Project Report

Abstract

This project implements a blockchain-based digital asset marketplace using Solidity for smart contract development and a React-based frontend with Bootstrap and Iconsax. The primary objective is to enable users to register, manage, and trade digital assets securely. The project integrates Ethers.js, Hardhat, Waffle, and Truffle for development and testing, with Remix IDE used for contract verification. While functional, this marketplace is a small-scale project and does not meet enterprise security standards.

Introduction

The protection and ownership verification of digital assets has become crucial in the era of Web3 and decentralized finance. Blockchain technology offers immutable, transparent, and secure solutions to register and trade assets. This project aims to demonstrate how a smart contract can be leveraged to create a digital asset marketplace, allowing users to register assets via the payToMint function, retrieve asset information (hashes, timestamps, and metadata URIs), and trade assets between users. However, due to the use of setApprovalForAll, all users are automatically approved, presenting security risks.

Methodology

Design Decisions

- **Smart Contract:** Developed using Solidity, with OpenZeppelin libraries for security and token management.
- **Frontend:** Built with React, Bootstrap, and Iconsax for an intuitive user experience.
- **Blockchain Interaction:** Utilizes Ethers.js for seamless communication with the Ethereum network.
- **Testing Frameworks:** Hardhat, Waffle, and Truffle ensure contract functionality and security.

Architecture

1. **Frontend (React + Bootstrap)**

    - User interface for asset registration, viewing, and transactions.
    - Integrates Ethers.js for blockchain interactions.
2. **Smart Contract (Solidity)**
    - Implements ERC-721 tokens to represent digital assets.
    - Tracks asset ownership, metadata, and pricing.
    - Handles asset trading through the buyToken function.
3. **Blockchain Transactions**
    - Users interact with smart contracts to mint and trade assets.
    - Transactions are recorded on the Ethereum blockchain for transparency.

Implementation

Smart Contract Structure

```solidity
// SPDX-License-Identifier: MIT
// Compatible with OpenZeppelin Contracts ^5.0.0
pragma solidity ^0.8.22;

import {ERC721} from "@openzeppelin/contracts/token/ERC721/ERC721.sol";
import {ERC721URIStorage} from "@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";
import {Ownable} from "@openzeppelin/contracts/access/Ownable.sol";
import {IERC721Receiver} from "@openzeppelin/contracts/token/ERC721/IERC721Receiver.sol";
import "@openzeppelin/contracts/token/ERC721/IERC721.sol";

contract Krypton is ERC721, ERC721URIStorage, Ownable, IERC721Receiver {
    uint256 private _nextTokenId = 1;

    mapping(string => uint8) existingURIs;
    mapping(uint256 => uint256) public tokenPrices;

    mapping(uint256 => bytes32) public tokenHashes;      // Stores token hash
    mapping(uint256 => uint256) public tokenTimestamps; // Stores last update timestamp
    mapping(uint256 => uint256) private transferCounts;

    uint256 public alertThreshold = 3; // Example: More than 3 transfers in a short time

    event NFTMinted(address indexed owner, uint256 tokenId, string metadataURI);
    event NFTPurchased(address indexed buyer, address indexed seller, uint256 tokenId, uint256 price);
    event SuspiciousActivityDetected(uint256 tokenId, address indexed owner);
    event ReceivedETH(address sender, uint256 amount);

    constructor(address initialOwner)
        ERC721("Krypton", "KTN")
        Ownable()
    {
        _transferOwnership(initialOwner); // Set the owner
    }
```

- payToMint: Mints a new NFT when a user pays the required fee.

```
function payToMint(address recipient, string memory metadataURI)  public payable returns (uint256) {
    require(existingURIs[metadataURI] == 0, "URI already used");
    require (msg.value >= 0.05 ether, "Need to pay up!");
    existingURIs[metadataURI] = 1;
    uint256 tokenId = _nextTokenId++;
    _safeMint(recipient, tokenId);
    _setTokenURI(tokenId, metadataURI);

    tokenTimestamps[tokenId] = block.timestamp; // Store minting time
    tokenHashes[tokenId] = keccak256(abi.encodePacked(tokenId, recipient, block.timestamp)); // Store hash

    emit NFTMinted(recipient, tokenId, metadataURI);
    return tokenId;
}
```

- setTokenPrice: Allows users to set a price for their assets.

```
function setTokenPrice(uint256 tokenId, uint256 price) public onlyTokenOwner(tokenId) {
    tokenPrices[tokenId] = price;
}
```

- buyToken: Enables users to purchase assets from others.

```
function buyToken(uint256 tokenId) public payable {
    uint256 price = tokenPrices[tokenId];
    address seller = ownerOf(tokenId);

    require(price > 0, "Token not for sale");
    require(msg.value >= price, "Insufficient funds");

    // Transfer funds to the seller
    (bool success, ) = payable(seller).call{value: price}("");
    require(success, "Transfer failed");
    // Transfer ownership of the token
    safeTransferFrom(seller, msg.sender, tokenId);

    transferCounts[tokenId]++;

    // Suspicious activity detection
    if (transferCounts[tokenId] > alertThreshold) {
        emit SuspiciousActivityDetected(tokenId, msg.sender);
    }

    // Clear the sale price after purchase
    tokenPrices[tokenId] = 0;

    emit NFTPurchased(msg.sender, seller, tokenId, price);
}
```

- getTokenHash: Retrieves the hash of a specific token.

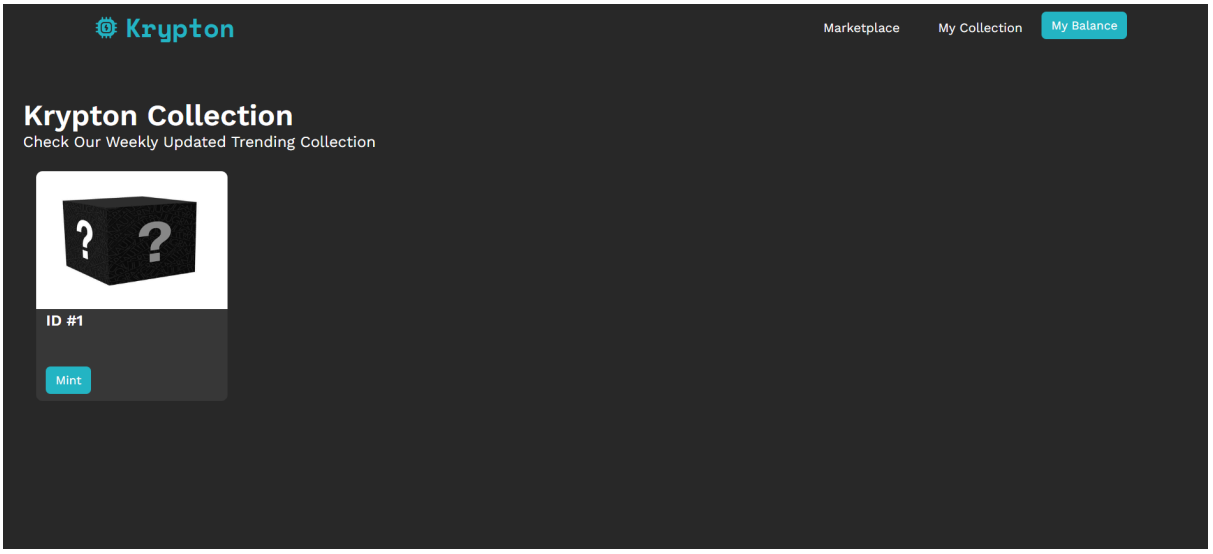
```
function getTokenHash(uint256 tokenId) public view returns (bytes32) {
    require(_exists(tokenId), "Token does not exist");
    return tokenHashes[tokenId];
}
```

- getTokenTimestamp: Fetches the timestamp of the asset's last transaction.

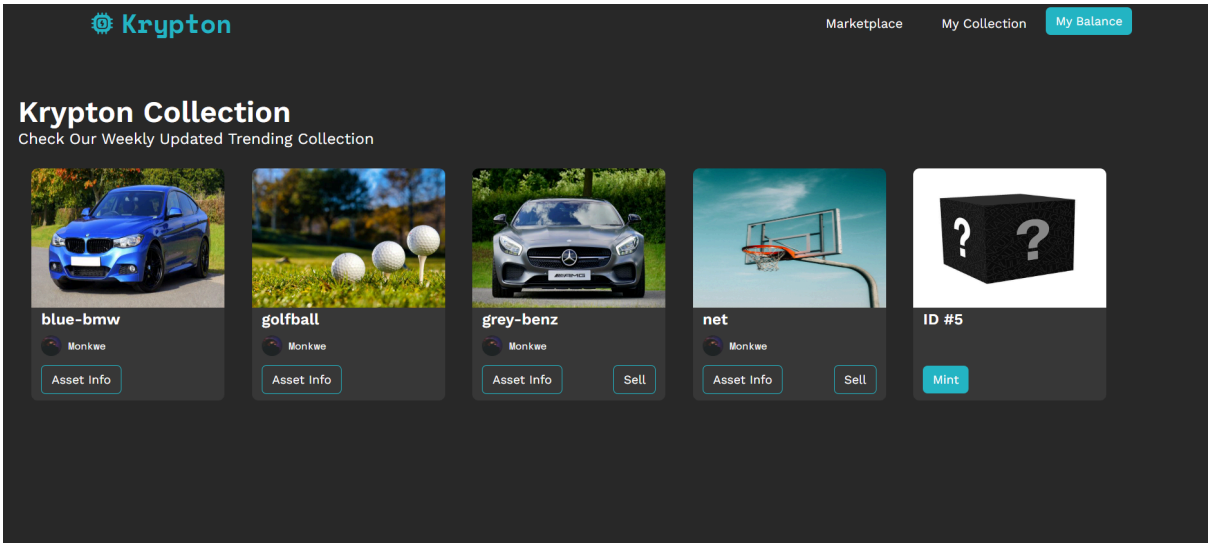
```
function getTokenTimestamp(uint256 tokenId) public view returns (uint256) {
    require(_exists(tokenId), "Token does not exist");
    return tokenTimestamps[tokenId];
}
```

Frontend UI Overview

- **Asset Registration Page:** Users submit metadata to mint NFTs.



- **Asset Marketplace:** Displays available assets for purchase.



GitHub Repository

[GitHub Repo Link]

Limitations & Security Considerations

- The project is a small-scale implementation and lacks enterprise-level security.
- All users are approved via setApprovalForAll, making unauthorized transfers possible.
- No advanced authentication or security measures are enforced.
- It should not be used in a real-world production environment without security improvements.