ERC 20

A MINI-PROJECT REPORT

Submitted by

ADITYA JAIN RA2011050010024
DHRUV MEHTA RA2011050010046
Studying
B. Tech

Under the Guidance of

Dr. SV. Shri Bharathi
Assistant Professor, Department of DSBS

DEPARTMENT OF DATA SCIENCE AND BUSINESS SYSTEMS

FACULTY OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND
TECHNOLOGY KATTANKULATHUR- 603 203

NOVEMBER 2022

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
KATTANKULATHUR – 603 203

BONAFIDE CERTIFICATE

Certified that this B.Tech mini-project report titled ERC 20 is the bonafide work of **Aditya Jain and Dhruv Mehta** who carried out the project work under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form part of any other thesis or dissertation on the basis of which a degree or award was conferred on an earlier occasion for this or any other candidate.

Dr. SV. SHRI BHARATHI
SUPERVISOR
Assistant Professor
Department of DSBS

Dr. M. Lakshmi
PROFESSOR & HOD
Department of DSBS

Signature of Internal Examiner **Signature of External Examiner**

ABSTRACT

In Ethereum, an ERC is an *Ethereum Request for Comments*. These are technical documents that outline standards for programming on Ethereum.

They're not to be confused with Ethereum Improvement Proposals (EIPs), which, like Bitcoin's BIPs, suggest improvements to the protocol itself.

ERCs instead aim to establish conventions that make it easier for applications and contracts to interact with each other.

Authored by Vitalik Buterin and Fabian Vogelsteller in 2015, ERC-20 proposes a relatively simple format for Ethereum-based tokens.

Once new ERC-20 tokens are created, they're automatically interoperable with services and software supporting the ERC-20 standard (software wallets, hardware wallets, exchanges, etc.).

TABLE OF CONTENTS

CHAPTER 1
INTRODUCTION

What makes ERC20 tokens so attractive and successful? There are several factors in play:

- ERC20 tokens are simple and easy to deploy, as you will see in this tutorial.

- The ERC20 standard solves a significant problem, as blockchain-based marketplaces and crypto-wallets need a single, standardized set of commands to communicate with the range of tokens they manage. This includes interaction rules between different tokens, as well as token purchase rules.

- It was the first popular specification to offer Ethereum token standardization. It was not by any means *the first*, but thanks to its popularity, it quickly became the industry standard.

Just like other Ethereum tokens, ERC20 tokens are implemented as smart contracts and executed on the Ethereum Virtual Machine (EVM) in a decentralized manner.

Ethereum smart contracts are written in Solidity. While there are alternative languages, hardly anyone uses them for this purpose.

Chapter 3
SYSTEM ANALYSIS

1. PROBLEMSTATEMENT:

The following standard allows for the implementation of a standard API for tokens within smart contracts. This standard provides basic functionality to transfer tokens, as well as allow tokens to be approved so they can be spent by another on-chain third party. A standard interface allows any tokens on Ethereum to be re-used by other applications: from wallets to decentralized exchanges.

2. PROPOSED SOLUTION:

An ERC20 token contract keeps track of *fungible* tokens: any one token is exactly equal to any other token; no tokens have special rights or behavior associated with them. This makes ERC20 tokens useful for things like a **medium of exchange currency**, **voting rights**, **staking**, and more.

OpenZeppelin Contracts provides many ERC20-related contracts. On the API reference you'll find detailed information on their properties and usage.

Solidity and the EVM do not support this behavior: only integer (whole) numbers can be used, which poses an issue. You may send `1` or `2` tokens, but not `1.5`.

3. SOFTWARE and HARDWARES

1. Software Requirements

Operating System: Windows 8/9/10, Linux Ubuntu/ MAC OS

Tools: VS Code, MetaMask, MangoDB

2. Hardware requirements: NoneProcessor: intel core i5 10th Gen with Nvidia 2gb graphics

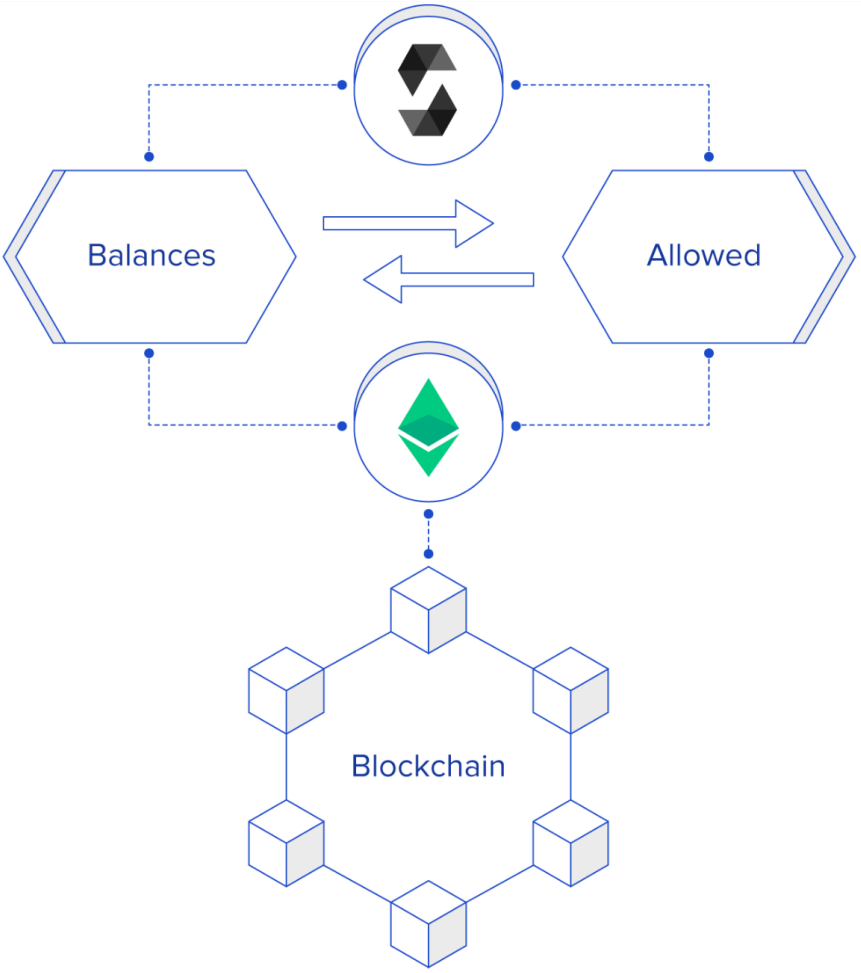
Hard disk: SK Hynix BC511

HFM512GDJTNI-82A0ARAM: 4GB/8GB

Chapter 4

SYSTEM DESIGN AND IMPLEMENTATION

ARCHITECTURE DIAGRAM:



Description of Architecture Diagram:

Put simply, the ERC20 standard defines a set of functions to be implemented by all ERC20 tokens so

```
function totalSupply() public view returns (uint256);
function balanceOf(address tokenOwner) public view returns (uint);
function allowance(address tokenOwner, address spender)
public view returns (uint);
function transfer(address to, uint tokens) public returns (bool);
function approve(address spender, uint tokens)  public returns (bool);
function transferFrom(address from, address to, uint tokens) public returns (bool);
```

as to allow integration with other contracts, wallets, or marketplaces. This set of functions is rather

ERC20 functions allow an external user, say a crypto-wallet app, to find out a user's balance and transfer funds from one user to another with proper authorization.

The smart contract defines two specifically defined events:

```
event Approval(address indexed tokenOwner, address indexed spender,
 uint tokens);
event Transfer(address indexed from, address indexed to,
 uint tokens);
```

These events will be invoked or *emitted* when a user is granted rights to withdraw tokens from an account, and after the tokens are actually transferred.

In addition to standard ERC20 functions, many ERC20 tokens also feature additional fields and some have become a de-facto part of the ERC20 standard, if not in writing then in practice. Here are a few examples of such fields.

Here are a few points regarding ERC20 and Solidity nomenclature:

- A `public` function can be accessed outside of the contract itself
- `view` basically means constant, i.e. the contract's internal state will not be changed by the function
- An `event` is Solidity's way of allowing clients e.g. your application frontend to be notified on specific occurrences within the contract

Most Solidity language constructs should be clear if you already possess essential Java/JavaScript skills.

FRONT END

HTML:

The Hypertext Markup Language or HTML is the standard markup language for documents designed to be displayed in a web browser. It can be assisted by technologies suchas Cascading Style Sheets (CSS) and scripting languages such as JavaScript.

Web browsers receive HTML documents from a web server or from local storage and render the documents into multimedia web pages. HTML describes the structure of a webpage semantically and originally included cues for the appearance of the document.

HTML elements are the building blocks of HTML pages. With HTML constructs, images, and other objects such as interactive forms may be embedded into the rendered page. HTML provides a means to create structured documents by denoting structural semantics for text such as headings, paragraphs, lists, links, quotes, and other items. HTML elements are delineated by *tags*, written using angle brackets.

CSS:

Cascading Style Sheets (**CSS**) is a style sheet language used for describing the presentation of a document written in a markup language such as HTML or XML (including XML dialects such as SVG, MathML or XHTML).CSS is a cornerstone technology of the World Wide Web, alongside HTML and JavaScript.

CSS is designed to enable the separation of content and presentation, including layout, colors,and fonts. This separation can improve content accessibility; provide more flexibility and control in the specification of presentation characteristics; enable multiple web pages to share formatting by specifying the relevant CSS in a separate .css file, which reduces complexity and repetition in the structural content; and enable the .css file to be cached to improve the page load speed between the pages that share the file and its formatting.

JAVASCRIPT:

JavaScript is a text-based programming language used both on the client-side and server-sidethat allows you to make web pages interactive. Where HTML and CSS are languages that give structure and style to web pages, JavaScript gives web pages interactive elements that engage a user. Common examples of JavaScript that you might use every day include the search boxon Amazon, a news recap video embedded on The New York Times, or refreshing your Twitter feed.

BACK END

SOLIDITY:

Solidity is an object-oriented programming language for implementing smart contracts on various blockchain platforms, most notably, Ethereum. It was developed by Christian Reitwiessner, Alex Beregszaszi, and several former Ethereum core contributors. Programs in Solidity run on Ethereum Virtual Machine.

COMPILER

REMIX:

Remix IDE, is a no-setup tool with a GUI for developing smart contracts.
Used by experts and beginners alike, Remix will get you going in double time. Remix plays well with other tools, and allows for a simple deployment process to the chain of your choice. Remix is famous for our visual debugger. Remix is the place everyone comes to learn Ethereum.

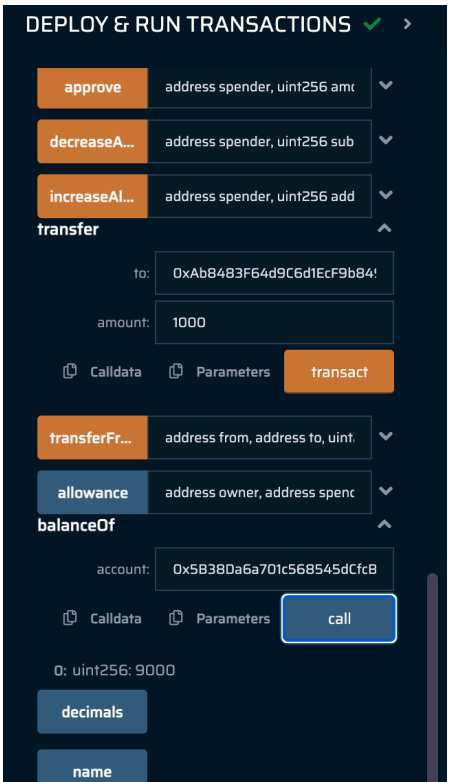
CODE AND OUTPUT EXPLANATION

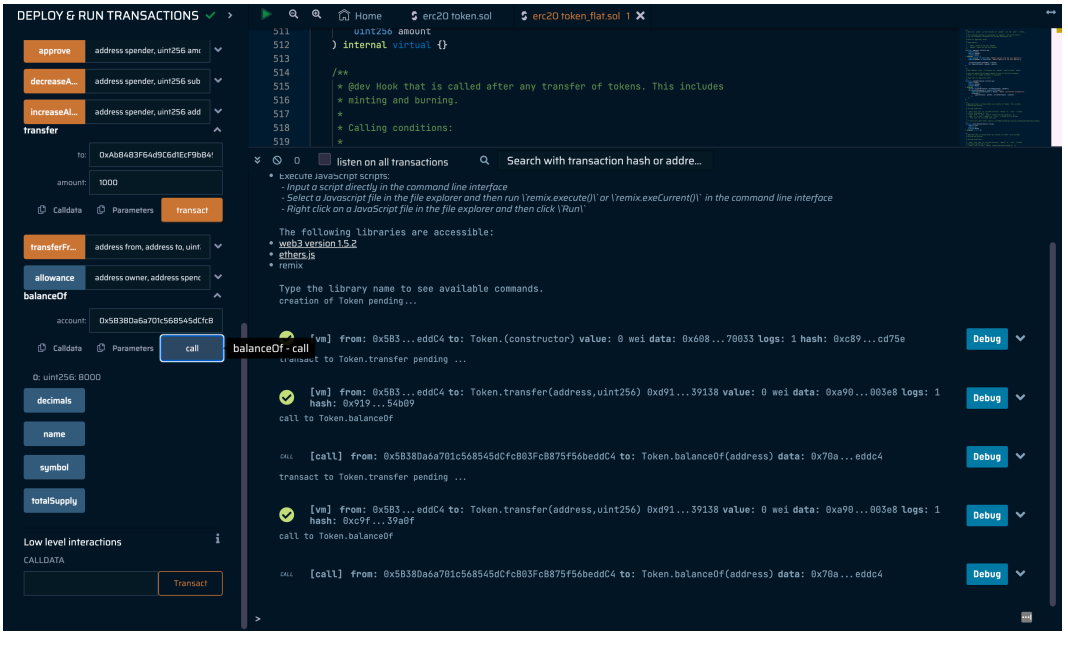
```
// File: @openzeppelin/contracts/utils/Context.sol


// OpenZeppelin Contracts v4.4.1 (utils/Context.sol)

pragma solidity ^0.8.0;

contract ERC20 is Context, IERC20, IERC20Metadata {
    mapping(address => uint256) private _balances;

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

    uint256 private _totalSupply;

    string private _name;
    string private _symbol;

    /**
     * @dev Sets the values for {name} and {symbol}.
     *
     * The default value of {decimals} is 18. To select a different value for
     * {decimals} you should overload it.
     *
     * All two of these values are immutable: they can only be set once during
     * construction.
     */
    constructor(string memory name_, string memory symbol_) {
        _name = name_;
        _symbol = symbol_;
    }

    /**
     * @dev Returns the name of the token.
     */
    function name() public view virtual override returns (string memory) {
        return _name;
    }

    /**
     * @dev Returns the symbol of the token, usually a shorter version of the
     * name.
     */
    function symbol() public view virtual override returns (string memory) {
        return _symbol;
    }

    /**
     * @dev Returns the number of decimals used to get its user representation.
     * For example, if `decimals` equals `2`, a balance of `505` tokens should
     * be displayed to a user as `5.05` (`505 / 10 ** 2`).
     *
     * Tokens usually opt for a value of 18, imitating the relationship between
     * Ether and Wei. This is the value {ERC20} uses, unless this function is
     * overridden;
     *
     * NOTE: This information is only used for _display_ purposes: it in
     * no way affects any of the arithmetic of the contract, including
     * {IERC20-balanceOf} and {IERC20-transfer}.
     */
    function decimals() public view virtual override returns (uint8) {
```

```
        return 18;
    }

    /**
     * @dev See {IERC20-totalSupply}.
     */
    function totalSupply() public view virtual override returns (uint256) {
        return _totalSupply;
    }

    /**
     * @dev See {IERC20-balanceOf}.
     */
    function balanceOf(address account) public view virtual override returns (uint256) {
        return _balances[account];
    }

    /**
     * @dev See {IERC20-transfer}.
     *
     * Requirements:
     *
     * - `to` cannot be the zero address.
     * - the caller must have a balance of at least `amount`.
     */
    function transfer(address to, uint256 amount) public virtual override returns (bool) {
        address owner = _msgSender();
        _transfer(owner, to, amount);
        return true;
    }

    /**
     * @dev See {IERC20-allowance}.
     */
    function allowance(address owner, address spender) public view virtual override returns
(uint256) {
        return _allowances[owner][spender];
    }

    /**
     * @dev See {IERC20-approve}.
     *
     * NOTE: If `amount` is the maximum `uint256`, the allowance is not updated on
     * `transferFrom`. This is semantically equivalent to an infinite approval.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     */
    function approve(address spender, uint256 amount) public virtual override returns (bool) {
        address owner = _msgSender();
        _approve(owner, spender, amount);
        return true;
    }

    /**
     * @dev See {IERC20-transferFrom}.
     *
     * Emits an {Approval} event indicating the updated allowance. This is not
     * required by the EIP. See the note at the beginning of {ERC20}.
     *
     * NOTE: Does not update the allowance if the current allowance
```

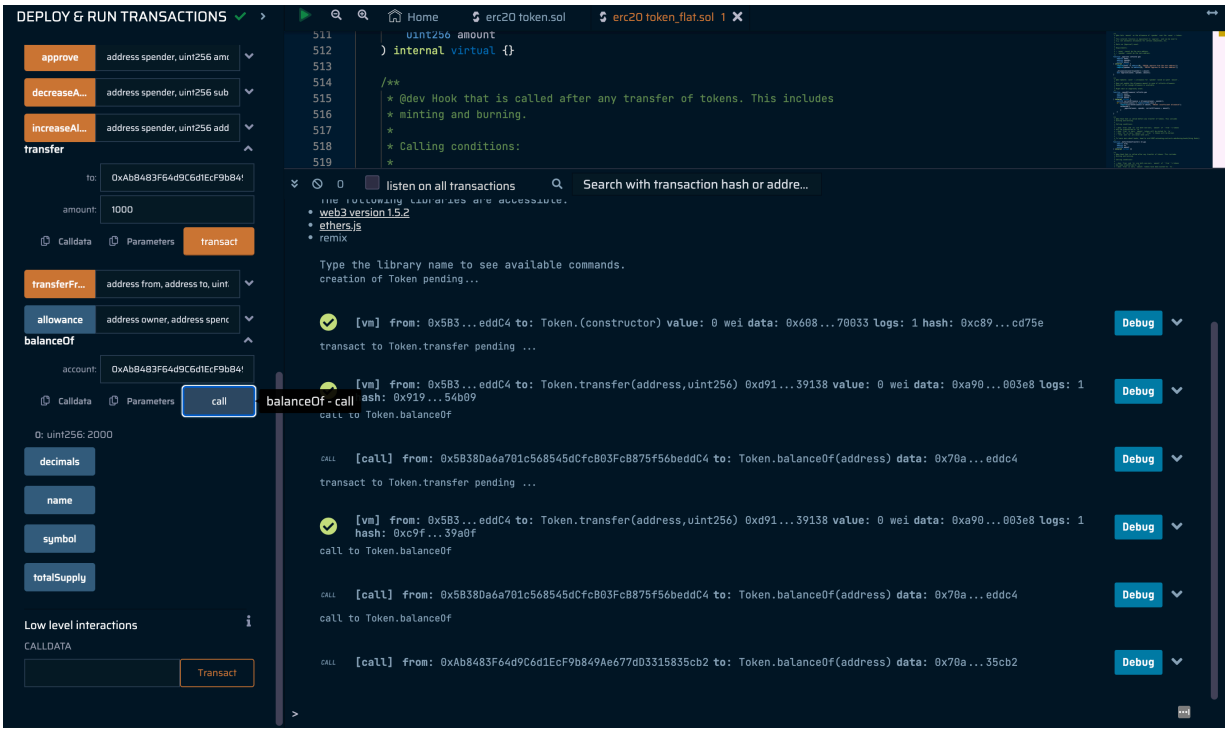
```
     * is the maximum `uint256`.
     *
     * Requirements:
     *
     * - `from` and `to` cannot be the zero address.
     * - `from` must have a balance of at least `amount`.
     * - the caller must have allowance for ``from``'s tokens of at least
     * `amount`.
     */
    function transferFrom(
        address from,
        address to,
        uint256 amount
    ) public virtual override returns (bool) {
        address spender = _msgSender();
        _spendAllowance(from, spender, amount);
        _transfer(from, to, amount);
        return true;
    }

    /**
     * @dev Atomically increases the allowance granted to `spender` by the caller.
     *
     * This is an alternative to {approve} that can be used as a mitigation for
     * problems described in {IERC20-approve}.
     *
     * Emits an {Approval} event indicating the updated allowance.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     */
    function increaseAllowance(address spender, uint256 addedValue) public virtual returns
(bool) {
        address owner = _msgSender();
        _approve(owner, spender, allowance(owner, spender) + addedValue);
        return true;
    }

    /**
     * @dev Atomically decreases the allowance granted to `spender` by the caller.
     *
     * This is an alternative to {approve} that can be used as a mitigation for
     * problems described in {IERC20-approve}.
     *
     * Emits an {Approval} event indicating the updated allowance.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     * - `spender` must have allowance for the caller of at least
     * `subtractedValue`.
     */
    function decreaseAllowance(address spender, uint256 subtractedValue) public virtual
returns (bool) {
        address owner = _msgSender();
        uint256 currentAllowance = allowance(owner, spender);
        require(currentAllowance >= subtractedValue, "ERC20: decreased allowance below
zero");
        unchecked {
            _approve(owner, spender, currentAllowance - subtractedValue);
        }
```

```
        return true;
    }

    /**
     * @dev Moves `amount` of tokens from `from` to `to`.
     *
     * This internal function is equivalent to {transfer}, and can be used to
     * e.g. implement automatic token fees, slashing mechanisms, etc.
     *
     * Emits a {Transfer} event.
     *
     * Requirements:
     *
     * - `from` cannot be the zero address.
     * - `to` cannot be the zero address.
     * - `from` must have a balance of at least `amount`.
     */
    function _transfer(
        address from,
        address to,
        uint256 amount
    ) internal virtual {
        require(from != address(0), "ERC20: transfer from the zero address");
        require(to != address(0), "ERC20: transfer to the zero address");

        _beforeTokenTransfer(from, to, amount);

        uint256 fromBalance = _balances[from];
        require(fromBalance >= amount, "ERC20: transfer amount exceeds balance");
        unchecked {
            _balances[from] = fromBalance - amount;
            // Overflow not possible: the sum of all balances is capped by totalSupply, and the
sum is preserved by
            // decrementing then incrementing.
            _balances[to] += amount;
        }

                emit Transfer(from, to, amount);

        _afterTokenTransfer(from, to, amount);
    }
```





/** @dev Creates `amount` tokens and assigns them to `account`, increasing



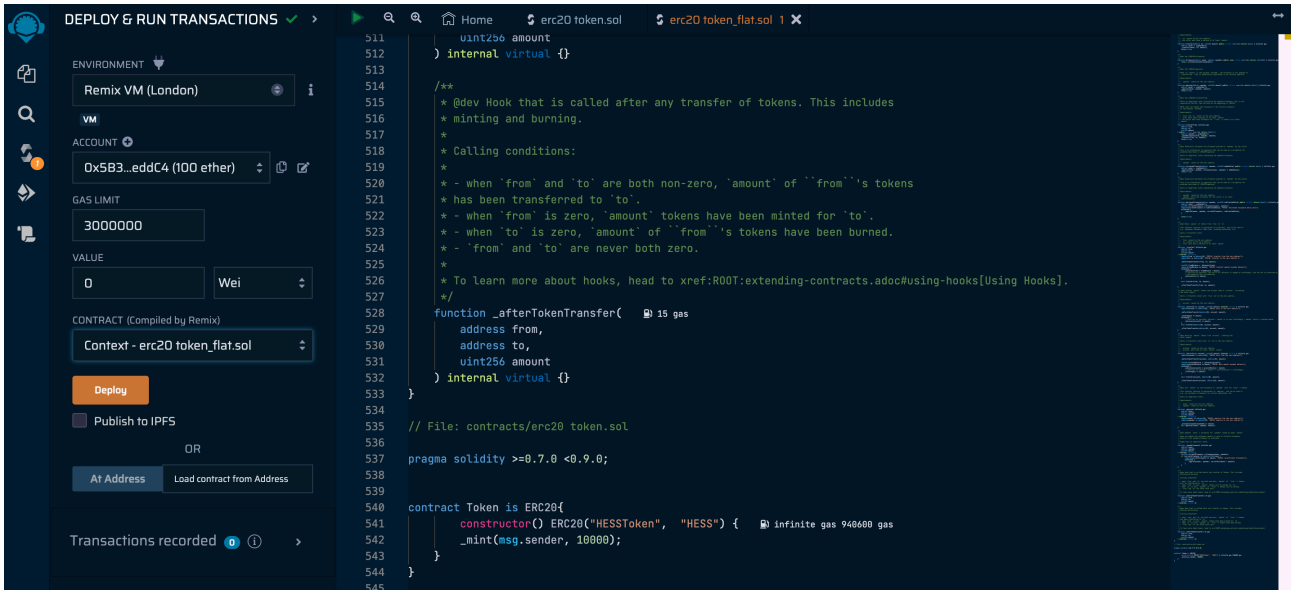

```
     * the total supply.
     *
     * Emits a {Transfer} event with `from` set to the zero address.
     *
     * Requirements:
     *
     * - `account` cannot be the zero address.
     */
    function _mint(address account, uint256 amount) internal virtual {
        require(account != address(0), "ERC20: mint to the zero address");

        _beforeTokenTransfer(address(0), account, amount);

        _totalSupply += amount;
        unchecked {
            // Overflow not possible: balance + amount is at most totalSupply + amount, which is
checked above.
            _balances[account] += amount;
        }
        emit Transfer(address(0), account, amount);

        _afterTokenTransfer(address(0), account, amount);
    }

    /**
     * @dev Destroys `amount` tokens from `account`, reducing the
     * total supply.
     *
     * Emits a {Transfer} event with `to` set to the zero address.
     *
     * Requirements:
     *
     * - `account` cannot be the zero address.
     * - `account` must have at least `amount` tokens.
     */
```

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

    _beforeTokenTransfer(account, address(0), amount);

    uint256 accountBalance = _balances[account];
    require(accountBalance >= amount, "ERC20: burn amount exceeds balance");
    unchecked {
        _balances[account] = accountBalance - amount;
        // Overflow not possible: amount <= accountBalance <= totalSupply.
        _totalSupply -= amount;
    }

    emit Transfer(account, address(0), amount);

    _afterTokenTransfer(account, address(0), amount);
}

/**
 * @dev Sets `amount` as the allowance of `spender` over the `owner` s tokens.
 *
 * This internal function is equivalent to `approve`, and can be used to
 * e.g. set automatic allowances for certain subsystems, etc.
 *
 * Emits an {Approval} event.
 *
 * Requirements:
 *
 * - `owner` cannot be the zero address.
 * - `spender` cannot be the zero address.
 */
function _approve(
    address owner,
    address spender,
    uint256 amount
) internal virtual {
    require(owner != address(0), "ERC20: approve from the zero address");
    require(spender != address(0), "ERC20: approve to the zero address");

    _allowances[owner][spender] = amount;
    emit Approval(owner, spender, amount);
}

/**
 * @dev Updates `owner` s allowance for `spender` based on spent `amount`.
 *
 * Does not update the allowance amount in case of infinite allowance.
 * Revert if not enough allowance is available.
 *
 * Might emit an {Approval} event.
 */
function _spendAllowance(
    address owner,
    address spender,
    uint256 amount
) internal virtual {
    uint256 currentAllowance = allowance(owner, spender);
    if (currentAllowance != type(uint256).max) {
        require(currentAllowance >= amount, "ERC20: insufficient allowance");
        unchecked {
            _approve(owner, spender, currentAllowance - amount);
        }
```

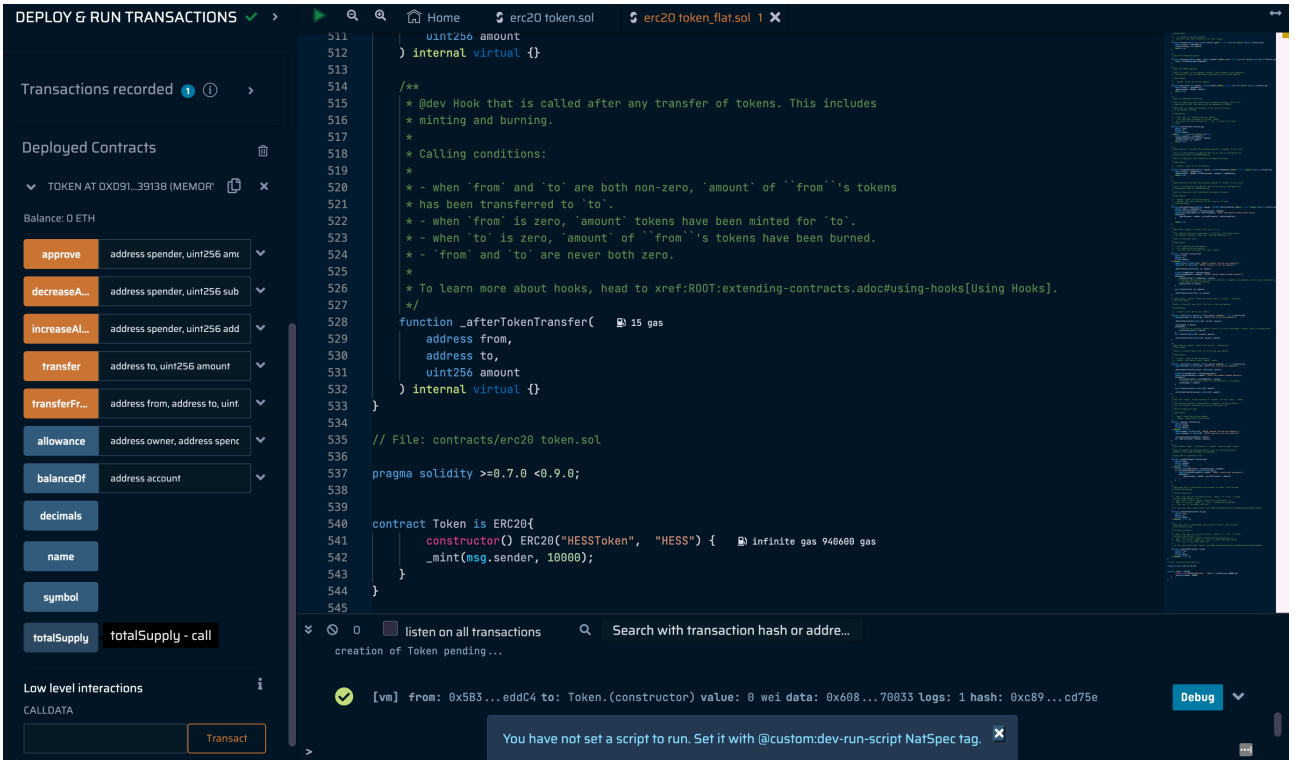
```
        }
    }

    function _beforeTokenTransfer(
        address from,
        address to,
        uint256 amount
    ) internal virtual {}

    function _afterTokenTransfer(
        address from,
        address to,
        uint256 amount
    ) internal virtual {}
}
```

```
// File: contracts/erc20 token.sol

pragma solidity >=0.7.0 <0.9.0;


contract Token is ERC20{
        constructor() ERC20("HESSToken",  "HESS") {    infinite gas 940600 gas
        _mint(msg.sender, 10000);
    }
}
```

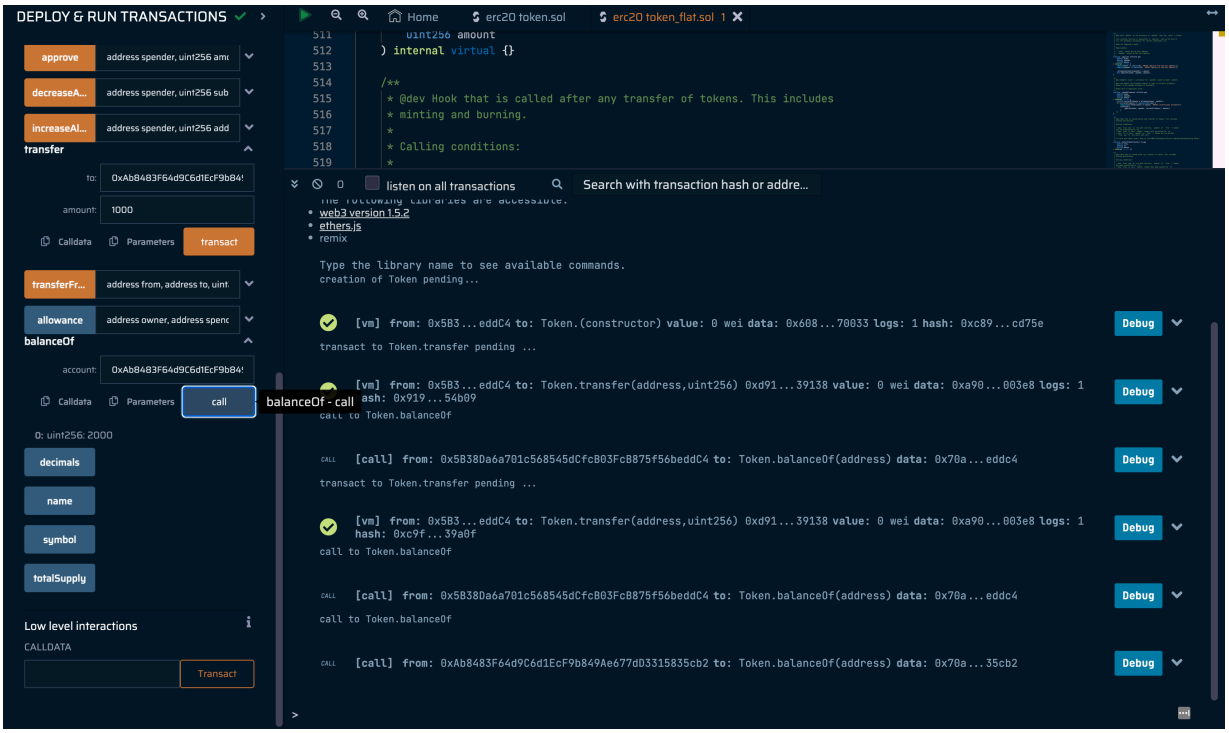
Final code.



Before Deploying the contract



After Deployed contract



FINAL SUBMISSION AFTER DEPLOYED INSTANCE

Chapter 5

CONCLUSIONS

ERC-20 was the first (and, to date, the most popular) Ethereum token standard, but it's by no means the only one. Over the years, many others have emerged, either proposing improvements on ERC-20 or attempting to achieve different goals altogether.

Some of the less common standards are the ones used in non-fungible tokens (NFTs). Sometimes, your use case actually benefits from having unique tokens with different attributes. If you wanted to tokenize a one-of-a-kind piece of art, in-game asset, etc., one of these contract types might be more appealing.

The ERC-20 standard has dominated the crypto asset space for years, and it's not hard to see why. With relative ease, anyone can deploy a simple contract to suit a wide range of use cases (utility tokens, stablecoins, etc.). That said, ERC-20 does lack some of the features brought to life by other standards. It remains to be seen whether subsequent types of contracts will take its place.

REFERENCES

- https://eips.ethereum.org/EIPS/eip-20
- https://remix.ethereum.org/
- https://github.com/ethereum/wiki/wiki/.
- Standardized_Contract_APIs/499c882f3ec123537fc2fccd57eaa29e6032fe4a
- Reddit discussion: https://www.reddit.com/r/ethereum/comments/3n8fkn/lets_talk_about_the_coin_standard/