Create and sell an ERC721 token collection

Contract overview

Create and sell an ERC721 token collection is a feature provided by 2 custom smart contracts used together: **CollectionMinter.sol** and **Erc721Collection.sol**

The **CollectionMinter** contract serves as a flexible intermediary, enabling users to purchase ERC721 NFTs from a specified collection (**Erc721Collection** contract) during various sale phases.

Key features of the *CollectionMinter* contract include:

Intermediary functionality:

- Acts as an intermediary allowing users to buy ERC721 NFTs from a specified collection through various sale phases.

Sale phase features:

- Duration: Specifies a block range [start, end] defining the active period of the sale phase.
- Payment Method: Supports either the native chain currency or a specific ERC20 token.
- Mint Price: Establishes a price for minting one token during the sale phase.
- Whitelisted Required: If set to 'true,' only wallets with whitelist permission can mint during the phase.
- Sale Phase Cap: Sets the maximum overall amount of NFTs mintable in a specific sale phase.
- Wallet Limit: Defines the maximum number of NFTs that can be minted by a single wallet in the sale phase.

Payment mechanics:

- Funds from each payment in a sale phase are deposited into the contract.
- Only the admin (DEFAULT_ADMIN_ROLE) can decide the funds receiver via the {withdrawFunds} function.
- {withdrawFunds} allows the manager to trigger fund withdrawal and specify the currency to withdraw.
- A fee is applied to every mint operation, where the {contractProvider} receives a fee in the blockchain native currency equal to {providerMintFee}.

Key features of the *Erc721Collection* contract include:

Custom token IDs and URIs:

- Specifies a custom range for token IDs during deployment using {_minimumNftTokenId} and {_maximumNftTokenId}.
- Allows customization of token URIs during deployment through a partial metadata URL.
- Enables subsequent customization of existing token URIs through a custom function.

Minting from third parties:

- Permits third parties, including external EOAs or smart contracts, to mint tokens from this contract.
- Facilitates the implementation of drop logic in a different contract while maintaining collection constraints.

Owner control:

- Supports the Ownable interface in addition to the AccessControl.
- Facilitates the transfer of collection ownership for future marketplace handling, providing flexibility beyond the deployer's wallet.

1. Functional requirements

1.1. Roles

CollectionMinter contract has 3 roles:

1. **Manager Role (MANAGER_ROLE)**: Manages creation, disabling, and whitelists for sale phases.
2. **Team Minter Role (TEAM_MINTER_ROLE)**: Enables the team to mint NFTs outside sale phases, respecting collection max supply.
3. **Pauser Role (PAUSER_ROLE)**: Pauses minting, whitelist management, and creation of new sale phases.

Erc721Collection has 3 roles:

1. **Pauser Role (PAUSER_ROLE)**: Enables/disables the ability to mint new tokens and allows changing the URI of existing tokens.

2. **Minter Role (MINTER_ROLE)**: Has the ability to call the {safeMint} function to mint new tokens for the collection.
3. **URI Editor Role (URI_EDITOR_ROLE)**: Can change the URI of an existing token within the collection.

1.2. Features

Erc721Collection has the following features:

1. Mint new tokens up to the max collection supply (MINTER_ROLE)
2. Change the URI of an existing token (URI_EDITOR_ROLE)
3. Pause and resume the possibility of minting new tokens and change the URI of existing ones (PAUSER_ROLE),

CollectionMinter has the following features:

1. Create a new sale phase specifying the token payment method (ERC20 or native chain currency), whitelist enabled, limitations and duration. (MANAGER_ROLE)
2. Check which is the ID of the last sale phase created
3. Grant and revoke the whitelist to a wallet to access a specific sale phase
4. Pause and unpause the overall mint (DEFAULT_ADMIN_ROLE)
5. Check if a sale phase is active
6. Disable a sale phase (MANAGER_ROLE)
7. Set and update the wallet that is able to withdraw the funds collected during the different sale phases (DEFAULT_ADMIN_ROLE)
8. Let the a wallet with the MANAGER_ROLE role to withdraw funds to the funds receiver wallet
9. Let a wallet to mint one or more tokens in a specific sale phase (if sale constraints are satisfied)
10. Let the project team to mint directly new tokens without the need to select a specific sale phase and for free (TEAM_MINTER_ROLE)

1.3. Use cases

1.3.1. Erc721Collection use cases

1. The MINTER_ROLE mints to his wallet 10 new tokens.
2. The PAUSER_ROLE stops the mint and the token URI changing features.
3. The URI_EDITOR_ROLE changes the token URI of the existing token with ID 9.

1.3.2. CollectionMinter use cases

4. The MANAGER_ROLE creates a new sale phase on Ethereum setting the mint price to 1 ETH per token, with a whitelist needed in order to be able to mint with a duration that starts from the current block and goes on for 1000 blocks. The admin also sets other sale details like the number of tokens mintable per wallet and the max collection supply that the community can reach by minting at 100 tokens. After setting up the sale the MANAGER_ROLE adds the wallets that can mint in this sale phase and enables the mint.
5. The MANAGER_ROLE removes a whitelisted wallet from a sale phase that has not yet started.
6. Users mint through an existing sale phase paying with the select ERC20 token set for the related sale phase.
7. After some successful sales the MANAGER_ROLE decides to withdraw the funds from the smart contract specifying that he wants to withdraw only the gain in native currency and waiting another moment to withdraw the ERC20 tokens gained in other past sale phases. The funds have been sent to the funds receiver set on the contract deployment time since it has never been changed.
8. The DEFAULT_ADMIN_ROLE changes the funds receiver.
9. The MANAGER_ROLE stops an ongoing sale phase since it was set up in a wrong way.

2. Technical requirements

This project has been developed with **Solidity** language, using Hardhat as the development environment. **Javascript** has been the language used for testing and scripting.

In addition, **OpenZeppelin**'s libraries have been used in the project to handle features like Access Control and Reentrancy Guard issues. All the documentation related to the used libraries is available in their Github repository.

Related to this contract the involved files in the repository are the following ones in bold red:

```
├── README.md
├── contracts
│   ├── CollectionMinter.sol
│   ├── Erc1155Claimer.sol
│   ├── Erc721Collection.sol
│   ├── SimpleErc1155.sol
│   ├── SimpleErc721.sol
│   ├── SnowMarketplace.sol
│   ├── SnowTracker.sol
│   ├── gasReport.txt
│   └── linesCounter.js
├── coverage.json
├── gasReport.txt
├── hardhat.config.js
├── package-lock.json
├── package.json
├── scripts
└── test
    ├── collectionMintTest.js
    ├── erc1155ClaimerTest.js
    └── test.js
```

Start with README.md to find all basic information about project structure and scripts that are required to test the contracts.

Inside the **./contracts folder**, CollectionMinter.sol contains the smart contract with the minting functionality explained in section 1.2 of this document. This contract **needs to be paired with the smart contract Erc721Collection** that is contained in the Erc721Collection.sol file. This pairing is done by first specifying at the deployment time the instance of a deployed Erc721Collection contract and second by granting from the same Erc721Collection instance the role MINTER_ROLE to the CollectionMinter instance. In this way the CollectionMinter instance will act as an intermediary allowing users to mint Erc721Collection instance tokens with the set sale phases logic of the CollectionMinter contract instance.

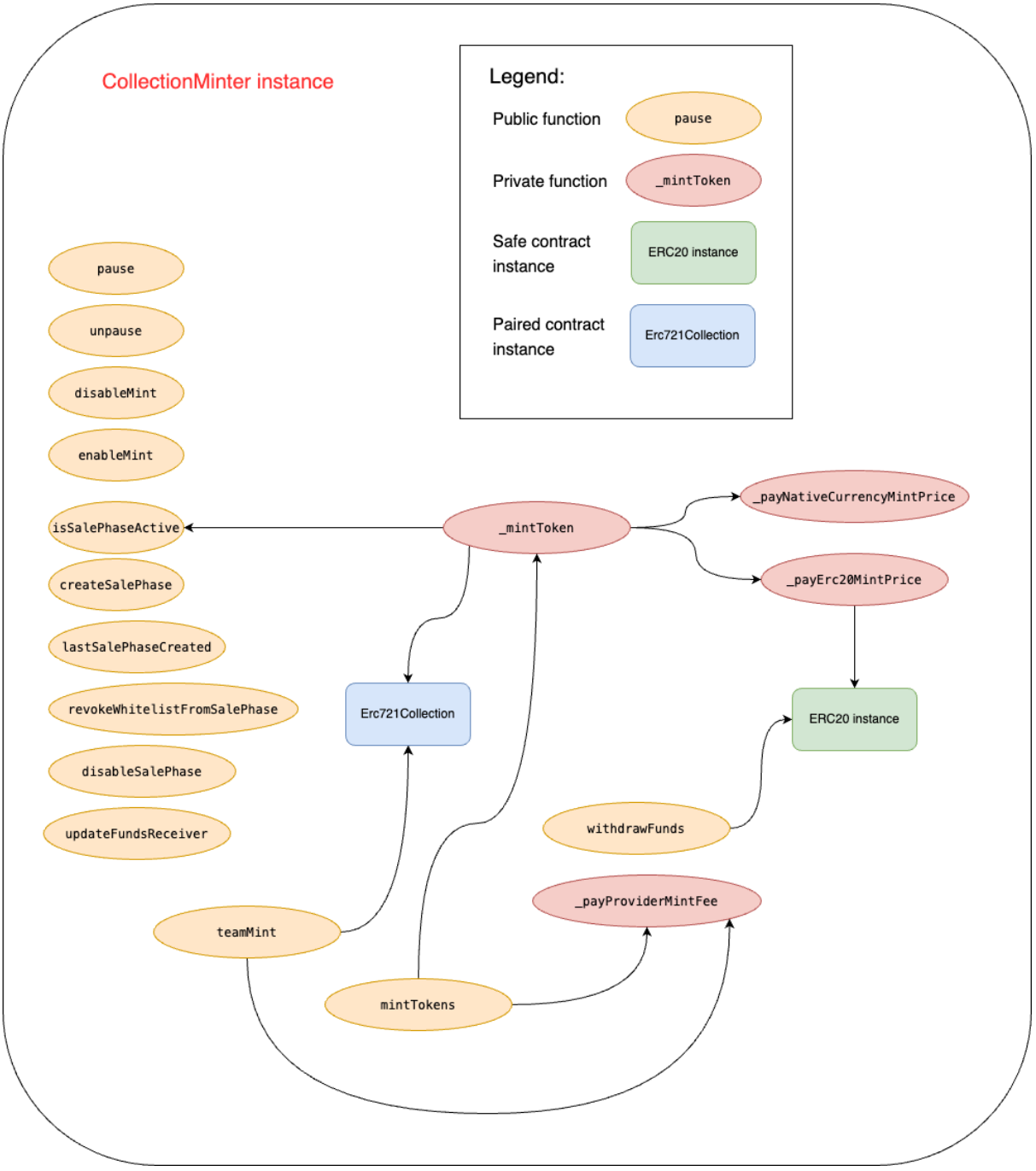
Automated tests have been created to test this contract and can be found in the file located at **./test/collectionMintTest.js**

2.1. Deployment instructions

The main contract, CollectionMinter, can be deployed by cloning the repository using the Remix IDE and following the steps below:

1. **Connect the walle**t to the desidered chain.
2. Compile the contract with a Solidity version that is **at least 0.8.20** (this one is the suggested target version to use in order to avoid compiling issues).
3. Set the contract parameters values (_minimumNftTokenId, _maximumNftTokenId, _contractRoyaltiesBps, _partialMetadataUri), **deploy an instance of the Erc721Collection** contract and save the resulting contract address.
4. Set the deployment parameter values (_collectionContract, _providerMintFee, _contractProvider), set the _collectionContract parameter value as the contract address of the previous deployed instance and **deploy an instance of the contract CollectionMinter**.
5. **Call the grantRole** function from the Erc721Collection contract instance and grant the MINTER_ROLE role to the deployed instance of the CollectionMinter contract.
6. Now is possible to create a new sale phase and mint new Erc721Collection collection tokens through the logic contained in the sale phases of the CollectionMinter instance.

2.2. Architecture overview



2.3. Contract information

This section contains useful information about the contracts used for this logic block.

2.3.1. Erc721Collection.sol

The aim of this contract is to manage a traditional ERC721 collection with customized token IDs, flexible URI options, and control over minting, access, and ownership. It enables additional features such as minting from third parties and easy transfer of ownership for future marketplace handling.

This contract has no particular issues or attention that need to be taken care about and for this reason its functions behavior (parameters, result and exceptions) have been reported in NatSpec format directly in the smart contract code.

2.3.2. CollectionMinter.sol

This contract serves as an intermediary, allowing users to purchase ERC721 NFTs from a specified collection through various sale phases. Key features include customizable sale phases with options for duration, payment methods, mint prices, whitelisting, and specific limits. The contract manager can create and disable sale phases, manage whitelists, and handle fund withdrawals. The team can mint NFTs outside sale phases, and a pauser can control minting, whitelist management, and sale phase creation. Payment mechanics involve fund deposits, fund withdrawal control, and the application of fees to each mint operation.

This contract has no particular issues or attention that need to be taken care about and for this reason its functions behavior (parameters, result and exceptions) have been reported in NatSpec format directly in the smart contract code.

2.3.3. Emitted events

2.3.3.1. Erc721Collection.sol events

No custom events have been defined for this contract.

2.3.3.2. CollectionMinter.sol events

The custom events triggered by this contract are the following:

- **NftsMinted**(_from, _amount, _salePhaseId, _blockNumber): this event is triggered when new NFTs are minted. In particular it records who has minted (_from), how many NFTs have been minted (_amount), in which sale phase (_salePhaseId) and at which block number (_blockNumber**).**
- **TeamMintedNfts(_from, _amount)**: this event is triggered when new NFTs are minted by a wallet with the TEAM_MINTER_ROLE. The event records information like the minter (_from) and the amount of tokens minted (_amount).
- **FundsWithdrawn**(_from, _amount): this event is triggered when a MANAGER_ROLE wallet withdraws the funds in the smart contract. The event records information like from who has been called the function (_from) and the amount of tokens withdrawn (_amount).
- **NativePaymentReceived**(_from, _amount): this event is triggered when the smart contract receives the funds in the blockchain native currency from a token sale. The information recorded by the event are from who the funds come (_from) and the amount received (_amount).

2.3.4. Custom errors

2.3.4.1. Erc721Collection.sol custom errors

No custom errors have been set up for this contract.

2.3.4.2. CollectionMinter.sol custom errors

WrongMessageValue(currentValue, requiredValue): This error is triggered when the value sent to the contract during a mint is not correct (either more or less than expected). The information recorded by the error are the current value set (currentValue) and the value that the contract was expecting (requiredValue). This error is triggered in 2 public functions: **mintTokens** and

teamMint. *Note*: even if the team is the contract owner it has to pay the fee for minting to the contract provider.

2.3.5. Custom structs

2.3.5.1. Erc721Collection.sol custom structs

No custom structs have been set up for this contract.

2.3.5.2. CollectionMinter.sol custom structs

Only 1 custom struct has been created for this contract and its name is **Sale Phase** and below are detailed the fields that characterize it:

- **phaseId** (uint256): Represents the unique identifier for the sale phase, starting from 0.
- **whitelistRequired** (bool): Indicates whether only whitelisted wallets have the ability to mint during this sale phase.
- **startBlock** (uint256): Specifies the block number at which the sale phase commences.
- **endBlock** (uint256): Specifies the block number at which the sale phase concludes.
- **payInNativeCurrency** (bool): If set to {true}, denotes that the sale transactions are conducted using the blockchain's native currency; otherwise, set to {false}.
- **paymentToken** (address): Signifies the address of the token utilized for payment transactions during this sale phase.
- **mintPrice** (uint256): Represents the cost associated with minting during the current sale phase.
- **maxMintsPerWallet** (uint256): Denotes the maximum allowable number of mints per wallet during this sale phase (0 signifies unlimited).
- **maxSalePhaseCap** (uint256): Specifies the maximum supply achievable in this particular sale phase.

2.3.6. Smart contract functions

All functions within the smart contract are meticulously documented in the code. For a comprehensive understanding of

the contract's intricacies, readers are encouraged to refer directly to the code.

Each function includes a detailed explanation of its behavior, with both technical and abstract descriptions when necessary. Additionally, the documentation outlines the inputs, expected results, and potential exceptions for each function, providing a comprehensive guide to the contract's functionality.