MSE | MASTER OF SCIENCE IN ENGINEERING

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts Western Switzerland

Master of Science HES-SO in Engineering
Av. de Provence 6
CH-1007 Lausanne

Master of Science HES-SO in Engineering

Orientation: Computer Science (CS)

CRYPTOCURRENCY TOKEN FOR INTERNATIONAL SETTLEMENTS

Author:
Luca Srdjenovic

Under the direction of:
Prof. Dr. Ninoslav Marina
HE-Arc

External expert:

Lausanne, HES-SO//Master, May 30, 2023

Information about this report

Contact information

Author: Luca Srdjenovic
MSE Student
HES-SO//Master
Switzerland
Email: *luca.srdjenovic@master.hes-so.ch*

Declaration of honor

I, undersigned, Luca Srdjenovic, hereby declare that the work submitted is the result of a personal work. I certify that I have not resorted to plagiarism or other forms of fraud. All sources of information used and the author quotes were clearly mentioned.

Place, date: _____

Signature: _____

Validation

Accepted by the HES-SO//Master (Switzerland, Lausanne) on a proposal from:

Prof. Dr. Ninoslav Marina, Thesis project advisor
, , Main expert

Place, date: _____

Prof. Dr. Ninoslav Marina
Advisor

Dean, HES-SO//Master

Abstract

Key words: Blockchain, Cryptography, Ethereum, Smart Contracts, ERC20

Contents

1 Methodology & models

This chapter describes the methodology and models used to design and implement our international settlement system to achieve the objectives of this project, which focus on creating a decentralized and secure platform for token exchange. We use a systematic methodology and develop two proof-of-concept implementations. These proofs of concepts serve as concrete examples to demonstrate the functionalities and feasibility of our proposed solution.

In section 1.1, we present the first proof of concept, which is a development of a wallet for fungible tokens. This component enables users to securely store and manage their digital assets. By implementing a wallet, we aim to give to the users the ability to store, send, and receive tokens within our international settlement system, which is decentralized and secure. This proof of concept is the foundation for our overall project and provides the basis for developing the second proof of concept.

After developing the first proof of concept, the section 1.2 presents our second proof of concept, focusing on creating a decentralized exchange for token exchange. This component is the core of our international settlement system, enabling users to exchange tokens transparently and securely as a decentralized marketplace. In developing our platform exchange, we aim to demonstrate the need for a reliable and secure environment where users can confidently engage in token exchange without intermediaries. This proof of concept demonstrates the functionalities of our international settlement system and serves as a concrete example of its potential to replace the current financial system.

1.1 Wallet for fungible tokens

This section presents the design and implementation of our first proof of concept, which entails the development of a digital asset wallet for our international settlement system. The digital asset is a fungible "Universal Exports Token" (UET).

To simplify the design and implementation of the smart contract, we leverage the ERC20 standard [1] in conjunction with the OpenZeppelin library [2]. This combination reduces the complexity involved in creating the smart contract. Once the smart contract is deployed, we build a blockchain application that serves as a digital asset wallet. The application is developed using the Truffle framework [3] and the ReactJS framework [4]. These frameworks enable us to seamlessly connect to a local Ethereum network facilitated by Ganache-cli [5]. This local network provides a controlled environment for testing and interaction with the digital asset wallet.

1.1.1 Token Smart Contract

For our digital asset, we create a smart contract in Solidity [6] that implements the ERC20 standard [1] using the OpenZeppelin library [2]. We are using the latest version of the OpenZeppelin library, version 4.8.3, which is compatible with Solidity version 0.8.0 in the writing of this report.

The diagram in Figure 1.1 shows the class diagram of the *UEToken* smart contract to have a better understanding of the design of the smart contract. The diagram shows the inheritance of the *UEToken* smart contract from the *ERC20Capped* and *Ownable* smart contracts. The *ERC20Capped* smart contract inherits from the *ERC20* smart contract, which implements the ERC20 standard. This design choice allows us

to leverage the features and security enhancements the OpenZeppelin library provides, ensuring a standardized implementation of the ERC20 standard and incorporating best practices for secure smart contract development.

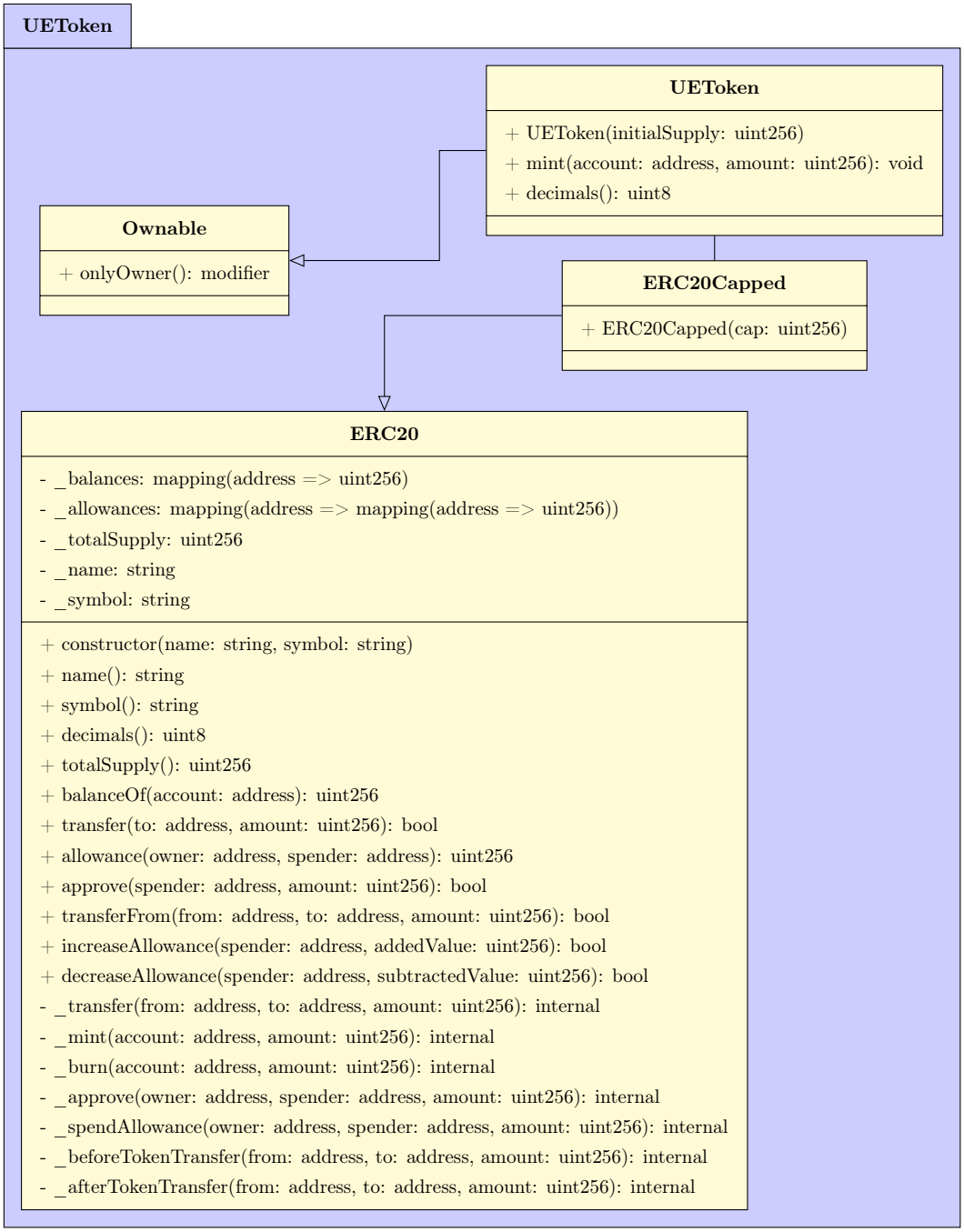

Figure 1.1: Class diagram of the UEToken smart contract using OpenZeppelin's ERC20 implementation.

Now that we have a better understanding of the design of the smart contract, we can see the implementation of the *UEToken* smart contract in Listing 1.1. The code snippet in Listing 1.1 represents the Solidity source code of our *UEToken* smart contract.

```
1  // contracts/GLDToken.sol
2  // SPDX-License-Identifier: MIT
3  pragma solidity ^0.8.0;
4
5  import "@openzeppelin/contracts/token/ERC20/extensions/ERC20Capped.sol";
6  import "@openzeppelin/contracts/access/Ownable.sol";
7
8  contract UEToken is ERC20Capped, Ownable {
9      constructor(
10         uint256 initialSupply
11     ) ERC20("Universal Export Token", "UET") ERC20Capped(1000000000) {
12         _mint(msg.sender, initialSupply * 10 ** decimals());
13     }
14
15     function mint(address account, uint256 amount) public onlyOwner {
16         _mint(account, amount);
17     }
18
19     function decimals() public view virtual override returns (uint8) {
20         return 2;
21     }
22 }
```

Listing 1.1: Solidity source code of the UEToken smart contract.

Line 3 of the code snippet in Listing 1.1 shows the Solidity version used for the smart contract. For the writing of this report, we are using the latest version of Solidity, version 0.8.0.

Lines 5 and 6 illustrate the import of the *ERC20Capped* and *Ownable* smart contracts from the OpenZeppelin library. Let us dive into the *ERC20Capped* smart contract to understand the design of the smart contract. In figure 1.2, we can show the design of the *ERC20Capped* smart contract in detail. As illustrated in Figure 1.1, the *ERC20Capped* smart contract inherits from the *ERC20* smart contract. ERC20Capped is an abstract contract that inherits from the *ERC20* smart contract and overrides the `_mint` function to add a token supply cap by verifying that the total supply of tokens does not exceed the cap. The *ERC20Capped* smart contract is designed to be inherited by other smart contracts, which is the case for our *UEToken* smart contract.

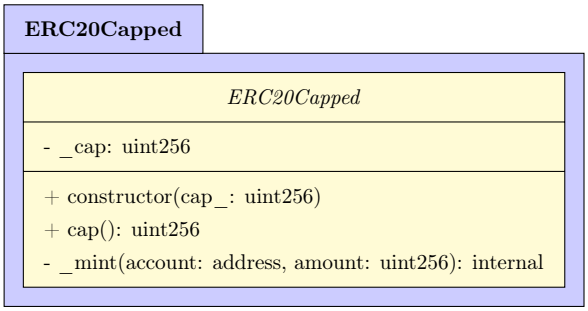

Figure 1.2: Class diagram of the OpenZeppelin ERC20Capped smart contract.

By extending the `ERC20Capped` contract, our token has a capped supply, meaning that the total supply of tokens cannot exceed a specified limit. By setting this cap on the supply of tokens, we control the maximum number of tokens that can never

be created. It can be helpful in some use cases, such as creating a stablecoin, where the supply of tokens is backed by a reserve of assets, or in maintaining a fixed token economy. In our use case, we are not backing our token with a reserve of assets or fiat currency, but we are using the cap to limit the supply of tokens to a maximum number of tokens. In our use case, we set the cap to 1 billion tokens, the maximum number of tokens that can be minted. This provides us transparency and trust by incorporating this cap to token holders and users of our token. This can build trust and confidence in the token's ecosystem. The cap is set in the constructor of the `UEToken` smart contract, as shown in Listing 1.1. The other smart contract we import from the OpenZeppelin library is the `Ownable` smart contract. We illustrate in Figure 1.3 the detailed design of the OpenZeppelin `Ownable` smart contract.

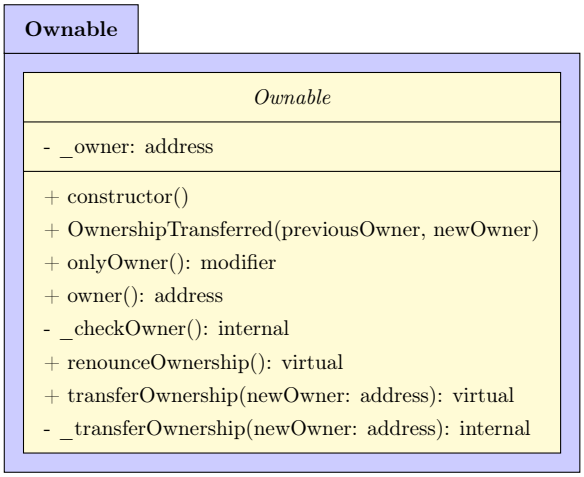

Figure 1.3: Class diagram of the OpenZeppelin Ownable smart contract.

The `Ownable` contract implements a direct access control mechanism in our `UEToken` smart contract. Although, by incorporating the `Ownable` smart contract into our token implementation, we can grant exclusive access to specific functions, such as the `mint` function used for creating new tokens, through inheritance from the `Ownable` contract, our token uses the `onlyOwner` modifier, which limits the execution of these functions to the contract owner. Consequently, only the owner who deployed the smart contract can execute the `mint` function, as it is annotated with the `onlyOwner` modifier. In our case, we aim to demonstrate the controlled minting of new tokens exclusively by the contract owner, preventing the possibility of unauthorized minting that could devalue the tokens. By default, the `Ownable` contract sets the contract owner in its constructor. However, the ownership can be transferred by calling the `transferOwnership` function, offering flexibility in contract management when necessary. The `Ownable` contract is utilized in scenarios where specific critical functions, such as the `mint` function, or administrative tasks must be restricted to the contract owner.

Lastly, in line 11 of the code snippet in Listing 1.1, the name and symbol of the token are defined in the constructor using the `ERC20` constructor inherited from the *ERC20Capped* contract. As explained in the previous chapter (Chapter **??**), specifically in Section **??**, the ERC20 token provides a standard interface for fungible tokens. Figure 1.1 illustrates the class diagram of our token smart contract. The `ERC20` constructor accepts the name and symbol of the token as parameters, which

are set to `UEToken` and `UE`, respectively. Additionally, our constructor takes an `initialSupply` parameter, which sets the initial supply of tokens upon deployment of the smart contract.

In line 12, we invoke the `_mint` function from the *ERC20Capped* contract to mint the initial supply of tokens. The `_mint` function requires the address of the account to be minted (i.e., the contract owner) and the number of tokens to be minted (i.e., the initial supply of tokens). Furthermore, in our contract (line 15), we create the `mint` function, utilizing the `_mint` private function from the *ERC20Capped* contract. The `mint` function accepts the account's address to be minted and the number of tokens to be minted as parameters. The `onlyOwner` modifier is applied to the `mint` function to restrict its execution solely to the contract owner. Our platform aims to demonstrate the contract owner's minting of new tokens exclusively.

In line 19, we override the `decimals` function from the *ERC20* contract to set the number of decimals to 2. By default, the `decimals` function returns 18 decimals. However, for our token representing standard currency (e.g., Euro), we set the number of decimals to 2 to align with the typical decimal representation of currency. This adjustment ensures compatibility with financial calculations and representations. This completes the necessary code for our proof of concept token smart contract. While we could have included additional functionality, such as a `burn` function for burning tokens, our focus remains on the primary feature of our platform, which is the exchange of tokens between users. The complete OpenZeppelin code can be found in the GitHub repository [7].

Our fully functional code serves as a template for creating other ERC20 tokens. For instance, we can create a stablecoin called `USDC` (USD Coin) by modifying the name, symbol, and maximum supply of tokens. Additionally, the return value of the `decimals` function can be adjusted to set the desired number of decimal places for the token. The extensibility of our token smart contract is one of the advantages of adhering to the ERC20 standard. We can leverage our token smart contract as a foundation to create multiple tokens according to our requirements.

1.1.2 Token Deployment

To deploy our token smart contract, *UEToken*, we can utilize Truffle [3], a development environment framework for Ethereum. Truffle offers a comprehensive suite of tools, including a development environment, testing framework, and asset pipeline for Ethereum. One of the critical features of Truffle is the ability to manage contract deployments through migrations.

Migrations in Truffle are written in JavaScript and allow us to define the deployment phases of our contracts. Each migration file represents a separate deployment step, and Truffle ensures that migrations are executed in the correct order. This ensures that the contracts are deployed in a coordinated manner.

Additionally, Truffle provides a console, a JavaScript runtime environment that exposes the Truffle environment to the command line. The console allows us to interact with our contracts and execute JavaScript commands against the deployed contracts.

Before deploying our contract, we need to compile it to transform the Solidity code into bytecode that can be executed by the Ethereum Virtual Machine (EVM) on the blockchain. Truffle simplifies the compilation process by providing a built-in compiler. The snippet in Listing 1.2 shows the compilation of our *UEToken* smart contract using the Truffle compiler :

```
$ truffle compile
Compiling your contracts...
===========================
> Compiling ./contracts/UEToken.sol
> Compiling @openzeppelin/contracts/access/Ownable.sol
> Compiling @openzeppelin/contracts/token/ERC20/ERC20.sol
> Compiling @openzeppelin/contracts/token/ERC20/IERC20.sol
> Compiling @openzeppelin/contracts/token/ERC20/extensions/ERC20Capped.sol
> Compiling @openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol
> Compiling @openzeppelin/contracts/utils/Context.sol
> Artifacts written to /truffle/build/contracts
> Compiled successfully using:
   - solc: 0.8.19+commit.7dd6d404.Emscripten.clang
```

Listing 1.2: Compiling the *UEToken* smart contract.

As we can see in Listing 1.2, Truffle searches for all Solidity files in the *contracts* directory and dependencies in the *node_ modules* directory. It compiles them according to the Solidity version specified in the *truffle-config.js* file. Finally, the compiled contracts are written to the *build/contracts* directory.

Once the compilation is successful, we can proceed to the migration process of our contract. To begin the migration process, we must create a new migration file, such as *uetoken_ migration.js*, in the Truffle migrations directory *migrations*. The migration file name should be descriptive of the contract that is being deployed. Therefore, we name the migration file *uetoken_ migration.js*, meaning we are deploying the *UEToken* contract. In this file, we need to specify the deployment code for our token contract. The code for deploying our token contract is shown in Listing 1.3.

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

module.exports = function(_deployer) {
  // Use deployer to state migration tasks.
  const initialSupply = 1000000
  _deployer.deploy(UEToken, initialSupply);
};
```

Listing 1.3: Deployment code for the *UEToken* smart contract.

In the above code, we use the `artifacts.require` function to import the *UEToken* contract artifact. Then, in the migration deployment function, we specify the initial supply of tokens to mint and deploy the *UEToken* contract using the `_deployer.deploy` function.

Once the migration file is created, we can deploy our token contract on a local test network. We use a local test network for our proof of concept because it is faster and cheaper to deploy our contract on a local test network than on the Ethereum mainnet or testnet such as Ropsten. We use Ganache-CLI [5] for the local test network, simulating our private Ethereum blockchain. Ganache-CLI is a command-line version of Ganache, a personal blockchain for Ethereum development. Ganache-CLI is a fast and customizable blockchain emulator. It allows us to make calls to the blockchain without the cost of running an actual Ethereum node.

We can start Ganache-CLI by running the following command:

```
$ ganache-cli -m "other strike boat weekend address want pink oval sister cry
↪  excuse myth" --chainId 1 --networkId 1337

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

Available Accounts
==================
(0) 0x216384d2f868d00ddC2907151F01b392aE0de155 (100 ETH)
[snip]
(9) 0x2d9d21e8ef8b0ebb4bcba4a9b79cd5d8dcfba7a7 (100 ETH)

Private Keys
==================
(0) 0xb406b5f9eb128076dea028c9c149918a8811d193d93dfbde0489c316129d8f31
[snip]
(9) 0x94dec33a7e428dbd78a527fb9e9837645f2b488c040f54e90847d401a1008767
```

Listing 1.4: Starting Ganache-CLI.

The output shown in Listing 1.4 indicates that Ganache-CLI has successfully started a local Ethereum blockchain with ten accounts, each having a balance of 100 ETH. To ensure consistency, we utilize the `-m` argument to specify a mnemonic phrase for the blockchain. This allows us to start the identical blockchain with the same accounts whenever needed, which is handy for testing. Additionally, we use the `-chainId` argument to set the chain ID of the blockchain. The chain ID is used to identify the specific blockchain network, and in our case, we set it to one for testing purposes. This aligns with the configuration we use in Metamask [8] to connect to our local blockchain. It is worth mentioning that when importing ERC20 tokens into Metamask, we need to specify the chain ID as 1 (Ethereum mainnet) due to a bug in Metamask [9]. Furthermore, we utilize the `-networkId` argument to define the network ID of the blockchain. The network ID serves as an identifier for the blockchain network, and for testing purposes, we set it to 1337.

To deploy our token contract on the Ganache-CLI blockchain, we need to execute the following command:

```
$ truffle migrate --network ganache

[snip]

uetoken_migration.js
====================

   Replacing 'UEToken'
   -------------------
   > transaction hash:
   ↪  0xb79defe9f743e4592560da60dabee61e0a66ee017a4f90a0c1d20720061659e0
   > Blocks: 0            Seconds: 0
   > contract address:    0xd543BdeE60836107Ad2F70b2384a01827a61AF12
   > block number:        2
   > block timestamp:     1685382153
   > account:             0x216384d2f868d00ddC2907151F01b392aE0de155
   > balance:             99.93655312
   > gas used:            1586172 (0x1833fc)
   > gas price:           20 gwei
   > value sent:          0 ETH
   > total cost:          0.03172344 ETH

   > Saving artifacts
   -------------------------------------
   > Total cost:          0.03172344 ETH

Summary
=======
> Total deployments:   1
> Final cost:          0.03172344 ETH
```

Listing 1.5: Deploying the token contract on the ganache-cli blockchain.

In Listing 1.5, we can see that the command instructs Truffle to execute the migration script `uetoken_migration.js` to the specified network. In this case, the network is `ganache`, defined in the `truffle-config.js` file, and is the local blockchain we started with Ganache-CLI. Upon successful deployment, we can see in Listing 1.5 that Truffle summarizes the migration process, including the transaction hash, the contract address, the block number, and transaction details. We can then interact with the deployed contract using the contract address.

To call to explain one functionality of our token from Truffle to verify that it works as expected, we can run the following command:

```
$ truffle console --network ganache

truffle(ganache)> let instance = await UEToken.deployed()
undefined
truffle(ganache)> let accounts = await web3.eth.getAccounts()
undefined
truffle(ganache)> let name = await instance.name()
undefined
truffle(ganache)> name
'UEToken'
```

Listing 1.6: Calling the `name()` function our token contract from Truffle.

In Listing 1.6, we can see that we first get an instance of the deployed token contract, then we get the list of accounts from the blockchain, and finally, we call the `name()` function of the token contract. The result is the name of the token, which is `UEToken`. We can easily interact with the deployed token contract by leveraging Truffle's console.

1.1.3 Frontend Development

Now that we have successfully deployed our token contract on the blockchain, the next step is to develop a front-end application that allows users to interact with the token contract. As this project is a proof-of-concept, we focus on creating a simple front-end application that demonstrates the core functionalities of our token. Therefore, this section provides an overview of the main components of our front-end application and how we interact with the token contract from the front end.

For the frontend development, we utilize React [4], a widely-used JavaScript library for building user interfaces. Additionally, we use the Web3.js [10] library (refer to Section **??**), which provides a collection of modules that enable interaction with Ethereum and our deployed token contract.

Web3.js is a package that can be installed using the Node Package Manager (NPM). It allows us to connect to an Ethereum network and interact with our deployed token contract within our front-end application. In addition, the library offers various modules to handle different tasks, such as contract instantiation, function calls, and even listening.

We first connect with our web3 instance to initiate our development process. In our case, we rely on Metamask[1] to connect to our local blockchain. Metamask injects a global variable called `web3` into the browser's window object.

We initialize our web3 instance by verifying the installation and unlocking of Metamask. Once confirmed, we instantiate our web3 instance using the `Web3` constructor. Below is a code snippet demonstrating our web3 initialization process:

[1] Metamask is a browser extension that enables us to connect to an Ethereum network directly from our browser.

```
import Web3 from 'web3';
// Code snipped for brevity
mount = async () => {
var account;
    if (window.ethereum) {
        try {
            await window.ethereum.request({ method: 'eth_requestAccounts'
            ↪ }).then((accounts) => {
            account = accounts[0];
            this.web3.eth.defaultAccount = account;
            // Code snipped for brevity
            erc20Tokens.forEach((token) => {
                let erc20Token = new this.web3.eth.Contract(token.abi, token.address);
                // Call any function to get the token name
                erc20Token.methods.name().call().then((name) => {
                app.setState({
                    tokens: [...app.state.tokens, { ...token, name }]
                    });
                });
            });
        });
        } catch (error) {
            console.error('Error connecting to web3:', error);
        }
        const ethereum = window.ethereum;
        window.web3 = new Web3(ethereum);
        this.web3 = new Web3(ethereum);
    } else {
        console.log('No ethereum browser detected');
    }
}
```

Listing 1.7: Web3 initialization.

In Listing 1.7, we demonstrate how to manage the web3 instance and the connected account state in our React component. Upon refreshing the browser, we utilize the `mount()` lifecycle method to handle this process. Firstly, we check for the availability of MetaMask and request the user's accounts. Once connected, we create a new instance of the `Web3` object and set the default account to the first account in the list provided by MetaMask.

To interact with our deployed token contract, we leverage the web3 instance. For instance, we can call various functions of our token contract, such as `name()`, `symbol()`, `totalSupply()`, and `balanceOf()`. This involves accessing the `methods` object of our token contract instance and invoking the respective functions. By retrieving the returned values, such as the token name (`UEToken`), symbol, total supply, and balance, we can update the state of our React component accordingly.

To obtain an instance of our token contract, we must provide the token contract's ABI and address to the `Contract` constructor of the `web3.eth` object. The ABI is obtained from the JSON file located in the `build/contracts` directory, as demonstrated in the deployment section (Section 1.1.2). It serves as a blueprint for Web3, enabling it to interact correctly with our token contract by understanding its functions and events.

In our proof-of-concept wallet platform, we can list all the user's tokens. We need to supply the respective ABI and contract addresses for each token. Our frontend application, implemented as a React component, renders an array of tokens, each with its own instantiated `Contract` object. This allows us to call the `balanceOf()`

function of the token contract, retrieving the user's balance for that particular token. Additionally, buttons associated with each token enable users to perform actions such as transferring tokens, approving transfers, and minting new tokens.

For example, to call the `transfer()` function, the following code snippet can be utilized:

```
Transfer = async () => {
    // Code snipped for brevity
    tokenContract = new this.web3.eth.Contract(this.state.transferDetail20.abi,
    ↪  this.state.transferDetail20.address);
    var amount = this.state.fields.amount * (Math.pow(10,
    ↪  this.state.transferDetail20.decimal));
    try {
        await tokenContract.methods.transfer(recipient, amount).send({ from:
        ↪  this.web3.eth.defaultAccount });
        console.log('Transfer successful!');
    } catch (error) {
        console.error('Error transferring tokens:', error);
    }
}
```

Listing 1.8: Calling the `transfer()` function of the web3 token contract instance.

In the provided code snippet (Listing 1.8), we begin by creating a new instance of the `Contract` object, which requires the ABI (Application Binary Interface) and address of the token contract. Then, to ensure the accuracy of the transfer amount, we multiply the user-entered amount by the decimal value of the token (as described in Equation **??**). Next, we call the `transfer()` function of the token contract, passing the recipient's address and the desired transfer amount as arguments. This function call returns a JavaScript promise, allowing us to handle the response using the `then()` function. Through this, we can update the state of our React component accordingly. The exact process applies to other functions, such as `approve()` and `mint()`, with their respective arguments.

1.2 Decentralized Exchange (DEX)

This section describes our second proof-of-concept application, a decentralized exchange (DEX) platform responsible for trading ERC20 tokens, as mentioned in the objectives of this project. Traditionally, trading exchanges are centralized, meaning that they are owned and operated by an authority for facilitating the trading between a buyer and a seller. This authority is responsible for posting a buy or sell order on behalf of the user where the order is visible to other users of the exchange. Once a match is found, the exchange executes the trade and updates the users' balances accordingly. This process is called an order book, where the exchange maintains a list of buy and sell orders submitted by users. Traders can submit orders to the order book, and the exchange will match the orders based on the price and quantity of the order. In the case of a match, the exchange will execute the trade and update the users' balances accordingly. In our proof-of-concept, we implement a decentralized version of this process, meaning we remove the centralized authority and allow users

to interact directly with each other via a smart contract replacing the order book. In our use case, we implement a simple order book that allows traders to trade our ERC20 token `UEToken` for a stablecoin[2] , `USDC`[3] which are also an ERC20 token.

In our project, we are building the following components:

Two ERC20 tokens

The UEToken is the token we created in the previous section and represents the base asset[4] of our exchange. The USDC token is a stablecoin that is pegged to the US dollar and represents the counter asset[5] of our exchange. Due we are using our private Ethereum network, we need to simulate the USDC token. To do so, we create a new ERC20 token contract and mint 1000 USDC tokens, for example. In a real case, the USDC token is issued by a centralized authority, Circle Internet Financial Ltd. [11]. If we were to deploy our exchange on the Ethereum mainnet, we would use the real USDC token with the address `0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48` [6].

Order book smart contract

The order book smart contract matches buy and sell orders submitted by traders. The order book is implemented as a smart contract deployed on the Ethereum blockchain. The order book contract keeps track of the buy and sell orders submitted by traders and allows traders to view the order book and submit orders against an existing order stored in the contract. The order book contract is implemented in Solidity and is deployed on the local Ethereum blockchain using the Truffle framework with the help of the Ganache CLI.

Trading platform

The trading platform is a web application that allows traders to interact with the order book smart contract. It shows users their balances of the two tokens, base token and counter token, and allows them to submit buy and sell orders to the order book. When a new order is submitted, an engine verifies if the new request matches an existing order in the order book. If a match is found, the trade is executed, and the users' balances are updated accordingly by invoking the order book smart contract trade function. On the other hand, if no match is found, the order is added to the order book and is visible to other traders by calling the add order function of the order book smart contract. The trading platform is implemented using the React JavaScript framework and the web3.js library for interacting with the Ethereum blockchain.

1.2.1 Design of the Order Book Smart Contract

The order book smart contract is designed to be as simple as possible, facilitating the trading of ERC20 tokens between buyers and sellers. It keeps track of the buy and sell orders and executes the trades when conditions are met. The design consists of the following components as shown in the class diagram in Figure 1.4.

[2]A stablecoin is a cryptocurrency that is pegged to a stable asset such as gold or fiat currencies.

[3]USDC is a stablecoin pegged to the US dollar, meaning that 1 USDC is always equal to 1 USD. `https://en.wikipedia.org/wiki/USD_Coin`

[4]The base asset is the asset that is being traded.

[5]The counter asset is the asset that is being traded against.

[6]The USDC token contract address on the Ethereum mainnet. `https://etherscan.io/token/0xa0b86991c6218b36c1d19d4a2e9eb0ce3606eb48`

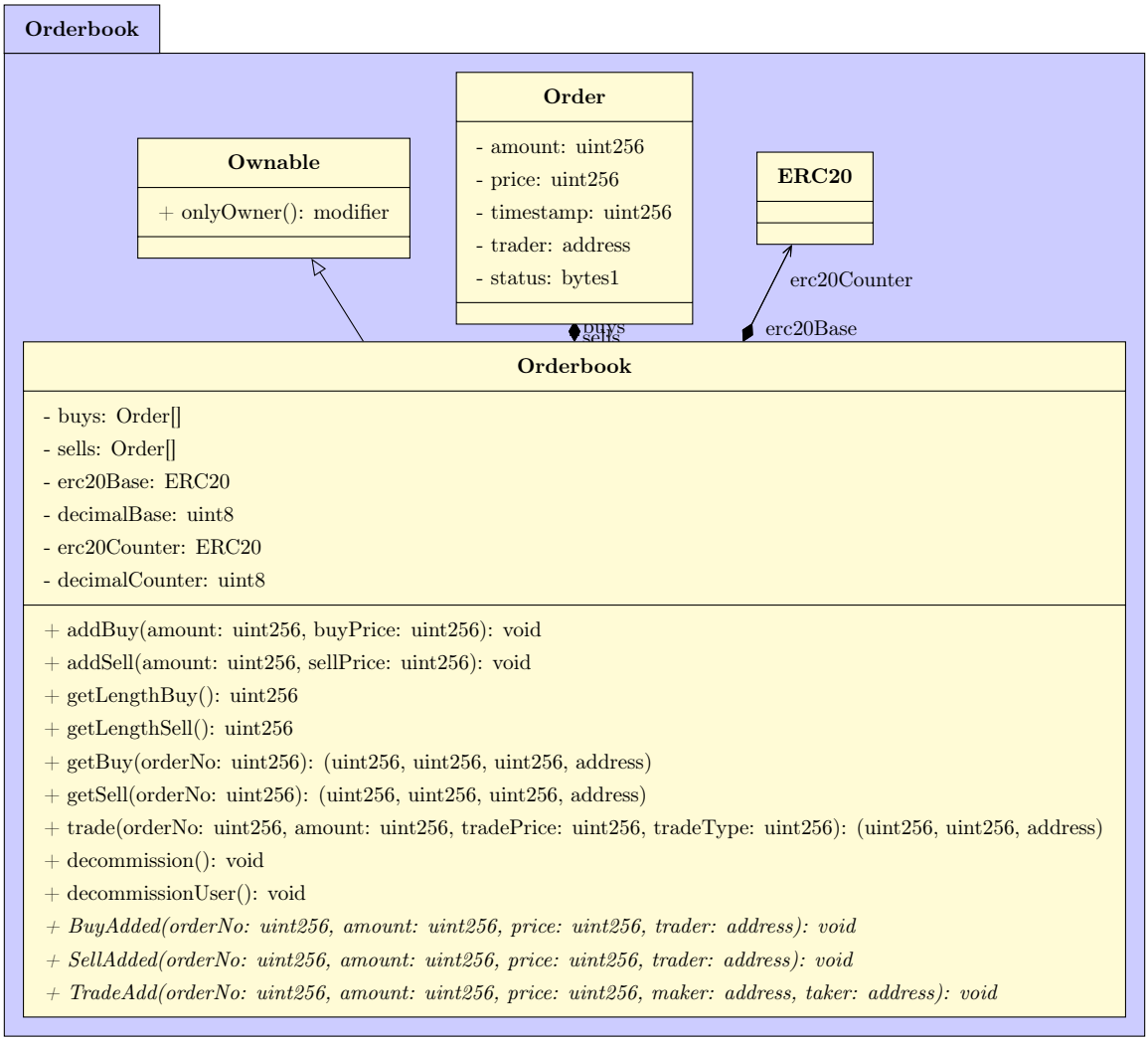

Figure 1.4: Class diagram of the Orderbook smart contract.

Struct Order

The `Order` struct is used for an individual buy or sell order. It contains the following fields:

- **amount**: The number of tokens bought or sold.
- **price**: The price at which the tokens are bought or sold.
- **timestamp**: When the order was created.
- **trader**: The trader's address who placed the order.
- **status**: The status of the order, represented as a two-byte value (e.g., "A" for active, "T" for traded).

buys and sells Arrays

The `buys` and `sells` arrays are used to store the buy and sell orders, respectively. Each array stores instances of the `Order` struct, representing the active orders in the order book.

erc20Base and erc20Counter Contracts

The `erc20Base` and `erc20Counter` contracts represent the ERC20 token being traded in the order book. They are instances of the ERC20 contract from the OpenZeppelin library [2]. The `erc20Base` contract represents the base asset while the `erc20Counter` contract represents the counter asset. Our order book smart contract is designed for only two tokens at a time, meaning that the base and counter assets are fixed. In our use case, the base asset is the `UEToken` ERC20 token while the counter asset is the `USDC` ERC20 token.

Constructor

The constructor of the order book smart contract is responsible for initializing the `erc20Base` and `erc20Counter` contracts. It takes as parameters the addresses of the two contracts and initializes the two contracts, and retrieves the decimals places for each token. The decimal places convert the tokens from the base unit to the token unit. For example, the `UEToken` ERC20 token has two decimals places, meaning that 1 UEToken is equal to 100 base units (refer to Equation **??** for further clarification).

Events

The contract defines three events:

- **BuyAdded**: Triggered when a buy order is added to the order book. It includes the order number, amount, price, and trader's address.
- **SellAdded**: Triggered when a sell order is added to the order book. It includes the order number, amount, price, and trader's address.
- **TradeAdd**: Triggered when a trade is executed between a buy and sell order. It includes the order number, amount, price, maker's address, and taker's address.

addBuy Function

The `addBuy` function allows a trader to add a buy order to the order book. It takes as parameters the amount and the desired buy price. The function transfers the total amount of tokens to the order from the trader's account to the order book contract using the `transferFrom` method of the `erc20Base` contract. It then adds the order to the `buys` array and emits the `BuyAdded` event.

addSell Function

The `addSell` function allows a trader to add a sell order to the order book. It takes as parameters the amount and the desired sell price. The function transfers the total amount of tokens to the order from the trader's account to the order book contract using the `transferFrom` method of the `erc20Counter` contract. It then adds the order to the `sells` array and emits the `SellAdded` event.

getLengthBuy and getLengthSell Functions

The `getLengthBuy` and `getLengthSell` functions return the length of the `buys` and `sells` arrays, respectively.

getBuy and getSell Functions

The `getBuy` and `getSell` functions allow users to view the details of a specific buy or sell order. They take the order number as a parameter and return the amount, price, timestamp, trader's address, and order status.

trade Function

The `trade` function executes a trade between a buy and sell order. It takes as parameters the order number, amount, trade price, and trade type (e.g., "B" for buy, "S" for sell). The function verifies the order number and trade parameters and then applies the trade whether the trade is valid or not. If the trade is valid, the function transfers the tokens between the trader and the counterparty using the `transfer` and `transferFrom` methods of the `erc20Base` and `erc20Counter` contracts, respectively. It updates the status of the order, emits the `TradeAdd` event, and returns the order number, amount, and trader's address of the trade.

`transferForm` is used when transferring tokens from the `msg.sender` (i.e., the caller of the function) to the counterparty of the trade, which can be either the buyer or seller, depending on the trade type. `transferForm` is used when an external account initiates the token transfer and requires approval from the account holder (i.e., the `msg.sender`). In this case, the `msg.sender` needs to approve the Orderbook contract to spend a certain amount of tokens on the token contract before initiating the `trade` function. Once the approval is granted, the Orderbook contract can execute the `transferFrom` function method to transfer the tokens from the `msg.sender`'s account to the counterparty's account. This ensures that the token transfer is authorized and controlled by the account holder, preventing the Orderbook contract from transferring tokens without the account holder's permission.

`transfer` is used when transferring tokens from the Orderbook contract to the `msg.sender`, which can be either the buyer or seller receiving their respective tokens after successfully executing the trade. In this case, the Orderbook contract acts as the account holder and can transfer tokens to the `msg.sender` without requiring any approval. Since the Orderbook contract holds the tokens in its balance, it can use the `transfer` method to send the tokens to the respective party.

decommission Function

The `decommission` function allows the contract owner to decommission the order book smart contract. It transfers any remaining tokens in the active orders back to the respective traders using the `transfer` method of the `erc20Base` and `erc20Counter` contracts. It then clears the `buys` and `sells` arrays, resetting the order book to its initial state. Only the contract owner can call this function with the `onlyOwner` modifier.

decommissionUser Function

The `decommissionUser` function allows a trader to decommission all of his or her active orders in the order book. First, check if the trader has active orders in the order book. If the trader has active orders, the function transfers any remaining tokens in the active orders back to the respective trader using the `transfer` method of the `erc20Base` and `erc20Counter` contracts. It then clears the transferred orders from the `buys` and `sells` arrays. It is helpful if a trader wants to decommission all active orders in the order book without waiting to validate the order book contract owner.

The order book smart contract provides a reliable and efficient way to trade ERC20 tokens, ensuring that the trades are executed fairly and transparently without the need for a trusted third party.

1.2.2 Migration of the order book smart contract

To deploy and migrate the Orderbook smart contract to a local blockchain, we use Truffle's migration feature, as explained in Section 1.1.2. Before deploying the Orderbook smart contract, we need to deploy the `UEToken` and `USDC` ERC20 tokens to the local blockchain, as explained in Section 1.1.2. We need this to have our local blockchain, such as Ganache, have the tokens for trading. Once the tokens are deployed, we can deploy the Orderbook smart contract using the following steps.

Deployment Script

In the *migrations* folder, we create a new file called *1_initial_migration.js*. This file is responsible for deploying smart token contracts. Then we create a new file called *2_deploy_orderbook.js* to deploy the Orderbook smart contract in the same folder to handle the deployment of the Orderbook smart contract.

Migration scripts

In the deployment script, we first import the `UEToken` and `USDC` smart contracts artifacts from the *build/contracts* folder. These contracts are generated by Truffle when compiling the smart contracts. Then we need to also get their respective addresses from the summary of the deployment of the tokens. The development script is shown in Listing 1.9.

```
const Orderbook = artifacts.require('Orderbook');

module.exports = function (_deployer) {
  const baseTokenAddress = '0x...'; // Address of the base token ERC20 contract
  const counterTokenAddress = '0x...'; // Address of the counter token ERC20 contract
  _deployer.deploy(Orderbook, baseTokenAddress, counterTokenAddress);
};
```

Listing 1.9: Orderbook migration script for development.

Deployment

Before deploying the Orderbook smart contract, we must ensure that the network configuration in the *truffle-config.js* file is set to the local blockchain network. Then we can run the migration command to display the Orderbook contract. In the command line, we run the following command `truffle migrate --reset`,

Deployment on the Rinkeby testnet

To deploy the Orderbook smart contract on the Rinkeby testnet, we need to ensure that the network configuration in the *truffle-config.js* file is set to the Rinkeby testnet. Then we can run the migration command to display the Orderbook contract. In the command line, we run the following command `truffle migrate --reset --network rinkeby`.

Verify the deployment

After successfully migrating the Orderbook smart contract, Truffle generates a summary of the deployment, which includes the addresses of the deployed contracts on the local blockchain.

Interacting with the order book smart contract

We can use the Truffle console to interact with the Orderbook smart contract. In the command line, we run the following command `truffle console`. This will open the Truffle console, which allows us to interact with the deployed smart contracts. We can then use the `Orderbook.deployed()` command to get the deployed Orderbook smart contract instance. We can then use the instance to call the functions of the Orderbook smart contract. For example, we can call the `addBuy` function to place a buy order in the order book. The following code snippet in Listing 1.10 shows how to place a buy order in

```
const orderbook = await Orderbook.deployed();
const amountUEToken = web3.utils.toWei('100', 'ether');
const amountBuyPrice = web3.utils.toWei('0.0001', 'ether');
await orderbook.addBuy(amountUEToken, amountBuyPrice);
const buy = await orderbook.getBuy(orderbook.getBuyLength() - 1);
console.log(buy);
```

Listing 1.10: Placing a buy order in the order book with Truffle console.

1.2.3 Implementation of the trading platform

The same as in Section 1.1.3, we use React to implement the trading platform. In this section, we explain the main components of the trading platform and how they interact with the Orderbook smart contract without explaining the implementation details. The main components of the trading platform are the following:

Event listener

The order book is displayed in a table with two columns, one for the buy orders and one for the sell orders. Each row of the table represents an order in the order book. The order book is updated in real-time using the `getSell` and `getBuy` functions of the Orderbook smart contract. When populating the order `buys` and `sells` arrays,

an event is emitted by the Orderbook smart contract. We can listen to these events in our trading platform to update the order book in real-time with new orders. The following code snippet shows how to listen to, e.g., the `BuyOrder` event emitted by the Orderbook smart contract can be caught in the trading platform.

```
watchOrderbook = async () => {
    await this.watchweb3.Contract(this.orderbookABI, orderbook.address).then((contract) =>
    ↪ {
        // Update the order book with the new buy ordersu
    });
}
```

Listing 1.11: Listening to the Orderbook smart contract events in the trading platform.

In Listing 1.11, we use the `watchweb3` library to listen to the events emitted by the Orderbook smart contract. Then, when a new event is emitted, we update the order book with the new orders and refresh the table to display the new orders.

Buy and sell orders

The following method in our trading platform is the ability to place buy and sell orders. To place a buy order, call the `addBuy` function of the Orderbook smart contract. The following code snippet shows how to place a buy order in the order book.

```
addBuy = async () => {
    const app = this;
    const usdcAmount = this.state.fields.buyAmount
    const amountBuyPrice = this.state.fields.buyPrice;
    const approvalAmount = usdcAmount * amountBuyPrice;
    const usdcContract = await this.web3.eth.Contract(app.USDC.abi, app.USDC.address);
    const uetContract = await this.web3.eth.Contract(app.UET.abi, app.UET.address);
    await this.web3.eth.Contract(app.orderbookABI.abi,
    ↪  app.orderbookABI.address).then(async (contract) => {
        // check if the sells array is empty
        if(this.state.sells.length === 0)
        {
            await usdcContract.methods.approve(app.orderbookABI.address, approvalAmount*
            ↪  10 ** 2).send({ from: app.web3.eth.defaultAccount }).then(async (receipt)
            ↪  => {
                await contract.methods.addBuy(usdcAmount, amountBuyPrice).send({ from:
                ↪  app.web3.eth.defaultAccount }).then(async (receipt) => {
                    // Update the order book with the new buy orders
                });
            });
        } else {
            // Look for a matching sell order
            // Execute the trade
        }
    });
}
```

Listing 1.12: Placing a buy order in the order book from the trading platform.

In Listing 1.12, we start by getting the amount of UEToken and the buy price from the state of the trading platform. Then we call the `addBuy` function of the Orderbook smart contract with the amount of UEToken and the buy price as parameters. We also specify the account that is placing the buy order. Then, we check if the `sells` array is empty, meaning there are no sell orders in the order book. If the `sells` is empty, then we skip the next step, looking for a matching sell order. We look for a matching sell order if the `sells` array is not empty. If we find a matching sell order, then we execute the trade.

For the empty, we first use the `approve` function of the USDC smart contract to approve the Orderbook smart contract to transfer the amount of USDC specified in the buy order that is worth the same as the order amount from the user's account. The approval amount is multiplied by the decimals of the USDC, which is 2. Then, to receive the successful transaction receipt, we call the `addBuy` function of the Orderbook smart contract to place the buy order in the order book. Finally, with our `watchOrderbook` method, we update the order book with the new buy orders.

Matching buy and sell orders

If the `sells` array is not empty, then we need to look for a matching sell order. To do so, we loop through the `sells` array and check if the amount of UEToken of the sell order is greater than or equal to the amount of UEToken of the buy order. If the amount of UEToken of the sell order is greater than or equal to the amount of UEToken of the buy order, then we execute the trade. The following code snippet shows how to execute the trade.

```
1  // approve the Orderbook smart contract to transfer the amount of UEToken specified in the
   ↪  buy order from user's account
2  await uetContract.methods.approve(app.orderbookABI.address, amountUEToken * 10 **
   ↪  18).send({ from: app.web3.eth.defaultAccount }).then(async (receipt) => {
3      // look for a matching sell order
4      while ( i < app.state.sells.length && amountUEToken > 0) {
5          let price = app.state.sells[i].price;
6          let sellAmount = app.state.sells[i].amount;
7          if (sellAmount >= amountUEToken) {
8              await contract.methods.trade(app.state.sells[i].orderNo, amountUEToken, price,
                   ↪  1).send({ from: app.web3.eth.defaultAccount }).then(async (receipt) => {
9                  // Update the order book with the new buy orders
10             });
11             amountUEToken = 0;
12         } else {
13             await contract.methods.trade(app.state.sells[i].orderNo, sellAmount, price,
                   ↪  1).send({ from: app.web3.eth.defaultAccount }).then(async (receipt) => {
14                 // Update the order book with the new buy orders
15             });
16             amountUEToken -= sellAmount;
17         }
18         i++;
19     }
20 });
```

Listing 1.13: Executing a trade from the trading platform.

In listing 1.13, in each iteration of the while loop, we check if the amount of UEToken of the buy order can fill the amount of UEToken of the sell order. If the amount of UEToken of the sell order can be filled by the amount of UEToken of the

buy order, then we execute the trade by calling the `trade` function of the Orderbook smart contract. Otherwise, we partially fill the sell order by calling the `trade` the function of the Orderbook smart contract until the amount of UEToken of the buy order is filled, then we stop the while loop.

Here, we describe adding a new buy order to the order book. The process of adding a new sell order to the order book is similar to the process of adding a new buy order to the order book. The only difference is that we call the `addSell` function of the Orderbook smart contract instead of the `addBuy` function of the Orderbook smart contract. We change all the `buy` keywords to `sell` keywords in the code snippet in Listing 1.12 and Listing 1.13.

1.2.4 Scenario

In this section, we describe a scenario of how the trading platform interacts with the Orderbook smart contract. The scenario is illustrated in Figure 1.5. The sequence diagram shows the interaction between the Buyer (Alice), the Seller (Bob), the Platform, and the Orderbook smart contract while placing a buy order and executing a trade. The following steps describe the scenario as follows:

1. Alice, the buyer, initiates a buy order by calling the `addBuy` function on the platform. She specifies the amount of 100 UET and the buy price of 10 USDC per UEToken.

2. The platform receives the buy order from Alice and forwards the buy order to the Orderbook smart contract by calling the `approve` function of the USDC smart contract to approve the Orderbook smart contract to transfer the amount of 1000 USDC from Alice's account. This approval ensures that the Orderbook can transfer the tokens on behalf of Alice when a matching sell order is found. Once the approval is successful, the platform calls the `addBuy` function of the Orderbook smart contract to place the buy order in the order book.

3. The Orderbook smart contract adds the buy order to its buys list. It updates the internal state and emits the 'BuyAdded' event with the order details (order number, amount, price, and trader).

4. The Platform verifies if there are any matching sell orders in the order book. In this case, there are no immediate matches.

5. The platform calls the `watchOrderbook` function of the Orderbook smart contract to update the order book with the new buy order. It receives the event `BuyAdded` from the Orderbook smart contract and updates the order book with the new buy order to display the order book to Alice and Bob.

6. Bob, the seller, initiates a sell order by calling the `addSell` function on the platform. He specifies the amount of 100 UET and the selling price of 10 USDC per UEToken.

7. The platform receives the sell order from Bob and forwards the sell order to the Orderbook smart contract by calling the `approve` function of the UEToken smart contract to approve the Orderbook smart contract to transfer the amount of 100 UET from Bob's account. This approval ensures that the Orderbook can transfer the tokens on behalf of Bob when a matching buy order is found.

8. Once the approval is successful, the platform calls the `addSell` function of the Orderbook smart contract to place the sell order in the order book. It provides the amount of 100 UET and the selling price of 10 USDC per UEToken.

9. The Orderbook smart contract adds the sell order to its sales list. It updates the internal state and emits the `SellAdded` event with the order details (order number, amount, price, and trader).

10. The Platform checks for matching buy orders in the order book. In this case, there is a match with Alice's buy order.

11. The Platform initiates the trade by calling the `trade` function on the Orderbook smart contract. It provides the necessary parameters (order number, amount, price, and trade type).

12. The Orderbook smart contract verifies the trade conditions and executes the trade. It transfers the required tokens between the Buyer (Alice) and the Seller (Bob) using the `transferFrom` and `transfer` functions.

13. The Orderbook emits the `TradeAdd` event with the trade details (order number, amount, price, maker, and taker).

14. The order book updates its internal buys and sells lists by removing the executed orders.

15. The Platform refreshes the buys and sells lists for Alice and Bob.

The sequence diagram illustrates the flow of interactions between the different actors and the Orderbook smart contract. It shows the process of adding orders, verifying matches, executing trades, and updating the order lists.

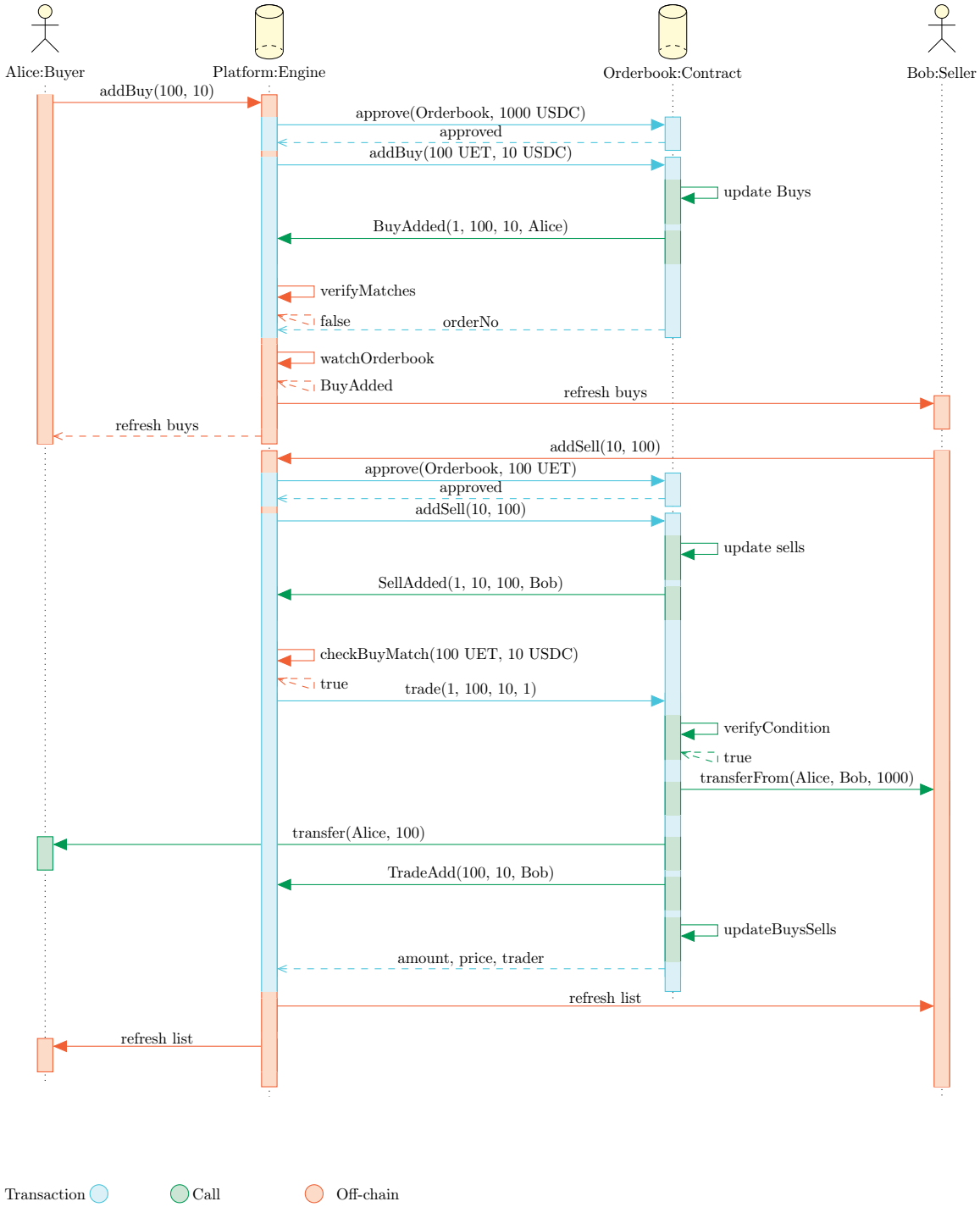

Figure 1.5: Sequence diagram of the Orderbook smart contract.

List of Figures

List of Tables

List of sources

A | Code of the UEToken smart contract

References

[1] Fabian Vogelsteller and Vitalik Buterin. *ERC-20 Token Standard*. Nov. 2015. URL: https://eips.ethereum.org/EIPS/eip-20 (visited on 05/23/2023).

[2] OpenZeppelin. *ERC20*. Version 4.x. 2023. URL: https://docs.openzeppelin.com/contracts/4.x/erc20 (visited on 05/23/2023).

[3] Truffle Suite. *Truffle Suite*. 2023. URL: https://www.trufflesuite.com/ (visited on 05/23/2023).

[4] Facebook. *React*. 2023. URL: https://reactjs.org/ (visited on 05/23/2023).

[5] Truffle Suite. *Ganache CLI*. https://github.com/trufflesuite/ganache-cli-archive. 2023. (Visited on 05/23/2023).

[6] Ethereum Foundation. *Solidity Documentation*. 2019. URL: https://docs.soliditylang.org/en/v0.8.20/ (visited on 05/23/2023).

[7] OpenZeppelin. *Github Repository of OpenZeppelin Contracts*. Version 4.x. 2023. URL: https://github.com/OpenZeppelin/openzeppelin-contracts/tree/master/contracts (visited on 05/23/2023).

[8] Metamask. *Metamask*. 2023. URL: https://metamask.io/ (visited on 05/23/2023).

[9] Metamask Community. *Not able to import custom tokens on MM from Ganache*. 2023. URL: https://community.metamask.io/t/not-able-to-import-custom-tokens-on-mm-from-ganache/22939 (visited on 05/23/2023).

[10] Ethereum Foundation. *Web3.js*. 2016. URL: https://web3js.readthedocs.io/en/v1.5.2/ (visited on 05/23/2023).

[11] Centre Consortium. *USD Coin*. 2023. URL: https://www.centre.io/usdc (visited on 05/23/2023).