration. Transfers the deposited amount in Ether and pays the interest by minting BBSE tokens to the investor address on the BBSE token contract. Finally, resets (i.e. deactivates) the investor's deposit.

- The requirements for the *BBSEToken.sol* contract are:
  - It should be an ERC20 token contract that uses the ERC20 token standard implementation by OpenZeppelin.

    OpenZeppelin provides a library for secure smart contract development. It is built on a solid foundation of community-vetted code. Documentation about available contracts can be found here `https://docs.openzeppelin.com/contracts/4.x/`. *The actual implementations of the contracts are available on* `https://github.com/OpenZeppelin/openzeppelin-contracts`. *OpenZeppelin contracts can be installed using npm and imported directly into a contract.*

    The ERC20 implementation by OpenZeppelin is a standard that is also recognized by the official EIP20 documentation. You can find the implementation here `https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol`. *It is a pretty common practice to use the ERC20 implementation by OpenZeppelin when creating ERC20 token contracts, instead of explicitly implementing the ERC20 interface inside the contract.*

  - It should set the token name to *BBSE TOKEN* and the symbol to *BBSE*.
  - It should only allow the initialized minter to mint new tokens.
  - The minter role should be transferable to another account by the current minter.

3 Testing in Truffle

Truffle comes with a standard automated testing framework for testing your smart contracts. It uses the Mocha testing framework (`https://mochajs.org/`) and Chai (`https://www.chaijs.com/`) for assertions to provide you with a solid framework from which to write your JavaScript tests. It is important to test your contracts before deploying them in order to verify that they behave as expected.

For this exercise, we have provided you with two test files (*bbsebank.test.js* and *bbsetoken.test.js*) that test out the behavior of BBSEBank and BBSEToken smart contracts. Take a look at these files and try to understand which conditions are tested.

4 Tasks

Complete the following tasks using the provided project in the GitHub repository. Don't forget to run `npm i` inside the root directory when you first clone the project.

- Complete the implementation of the BBSEToken contract **constructor** based on the provided comments.
- Complete the implementation of the **mint** function on the BBSEToken contract based on the provided comments.
- Complete the implementation of the **passMinterRole** function on the BBSEToken contract based on the provided comments.
- Make all tests in **bbsetoken.test.js** pass. You can run the test file using `truffle test test/bbsetoken.test.js` command.
- Add the missing state variables to the BBSEBank contract.

- Complete the implementation of the BBSEBank contract **constructor** based on the provided comments.
- Complete the implementation of the **deposit** function on the BBSEBank contract based on the provided comments.
- Complete the implementation of the **withdraw** function on the BBSEBank contract based on the provided comments.
- Make all tests in **bbsebank.test.js** pass. You can run the test file using `truffle test test/bbsebank.test.js` command.
- Deploy the contracts to Ganache network.

5 Tips

- Refer to the test cases to better understand what's the expected behavior of a function.
- You can use *require* statements instead of writing full modifiers.
- Expected error messages can be found in test cases.
- Message value is represented in Wei, not Ether. Don't forget to do the necessary conversions!