Hochschule Konstanz
Department of Computer Science

Submitted by
Thanh Son Dang
Student Number 297180

th851dan@htwg-konstanz.de

Bachelor Thesis

Creation of an ERC-20 Token for backing Play2Earn concept in practice

Konstanz, 09.02.2022

Hochschule Konstanz
Department of Computer Science

Bachelor Thesis

Creation of an ERC-20 Token for backing Play2Earn concept in practice

by

Thanh Son Dang

in Partial Fulfillment of the Requirements for the Degree of

Bachelor of Science
in Applied Computer Science

at the Hochschule Konstanz University of Applied Sciences,

Student Number: 297180

Date of Submission: 09.02.2022

Supervisor: **Prof. Dr. Marko Boger**
Second Examiner: **BsC. Pascal Pichler**

An electronic version of this thesis is available at http://repository.tudelft.nl/.

Abstract

Promising people an incentive can be seen as one of the most efficient way to win their desire to use a product. Most likely it will be utilized in the shape of a cashback or point collecting system. One way or the other, there is no guarantee that this kind of incentive belongs to users once earned, as it is still always under the provider's control and can be reclaimed in the provider's interest at anytime. Blockchain with its nature, the future carrying technology at the moment, appeared to strengthen ownership of users by removing all kind of trusted third parties.

This project describes the creation of a cryptocurrency following the ERC-20 Token Standard, which can be addressed as **Squib Token**. **Squib Token** is able to be issued in a blockchain network running Ethereum Virtual Machine, where it can be manipulated at the free will of token's owner under no government's control, or, in other words, under the observation of everyone in the network.

Contents

List of Figures

List of Tables

Listings

1

Introduction

Motivated and supported by Smart-up[1], the support program for innovation, entrepreneurship and self-employment at the Lucerne University of Applied Sciences and Arts, Squib Ltd. was founded in 2020 with the purpose of providing a state-of-the-art feedback tool for companies and individuals. As simple as uploading two or more pictures and a related inquiry, Squib's users acquire the opportunity to reach their friends, customers, or their followers right where they are.

1.1. Current Situation

In order to incentivise users to contribute by voting and leaving feedbacks, a concept of Gamification "Vote-to-Earn", based on the Business Model "Play-to-Earn", is used. More precisely, users are able to benefit from a considerable added value by participating in "Voting" activities. This somewhat utilises the "Reward System", which causes users "want" to vote more, independently to what extend they "like" to vote[BK13]. The "Voting" activity thereby becomes a desirable and attractive goal, which is converted from a mere visual experience into something that commands attention so that it will be sought out.[Ber12]. It is necessary to mention that in the early phase, only the "Sponsored Votings", which are created by the corporate enterprises, are profitable.

[1] Visit the Smart-up site https://www.hslu.ch/en/lucerne-university-of-applied-sciences-and-arts/about-us/smart-up/.

1.2. Vision

The aforementioned “Play-to-Earn“ concept is preferably deployed in the decentralized blockchain network, where the in-game earned assets in any form can be transferred to others in the real world as a valuable resource[Coi21]. To enable that ability for the users, an approach to the decentralized world is indispensable. Taking that into consideration, the aim is to create a new kind of token, called **Squib Token**, which follows the ERC-20 standard. ERC-20 tokens are blockchain-based assets that have values and can be sent and received, which are similar, in some respects, to Bitcoin, Litecoin, and any other cryptocurrency[Nat21]. In addition to the essential functions of a ERC-20 Token, **Squib Token** is enhanced by several modifications to increase flexibilities such as ***role based access control mechanism*** and ***redeeming***.

Besides, the implemention of the following user interfaces is necessary for exposing the preceding modifications:

- **Admin Interface:** Basically accessible for everyone. However, only certain roles are able to perform the administrative jobs such as *minting*, *granting role*.
- **User Wallet:** Containing basic functions for all users: *displaying token balance*, *transfering* or *redeeming* when a certain amount of tokens is reached.

2

Theorectical Background

This chapter presents the theoretical background for the thesis in detail.

In section 2.1, the blockchain, known as a system of recording information in a way that makes it difficult or impossible to hack or alter the system, which is one of the up-and-coming technologies at the moment, is explained intensively.

Section 2.2 focuses on the Ethereum blockchain and its native cryptocurrency Ether, which is a platform exploited blockchain technology that beyonds exchanging cryptocurrency between accounts in the manner of Bitcoin.

At last, the ERC-20 token standard is decoded in section 2.3, broadening the way for the implementation of the Squib Token in the next chapters.

2.1. Blockchain Technology

The first widespread application of blockchain was brought to life by Satoshi Nakamoto, the unidentified creator of Bitcoin(which can be a person or a group of people)[LS]. According to the whitepaper published in 2008, Bitcoin is ”a purely peer-to-peer version of electronic cash”[Nak08]. In order to work as cash, Bitcoin must have the capability of changing hands without being redirected to the wrong account and preventing double-spending(the same coin being spent twice by the same person)[15]. Before Bitcoin, there was no choice other than relying on financial institutions such as banks serving as trusted third parties which stand behind electronic payments. To achieve the outcome of a decentralized system and replace theses trusted third parties, Satoshi Nakamoto made use of blockchain technology.

2.1.1. Definition

Blockchain is designed as a shared database to stores information electronically in digital format, which is publicly available and distributed over a computer network, where each computer is called a node of that network[Ada22]. To differentiate from a typical database, which stores its data into tables, blockchain structures its data in groups, known as blocks that are strung together. Blocks have certain storage capacities and when one is filled, it is closed and linked to the previously filled block, forming a chain of data, hence ”blockchain”[Ada22]. As new data comes in, it is compiled into a newly formed block and once filled, it is also chained onto the previous block[Ada22]. The exact time stamp, when a filled block is added to the chain, is recorded to form a chronological order[Ada22]. Blocks are always added to the ”end” of the chain and contains a cryptographic hash of the previous block, alongside with the previously mentioned time stamp and its transaction data[Ada22]. That cryptographic hash is created by a mathematical function using this information that turns it into a completely random string and once that information is altered in any way, the hash code changes entirely as well[Ada22]. Therefore, blockchains are invulnerable to modification of their data, because the data of any given block cannot be adjusted without alternating all subsequent blocks.

2.1.2. Features

Decentralization

Blockchain allows its data to store across its peer-to-peer network around the world. In this way it can be used as a distributed ledger, which is replicated on several network nodes at various locations and no node is trusted more than any other[15]. This solves the problem of single point of failure, says when a node goes down because of being attacked or lost of electricity, the data is still recorded in many other locations and therefore able to be replicated persistently. Besides, blockchain doesn't require any governing authority looking after the system, only the nodes maintain the network and making it decentralized[Gwy21a]. If a record at one node of the database is tampered with, all other nodes, which would not be altered, will cross-reference each other to track down the node being attacked[Ada22]. That makes the information stored on blockchain irreversible. This information might not only be transaction history(in the case of cryptocurrencies), but also a variety of other information like important documents, contracts or state identifications.

Immutability

Since every node on the system has a copy of the digital ledger, each of them also needs to check the validity of the new entry or record. ”A majority of the decentralized network’s computing power would need to agree on it”[Ada22], so the new data is validated to the block. One key fact is that, blocks always get added to the ”end” of the ledger. Hence, an arbitrary block can not be edited, removed or updated by any user on the network without entirely modifying the subsequent blocks, violating the system and hence being caught easily.

Security

As mentioned before, every block contains its own hash implied by cryptographic hash function such as SHA-256 using the following components: the time stamp when it is added to the chain, the information and the hash of the previous block. In that way, blocks are always stored chronologically and new blocks are always added to the ”end” of the blockchain. Remarkably, a tiny change in the data input for the hash function will result in a completely different output. To succeed a hack into the blockchain system, the attacker must posses 51% the computational power of the network in order to make his copy the majority copy and become the agreed-upon chain[Ada22]. An enormous amount of money and resources is required for such an attack, which would not happen unnoticeably because the altered blocks have different time stamps and hash codes, drawing attention from the network members for this forceful alteration. They would then ”hard fork off to a new version of the chain that has not been affected”[Ada22], making the attacked version worthless. The more nodes participating in the system, the more secure the blockchain is.

Consensus

For the blockchain to work robustly against double spending where no trusted third parties required, this distributed ledger apparently needs a ***consensus algorithm*** that all nodes can rely on. Basically, a ***consensus algorithm*** makes the decisions whether a transaction is kept to be shared across the network or discarded to avoid confusions, ensuring the system can stand all kinds of attacks or failures when there is more than one node involved[LT21a].

Nowadays, there are two main consensus algorithms considered by its wide range of utilization:

- **Proof-of-Work:** A new transaction block is generated by resolving a computationally expensive problem, ”guaranteeing only blocks with a valid proof of resolution to the problem - proof-of-work - are accepted as valid blocks; blocks without a valid Proof-of-Work are rejected by the network nodes”[LT21b]. This is the case

of the most popular blockchain at this time: Bitcoin. Note that the mechanism was designed so that finding a solution is very expensive in terms of money and time while verifying the solution is a trivial and deterministic process[LT21b].

- **Proof-of-Stake:** Rather than computing a randomly complex problem, with Proof-of-Stake each user can directly vote to validate the next block with a digital signature, as long as the user staked a certain amount of coins and officially participating in the voting process[LT21c]. "The voting influence is proportional to the amount of staked coins — larger financial commitments to the voting process leading to a larger influence."[LT21c]

2.1.3. Types of blockchain

There are two main types of blockchain distinguished in terms of access control - who can participate in the blockchain, submit the transactions to it or play a role in the voting process:

- **Public blockchains:** Anyone can join public blockchains. The network remains permissionless and users can anonymously submit transactions to it. To help secure the network, the users are incentivized to join the system, being rewarded if they involve in the consensus process, says solving a computationally intensive problem in Proof-of-Work or staking and voting in Proof-of-Stake mechanism[WHB18].
- **Private blockchain:** This kind of blockchain offers a private network and users have to be invited and approved to join this system. This somewhat causes the blockchain less decentralized. In spite of that, it stills leverages the aforementioned features of blockchain, encouraging organizations and enterprises, where information should stay privately, to embrace blockchain[Gwy21b].

The purpose of this bachelor project is to create a token for everyone across the network to use as a cryptocurrency. Therefore, only public blockchains are relevant.

2.2. Ethereum

"What Ethereum intends to provide is a blockchain with a built-in fully fledged Turing-complete programming language that can be used to create 'contracts' that can be used to encode arbitrary state transition functions, allowing users to create any of

> the systems described above, as well as many others that we have not yet imagined, simply by writing up the logic in a few lines of code."
>
> *Ethereum whitepaper*[But14]

2.2.1. Introduction

Whereas Bitcoin can be thought of a state transition system, as it has a "state" consisting of the ownership of all existing bitcoins, focusing only on regulating money flows, with Ethereum users are able to deploy arbitrary code on a decentralized compute platform[But14]. That deployed logic is called "Smart Contract", which fundamentally remains unchangeable on the Ethereum Virtual Machine(EVM) and automatically executes when needed, enforcing payment agreements between parties[Dan17a]. The Ethereum's native cryptocurrency, ETH, merely acts as a token financing the transactions executed on the network, which may be transferring tokens from one account to the other, deploying a bunch of codes, or calling a state-changing function in a existing smart contract[LT21d].

2.2.2. Ethereum account types

There are two types of accounts in Ethereum:

Externally owned accounts(EOAs) [Coma]
An externally controlled account

- has an ether balance,
- can send transactions (ether transfer or trigger contract code),
- is controlled by private keys,
- has no associated code.

Contract accounts [Coma]
A contract

- has an ether balance,
- has associated code,
- code execution is triggered by transactions or messages (calls) received from other contracts.

- when executed - perform operations of arbitrary complexity (Turing completeness) - manipulate its own persistent storage, i.e., can have its own permanent state - can call other contracts.

An EOA with no associated code is controlled by a private key, can only send and receive ETH, trigger actions on smart contracts[LT21e].

A Contract Account has its behaviour and its ETH holdings controlled by code, can also store ETH and trigger actions on other smart contracts[LT21e].

2.2.3. Ethereum Virtual Machine

The Ethereum Virtual Machine is basically a virtual machine running in Ethereum environment, except it runs on every Ethereum node in order to maintain consensus across the blockchain[Dan17b]. This does not provide horizontal scaling at the moment, which is the acceleration to solve an overall problem by sharding the computational efforts across distributed nodes[LT21f]. On the contrary, Ethereum's goal is having all node execute the same computation. In spite of causing a lot of limitations, that is vital to achieve a high level of trust, enabling the system to work without any trusted third parties. Ethereum is "Turing complete" so developers can create applications and freely decide what it should be used for under the consideration that some types of applications benefit more than others[Comm].

2.2.4. Consensus

Like Bitcoin, Ethereum currently uses **Proof-of-Work** consensus protocol. However, because of some critical downsides such as consuming an important amount of energy as the network growing or producing blocks at a slow rate, Ethereum has plans to fully transition to **Proof-of-Stake** mechanism[LT21g]. This will start "the era of more sustainable, eco-friendly Ethereum"[Comj].

2.2.5. Smart Contracts

A "smart contract" is naturally a unit of functionality that runs on Ethereum blockchain. The term *dapp*, or *distributed application* expresses a web- or smartphone-accessible front end that uses the EVM as its back end[Dan17c]. The functions and states of a smart contract reside at a specific address on the Ethereum blockchain. Being a type of Ethereum account, Smart Contracts can hold a balance and send transac-

tions. In contrast to EOAs being controlled by users, Smart Contracts are deployed programmatically. They define rules and and automatically enforce them via code. A simple metaphor to Smart Contracts could be a digital vending machine: The right inputs guarantees certain outputs, but with the differences: Smart Contracts, once deployed to Ethereum blockchain, cannot be deleted, and interactions with them are irreversible[Comh].

Ethereum allows its users to write and deploy Smart Contracts to the network. Solidity and Vyper are the most popular smart contract languages. To deploy the Smart Contracts, they must be compiled and their owner must pay Gas in the same way of paying gas for a ETH transfer, because it's technically a transaction[Comh].

Smart Contracts's functions can be called by other Smart Contracts to greatly extend the usabilities, because they are all public on Ethereum blockchain and can be thought of as open APIs[Comh].

2.2.6. Gas

As simple as cars need gasoline to run, specific operations on Ethereum network also need gas to be executed. Gas is the unit that measures the amount of computational effort required to conduct a transaction on Ethereum successfully[Comf]. Gas fees are paid in ETH. The prices are denoted to gwei, meaning "giga-wei", which is equal to 1,000,000,000(10^9) wei. Wei is the smallest unit of ETH.

$$1\ ETH = 1,000,000,000\ gwei = 10^9\ gwei = 10^{18}\ wei$$

Gas fees help secure the Ethereum network by requiring a fees for every computational efforts requested by the network, avoiding spamming from bad actors. It is compulsory for every transaction to set a gas limit for the prevention of accidentally infinite loops or unwanted wastage in code[Comf]. Despite of that, the unused gas after fulfilling a transaction is returned to the user. It is preferable to set the gas limit generously for the transaction to be mined.

For example, a transaction with gas limit of 50,000 requires 30,000 gas to execute, 20,000 gas will be returned. However, if the gas limit is set to 20,000, the EVM will attempt to fulfill the transaction and consume the 20000 gas units, but the transaction is not complete. The EVM then reverts any changes and the 20,000 gas is not returned, as it is consumed for the computational work made by the miner.

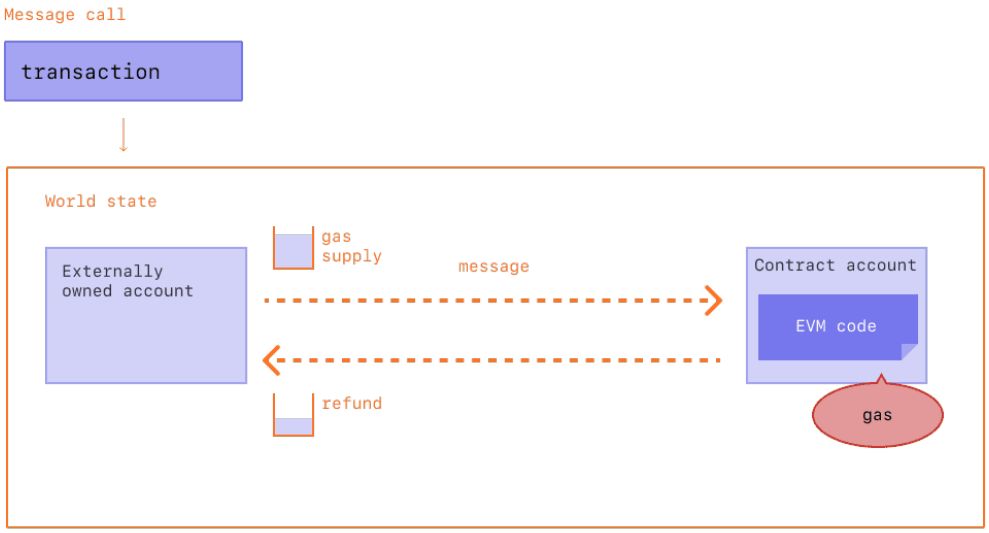


Figure 2.1: Gas and fee [Tak18]

2.3. ERC-20 Token Standard

Token is ”blockchain’s most powerful and most misunderstood tool”[Opeu].

Token represents *something* in the blockchain. This *something* can be anything from money, services, a song, a virtual artwork, shares in a company or even real-world, tangible objects. When things are represented by tokens, it enables the Smart Contracts to interact with them, exchange, create or destroy them[Opeu].

2.3.1. Different Kinds of Tokens

Depending on the fungibility of goods, whether they are equal to all others, for instance votes or fiat currencies, or they have differences in prices such as houses or services, tokens can be distinguished into two kinds[Opeu]:

- **Fungible tokens:** The goods that are identical and interchangeable. Any two tokens represent the same value. When dealing with them, the amount of tokens matters.

- **Non-fungible tokens:** The goods that are unique and in contrast to fungible tokens, they have their own value.

2.3.2. Standards

In the public world of Ethereum blockchain, there are no rules about what Smart Contracts have to do, the Ethereum community has developed many standards to define how a contract can interoperate with other contracts across complex implementation, ensuring them remain composable[Comc].

Token interfaces play an important role in the Ethereum development standards. A list of the most popular token standards on Ethereum can be found here [Comk]:

- **ERC-20** - A standard interface for fungible (interchangeable) tokens, like voting tokens, staking tokens or virtual currencies.
- **ERC-721** - A standard interface for non-fungible tokens, like a deed for artwork or a song.
- **ERC-777** - ERC-777 allows people to build extra functionality on top of tokens such as a mixer contract for improved transaction privacy or an emergency recover function.
- **ERC-1155** - ERC-1155 allows for more efficient trades and bundling of transactions – thus saving costs.

2.3.3. ERC-20 Token Standard

ERC-20 is one of the most significant smart contract standards on Ethereum, proposed by Fabian Vogelsteller in November 2015, is the norm for fungible token implementations on Ethereum blockchain[Comb].

A ERC-20 implementation must provide following functionalities[Comb]:

- transfer tokens from one account to another
- get the current token balance of an account
- get the total supply of the token available on the network
- approve whether an amount of token from an account can be spent by a third-party account

To fulfill those features, six mandatory public functions must be implemented in a ERC-20 contract[VB15]:

- `totalSupply(), returns(uint256)`: returns the total token supply.

- `balanceOf(address __owner), returns(uint256 balance)`: returns the account balance of another account with address `__owner`.
- `transfer(address __to, uint256 __value)`: Transfers `__value` amount of tokens to address `__to`.
- `transferFrom(address __from, address __to, uint256 __value)`: transfers __value amount of tokens from address __from to address `__to`.
- `approve(address __spender, uint256 __value)`: allows `__spender` to withdraw from caller's account multiple times, up to the `__value` amount. If this function is called again it overwrites the current allowance with `__value`.
- `allowance(address __owner, address __spender)`: returns the amount which `__spender` is still allowed to withdraw from `__owner`.

3

Development Environment

In this chapter, a brief introduction of the technologies used in this project is demonstrated. This contains:

- Solidity - The programming language of Smart Contracts.
- Truffle - The development environment, testing framework for Smart Contracts.
- Ganache - The one click blockchain.
- OpenZeppelin - Library for secure smart contract development[1].
- Metamask - The bridge.
- useDApp - Framework for rapid *dapp* development.

3.1. Solidity

Invented for addressing the implementation of Smart Contracts, Solidity appeared as an object-oriented, high-level programming language, designed to target the Ethereum Virtual Machine[Comi]. As a relatively young language, initially proposed by Polkadot[2] founder Gavin Wood, Solidity has been developed and improved by Ethereum Community at a rapid speed. Being a curly-bracket language that has been influenced by several well-known programming languages such as C++, Javascript, Python[Comg],

[1] https://docs.openzeppelin.com/contracts/4.x/
[2] Check out the Polkadot blockchain https://polkadot.network/about/

"Solidity is statically typed, supports inheritance, libraries and complex user-defined types among other features"[Comi]. Following is a simple "Hello World" smart contract:

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.10;
3
4 contract HelloWorld {
5     string public greet = "Hello World!";
6 }
```

Listing 3.1: `HelloWorld` smart contract

An extremely important keynote must be taken in consideration, that due to the immutable nature of blockchain, once a contract is deployed, it is principally unmodifiable and lives forever in that blockchain's lifetime, meaning every line of code written in Solidity has to be flawless. Otherwise, it would be put at risk of being hacked or attacked by every kind of exploitation.

This project works with the lastest version of Solidity 0.8.11[3].

3.2. Truffle

In this work, Truffle is used as a full package for compiling, linking and deploying for Smart Contracts. This framework is open source and can be easily installed using `npm`[4] command: `npm install truffle`.

Furthermore, Truffle provides a powerful interactive console, enabling direct communication with deployed contracts in a built-in blockchain, where all truffle commands are available. In a Truffle project, developers can, parallel to `npm`, use `ethPM` for Package Management, which is the new Package Registry for Ethereum and has been supported from many diverse Ethereum development tools[Suic]. A base Truffle project created using `truffle init` with no smart contracts in has following items[Suia]:

- `contracts/`: Directory for Solidity Smart Contracts.
- `migrations/`: Directory for scriptable deployment files.
- `test/`: Directory for test files for testing the application and contracts.
- `truffle-config.js`: Truffle configuration file.

[3] Solidity Version 0.8.11 Release Notes https://github.com/ethereum/solidity/releases/tag/v0.8.11

[4] Node package manager

3.3. Ganache

Ganache is a personal blockchain, which can be fired up with a single click, providing a safe and deterministic environment for rapid Ethereum distributed application development, available for Mac, Linux, and Windows[Suib]. Beside a CLI, Ganache provides an user-friendly User Interface(UI) exposing its feature-rich functionalities such as: **Visual Mnemonic & Account Info** , **Blockchain Log Output**, or even a **Built-In Blockchain Explorer**.

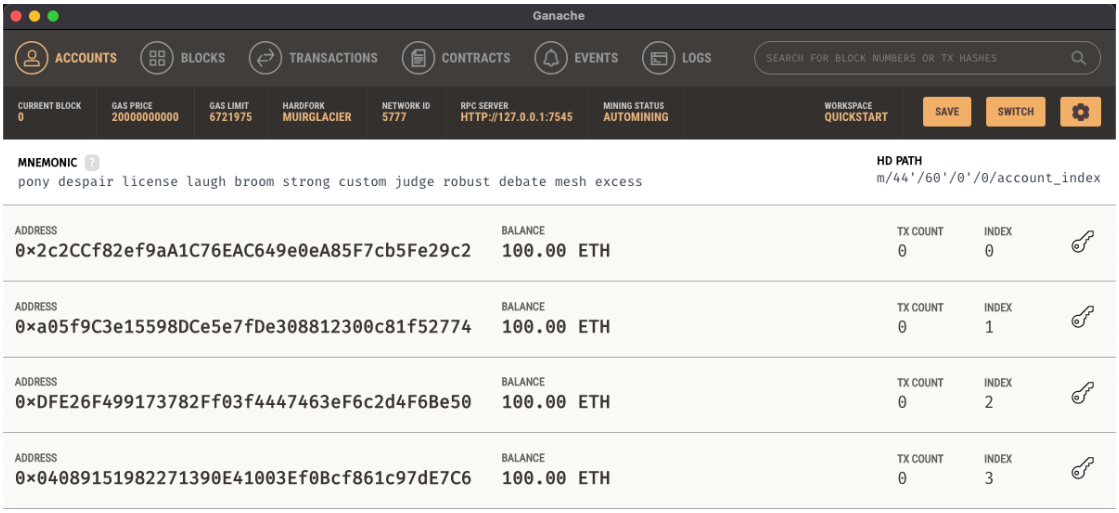


Figure 3.1: Ganache UI

The Squib Token Smart Contract will be deployed locally to Ganache.

3.4. OpenZeppelin

Currently there are already many implementations of the ERC-20 Token Standard. OpenZeppelin, which is one of the most trusted, leading companies in crypto cybersecurity technology and its services, provides an Open Source Library of reusable and secure Smart Contracts written in Solidity for blockchains such as Ethereum[Opez]. Its code is well tested, reviewed and wide-used by the Blockchain Community. This library also contains an implementation of the ERC-20 Token Standard, which theoretically does not require anymore Solidity code written to be deployed, additionally enriched with many other separated extensions such as `AccessControl` that enables a general role based access control mechanism, or `ERC20Burnable` that allows tokens to be destroyed from circulation by token holders, promising the Blockchain Community a great

deal of customization and application potential. Not to mention the fact that it is wise to take advantage of this work to save the efforts writing code from scratch and reduce the risk of vulnerabilities, alongside with essential modifications according to Squib Ltd. requirements.

Here are a list of contracts used in this project:

- `ERC20` : Fully implemented and ready-to-deploy contract of the ERC-20 Token Standard.
- `ERC20Burnable` : Extension of `ERC20` that allows token holders to destroy both their own tokens and those that they have an allowance for[Opeo].
- `Ownable` : Contract module which provides a basic access control mechanism, where there is an account (an owner) that can be granted exclusive access to specific functions[Opes].
- `AccessControlEnumerable` : Contract module that allows children to implement role-based access control mechanisms and enumerate the members of each role[Opeh]

More details will be discussed in chapter 4.

3.5. Metamask

Interaction with deployed Smart Contracts on Ethereum blockchain requires tremendous precautions, as there is no trusted third parties relied on in case of mistaken Ether transfers or gas refunds. Tokens are gone forever once they are released from one's blockchain wallet. Metamask offers a perfect solution for users to store and manage multiple blockchain accounts, sign transactions, send and receive cryptocurrencies and tokens available on Ethereum or other similar blockchains.

Users can securely connect to a blockchain through compatible web browsers(Chrome, Firefox) using the Metamask extension, or the Metamask mobile app available on both iOS and Android. In this project Metamask serves mostly as a bridge between the deployed **Squib Token** smart contract and the Frontend: Metamask will prompt the user to sign every transaction wishing to be broadcasted.

3.6. useDApp

While **Web2**, whose users's data is collected and exploited, is dominating our internet today, blockchain promises a new era of the internet, where anonymity and equality are honored: **Web3**. More and more *dapps* are built to take advantage of blockchain's features. However, writing raw request to the Ethereum network in order to interact with Smart Contracts is to some degree a repeating and cummbersome task, not to mention the complication while managing the state of each request.

Addressing exactly that, useDApp is a simple but feature-rich framework, help *dapp* developers avoid writing boilerplate code to interact with Ethereum blockchain. useDApp can be viewed as an Ethereum Javascript API, built on top of other modern stacks such as `ethers.js`[5] or `Waffle`[6], providing helpful interfaces from etablishing connection to blockchain, communicating with Smart Contracts, or automatically refreshing on new block[ETH].

This project mainly makes use of hooks, or extends them to talk with the **Squib Token** Smart Contract.

[5]ethers.js library `https://docs.ethers.io/v5/`
[6]Waffle - Sweet tool for smart contracts testing `https://getwaffle.io`

4

Implementation

4.1. Mandatory Functions

Following are a fully implemented, well-tested version of 6 essential methods in regard to a ERC-20 Token Standard, provided by OpenZeppelin which does not require anymore modifications for the purpose of this project.

4.1.1. totalSupply

```
1 function totalSupply() public view virtual override returns
    (uint256) {
2         return _totalSupply;
3     }
```

Listing 4.1: totalsupply function[Opev]

This function retrieves the current total supply of all tokens in circulation, which is stored by a private variable _totalsupply.

4.1.2. balanceOf

```
1 function balanceOf(address account) public view virtual
    override returns (uint256) {
```

```
2         return _balances[account];
3     }
```

Listing 4.2: `balanceOf` function[Opel]

returns the balance of the account with address `account`. Note that `_balances` is a Solidity private mapping:

```
1 mapping(address => uint256) private _balances;
```

Listing 4.3: `_balances` mapping

4.1.3. `transfer`

```
1 function transfer(address recipient, uint256 amount) public
    virtual override returns (bool) {
2         _transfer(_msgSender(), recipient, amount);
3         return true;
4     }
```

Listing 4.4: `transfer` function[Opew]

The caller(`_msgSender()` returns the caller's address) transfers `amount` tokens to `recipient` by calling the internal function `_transfer`:

```
1  function _transfer(address sender, address recipient,
     uint256 amount) internal virtual {
2          require(sender != address(0), "ERC20: transfer from
             the zero address");
3          require(recipient != address(0), "ERC20: transfer to
              the zero address");
4
5          _beforeTokenTransfer(sender, recipient, amount);
6
7          uint256 senderBalance = _balances[sender];
8          require(senderBalance >= amount, "ERC20: transfer
             amount exceeds balance");
9          unchecked {
10             _balances[sender] = senderBalance - amount;
11         }
12         _balances[recipient] += amount;
13
```

```
14          emit Transfer(sender, recipient, amount);
15
16          _afterTokenTransfer(sender, recipient, amount);
17      }
```

Listing 4.5: `_transfer` function[Opef]

Beside checking on the prerequisite for the transfer to happen, there are basically two simple arithmetic expressions done:

- deducting `amount` tokens from sender's balance,
- and adding up `amount` tokens to recipient's balance.

Afterwards, a `Transfer` event is obligatorily sent over the network, indicating a transfer done. `Transfer` event is declared as following

```
1 event Transfer(address indexed from, address indexed to,
    uint256 value);
```

Listing 4.6: `Transfer` event[VB15]

As stated in [VB15], `Transfer` "MUST trigger when tokens are transferred, including zero value transfers", says in case of *minting* or *burning*.

`_beforeTokenTransfer` and `_afterTokenTransfer` are ***hooks***, which "are simply functions that are called before or after some action takes place[Opey]". These hooks are still empty functions in OpenZeppelin's implementations, staying untouch in this project.

4.1.4. `approve`

```
1 function approve(address spender, uint256 amount) public
    virtual override returns (bool) {
2           _approve(_msgSender(), spender, amount);
3           return true;
4       }
```

Listing 4.7: `approve` function[Opek]

calls the internal `_approve` function:

```
1 function _approve(address owner, address spender, uint256
    amount
2       ) internal virtual {
```

```
3         require(owner != address(0), "ERC20: approve from
              the zero address");
4         require(spender != address(0), "ERC20: approve to
              the zero address");
5
6         _allowances[owner][spender] = amount;
7         emit Approval(owner, spender, amount);
8     }
```

Listing 4.8: `_approve` function[Opea]

`_allowances` is a Solidity mapping, whose keys hold address of token owners and values are also of type Solidity mapping, which contain the address of the qualified spenders and the remaining associated allowances:

```
1 mapping(address => mapping(address => uint256)) private
      _allowances;
```

Listing 4.9: `_allowances` mapping[Opei]

Inevitably, an `Approval` event "MUST trigger on any successful call to `approve(address _spender, uint256 _value)`"[VB15].

4.1.5. `transferFrom`

```
1  function transferFrom(address sender, address recipient,
       uint256 amount) public virtual override returns (bool) {
2         uint256 currentAllowance = _allowances[sender][
              _msgSender()];
3         if (currentAllowance != type(uint256).max) {
4             require(currentAllowance >= amount, "ERC20:
                  transfer amount exceeds allowance");
5             unchecked {
6                 _approve(sender, _msgSender(),
                      currentAllowance - amount);
7             }
8         }
9
10        _transfer(sender, recipient, amount);
11
12        return true;
```

```
13      }
```

Listing 4.10: `transferFrom` function[Opex]

technically issues a token transfer from `sender` to `recipient`, in case the caller has been approved to spend tokens by `sender`. Sequently, a new allowance is set to `currentAllowance - amount` and the function `_transfer` is called.

4.1.6. `allowance`

```
1  function allowance(address owner, address spender) public
       view virtual override returns (uint256) {
2          return _allowances[owner][spender];
3      }
```

Listing 4.11: `allowance` function[Opej]

straightforwardly returns `spender`'s remaining allowance from `owner`.

4.2. Customized Functionalities

4.2.1. Role-based access control

Despite the fact that in Blockchain environment, there should be fundamentally no user in the network who is more important than the others, it is still significant for **Squib Token** to have some sort of administrative roles such as Minter Role, who can mint more tokens into circulation, or Default Admin Role, who can grant different roles to other user. These roles cannot be provided by default to everyone in the network, but only to a certain group of internally identified people, who also hold accordingly important positions in Squib Ltd. Nevertheless, unless there are serious problems from Squib Ltd, for example in the case of corruption, the decentralization of the blockchain is still honored, since once the tokens are rewarded to the users, they stay in the circulation until burned and there is no way for Squib Ltd. to manipulate these tokens.

Role Definition

New roles are defined by this command:

```
1  bytes32 public constant MY_ROLE= keccak256("MY_ROLE");
```

Listing 4.12: Role Definition

Then the ”MY_ROLE” role can be referred to by its MY_ROLE identifier, which is unique and exposable to the external API[Opeg]. Note that the keccak256 function is a cryptographic hash function built directly and used preferably in Solidity, which converts an arbitrary input into an unique 32 bytes hash[Tea].

From Squib Ltd.'s point of views in the early phase, only Default Admin Role and Minter Role are adequate to fulfill the predefined purposes. Default Admin Role, the admin role for all roles, has been already prebuilt in the parent contract AccessControl, which is derived by this contract's parent contract AccessControlEnumerable. Minter Role is defined as following:

```
1 bytes32 public constant MINTER= keccak256("MINTER");
```

Listing 4.13: Minter Role Definition

Role grant

Granting a defined role for an eligible user is straight-forward and as simple as calling the grantRole function prebuilt in the parent class AccessControl:

```
1 function grantRole(bytes32 role, address account) public
    virtual override onlyRole(getRoleAdmin(role)) {
2         _grantRole(role, account);
3     }
```

Listing 4.14: grantRole function[Oper]

Note that the onlyRole(getRoleAdmin(role)), one of the **Function Modifiers** predefined in parent contracts, which ”automatically check a condition prior to executing the function[Come]”, specifies only users possessing admin role of role are permitted to call this function. grantRole in turn calls the internal function _grantRole only available in base contract, where the actual work is done:

```
1 function _grantRole(bytes32 role, address account) internal
    virtual {
2         if (!hasRole(role, account)) {
3             _roles[role].members[account] = true;
4             emit RoleGranted(role, account, _msgSender());
5         }
6     }
```

Listing 4.15: _grantRole function[Opec]

This function basically first checks whether the account has role role. If not account will be assigned role role and an event GrantRole containing information, that account has been granted role role by whoever _msgSender() called the function in the first

place, is triggered over the network, which is compulsory for the arguments to be stored in the transaction's log for later access[Comd].

With those resources provided, granting roles in the Squib Token contract is as effortless as following:

Default Admin Role Granting `DEFAULT_ADMIN_ROLE`

```
1 function grantAdminRole(address account) public onlyRole(
    DEFAULT_ADMIN_ROLE) {
2         grantRole(DEFAULT_ADMIN_ROLE, account);
3     }
```

Listing 4.16: `grantAdminRole` function

Minter Role Granting `MINTER_ROLE`

```
1 function grantAdminRole(address account) public onlyRole(
    DEFAULT_ADMIN_ROLE) {
2         grantRole(MINTER_ROLE, account);
3     }
```

Listing 4.17: `grantMinterRole` function

For both functions it is comprehensible that only default admins are qualified to grant these roles. Therefore the modifier `onlyRole(DEFAULT_ADMIN_ROLE)` is used.

Role Revocation

When a role has been granted for an account, it should also be possibly revoked at a later time. Accordingly, there are also a `revokeRole` and a `_revokeRole` predefined in the base contract `AccessControl`:

```
1 function revokeRole(bytes32 role, address account) public
    virtual override onlyRole(getRoleAdmin(role)) {
2         _revokeRole(role, account);
3     }
```

Listing 4.18: `revokeRole` function[Opet]

```
1 function _revokeRole(bytes32 role, address account) internal
    virtual {
2         if (hasRole(role, account)) {
3             _roles[role].members[account] = false;
4             emit RoleRevoked(role, account, _msgSender());
```

```
5         }
6     }
```

Listing 4.19: `_revokeRole` function[Opee]

The function `_revokeRole` removes the role `role` granted for `account`, and correspondingly fires an `RoleRevoked` to the network.

And in the derived contract:

Default Admin Role Revoking `DEFAULT_ADMIN_ROLE`

```
1 function revokeAdminRole(address account) public onlyOwner {
2         revokeRole(DEFAULT_ADMIN_ROLE, account);
3     }
```

Listing 4.20: `revokeAdminRole` function

Minter Role Revoking `MINTER_ROLE`

```
1 function revokeMinterRole(address account) public onlyRole(
    DEFAULT_ADMIN_ROLE) {
2         revokeRole(MINTER_ROLE, account);
3     }
```

Listing 4.21: `revokeMinterRole` function

Note that it is self-explanatory default admins are eligible to revoke Minter Role. However, since default admins have same right Default Admin Role, they are not supposed to revoke it from each other. This is where the `Ownable` base contract comes into action, which provides the function modifier `onlyOwner`, stating only the contract owner, who deployed the token contract at the beginning, is worthy to invalidate Default Admin Role for default admins.

4.2.2. `mint`

This function helps generating new tokens, serving the **Squib Token's supply mechanism**.

Squib Token's supply mechanism *The initial tiny amount of tokens generated at contract deployment is undertaken by an equivalent amount of a fiat currency, which is stored in some sort of vault at Squib Ltd. Generating new tokens into circulation requires more cash at stake, so that a correspondence between tokens and real-world money meets. Squib Ltd. at any time must guarantee this connection.*

```
1 function mint(address to, uint256 amount) public onlyRole(
    MINTER_ROLE) {
2         _mint(to, amount);
3     }
```

Listing 4.22: `mint` function

The function modifier `onlyRole(MINTER_ROLE)` clearly declares the obvious fact that only minters is permitted to mint new tokens. Nonetheless, default admins cannot forge new tokens by default, unless they are minter as well, says, they have been granted Minter Role by other admins or themselves.

Let's take a closer look to the internal `_mint` function from `ERC20` base contract:

```
1  function _mint(address account, uint256 amount) internal
     virtual {
2         require(account != address(0), "ERC20: mint to the
            zero address");
3
4         _beforeTokenTransfer(address(0), account, amount);
5
6         _totalSupply += amount;
7         _balances[account] += amount;
8         emit Transfer(address(0), account, amount);
9
10        _afterTokenTransfer(address(0), account, amount);
11    }
```

Listing 4.23: `_mint` function[Oped]

Unquestionably, the address `account` that `amount` tokens are minted to must not be 0. Otherwise, the transaction will be reverted on alert: *"mint to the zero address"*. The mere logic behind minting is frankly adding the `amount` tokens to both token's total supply and the balance of `account` and firing the deterministic event `Transfer` with `sender` set to address 0, since `amount` tokens have been created plainly out of nowhere.

4.2.3. `reward`

The motivation behind this function is that after a certain period of time, users are rewarded a determined amount of tokens depending on the interaction between the users and Squib in that period of time. Technically, a reward transaction is merely a finite number of mint transactions. There are three approaches taken in consideration:

- **3 transactions with 2 mappings:** At each reward event, two correlative arrays, *an account array* and *an amount array*, are imported separately and stored as values in two independent mappings with the same *keys*, so that they can be retrieved accurately together later. Then a third transaction is explicitly required to process the actual rewarding job by fetching the arrays using the aforementioned *key* from the mappings, looping through them and performing `mint` for each element. In this way all reward events are stored on chain forever and can be accessed anytime in the future. Nevertheless, interacting with mappings in a transaction is considered gas-inefficient, not to mention the number of internal `mint` transactions called.

- **n transactions:** In the frontend application, looping through the reward list and calling `mint` provided by the Squib Token's Contract API for each account in the reward list. This demands a great deal of user interaction, since `mint` is called outside the contract's context and MetaMask will ask to sign the transaction at each loop. There is a possibility to bypass MetaMask by providing the private key of the caller account to sign the transaction when calling the contract function externally. However, doing so somewhat put the users at risk, as the *secret private key* is propagated over the internet.

- **1 transaction:** A contract function takes two correlative arrays, *an account array* and *an amount array*, looping through them accordingly and performing `mint` at each loop.

The third approach exploits the ideas of the others two, constructing the most appropriate version for rewarding eligible users.

```
1 function reward(address[] memory addresses, uint256[] memory
      amounts) public onlyRole(DEFAULT_ADMIN_ROLE)
2     {
3         require(addresses.length == amounts.length,
4             "Unequal quantity of elements in address and
                amount collections.");
5         uint256 count = addresses.length;
6         for (uint256 i = 0; i < count; i++) {
7             _mint(addresses[i], amounts[i]);
8         }
9     }
```

Listing 4.24: `reward` function

One condition is that the *addresses* array and *amounts* array must have the same lengths to match, not to mention the assumption they are imported precisely. It makes senses that only Default Admin Role is acceptable for this kind of action, whereas Minter Role is only responsible for single minting transaction. In the `for`-loop, the `internal virtual _mint` function in the base contract is directly called to reduce a level of abtraction.

4.2.4. `redeem`

According to **Squib Token's supply mechanism**, tokens acquire their tangible value since genesis. However, a certain extend of public approval for Squib Token must be gained in order for the token to get listed by a cryptocurrency exchange such as **Binance** or **Uniswap**, before Squib's users can truly trade it with other tokens effortlessly and independently. This is the reason `redeem` function is provided. It serves the purpose of promising users an incentive while they interact with Squib: *when a certain amount of tokens is reached, it can be exchanged with real-world valuable objects provided directly in the Squib Platform, says, a bank transfer or a shopping voucher.*

Technically, when tokens are redeemed, they are removed from circulation or in fact, they are ***burned***.

```
1 function redeem(uint256 amount) public {
2         burn(amount);
3     }
```

Listing 4.25: `redeem` function

It calls the `burn` function from parent contract `ERC20Burnable`:

```
1 function burn(uint256 amount) public virtual {
2         _burn(_msgSender(), amount);
3     }
```

Listing 4.26: `burn` function[Opem]

which in turn calls the `_burn` function from `ERC20` base contract:

```
1 function _burn(address account, uint256 amount) internal
    virtual {
2         require(account != address(0), "ERC20: burn from the
              zero address");
3
4         _beforeTokenTransfer(account, address(0), amount);
5
```

```
6         uint256 accountBalance = _balances[account];
7         require(accountBalance >= amount, "ERC20: burn
              amount exceeds balance");
8         unchecked {
9             _balances[account] = accountBalance - amount;
10        }
11        _totalSupply -= amount;
12
13        emit Transfer(account, address(0), amount);
14
15        _afterTokenTransfer(account, address(0), amount);
16    }
```

Listing 4.27: `_burn` function[Opeb]

It is reasonable users can only manipulate their own tokens or that they have been approved for, indicated in those burn functions that tokens are burned from caller balance(`_msgSender`) and the balance `_balances[account]` must greater or at least equal the amount burned `amount`. Subsequently, these tokens vanishes from circulation, which causes the an `amount` reduction of `_totalSupply`: `_totalSupply -= amount`. Not to mention the trigger for a Transfer event with the recipient address set to 0, which implicitly informs the network some tokens have been burned.

4.2.5. Getters

So far, all the functions discussed previously are considered **transactions**, since each of them to some extend modifies the state of the smart contract when executed. Transactions [Comn]:

- are computed by miners,
- cost Gas,
- and won't return a value.

On the contrary, there is a different kind of functions which only read data and don't change the state of the blockchain: **calls**, that [Comn]:

- are free,
- are processed immediately,

- will return a value.

Getter methods are **calls** that contain `view` in their signatures[Coml]. Squib Token contract API provides following public Getters for the convenience of its users:

```
1 function hasAdminRole(address account) public view returns (
    bool) {
2         return hasRole(DEFAULT_ADMIN_ROLE, account);
3     }
```

Listing 4.28: `hasAdminRole` getter

```
1 function hasMinterRole(address account) public view returns
    (bool) {
2         return hasRole(MINTER_ROLE, account);
3     }
```

Listing 4.29: `hasMinterRole` getter

```
1 function getAllMinters() public view returns (address[]
    memory) {
2         return getAllAddressesOfRole(MINTER_ROLE);
3     }
```

Listing 4.30: `getAllMinters` getter

```
1 function getAllAdmins() public view returns (address[]
    memory) {
2         return getAllAddressesOfRole(DEFAULT_ADMIN_ROLE);
3     }
```

Listing 4.31: `getAllAdmins` getter

`hasAdminRole` and `hasMinterRole` examine whether `account` holds the corresponding role. `getAllMinters` and `getAllAdmins` retrieve the whole list of respective role by calling another `internal view` function `getAllAddressesOfRole`:

```
1     function getAllAddressesOfRole(bytes32 role) internal
        view returns (address[] memory)
2     {
3         uint256 roleCount = getRoleMemberCount(role);
4         address[] memory roleAddresses = new address[](
            roleCount);
5         for (uint256 i = 0; i < roleCount; i++) {
```

```
6                 roleAddresses[i] = getRoleMember(role, i);
7             }
8             return roleAddresses;
9         }
```

Listing 4.32: `getAllAddressesOfRole` function

Again, `getAllAddressesOfRole` uses the predefined functions `getRoleMemberCount` and `getRoleMember` in the parent contract `AccessControlEnumerable` to access the private variable `_roleMembers`, which stores the actual contract data and cannot be retrieved even in the derived Squib Token contract.

```
1 function getRoleMemberCount(bytes32 role) public view
     virtual override returns (uint256) {
2             return _roleMembers[role].length();
3         }
```

Listing 4.33: `getRoleMemberCount` function[Opeq]

```
1 function getRoleMember(bytes32 role, uint256 index) public
     view virtual override returns (address) {
2             return _roleMembers[role].at(index);
3         }
```

Listing 4.34: `getRoleMember` function[Opep]

4.2.6. The constructor

In the constructor, initial statements must be declared, as it will be called at contract deployment. Granting the `DEFAULT_ADMIN_ROLE` to the contract owner is the most important one. Otherwise, there will be no new tokens generated. The token `name`, `symbol` are assigned by calling the constructor of base contract `ERC20`:

```
1 constructor(string memory name_, string memory symbol_) {
2             _name = name_;
3             _symbol = symbol_;
4         }
```

Listing 4.35: `ERC20` constructor[Open]

Besides, the contract owner gets the `MINTER_ROLE` assigned and receives 1500 tokens at contract deployment. The Squib Token smart contract is then completed:

see `SquibToken.sol`.

4.3. Deployment

Deploying a contract to blockchain is a transaction itself, meaning it requires Gas to be executed and the contract owner has to pay the gas fee, in case it is deployed to Ethereum Mainnet[1].

Therefore, it is wise to deploy the first prototype of Squib Token to a personal blockchain - Ganache - the one click blockchain, running directly on a local computer.

4.3.1. Compiling Contracts

Firstly the contract needs to be compiled to executable bytecode for the EVM(Ethereum Virtual Machine) to understand, using this command:

```
truffle compile
```

Listing 4.36: Compiling contract

Once successfully compiled, the artifacts of the compilation will be stored in the `build/contracts/` directory, which are `.json` files, whose name reflect the name of the **contract definition**(and not the name of **source file**). The file `SquibToken.json` is referred to as an *contract ABI*[2], which defines the methods and structures for the Frontend to interact with the binary contract.

4.3.2. Migrating Contracts

Thereafter, the compiled EVM bytecode has to be migrated to Ganache, which is currently running a local instance of Ethereum blockchain. The migration files will help achieve that, which are responsible for staging deployment tasks [Suid]. It is only necessary to create a new migration file for the single `SquibToken.sol` contract:

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

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

Listing 4.37: `2_deploy_squibtoken.js`

[1] Mainnet https://ethereum.org/en/developers/docs/networks/#mainnet
[2] Application Binary Interface https://www.quicknode.com/guides/solidity/what-is-an-abi

Note that migration files's name always begins with a number, specifying the order of the contract's bytecode to be migrated, which in this case 2, after the `1_initial_migration.js` file shipped with Truffle project by default.

Finally, execute the migrate command:

```
truffle migrate
```

Listing 4.38: Migrating contract

The information of the migration such as transaction hash, account, gas used, or address of deployed contract will be displayed in the console output or can be reviewed in Ganache. The *contract address* will vary each time the same contract is deployed. It will be used in the Frontend, aside from the *contract ABI*.

4.4. Frontend

The frontend applications developed to communicate with Smart Contracts on blockchain are called *dezentralized applications*, or *dapps*. Two separated *dapps* are developed to serve different purposes of this project, basically aiming to different user roles. Since both are developed using **React with Typescript** and useDApp, they have some general similarities.

useDApp makes it relatively simple to etablish the connection with blockchain by proving the hook: `userEthers`

```
const { account, deactivate, activateBrowserWallet, error,
    active } = useEthers()
```

Listing 4.39: `useEthers`

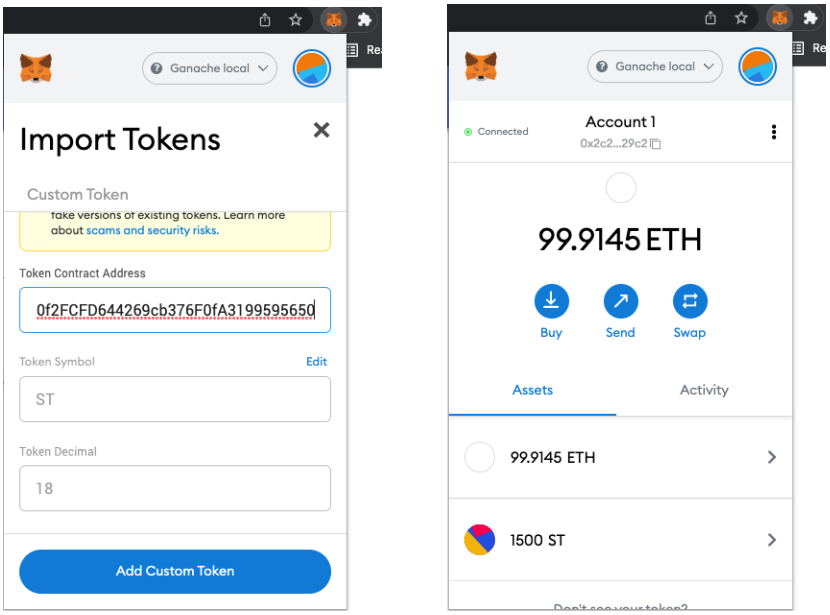
The functions `activateBrowserWallet` and `deactivate` can be called by a single button click to connect or disconnect with blockchain

```
{account ?
    <Button onClick={() => deactivateAccount()}>Disconnect</
        Button>
    : <Button onClick={() => activateBrowserWallet()}>
        Connect</Button>}
```

Listing 4.40: Connection on a single click

If Metamask is configured correctly using the account provided by Ganache, `account` will return the account's address and `active` is set to `true` in case of a successful connection, or else errors will be stored by `error`.

The Squib Token will not be listed in Metamask by default and has to be manually imported using the *contract address* from the contract migration:



(a) Import Squib Token (b) Squib Token listed in **Assets**

Figure 4.1: Metamask extension on Chrome

The `account`'s balance and token's total supply are simply retrieved and automatically updated using these hooks: `useTokenBalance` and `useTokenTotalSupply`. The *contract address* is used here as in `tokenAddress`.

```
1 const tokenBalance = useTokenBalance(tokenAddress, account);
2 const totalSupply = useTokenTotalSupply(tokenAddress);
```

Listing 4.41: `useTokenBalance` and `useTokenTotalSupply`

On top of that, useDApp maximizes customization potential by providing `useContracFunction` and `useContractCall` hooks, allowing Smart Contracts's customized functions to be exposed to. This is where the *contract ABI* comes into action. New custom hooks will be introduced in User Wallet and Admin frontends.

4.4.1. User Wallet

`useTokenRedeem`

This custom hook calls the `redeem` function from the **Squib Token** contract API, when a Squib user wishes to exchange their tokens with a real-world valuable object. Technically, on the blockchain these tokens will be removed from circulation, reducing the

token's total supply. This then signals the exchange process to carry on sending the chosen object to user, says by calling other API, which can be completely irrelevant to blockchain.

```
1 export const useTokenRedeem = (tokenAddress: string) => {
2  const contract = new Contract(tokenAddress, SquibToken.abi)
3  const {state: redeemState, send: redeem, events} =
     useContractFunction(contract, 'redeem', {transactionName
     : 'Redeem'})
4  return {redeemState,  redeem, events}
```

Listing 4.42: `useTokenRedeem`

`useContractFunction` takes an `Contract` object, which is declared using *contract address* and *contract ABI*, as the first argument, the respective contract function's name as the second argument. The third optional argument is of type `TransactionOverrides`[3], which can be used to manipulate transaction parameters such as transactionName, gasPrice any many others.

`state` stores the transaction status after `send` is called, and `events` collects all events fired during the transaction. All of the three variables can be renamed at will.

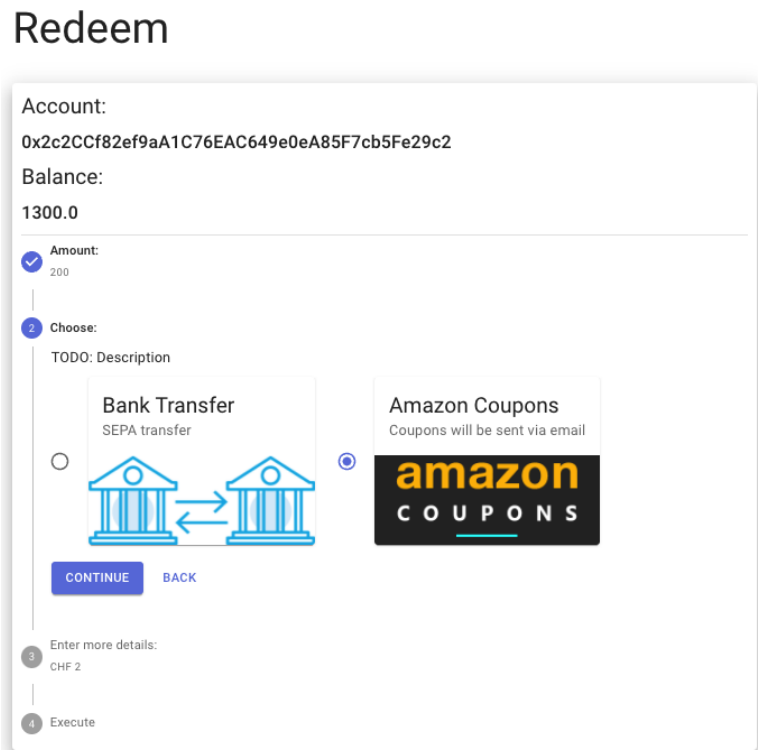


Figure 4.2: Redeem process

[3] https://usedapp.readthedocs.io/en/latest/core.html#usecontractfunction

4.4.2. Admin

As Admin will be deployed as a web application, it is accessible for users using a supported browser, as long as they have a Ethereum wallet. However, only certain users with administrative roles are permitted to broadcast transactions such as `reward`, `mint`, or `grantMinterRole`.

Respectively, Admin has following custom hooks:

`useMint`
see A.2

`useTokenReward`
see A.3

`useGrantAdminRole`
see A.4

`useRevokeAdminRole`
see A.5

`useGrantMinterRole`
see A.6

`useRevokeMinterRole`
see A.7

`useHasAdminRole`
`useContractFunction` is used instead of using `useContractFunction` to access the contract's Getters.

```
1  export const useHasAdminRole = (tokenAddress: string,
     account: string) => {
2      const [isAdmin] =
3          useContractCall({
4              abi: new utils.Interface(SquibToken.abi),
5              address: tokenAddress,
6              method: "hasAdminRole",
7              args: [account],
8          }) ?? [];
9
10     return isAdmin;
```

```
11 }
```

Listing 4.43: `useHasAdminRole`

`useHasMinterRole`
see A.8

`useGetAdminRole`
see A.9

`useGetMinterRole`
see A.11

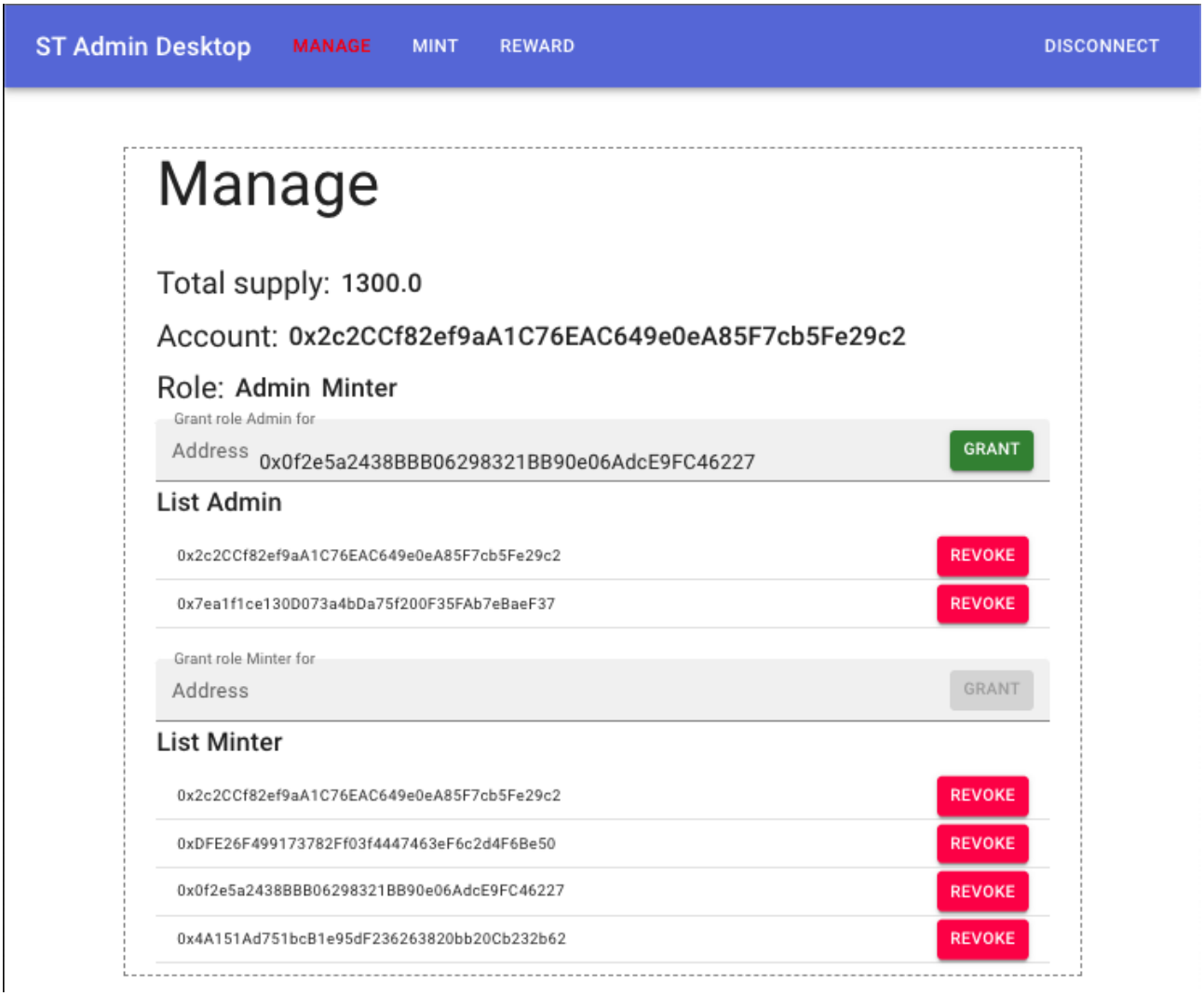


Figure 4.3: Squib Admin UI

5

Results and Discussion

The outcome of this bachelor project is the creation of a new ERC-20 Token in need of rewarding Squib users when they interact with the platform. This shapes a Win-Win-Win situation, where

- The more votings an user get involved in, the more tokens they earn.
- The more people involve in votings, the more insights enterprises gain.
- The more insights enterprises gain, the more votings they create, the more values Squib get.

The nature of blockchain grants **Squib Token** the best environment to be deployed in, where no intermediaries are required to keep track of transactions. Once rewarded to users, it's their decision to manipulate these tokens in their own wallet, allowing Squib to get rid of all the maintenance tasks.

However, there are certain inevitable drawbacks such as significant deployment and transaction costs, latency in mining process which causes a transaction might not be processed intermediately, or proneness to upgradability. Regarding efficiency, Squib Token can be deployed to Harmony[1] instead of Ethereum, a growing fully ERC-20 compatible blockchain running Ethereum Virtual Machine with block time in the matter of seconds and less transaction fees.

So far, the Squib Token implemented in this project is as alike as a StableCoin[2], whose value is always attached to that of a fiat currency, says 100 Squib Tokens are consistently worth CHF 1, as the price of a token never changes.

[1] Harmony blockchain https://www.harmony.one
[2] StableCoin https://www.investopedia.com/terms/s/stablecoin.asp

Squib's ambition is to make Squib Token fluctuate, causing it unique and its value will depend on Squib's reputation. An approach to achieve this goal is suggested in section 5.1.

5.1. "Unstable" Squib Token Strategy

The idea: *Limiting the **amount** of tokens, which could be maximal and minimal generated (minted) in a **period** of time. **period** and **amount** are to be discussed.*

$T_{min}|T_{max}$ are minimal — maximal **minted tokens** per Period P

Current Amount of Tokens $= CT$, each token currently values cV

Value of all tokens in current period $= cV * CT = VcP$ (Value current Period)

In the next period, T_{new} tokens were minted and T_{sold} were exchanged with enterprises. Note that:

$T_{min} \leq T_{new} \leq T_{max}\ and\ 0 \leq T_{sold} \leq T_{max}$

Because it's possible **not all generated tokens were sold** to enterprises. In this case:

$0 \leq T_{sold} \leq T_{min} = T_{new}$

T_{sold} was sold with V_{new} totally.

Then the new amount of tokens: NT (New Amount of Token):

$NT = CT + T_{new}$

And the new value of all tokens in new period:

$VnP = VcP + V_{new}$

Then we could calculate the new value of each token:

$$nV = \frac{VnP}{NT} = \frac{VcP + V_{new}}{CT + T_{new}}$$

Case 1: Not all new generated tokens were sold to enterprises

$0 \leq T_{sold} < T_{new}$

Each token should be sold with the value of last period: cV

Then:

$V_{new} = cV * T_{sold}$

$$nV = \frac{VcP + V_{new}}{CT + T_{new}} = \frac{cV * CT + cV * T_{sold}}{CT + T_{new}} = cV * \frac{CT + T_{sold}}{CT + T_{new}}$$

Due to $T_{sold} < T_{new}$

$\implies CT + T_{sold} < CT + Tnew$

$$\implies \frac{CT + Tsold}{CT + Tnew} < 1$$

$$\implies cV * \frac{CT + T_{sold}}{CT + T_{new}} < cV$$
$$\implies nV < cV$$

Summary: *The value of tokens **decreases** when not all the minted tokens were sold.*

Case 2: All generated tokens were sold to enterprises.

$T_{new} = T_{sold} = T$

$T_{min} <= T <= T_{max}$

After the sold, each token worths nV

Normally the price of each new token should be cV

Then the actual new value of new generated tokens:

$V_{new} = V_{actual} = cV * T$

$$nV = \frac{VcP + V_{new}}{CT + T} = \frac{cV * CT + cV * T}{CT + T}$$

If an enterprise was willing to pay an excess amount of money more (V_{excess}) for the same amount of tokens:

$V_{new} = V_{excess} + V_{actual} \geq V_{actual}$

$$\implies VcP + V_{new} \geq VcP + V_{actual}$$
$$\implies \frac{VcP + V_{new}}{CT + T} \geq \frac{VcP + V_{actual}}{CT + T}$$
$$\implies nV \geq cV$$

Case 2.1:

$nV = cV$ when $V_{excess} = 0$

Summary: *When no excess amount of money was paid for the same amount of tokens, the value of tokens **stays persistently.***

Case 2.2:

$nV > cV$ when $V_{excess} > 0$

Summary: *When someone paid more for the same amount of tokens, the value of tokens **increases**, under the condition that sold amount of tokens is the maximal amount of generated tokens in the period.*

Example:

$T_{min} = 1000, T_{max} = 10000$

Current Amount of Tokens $CT = 5000$, current value of each token $cV = CHF\ 1$

Value of all tokens in current period $VcP = cV * CT = CHF\ 5000$

Case 1:
Amount of tokens that is sold to enterprises $T_{sold} = 500$
with the price CHF 1 per token $\implies V_{new} = CHF\ 500$
Amount tokens generated $T_{new} = 1000$
Then the new amount of tokens: $NT = 6000$
And the new value of all tokens in new period:
$VnP = VcP + V_{new} = 5000 + 500 = 5500$
Then we could calculate the new value of each token:
$nV = \frac{VnP}{NT} = \frac{5500}{6000} = CHF\ 0,91667 <$ price of last period $CHF\ 1$

Case 2.1:
Amount tokens sold to enterprises $T_{sold} = 5000$ with the price $CHF\ 1$ per token.

Case 2.2:
Amount tokens sold to enterprises $T_{sold} = T_{max} = 10000$ with $Vexcess = CHF\ 2000$
The calculation is listed on the table 5.1 for better comparison.

5.2. Next steps

This work proposes the first prototype of the Squib Token. There's still lot of room for improvement, such as:

- Enable Upgradability for Squib Token Smart Contract.
- Extend the communication to talk with other popular wallets than Metamask: Coinbase Wallet[3], Trust Wallet[4].
- Integrate Admin and User Wallet in Squib's website.
- Dezentralize the Squib platform.

[3] https://www.coinbase.com/wallet
[4] https://trustwallet.com/

	Case 1	Case 2.1	Case 2.2
T_{min}	1000	1000	1000
T_{max}	10000	10000	10000
T_{new}	1000	5000	10000
T_{sold}	500	5000	10000
CT	5000	5000	5000
NT	6000	10000	15000
VcP	CHF 5000	CHF 5000	CHF 5000
V_{excess}	CHF 0	CHF 0	CHF 2000
$V_{actual} = cV * T_{sold}$	CHF 500	CHF 5000	CHF 10000
$V_{new} = V_{excess} + V_{actual}$	CHF 500	CHF 5000	CHF 12000
$VnP = VcP + V_{new}$	CHF 5500	CHF 10000	CHF 17000
cV	CHF 1	CHF 1	CHF 1
$nV = \frac{VnP}{NT}$	CHF 0,9167	CHF 1	CHF 1,13

Table 5.1: Squib Token price fluctuation

6

Summary

To sum up, the emergence of **Squib Token** enables the ”Play-to-Earn” concept to be deployed in the A/B Voting Platform of Squib Ltd. The token is developed following ERC-20 Token Standard with the help of the reliable OpenZeppelin contract library and Truffle development environment, deployed rapidly using Ganache One Click Blockchain. The **Squib Token** contract API is exposed and tested in two decentralized web application: User Wallet and Admin, bridging through Metamask.

Blockchain technology paves the way for the transition to the next generation of Internet, where users are able to control their assets better than ever. Aside from contributing to Squib Ltd., spending effort to dig deeper in the blockchain world gave me an overall insight of blockchain's applications, opening new opportunities and directions for my personal and career development in the future.

A

APPENDIX

A.1. SquibToken.sol

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.11;
3
4  import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
5  import "@openzeppelin/contracts/token/ERC20/extensions/
       ERC20Burnable.sol";
6  import "@openzeppelin/contracts/access/Ownable.sol";
7  import "@openzeppelin/contracts/access/
       AccessControlEnumerable.sol";
8
9  contract SquibToken is ERC20, ERC20Burnable, Ownable,
       AccessControlEnumerable {
10     bytes32 public constant MINTER_ROLE = keccak256("
           MINTER_ROLE");
11
12     constructor() ERC20("SquibToken", "ST") {
13         _mint(msg.sender, 1500 * 10**decimals());
14         _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
15         _grantRole(MINTER_ROLE, msg.sender);
16     }
17
```

```
18      function grantAdminRole(address account) public
19          onlyRole(DEFAULT_ADMIN_ROLE)
20      {
21          grantRole(DEFAULT_ADMIN_ROLE, account);
22      }
23
24      function revokeAdminRole(address account) public
           onlyOwner {
25          revokeRole(DEFAULT_ADMIN_ROLE, account);
26      }
27
28      function mint(address to, uint256 amount) public
           onlyRole(MINTER_ROLE) {
29          _mint(to, amount);
30      }
31
32      function grantMinterRole(address account) public
33          onlyRole(DEFAULT_ADMIN_ROLE)
34      {
35          grantRole(MINTER_ROLE, account);
36      }
37
38      function revokeMinterRole(address account) public
39          onlyRole(DEFAULT_ADMIN_ROLE)
40      {
41          revokeRole(MINTER_ROLE, account);
42      }
43
44      function redeem(uint256 amount) public {
45          burn(amount);
46      }
47
48      function reward(address[] memory addresses, uint256[]
           memory amounts)
49          public onlyRole(DEFAULT_ADMIN_ROLE)
50      {
51          require(
52              addresses.length == amounts.length,
```

```
53              "Unequal quantity of elements in address and
                    amount collections."
54          );
55          uint256 count = addresses.length;
56          for (uint256 i = 0; i < count; i++) {
57              _mint(addresses[i], amounts[i]);
58          }
59      }
60
61      function hasAdminRole(address account) public view
            returns (bool) {
62          return hasRole(DEFAULT_ADMIN_ROLE, account);
63      }
64
65      function hasMinterRole(address account) public view
            returns (bool) {
66          return hasRole(MINTER_ROLE, account);
67      }
68
69      function getAllMinters() public view returns (address[]
            memory) {
70          return getAllAddressesOfRole(MINTER_ROLE);
71      }
72
73      function getAllAdmins() public view returns (address[]
            memory) {
74          return getAllAddressesOfRole(DEFAULT_ADMIN_ROLE);
75      }
76
77      function getAllAddressesOfRole(bytes32 role) internal
            view returns (address[] memory)
78      {
79          uint256 roleCount = getRoleMemberCount(role);
80          address[] memory roleAddresses = new address[](
                roleCount);
81          for (uint256 i = 0; i < roleCount; i++) {
82              roleAddresses[i] = getRoleMember(role, i);
83          }
```

```
84          return roleAddresses;
85      }
86  }
```

Listing A.1: `SquibToken.sol`

A.2. Custom Hooks

```
1
2  export const useMint = (tokenAddress: string, signer?:
       Signer | undefined) => {
3      const contract = new Contract(tokenAddress, new utils.
           Interface(SquibToken.abi), signer);
4      const {state: mintState, send: mint, events} =
           useContractFunction(contract, 'mint', {
           transactionName: 'Mint'});
5      return { mintState, mint, events};
6
7  }
```

Listing A.2: `useMint`

```
1
2  export const useMint = (tokenAddress: string, signer?:
       Signer | undefined) => {
3      const contract = new Contract(tokenAddress, new utils.
           Interface(SquibToken.abi), signer);
4      const {state: mintState, send: mint, events} =
           useContractFunction(contract, 'mint', {
           transactionName: 'Mint'});
5      return { mintState, mint, events};
6
7  }
```

Listing A.3: `useReward`

```
1
2  export const useGrantAdminRole = (tokenAddress: string) => {
```

```
3     const contract = new Contract(tokenAddress, new utils.
          Interface(SquibToken.abi));
4     const {state: grantAdminState, send: grantAdmin, events}
          = useContractFunction(contract, 'grantAdminRole', {
          transactionName: 'Grant Admin Role'});
5
6     return {grantAdminState,  grantAdmin, events};
7 }
```

Listing A.4: `useGrantAdminRole`

```
1
2 export const useRevokeAdminRole = (tokenAddress: string) =>
      {
3     const contract = new Contract(tokenAddress, new utils.
          Interface(SquibToken.abi));
4     const {state: revokeAdminState, send: revokeAdmin,
          events} = useContractFunction(contract, '
          revokeAdminRole', {transactionName: 'Revoke Admin
          Role'});
5
6     return {revokeAdminState,  revokeAdmin, events};
7 }
```

Listing A.5: `useRevokeAdminRole`

```
1
2 export const useGrantMinterRole = (tokenAddress: string) =>
      {
3     const contract = new Contract(tokenAddress, new utils.
          Interface(SquibToken.abi));
4     const {state: grantMinterState, send: grantMinter,
          events} = useContractFunction(contract, '
          grantMinterRole', {transactionName: 'Grant Admin Role
          '});
5
6     return {grantMinterState,  grantMinter, events};
7 }
```

Listing A.6: `useGrantMinterRole`

```
1
2 export const useRevokeMinterRole = (tokenAddress: string) =>
      {
3     const contract = new Contract(tokenAddress, new utils.
        Interface(SquibToken.abi));
4     const {state: revokeMinterState, send: revokeMinter,
        events} = useContractFunction(contract, '
        revokeMinterRole', {transactionName: 'Revoke Admin
        Role'});
5
6     return {revokeMinterState,  revokeMinter, events};
7 }
```

Listing A.7: `useRevokeMinterRole`

```
1
2 export const useHasMinterRole = (tokenAddress: string,
    account: string) => {
3     const [isAdmin] =
4         useContractCall({
5             abi: new utils.Interface(SquibToken.abi),
6             address: tokenAddress,
7             method: "hasMinterRole",
8             args: [account],
9         }) ?? [];
10
11    return isAdmin;
12 }
```

Listing A.8: `useHasMinterRole`

```
1
2 export const useGetAdminRole = (tokenAddress: string) => {
3     const [listAdmins] =
4         useContractCall({
5             abi: new utils.Interface(SquibToken.abi),
6             address: tokenAddress,
7             method: "getAllAdmins",
8             args: [],
9         }) ?? [];
10
```

```
    return listAdmins;
}
```

Listing A.9: useGetAdminRole

```

export const useGetMinterRole = (tokenAddress: string) => {
    const [listMinters] =
        useContractCall({
            abi: new utils.Interface(SquibToken.abi),
            address: tokenAddress,
            method: "getAllMinters",
            args: [],
        }) ?? [];

    return listMinters;
}
```

Listing A.10: useGetMinterRole

```

export const useGetOwner = (tokenAddress: string) => {
    const [listMinters] =
        useContractCall({
            abi: new utils.Interface(SquibToken.abi),
            address: tokenAddress,
            method: "owner",
            args: [],
        }) ?? [];

    return listMinters;
}
```

Listing A.11: useGetMinterRole

References

[Nak08] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. http://www.bitcoin.org/bitcoin.pdf. 2008.

[Ber12] Kent C. Berridge. "From prediction error to incentive salience: mesolimbic computation of reward motivation". In: *The European Journal of Neuroscience* 35.7 (Apr. 2012), pp. 1124–1143. ISSN: 0953-816X. DOI: 10.1111/j.1460-9568.2012.07990.x. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3325516/ (visited on 01/23/2022).

[BK13] Kent C. Berridge and Morten L. Kringelbach. "Neuroscience of affect: Brain mechanisms of pleasure and displeasure". In: *Current opinion in neurobiology* 23.3 (June 2013), pp. 294–303. ISSN: 0959-4388. DOI: 10.1016/j.conb.2013.01.017. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3644539/ (visited on 01/23/2022).

[But14] Vitalik Buterin. *A Next-Generation Smart Contract and Decentralized Application Platform*. https://ethereum.org/en/whitepaper/. [last updated 01/02/2022]. 2014.

[15] "The great chain of being sure about things". In: *The Economist* (Oct. 2015). ISSN: 0013-0613. URL: https://www.economist.com/briefing/2015/10/31/the-great-chain-of-being-sure-about-things (visited on 02/02/2022).

[VB15] Fabian Vogelsteller and Vitalik Buterin. "EIP-20: Token Standard". In: *Ethereum Improvement Proposals* no. 20 (Nov. 2015). [Online serial]. URL: https://eips.ethereum.org/EIPS/eip-20.

[Dan17a] Chris Dannen. *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*. Apress, 2017, p. 10. ISBN: 9781484225356.

[Dan17b] Chris Dannen. *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*. Apress, 2017, p. 47. ISBN: 9781484225356.

[Dan17c] Chris Dannen. *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*. Apress, 2017, p. 90. ISBN: 9781484225356.

[Tak18] T. Takenobu. *Ethereum EVM illustrated*. https://takenobu-hs.github.io/downloads/ethereum_evm_illustrated.pdf. [Online; accessed 04/02/2022]. 2018.

[WHB18] Yingli Wang, Jeong Hugh Han, and Paul Beynon-Davies. "Understanding blockchain technology for future supply chains: a systematic literature review and research agenda". In: *Supply Chain Management: An International Journal* 24.1 (Jan. 2018), pp. 62–84. ISSN: 1359-8546. DOI: 10.1108/SCM-03-2018-0148. URL: https://doi.org/10.1108/SCM-03-2018-0148 (visited on 02/04/2022).

[Coi21] Coinmarketcap Alexandria. *Play2Earn (Play-to-Earn)*. https://coinmarketcap.com/alexandria/glossary/play2earn-play-to-earn. [Online; accessed 22/01/2022]. 2021.

[Gwy21a] Gwyneth Iredale. *6 Key Blockchain Features You Need to Know Now*. en-US. [Online; accessed 02/02/2022]. Nov. 2021. URL: https://101blockchains.com/introduction-to-blockchain-features/ (visited on 02/03/2022).

[Gwy21b] Gwyneth Iredale. *What Are The Different Types of Blockchain Technology?* en-US. [Online; accessed 04/02/2022]. Jan. 2021. URL: https://101blockchains.com/types-of-blockchain/ (visited on 02/04/2022).

[LT21a] Alexander Lipton and Adrien Treccani. *Blockchain and Distributed Ledgers Mathematics, Technology, and Economics*. World Scientific Publishing, 2021, pp. 60–61. ISBN: 9811221510.

[LT21b] Alexander Lipton and Adrien Treccani. *Blockchain and Distributed Ledgers Mathematics, Technology, and Economics*. World Scientific Publishing, 2021, pp. 61–62. ISBN: 9811221510.

[LT21c] Alexander Lipton and Adrien Treccani. *Blockchain and Distributed Ledgers Mathematics, Technology, and Economics*. World Scientific Publishing, 2021, p. 70. ISBN: 9811221510.

[LT21d] Alexander Lipton and Adrien Treccani. *Blockchain and Distributed Ledgers Mathematics, Technology, and Economics*. World Scientific Publishing, 2021, p. 205. ISBN: 9811221510.

[LT21e] Alexander Lipton and Adrien Treccani. *Blockchain and Distributed Ledgers Mathematics, Technology, and Economics*. World Scientific Publishing, 2021, p. 210. ISBN: 9811221510.

[LT21f] Alexander Lipton and Adrien Treccani. *Blockchain and Distributed Ledgers Mathematics, Technology, and Economics*. World Scientific Publishing, 2021, p. 211. ISBN: 9811221510.

[LT21g] Alexander Lipton and Adrien Treccani. *Blockchain and Distributed Ledgers Mathematics, Technology, and Economics*. World Scientific Publishing, 2021, p. 69. ISBN: 9811221510.

[Nat21] Nathan Reiff. *What Is ERC-20 and What Does It Mean for Ethereum?* https://www.investopedia.com/news/what-erc20-and-what-does-it-mean-ethereum/. [Online; accessed 22/01/2022]. 2021.

[Ada22] Adam Heyes. *Blockchain explained*. https://www.investopedia.com/terms/b/blockchain.asp/. [Online; accessed 02/02/2022]. 2022.

[Coma] Ethereum Community. *Account Types, Gas, and Transactions*. https://ethdocs.org/en/latest/contracts-and-transactions/account-types-gas-and-transactions.html. [Online; accessed 04/02/2022].

[Comb] Ethereum Community. *ERC-20 Token Standards*. https://ethereum.org/en/developers/docs/standards/tokens/erc-20/. [Online; accessed 05/02/2022; last edit 03/12/2021].

[Comc] Ethereum Community. *Ethereum Development Standards*. https://ethereum.org/en/developers/docs/standards/. [Online; accessed 05/02/2022; last edit 09/12/2021].

[Comd] Ethereum Community. *Events*. https://docs.soliditylang.org/en/v0.8.11/contracts.html#events. [Online; accessed 05/02/2022].

[Come] Ethereum Community. *Function Modifiers*. https://docs.soliditylang.org/en/v0.8.11/contracts.html#function-modifiers. [Online; accessed 05/02/2022].

[Comf] Ethereum Community. *Gas, Introduction to Smart Contracts*. https://ethereum.org/en/developers/docs/gas. [Online; accessed 04/02/2022; last edit 14/01/2022].

[Comg] Ethereum Community. *Laguage Influences*. https://docs.soliditylang.org/en/v0.8.11/language-influences.html. [Online; accessed 07/02/2022].

[Comh] Ethereum Community. *Smart Contracts, Introduction to Smart Contracts*. https://ethereum.org/en/developers/docs/smart-contracts. [Online; accessed 04/02/2022; last edit 10/01/2022].

[Comi] Ethereum Community. *Solidity*. https://docs.soliditylang.org/en/v0.8.11/. [Online; accessed 07/02/2022].

[Comj] Ethereum Community. *The merge*. https://ethereum.org/en/upgrades/merge. [Online; accessed 04/02/2022; last edit 01/02/2022].

[Comk] Ethereum Community. *Token Standards*. https://ethereum.org/en/developers/docs/standards/tokens/. [Online; accessed 05/02/2022; last edit 19/10/2021].

[Coml] Ethereum Community. *View Functions*. https://docs.soliditylang.org/en/v0.8.11/contracts.ht functions. [Online; accessed 06/02/2022].

[Comm] Ethereum Community. *What is Ethereum?* https://ethdocs.org/en/latest/introduction/what-is-ethereum.html. [Online; accessed 04/02/2022].

[Comn] Truffle Community. *Interacting with your contracts*. https://trufflesuite.com/docs/truffle/getting-started/interacting-with-your-contracts.html.

[ETH] ETHWORKS. *useDapp*. https://usedapp.io. [Online; accessed 07/02/2022].

[LS] L.S. "Who is Satoshi Nakamoto?" In: *The Economist* (). ISSN: 0013-0613. URL: https://www.economist.com/the-economist-explains/2015/11/02/who-is-satoshi-nakamoto (visited on 02/02/2022).

[Opea] OpenZeppelin. *_approve, ERC20.sol*. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol#L313. [Online; accessed 06/02/2022].

[Opeb] OpenZeppelin. *_burn, ERC20.sol*. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol#L283. [Online; accessed 06/02/2022].

[Opec] OpenZeppelin. *_grantRole, AccessControl.sol*. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/AccessControl.sol#L217. [Online; accessed 05/02/2022].

[Oped] OpenZeppelin. *_mint, ERC20.sol*. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol#L260. [Online; accessed 05/02/2022].

[Opee] OpenZeppelin. *_revokeRole, AccessControl.sol*. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/AccessControl.sol#L229. [Online; accessed 05/02/2022].

[Opef] OpenZeppelin. *_transfer, ERC20.sol*. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol#L229. [Online; accessed 06/02/2022].

[Opeg] OpenZeppelin. *AccessControl.sol*. https://docs.openzeppelin.com/contracts/4.x/api/access#AccessControl/. [Online; accessed 05/02/2022].

[Opeh] OpenZeppelin. *AccessControlEnumerable.sol*. https://docs.openzeppelin.com/contracts/4.x/api/access#AccessControlEnumerable/. [Online; accessed 05/02/2022].

[Opei] OpenZeppelin. *allowance mapping, ERC20.sol*. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol#L38. [Online; accessed 06/02/2022].

[Opej] OpenZeppelin. *allowance, ERC20.sol*. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol#L121. [Online; accessed 06/02/2022].

[Opek] OpenZeppelin. *approve, ERC20.sol*. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol#L135. [Online; accessed 06/02/2022].

[Opel] OpenZeppelin. *balanceof, ERC20.sol*. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol#L101. [Online; accessed 06/02/2022].

[Opem] OpenZeppelin. *burn, Burnable.sol*. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/extensions/ERC20Burnable.sol#L20. [Online; accessed 06/02/2022].

[Open] OpenZeppelin. *constructor, ERC20.sol*. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol#L54. [Online; accessed 07/02/2022].

[Opeo] OpenZeppelin. *ERC20Burnable.sol*. https://docs.openzeppelin.com/contracts/4.x/api/token/erc20#ERC20Burnable/. [Online; accessed 05/02/2022].

[Opep] OpenZeppelin. *getRoleMember, AccessControlEnumerable.sol*. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/AccessControlEnumerable.sol#L37. [Online; accessed 06/02/2022].

[Opeq] OpenZeppelin. *getRoleMemberCount, AccessControlEnumerable.sol*. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/AccessControlEnumerable.sol#L45. [Online; accessed 06/02/2022].

[Oper] OpenZeppelin. *grantRole, AccessControl.sol*. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/AccessControl.sol#L142. [Online; accessed 05/02/2022].

[Opes] OpenZeppelin. *Ownable*. https://docs.openzeppelin.com/contracts/4.x/api/access#Ownable/. [Online; accessed 05/02/2022].

[Opet] OpenZeppelin. *revokeRole, AccessControl.sol*. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/AccessControl.sol#L155. [Online; accessed 05/02/2022].

[Opeu] OpenZeppelin. *Tokens*. https://docs.openzeppelin.com/contracts/4.x/tokens. [Online; accessed 05/02/2022].

[Opev] OpenZeppelin. *totalsupply, ERC20.sol*. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol#L94. [Online; accessed 06/02/2022].

[Opew] OpenZeppelin. *transfer, ERC20.sol*. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol#L113. [Online; accessed 06/02/2022].

[Opex] OpenZeppelin. *transferfrom, ERC20.sol*. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol#L156. [Online; accessed 06/02/2022].

[Opey] OpenZeppelin. *Using Hooks*. https://docs.openzeppelin.com/contracts/4.x/extending-contracts#using-hooks. [Online; accessed 06/02/2022].

[Opez] OpenZeppelin. *We build technology to bring freedom to the world*. https://openzeppelin.com/about/. [Online; accessed 05/02/2022].

[Suia] Truffle Suite. *Creating a Project*. https://trufflesuite.com/docs/truffle/getting-started/creating-a-project.html. [Online; accessed 07/02/2022].

[Suib] Truffle Suite. *Ganache Overview*. https://trufflesuite.com/docs/ganache/index.html. [Online; accessed 07/02/2022].

[Suic] Truffle Suite. *Package Management via NPM*. https://trufflesuite.com/docs/truffle/getting-started/package-management-via-npm.html. [Online; accessed 07/02/2022].

[Suid] Truffle Suite. *Running Migrations*. https://trufflesuite.com/docs/truffle/getting-started/running-migrations.html. [Online; accessed 08/02/2022].

[Tea] TeamKeccak. *Keccak*. https://keccak.team/keccak.html. [Online; accessed 05/02/2022].