Ethereum Design Patterns: Applying Idioms and Patterns

1 Overview

In this exercise sheet, you are asked to add new functionalities to the BBSEBank decentralized application that you have built in the previous exercise sheet. You will be completing contract implementations and testing them against prepared test suites. The goal of this exercise sheet is to help you gain practical experience with how to use *Access Restriction Idiom* and *Oracle Design Pattern*.

- The GitHub repository for BBSEBank 2.0 including the incomplete contracts, migration files, and test suites can be found here `https://github.com/sebischair/bbse-bank-2.0-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

The first version of BBSEBank only supported deposit and withdrawal functionalities. In **BBSEBank 2.0**, you want to offer a borrowing functionality. The application should enable users to borrow ETH by collateralizing their BBSE tokens and pay back anytime they want.

In decentralized finance (DeFi), lending (i.e., borrowing) protocols require the users to collateralize an asset that has a greater value than the requested loan amount (usually at least 150%). This is called *over-collaterizing*. For example, if you want to take 1 ETH for a loan, you need to collateralize an asset worth at least 1.5 ETH. BBSEBank 2.0 users are also required to over-collateralize their loans.

- The new requirements for the *BBSEBank.sol* contract are:
  - It should inherit from the Ownable contract by OpenZeppelin.
    The Ownable contract by OpenZeppelin is used for implementing ownership logic in smart contracts. Read more here `https://docs.openzeppelin.com/contracts/2.x/access-control`.
  - Yearly return rate should be configurable anytime by the contract owner.
  - Collateralization ratio should be 150%.
  - %1 fee should be taken from every loan.
  - A **borrow** and a **payLoan** function should be available.
    - **borrow**: Checks, whether the borrower has no active loans and the bank contract has enough ETH in its balance to lend the loan. If both conditions are satisfied, transfers the requested ETH to the borrower and gets the collateral.
    - **payLoan**: Checks, whether the borrower has an active loan and the paid amount is equal to the loaned amount. If the conditions are satisfied, calculates the fee to be taken. Transfers back the collateral after cutting the fee. Finally, resets (i.e. deactivates) the borrower's loan.

In BBSEBank, loans are given in ETH while collateral is in the form of BBSE tokens. Since a BBSE token does not have the same value as ETH, we need to get the latest rate of ETH/BBSE, when finding the required collateral value. As the rate data is available outside the blockchain (e.g., through an API), we need to use an oracle.

Smart contracts can't access any data from outside the blockchain on their own. No HTTP or similar network methods are implemented to call or access external services. This is on purpose to prevent non-deterministic behavior once a function is called.

Currently, the only way to write smart contracts using external data (e.g., weather data, traffic data, etc.) is to use oracles. Oracles are third-party services that collect data from web services and write the data via a special smart contract to the blockchain. Other smart contracts can now call the Oracle contract to get the data.

- The requirements for the *ETHBBSEPriceFeedOracle.sol* contract are:
  - It should an inherit from the Ownable contract by OpenZeppelin.
  - It should provide the current rate of ETH/BBSE pair.
  - It should emit an event to trigger the off-chain oracle server when the last price update took place more than three blocks ago.
  - It should only let the owner of the contract update the rate.

3 Testing in Truffle

For this exercise, we have provided you three test files (*bbsetoken.test.js*, *oracle.test.js*, *bbsebank.test.js*) that test out the behavior of BBSEToken, ETHBBSEPriceFeedOracle, and BBSEBank smart contracts. 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 ETHBBSEPriceFeedOracle contract **constructor** based on the provided comments.
- Implement the **updateRate** function on the ETHBBSEPriceFeedOracle contract based on the provided comments.
- Implement the **getRate** function on the ETHBBSEPriceFeedOracle contract based on the provided comments.
- Make all tests in **oracle.test.js** pass. You can run the test file using `truffle test test/oracle.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.
- Implement the **updateYearlyReturnRate** function on the BBSEBank contract based on the provided comments.
- Complete the implementation of the **borrow** function on the BBSEBank contract based on the provided comments.
- Complete the implementation of the **payLoan** 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. Try to understand what's going on in the **setTheScene** function.
- Deploy the contracts to Ganache network with `truffle migrate` command.

5 Bonus

An oracle contract requires an off-chain oracle server to feed it with data. The oracle server should listen to events triggered by the oracle contract, and update it when new data is required. In ETHBBSEPriceFeedOracle's case, this data is a price rate. Since there is no ETH/BBSE in reality, we use an existing pair to mock the rate. For this purpose, we chose *AAVE* (`https://aave.com/`) which is a famous loan provider protocol on Ethereum.

- Under *oracle-server* folder, you will find a script (*index.js*) that is responsible for updating the oracle contract.
  1. Run `npm i` inside the directory.
  2. Create an *.env* file and store the following variables:
     - **API_KEY**: Your api key from *CoinAPI.io* - `https://docs.coinapi.io/#md-docs`
     - **API_HOST**: http://rest.coinapi.io
     - **BASE**: ETH
     - **QUOTE**: AAVE
- Run `node –experimental-json-modules index.js` to start the oracle server.
- Check the output on the console. You will see that *GetNewRate* event emitted from the constructor is caught and the oracle is updated with the new fetched rate of ETH/AAVE.
- Open a different terminal tab inside the project directory and run `truffle console`. We will be interacting with the Ganache network using this console.
- Run the following commands. To better understand what they are doing, you can also check *setTheScene* function in **bbsebank.test.js**.
  1. const bank = await BBSEBank.deployed()
  2. const token = await BBSEToken.deployed()
  3. await bank.deposit({from:accounts[1], value:10**19})
  4. await bank.deposit({from:accounts[2], value:10**18})
  5. await bank.deposit({from:accounts[3], value:10**18})

6. await bank.deposit({from:accounts[4], value:10**18})
7. await bank.withdraw({from:accounts[1]})
8. await token.approve(bank.address, web3.utils.toWei(”0.05”, ”ether”), {from:accounts[1]})
9. await web3.eth.sendTransaction({from:accounts[9], to: bank.address, value: web3.utils.toWei(”10”, ”ether”)})
10. await bank.borrow(web3.utils.toWei(”0.001”, ”ether”), {from:accounts[1]})

- Check the output on the oracle server console. You will see that *GetNewRate* event emitted from the **getRate** function is caught and the oracle is updated with the new fetched rate of ETH/AAVE.

6 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. If you are confused, use a converter tool like `https://eth-converter.com/`.
- Be careful when copying code or commands from the exercise sheet.