Documentation for NFT Wars Project

Introduction

NFT Wars Project, a collection of Solidity smart contracts developed by Berke Kiran. This project enables users to mint and manage ERC1155 NFTs using the `NFTContract` and engage in battles between NFTs with the `BattleContract`. This documentation covers both contracts and their functionalities.

Contracts Overview

- **NFTContract.sol**
  Purpose: The NFTContract allows users to mint ERC1155 tokens, manage token properties, set mint prices, and more.
  Inherits From: ERC1155 (from OpenZeppelin), AccessControl (from OpenZeppelin).
  Roles: ADMIN, FRIEND_CONTRACT.

- **BattleContract.sol**
  Purpose: The BattleContract enables NFT owners to engage in battles between two NFTs, determining a winner based on pseudo-random numbers.
  Inherits From: None.
  Dependencies: Relies on the INFTContract interface for NFT-related functions.

Usage

NFTContract

- **Minting NFTs**
  Use the mint() function to mint a single token.
  Use the mintBatch(amount) function to mint multiple tokens in a single transaction.
- **Burning NFTs**
  Use the burn(account, tokenId) function to burn tokens from an account.
- **Setting Mint Price**
  Use the setMintPrice(tokenId, price) function to set the mint price for a specific token.
- **Setting NFT Score**
  Use the setScore(tokenId, score) function to set the score for a specific token.
- **Other Functions**
  uri(tokenId): Returns the URI for a given token ID.
  supportsInterface(interfaceId): Checks if the contract supports a specific interface.
  getScore(tokenId): Gets the score for a given token ID.
  totalSupply(): Returns the total supply of tokens.

BattleContract

- **Constructor**
  Initialize the BattleContract by providing the address of the NFTContract interface.
- **Battle Functio**n
  Use the battle(firstNftId, secondNftId) function to initiate battles between two NFTs. The winner is determined based on pseudo-random numbers.

Events

The following events are emitted by the contracts in this project:

NFTContract Events

- **NFTMinted(tokenId, minter)**: Emitted when a single token is minted, indicating the tokenId and the minter address.
- **NFTBatchMinted(firstTokenId, lastTokenId, minter)**: Emitted when a batch of tokens is minted, indicating the range of tokenIds and the minter address.
- **NFTBurned(tokenId, burner)**: Emitted when a token is burned, indicating the tokenId and the burner address.
- **MintPriceUpdated(tokenId, price)**: Emitted when the mint price for a specific token is updated, indicating the tokenId and the new price.
- **ScoreChanged(tokenId, score)**: Emitted when the score for a specific token is changed, indicating the tokenId and the new score.
- **Withdrawn(amount, receiver)**: Emitted when contract balance in ETH is withdrawn, indicating the amount and the receiver address.

BattleContract Events

- **BattleCompleted(firstNftId, secondNftId, winnerNftId, caller)**: Emitted when a battle is completed in the BattleContract, indicating the firstNftId, secondNftId, winnerNftId, and the caller address.

These events provide essential information about the interactions and state changes that occur within the contracts, enabling users and developers to monitor and react to contract activities.

Compiler Version and Optimizations

This smart contracts was developed using Solidity version 0.8.18 with compiler optimizations enabled. The Solidity compiler settings used include an enabled optimizer with 200 runs, which helps ensure efficient contract execution on the Polygon Mumbai Testnet blockchain.

Installation and Setup

1. git clone [repository_url]
2. cd [project_directory]
3. Create an .env file and add the following line, replacing "your_wallet_private_key" with your actual wallet private key: PRIVATE_KEY=your_wallet_private_key
4. yarn or yarn install
5. npx hardhat compile
6. npx hardhat test

Deploying Contracts

To use the NFT Battle Project, you'll need to deploy the NFTContract and BattleContract contracts to an Polygon Mumbai Testnet network. This section provides step-by-step instructions on how to do this using the provided deployment script.

Deployment Steps

1. Open your terminal and navigate to the project directory.
2. Ensure that your Polygon Mumbai Testnet development environment is properly configured.
3. Run the deployment script using the following command:
   npx hardhat run scripts/deploy.ts --network polygon_mumbai

This command will execute the deployment script (deploy.js) located in the scripts directory.

Once the script has completed, you will see the addresses where both the NFTContract and BattleContract have been deployed.

You can now interact with the contracts using their respective addresses.

Deployed Contracts

The following contracts have been successfully deployed on the Polygon Mumbai Testnet.

NFT Contract

Address: 0xEd3ce7E110B96F53c9cC57ef914Dfa6802567d2b

Battle Contract

Address: 0x1f274D1C31D2c3473D375D3fEE9265FB95464dee

Unit Testing

Before deploying the contracts to a live Polygon Mumbai Testnet network, it's essential to thoroughly test the smart contracts to ensure they function as expected. This section describes how to run unit tests for the NFT Contract and Battle Contract using Hardhat and Chai.

Running Unit Tests

To run the unit tests for the NFT Contract and Battle Contract, follow these steps:

1. Open your terminal and navigate to the project directory.
2. Ensure that your Polygon Mumbai Testnet development environment is correctly configured.
3. Execute the following command to run the unit tests:
   npx hardhat test

This command will execute all the tests defined in the test files located in the test directory. The tests are written using Chai assertions and Hardhat for contract deployment and interaction.

Wait for the tests to complete. You will see the test results in your terminal, indicating whether each test passed or failed.

Contracts

NFT Contract

✔ Should mint an NFT
✔ Should not allow minting an NFT that would exceed the maximum supply
✔ Should not allow minting an NFT with insufficient funds
✔ Should emit NFT Minted event with right data when mint an NFT
✔ Should mint a batch of 100 NFTs
✔ Should not allow minting of zero amount
✔ Should not allow minting a batch of NFTs that would exceed the maximum supply
✔ Should not allow minting a batch of NFTs with insufficient funds
✔ Should emit NFT Batch Minted event with right data when mint a batch of 100 NFTs
✔ Should burn an NFT
✔ Should only allow administrators or authorized friend contracts to burn tokens
✔ Should not allow burning an NFT from the zero address
✔ Should not allow burning an NFT if there are no tokens
✔ Should withdraw
✔ Should only allow administrators to withdraw
✔ Should not allow withdrawing if there are no native tokens
✔ Should emit Withdrawn event with right data when withdraw
✔ Should set the mint price
✔ Should only allow administrators to set mint price
✔ Should set score
✔ Should only allow administrators or authorized friend contracts to set score
✔ Should return the URI

- ✔ Should return the total supply

Battle Contract

- ✔ Should battle
- ✔ Should both NFTs be owned by the caller
- ✔ Should emit Battle Completed event with right data when battle

Known Issues and Limitations

Random Number Generation

Description: The current implementation of the battle function in the Battle Contract uses a pseudo-random number generated from a combination of block.timestamp, msg.sender, firstNftId, and secondNftId. While this approach may work for testing and demonstration purposes, it is not secure or suitable for production use.

Issue: In a production environment, relying on such a method for generating random numbers is not recommended. The blockchain's state should be deterministic, and smart contracts should not rely on external factors like block.timestamp for randomness.

Recommendation: In production, this function should only emit a "battle started" event, signaling that a battle has been initiated. Subsequently, a secure off-chain process or API call should be used to generate and reveal the battle result. This ensures fairness, security, and unpredictability, which are essential for a production-ready battle system.

Note: The current implementation is for demonstration purposes only and should not be used in a real-world, production environment. For production use, consider implementing a secure and reliable random number generation mechanism as part of your smart contract architecture.

Project Development Timeline and Approach

I developed this project in a span of 6 hours, working continuously to create and test the smart contracts, write the documentation, and set up the necessary configurations. During this time, I faced no significant blockers or breaks, allowing for a focused and efficient development process. The approximate timeline for the project can be summarized as follows:

Contract Development: I began by writing the smart contracts for the NFT and Battle functionalities, following best practices and security measures. This phase took approximately 2 hours.

Documentation: After completing the contract code, I spent the next 30 minutes documenting the contracts, including explanations, functions, events, and usage instructions.

Testing: With the contracts and documentation in place, I allocated around 3 hours for testing. This involved writing comprehensive unit tests for both contracts, ensuring their functionality and security.

Deployment: Following successful testing, I deployed the contracts on the Polygon Mumbai Testnet. This process took about 5 minutes, including setting up deployment scripts and configurations.

Documentation Enhancement: After deployment, I updated the documentation to include information about contract deployment and provided examples. This phase took around 30 minutes.

The project was completed smoothly within the 6-hour timeframe, allowing for rapid development and deployment while maintaining code quality and security.

Author

This smart contract was authored by Berke Kiran. For more information or inquiries, you can reach out to the author at author's website or berkekiranofficial@hotmail.com.

License