Outcome of functions and examples

Imports

```
import
"https://github.com/OpenZeppelin/openzeppelin-contracts/blob/releas
e-v4.0/contracts/token/ERC721/ERC721.sol";
import
"https://github.com/OpenZeppelin/openzeppelin-contracts/blob/releas
e-v4.0/contracts/token/ERC721/extensions/ERC721Enumerable.sol";
import
"https://github.com/OpenZeppelin/openzeppelin-contracts/blob/releas
e-v4.0/contracts/access/Ownable.sol";
import
"https://github.com/OpenZeppelin/openzeppelin-contracts/blob/releas
e-v4.0/contracts/utils/structs/EnumerableSet.sol";
import
"https://github.com/OpenZeppelin/openzeppelin-contracts/blob/releas
e-v4.0/contracts/security/ReentrancyGuard.sol";
```

We are reporting files from OpenZeppelin with their ERC721 contract. Our tokens will be ERC721 non-fungible tokens

Contract

Contract Name: NFTSalon
constructor: Token name is called "SuperAsset", "SUPERASSET"
setPercentCut: Contract owner can change percentage cut at any time
setMetaUrl: used to connect to OpenSea

Structures for batch

```
uint256 public tokenBatchIndex; //Batch ID
mapping(uint256 => string) public tokenBatchHash; // Key -> Batch
ID  : Value -> File Hash
mapping(uint256 => string) public tokenBatchName; // Key -> Batch
ID  : Value -> Batch Title
mapping(uint256 => uint256) public tokenBatchEditionSize; // Key ->
Batch ID  : Value -> how many tokens can we mint in the same batch
(group)
mapping(uint256 => uint256) public totalMintedTokens; // Key ->
Batch ID  : Value -> ERC721 tokens already minted under same batch
```

```
mapping(uint256 => address) public tokenCreator; // Key -> Batch ID
: value -> address of creator
mapping(uint256 => string) public fileUrl; // Key -> Batch ID :
value -> fileUrl
mapping(uint256 => string) public thumbnail; // Key -> Batch ID :
value -> thumbnail url
mapping(uint256 => address payable [5]) public
royaltyAddressMemory; // Key -> Batch ID  : Value -> creator
(artist) address
mapping(uint256 => uint256[5]) public royaltyPercentageMemory;  //
Key -> Batch ID  : Value -> percentage cut  for artist and owner
mapping(uint256 => uint256) public royaltyLengthMemory; // Key ->
Batch ID  : Value -> Number of royalty parties (ex. artist1,
artist2)
mapping(uint256 => bool) public openMinting; // Key -> Batch ID  :
Value -> minting open or not
mapping(uint256 => uint256) tokenBatchPrice; // Key -> Batch ID  :
Value -> price of Batch
mapping(uint256 => bool) public isSoldorBidded;
```

Users can create a batch and in each batch, there can be several tokens.
Data structures linked to `tokenBatchIndex`:

1. `tokenBatchHash`: Hash of the uploaded file is generated to check and ensure there is only one copy of file
2. `tokenBatchName`: Title of the batch file that is used across all tokens minted from that batch
3. `tokenBatchEditionSize`: Number of tokens to be minted in the batch
4. `totalMintedTokens`: ERC721 tokens already minted under this batch
5. `tokenCreator`: Wallet address of the creator
6. `fileUrl`: Url to uploaded file
7. `thumbnail`: Url to uploaded thumbnail
8. `royaltyAddressMemory`: Up to 5 wallet addresses can be added for royalties split for each batch
9. `royaltyPercentageMemory`: Up to 5 percentages can be added for royalties split for each batch
10. `royaltyLengthMemory`: number of royalties that has been entered
11. `openMinting`: This is a feature where creators allow users to mint their own tokens at a set price. For example, a creator set 10 tokens for open minting at 0.1ETH. Users can mint a token that will go to their account after paying the 0.1ETH.
12. `tokenBatchPrice`: Price set by creator so that users can use `openMinting`. In this case, it will be 0.1ETH based on the previous example.
13. `isSoldorBidded`: Check if any token from that batch has been sold or bidded on

Tokens

```
    mapping(address => EnumerableSet.UintSet) internal tokensOwnedByWallet;
    mapping(address => uint256) internal userBalance;
    mapping(uint256 => bool) public isSellings;
    mapping(uint256 => uint256) public sellPrices;
    mapping(uint256 => uint256) public tokenEditionNumber;
    mapping(uint256 => uint256) public referenceTotokenBatch; //name : tokenIdToBatchId //
Key -> Token ID  : Value -> Batch Id to which it belongs to
    mapping(uint256 => Auction) public auctions;
```

These tokens are minted from the batch created
Token properties:

1. `tokensOwnedByWallet`: Storing all the tokens owned by a wallet address
2. `userBalance`: In the case where a contract address placed a bid and gets outbidded, sometimes their bid money cannot be returned to their wallet and will stay in the smart contract under userBalance. This individual can withdraw it later to the same contract address
3. `isSellings`: Is token listed for sale or not
4. `sellPrices`: Price set for sale for each token
5. `tokenEditionNumber`: Edition number for each token after batch minting (Eg. #1, #2, #3)
6. `referenceTotokenBatch`: Which batch this token belongs to
7. `auctions`:
    a. Substructure:
        i. bidder: Current user that adds a bid
        ii. bidPrice: Current bid price (Highest)
        iii. isBidding: If token is active on bidding, bidding time not ended yet
            - True => Auction for token is active
            - False => Auction has been closed
        iv. bidEnd: Bid end time
        v. seller: user who started the auction

Events

```
event newTokenBatchCreated(string tokenHash, string tokenBatchName, uint256 editionSize,
uint256 price, uint256 tokenBatchIndex, address creator, uint timestamp);
```

timestamp: when the event was emitted

```
event tokenCreated(uint indexed tokenId, address indexed tokenCreator, uint timestamp, uint indexed batchId);
```

```
event tokenPutForSale(uint indexed tokenId, address indexed seller, uint sellPrice, bool isListed, uint timestamp);
```

```
event tokenBid(uint indexed tokenId, address indexed bidder, uint tokenPrice, uint timestamp);
```

```
event bidStarted(uint indexed tokenId, address indexed lister, bool isBid, uint tokenPrice, uint endTime, bool isClosedBySuperWorld, uint timestamp);
```

isClosedBySuperWorld

- True => SuperWorld has closed the bid
- False => Owner/bidder has closed the bid

```
event tokenBought(uint indexed tokenId, address indexed newowner, address indexed seller, uint timestamp, uint tokenPrice);
```

Modifier

```
modifier ownerToken(uint256 tokenId) {
        require(ownerOf(tokenId) == msg.sender,"Not TokenOwner");
        _;
    }
modifier creatorToken(uint256 tokenBatchId) {
        require(tokenCreator[tokenBatchId] == msg.sender,"Not tokenCreator");
        _;
    }
```

ownerToken: only the owner of token allowed
creatorToken: only the creator of token allowed

Functions

createTokenBatch()

```
function createTokenBatch(string memory _tokenHash,  string memory _tokenBatchName, uint256 _editionSize, uint256 _price, string memory _fileUrl, string memory _fileThumbnail) public returns(uint256)
```

Creator can create a token batch.

openCloseMint()

```
function openCloseMint(uint256 tokenBatchToUpdate, uint256 _price, bool _isOpen) public creatorToken(tokenBatchToUpdate)
```

Creator can put it for open minting or end the open minting where buyers can mint tokens at the price set by the creator.

addTokenBatchRoyalties()

```
function addTokenBatchRoyalties(uint256 tokenBatchId, address[] memory _royaltyAddresses, uint256[] memory _royaltyPercentage) public creatorToken(tokenBatchId)
```

Only creators can add up to 5 royalties to a batch. Once a token from the batch is sold, creators cannot edit the royalties.

Requirements

```
require(_royaltyAddresses.length == _royaltyPercentage.length,"royaltyAddress not match royaltyPercentage length");
require(_royaltyAddresses.length <= 5,"Maximum size exceeded");
require(isSoldorBidded[tokenBatchId] == false,"Token already sold or bidded");

require(totalCollaboratorRoyalties <= 100,"Max percentage reached");
```

1. The number of wallet address has to equal the number of percentages added
2. There is a maximum of 5 royalties for input
3. If token has been sold or bidded, royalties cannot be changed
4. Maximum number of sum of percentages is equal to 100%

getRoyalties()

```
function getRoyalties(uint256 tokenBatchId) public view returns (address[5] memory addresses, uint256[5] memory percentages)
```

Get wallet addresses and royalties percentages

mintTokenBatch()

```
function mintTokenBatch(uint256 tokenBatchId, uint256 amountToMint) public payable
```

Calls mintToken() based on the value amountToMint is set to

mintToken()

```
function mintToken(uint256 tokenBatchId) public payable
```

Mint one token

1. If open minting enabled, buyers can mint at the price the creator has set
2. If open minting is disabled, only creator can mint

Requirements for open minting

```
if (openMinting[tokenBatchId]) {
        require(tokenBatchPrice[tokenBatchId] <= msg.value,"Less Value sent");
        require(safeState <= tokenBatchEditionSize[tokenBatchId],"Max Batch capacity exceeded");
```

This is similar to the buy function

1. Value input to mint has to be equal or greater than the tokenBatchPrice set by the creator
2. Number of tokens minted has been less or equal to the tokenBatchEditionSize set by the creator

Example

1. Batch is listed for open minting at 100ETH, and percentage cut is set to 15% to SuperWorld
2. When a user does open minting at 100ETH, the creator gets 85% of the sale which equates to 85ETH

3. That 85ETH becomes the priceAfterFee and that would be split among the royalties percentages and that amount will be transferred to the wallet addresses
4. Value:x is sent to royaltyPerson

Requirements for regular minting

```
else {
          require(tokenCreator[tokenBatchId] == msg.sender,"Not tokenCreator");
          require(safeState <= tokenBatchEditionSize[tokenBatchId],"Max Batch capacity
exceeded");
```

If open minting is not available

1. Only creator can mint
2. Creator can only mint up to the tokenBatchEditionSize

sale()

```
function sale(uint256 _tokenId, uint _sellPrice, bool isListed) public ownerToken(_tokenId)
```

Several use cases for this function

1. Listing token for sale where buyers can buy token at the set price
2. Relisting token at a different price
3. Delisting token and remove it for purchase

Requirements

```
require(auctions[_tokenId].isBidding == false,"Token on bidding");
```

Token should not be on bidding

bulkSale()

```
function bulkSale(uint256[] memory _tokens, uint _sellPrice, bool _isListed) public
```

Listing several tokens at a set price, this calls the sale function several times

getTokenBatchData()

```
function getTokenBatchData(uint256 tokenBatchId) public view returns (uint256 _batchId,
```

```
string memory _tokenHash, string memory _tokenBatchName, uint256 _unmintedEditions,
address _tokenCreator, string memory _fileUrl, string memory _fileThumbnail, uint256
_mintedEditions)
```

Getting token batch data

getTokenData()

```
function getTokenBatchData(uint256 tokenBatchId) public view returns (uint256 _batchId,
string memory _tokenHash, string memory _tokenBatchName, uint256 _unmintedEditions,
address _tokenCreator, string memory _fileUrl, string memory _fileThumbnail, uint256
_mintedEditions)
```

Getting token data and reference to which batch it belongs to

Requirements

```
require(_exists(tokenId), "Not exist");
```

Token needs to be already minted and existing

startBid()

```
require(isSellings[_tokenId] == false,"Token on sale");
require(auctions[_tokenId].isBidding == false, "Token already on auction");
        if (_isCountdown == false) {
            require(_endTimestamp > block.timestamp,"Extend EndTime");
            require(_endTimestamp < (block.timestamp + 31 days),"Reduce the end time");
        }
        else{
            require(_endTimestamp < 31 days,"Reduce the end days");
        }
```

```
function startBid(uint _tokenId, uint256 _startPrice, uint _endTimestamp) public
ownerToken(_tokenId)
```

Starting an auction with a starting price and ending date

Requirements

1. Token has to be not on sale
2. Token has to be not on auction
3. If the countdown timer off:

    - Auction ending date has to be in the future
    - Auction ending date cannot be more than 31 days
4. If the countdown timer on:
    - Auction time will start after the first bid
    - Auction time has to be less than 31 days

addBid()

```
function addBid(uint _tokenId) public payable
```

Users can add their bids to an active auction

Requirements

```
require(auctions[_tokenId].isBidding,"Auction ended");
require(msg.value > auctions[_tokenId].bidPrice,"Increase Bid");
        if (auctions[_tokenId].bidder == payable(address(0x0))) {
            if (auctions[_tokenId].isCountdown == false) {
               require(auctions[_tokenId].bidEnd > block.timestamp,"Auction ended");
            }
            else {
                auctions[_tokenId].bidEnd += block.timestamp;
            }
            auctions[_tokenId].bidder = payable(msg.sender);
            auctions[_tokenId].bidPrice = msg.value;
            auctions[_tokenId].isBidding = true;
            emit tokenBid(_tokenId, msg.sender, msg.value, block.timestamp);
        }
        else{
            require(auctions[_tokenId].bidEnd > block.timestamp,"Auction ended");
            (bool success, ) = (auctions[_tokenId].bidder).call{value:
auctions[_tokenId].bidPrice}("");
            if(success == false){
                userBalance[auctions[_tokenId].bidder] += auctions[_tokenId].bidPrice;
            }
            auctions[_tokenId].bidder = payable(msg.sender);
            auctions[_tokenId].bidPrice = msg.value;
            emit tokenBid(_tokenId, msg.sender, msg.value, block.timestamp);
        }
```

1. Auction needs to be active
2. Bid value needs to be higher than current bid

If no bidder yet,

- If no countdown timer, we check if auction still active
- If there is countdown timer, timer starts after the first bid

If there is already bidder,

- We check if auction is active
- Return previous bid to previous bidder and store latest bid into mapping

Example

1. If payable(address(0x0), no one has placed a bid yet
2. Once bid is put in, money will be put into the smart contract and the bidder's wallet address is stored
3. If there is already a previous bidder, the amount in the smart contract will returned to the previous bidder and the new bidder's amount will be stored in the smart contract and wallet address will be updated

closeBid()

```
function closeBid(uint _tokenId) public onlyOwner
```

Allows the contract owner (SuperWorld) to close a bid to return the money to the bidder and the token will not be on auction anymore. This will be done if the creator who started the bid did not end the bid after a long time.

Requirements

```
require(auctions[_tokenId].bidEnd < block.timestamp,"Active Auction");
```

1. Auction needs to have time ended

tokenURI()

```
function tokenURI(uint256 tokenId) public view override returns (string memory)
```

URL for OpenSea to track the token from our contract

_beforeTokenTransfer()

```
function _beforeTokenTransfer(address from, address to, uint256 tokenId) internal virtual override
```

Send bid funds back to bidder on SuperWorld before token transfer to new owner who purchased on OpenSea

buyToken()

```
function buyToken(uint256 _tokenId) public payable returns(bool)
```

This is called when there is a transfer of a token from one hand to another. This is used during

- Buy Now
- Bidding functionalities (except start and add bid)
- Close bids (by SuperWorld, by Buyer, by Owner)

Requirements

```
require(isSellings[_tokenId],"Token not selling");
require(msg.value >= sellPrices[_tokenId],"Add more value");
```

1. Token needs be to listed for selling
2. Value input has to be more or equal to selling price

_buyToken()

```
function _buyToken(uint256 _tokenId, address payable addr, uint256 _price, uint8 _type)
private returns(bool)
```

This is the main buy token functionality
After this is function is called,

- Royalties cannot be changed
- % fee will go to SuperWorld
- Updated total money (priceAfterFee) = Initial total money - fee to SuperWorld
- Fee will be added to SuperWorld balance (totalBalance)
- Loop through royalty addresses and percentages and those will be sent to the wallet addresses, remaining will be sent to the seller. (total of max 5 wallet addresses)

Used in:

- closeBidBuyer

closeBidOwner()

```
function closeBidOwner(uint _tokenId) public ownerToken(_tokenId) returns(bool)
```

1. If no one added bid and timer ended, owner can close bid and nothing happens. Token will be ready for sale or auction again.
2. If someone has added a bid, the token owner can only close the bid after the bid has expired. Once closed, money will go to the owner/seller and the token will go to the highest bidder.

Requirements

```
require(auctions[_tokenId].seller == ownerOf(_tokenId),"Starter of bid not owner");
require(auctions[_tokenId].bidEnd < block.timestamp,"Active Auction");
require(auctions[_tokenId].isBidding,"Token not bidding");
return _buyToken(_tokenId, auctions[_tokenId].bidder, auctions[_tokenId].bidPrice, 2);
```

If someone has put in a bid, we will go with the _buyToken function where the token will be transferred to the person who added the last bid before the timer ended.

1. Only the owner can start the bid
2. Auction timer need to have ended
3. Token needs to be on bidding

closeBidBuyer()

```
function closeBidBuyer(uint _tokenId) public returns(bool)
```

1. If timer ends, and seller did not end auction, last bidder can close the bidding to get the token and their money will be transferred to the seller

Requirements

```
require(auctions[_tokenId].bidEnd < block.timestamp,"Active Auction");
require(auctions[_tokenId].bidder == msg.sender,"Not Bidder");
require(auctions[_tokenId].isBidding,"Not on bidding");
return _buyToken(_tokenId, auctions[_tokenId].bidder, auctions[_tokenId].bidPrice, 2);
```

1. Auction timer needs to have ended
2. Only the last bidder can call this function
3. The token is not on bidding anymore

getOwnedNFTs()

```
function getOwnedNFTs(address _owner) public view returns(string memory)
```

Get all tokens of Owner for viewing for a specific wallet address

withdrawBalance()

```
function withdrawBalance() public payable onlyOwner() nonReentrant()
```

Owner of the contract (SuperWorld) can get their money from the contract from the fees.

Requirements

```
require(totalBalance > 0,"Not enough funds")
```

1. There needs to be balance from the fees to withdraw it

withdrawUserBalance()

```
function withdrawUserBalance() public payable nonReentrant()
```

Owner of the wallet can withdraw the their balance in the smart contract

Requirements

```
require(userBalance[msg.sender] > 0,"Not enough funds")
```

Require balance in smart contract

Utility Functions

toString()

```
function toString(uint256 _i)internal pure returns (string memory str)
```

Converts an integer to a string

toString()

```
function toString(address _i) internal pure returns (string memory str)
```

Converts an address to a string

toString()

```
function toString(bool _i) internal pure returns (string memory str)
```

Converts an bool to a string