# CSE 526 Blockchain Application Development

**PHASE 3**

## 1) Title of the project:

Museum Exhibit NFT

## 2) Team:

| | |
|---|---|
| • Name 1: Vijayavani Bandi | Name 2: Vaishnavi Koyyada |
| • email: vijayava@buffalo.edu | email: vkoyyada@buffalo.edu |
| • UBPerson number1: 50465139 | UBPerson number: 50468340 |

## 3) Issue(s) addressed:

In museum exhibits the issues addressed are

1.Authentication: This is used to authenticate ownership by creating NFTs

2.Provenance: It describes about the history of item, value of an item and the transfer of ownership.

3.Ownership: Gives proof about the owner by creating NFTs which authenticates the owners of items.

4.Trade: Trading an item defines that the items are in sale in return it generates revenue for the sold items.

5.Transparency: By creating NFTs it increases transparency by providing records of the owners

which prevents losses or fraudulences, and customers have a complete access to the records.

6.Accessibility: By creating NFTs a wide range of customers can be able to access and can purchase the items from the museum.

7.Preservation: Preserving records will help the customers to know the history of the items and the value which prevents from high loss.

8.Protection: Provides protection to artists ensuring that they receive some royalties for the item they have sold. It happens through NFTs.

9.Traceability: Provides traceability of items that helps in tracking the movement of items overtime.

10. Immutability: The transactions recorded in blockchain cannot be reverted and keeps the transactions secure.

11. Transaction Fee: Block chain uses digital currency system for the transactions which has less fee that the other payment methods.

12. Secured payments: Secured payments are done through smart contracts to prevent fraudulent transactions and ensures efficient payments.

## 4) Abstract

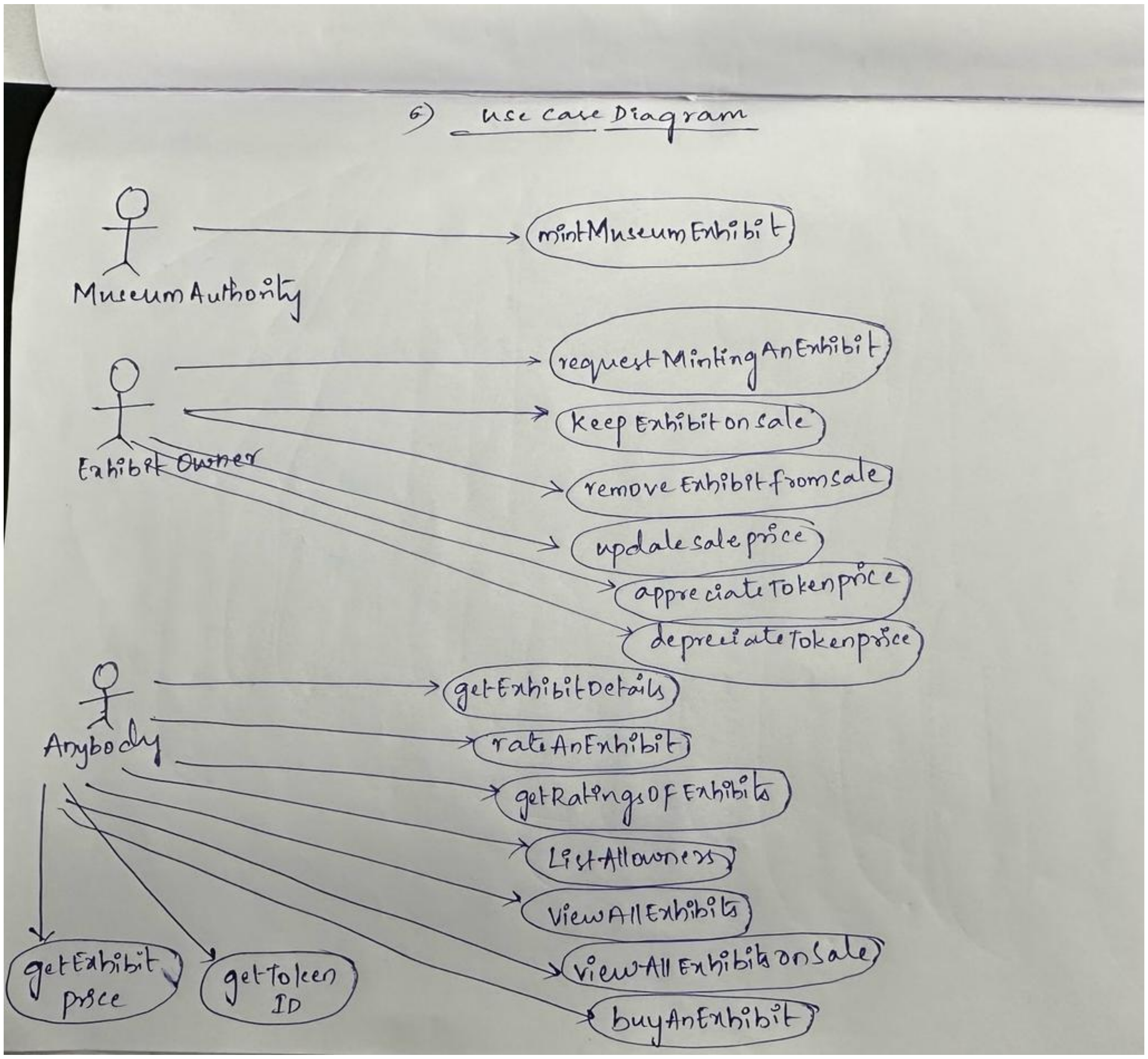
Museum Exhibits can be used as a Non-Fungible Tokens (NFTs) to authenticate the ownership and to include its detailed history of origin of museum exhibits. The idea is to create NFT that represents a unique digital identity for each exhibit, containing information such as its creator or the collector, year, previous owners, and other relevant information. These NFTs would be linked to the exhibits physically displayed in the museum, providing an immutable record of its ownership and history. By using blockchain technology, the authenticity of the exhibits and its ownership can be verified with a high degree of certainty, providing a solution to the problem of fraud and forgeries. Additionally, these NFTs could provide proof of ownership, simplifying the process of transferring ownership of exhibits between different museums or to the buyer. Everything is transparent to everyone. The creation of NFTs for museum exhibits will help to increase the transparency and trust and benefit the preservation and appreciation of these exhibits

## 5) Digital asset and Token:

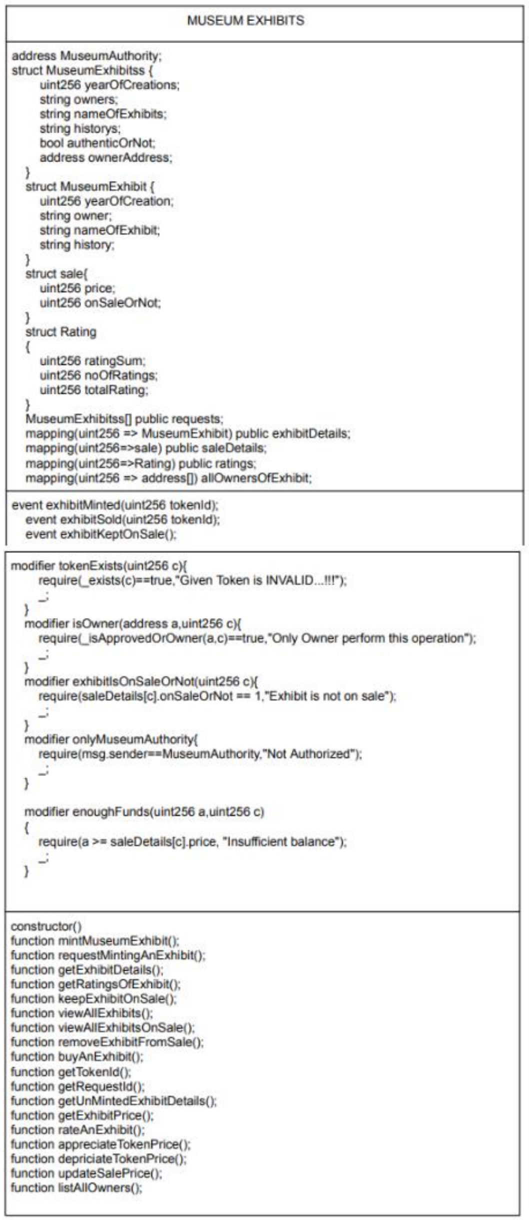
Considering exhibit as the digital asset in museum exhibits. Here the NFTs is used to give the information about the history of exhibits and the details of the previous owner when the exhibit been sold to another person. Generally, this data is stored in the blockchain records which helps the customers to know the information about the exhibit easily.

Token is created for NFTs for featuring the exhibits in the museum. ERC 721 is a token standard used in creation. In our case the museum exhibits are minted with a tokenId which helps to give the information about the exhibit which involves year, owner name, exhibit name and history. Tokens are also used to track the ownership of exhibit. Helps customers and owners engage in transactions without any intermediaries.

6)Usecase Diagram:



## 7) **Contract Diagram**



MUSEUM EXHIBITS

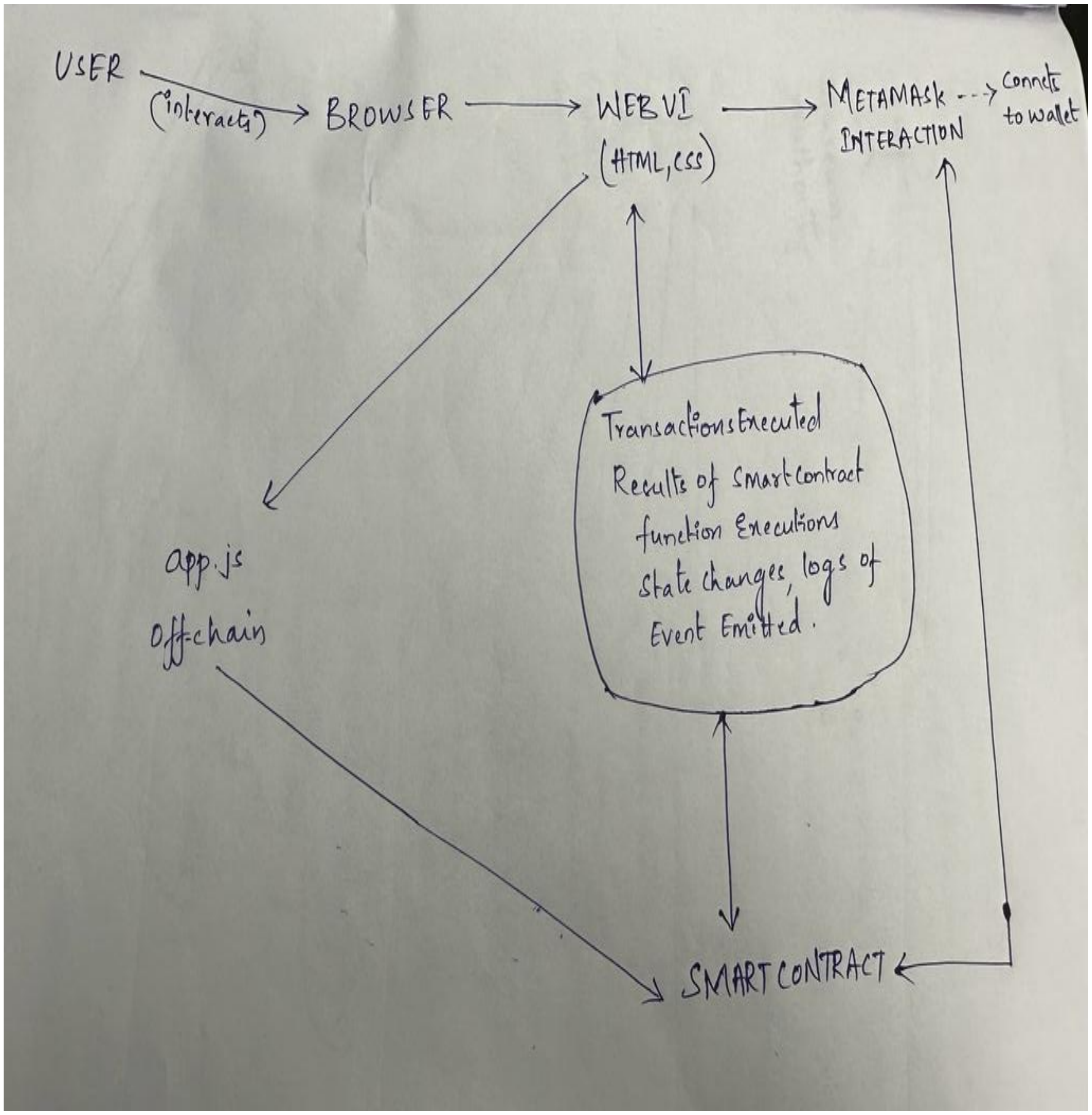
```
address MuseumAuthority;
struct MuseumExhibitss {
      uint256 yearOfCreations;
      string owners;
      string nameOfExhibits;
      string historys;
      bool authenticOrNot;
      address ownerAddress;
  }
  struct MuseumExhibit {
      uint256 yearOfCreation;
      string owner;
      string nameOfExhibit;
      string history;
  }
  struct sale{
      uint256 price;
      uint256 onSaleOrNot;
  }
  struct Rating
  {
      uint256 ratingSum;
      uint256 noOfRatings;
      uint256 totalRating;
  }
  MuseumExhibitss[] public requests;
  mapping(uint256 => MuseumExhibit) public exhibitDetails;
  mapping(uint256=>sale) public saleDetails;
  mapping(uint256=>Rating) public ratings;
  mapping(uint256 => address[]) allOwnersOfExhibit;
```

```
event exhibitMinted(uint256 tokenId);
  event exhibitSold(uint256 tokenId);
  event exhibitKeptOnSale();
```

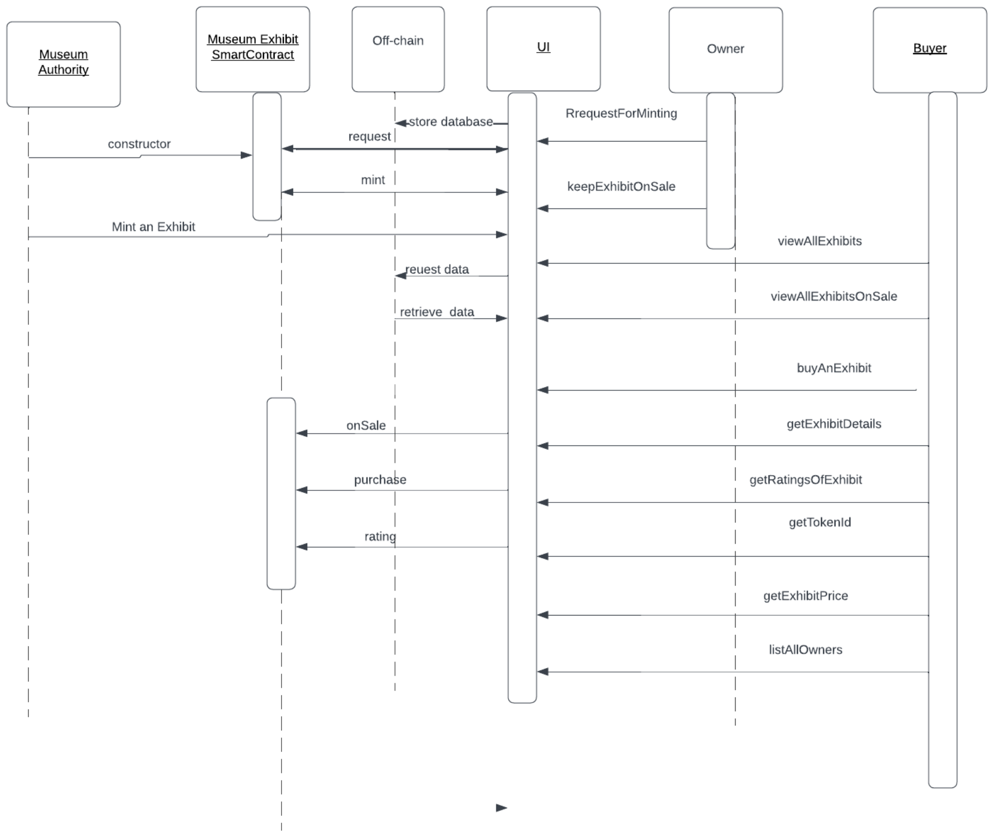
```
modifier tokenExists(uint256 c){
      require(_exists(c)==true,"Given Token is INVALID...!!!");
      _;
  }
  modifier isOwner(address a,uint256 c){
      require(_isApprovedOrOwner(a,c)==true,"Only Owner perform this operation");
      _;
  }
  modifier exhibitIsOnSaleOrNot(uint256 c){
      require(saleDetails[c].onSaleOrNot == 1,"Exhibit is not on sale");
      _;
  }
  modifier onlyMuseumAuthority{
      require(msg.sender==MuseumAuthority,"Not Authorized");
      _;
  }

  modifier enoughFunds(uint256 a,uint256 c)
  {
      require(a >= saleDetails[c].price, "Insufficient balance");
      _;
  }
```

```
constructor()
function mintMuseumExhibit();
function requestMintingAnExhibit();
function getExhibitDetails();
function getRatingsOfExhibit();
function keepExhibitOnSale();
function viewAllExhibits();
function viewAllExhibitsOnSale();
function removeExhibitFromSale();
function buyAnExhibit();
function getTokenId();
function getRequestId();
function getUnMintedExhibitDetails();
function getExhibitPrice();
function rateAnExhibit();
function appreciateTokenPrice();
function depriciateTokenPrice();
function updateSalePrice();
function listAllOwners();
```

8) Architecture Diagram



**9) Sequence Diagram:**



**10) Smart Contract Rules:**

1.Here, only the MuseumAuthority who is the deployer of the contract can mint a new exhibit token using the mintMuseumExhibit() function.

2.Anyone can request the minting of an exhibit by providing the required information using the requestMintingAnExhibit() function. The exhibit will only be minted if the MuseumAuthority approves it.

3.An exhibit owner can put their exhibit on sale and can remove their exhibit from sale with a specified price.

4.An exhibit owner can update the sale price of their exhibit.

5.An exhibit owner can appreciate or depreciate the price of their exhibit .

6.Buyers can purchase an exhibit which is on sale. The buyer must have enough funds to cover the sale price, and they cannot buy their own exhibit.

7.Users can rate an exhibit.

8. Following these rules, the smart contract ensures a secure and controlled environment for managing museum exhibits as NFTs, their minting, sales, and ratings.

9. All the prices of the exhibit are in Wei.

**11) Steps For Deployment:**

**a) Contract Deployment –**

To deploy the smart contract we need Gannache and Truffle.

1. Install Gannache and Truffle.
2. In command prompt navigate to dapp-contract Folder and initialize the truffle, the command we use here is **tuffle init.**
3. After initializing the truffle compile and deploy it using the command **truffle compile .**
4. Run the migration scripts for deploying smart contracts using the command **truffle migrate.**
5. Copy the address generated and update it in **app.js**

**b) Web app Deployment:**

To deploy the web-app we need to download and install Node.js and node package manager.

1. After installing npm navigate to dapp-app folder .
2. Initialize the node using **npm init**.
3. Install the node modules using **npm install.**
4. start the Node.js server using **npm start.**

**12) Deployment on Infura and cloud**

To deploy our Smart Contract on Infura, first create your account on Infura.

1. Create a folder for you Daap and then we get an Web3 API key for it.
2. We then install the hdwallet-provider in our MuseumExhibit-Contract folder using the command, **npm install @truffle/hdwallet-provider**
3. Then we are deploying our contract in the Sepolia. To do so, we need to make the following changes in our truffle-config.js file
   Put the API key which you got from Infura in a variable named infuraApikey and initialize it using **const infuraApiKey = '';.**
   Put the private key of your account from which you want to deploy your smart contract in a variable named key and initialize it using **const key = '';.**
   Now update the networks to below

```
networks: {
  sepolia: {
    provider: () => new
HDWalletProvider(key,`https://sepolia.infura.io/v3/${infuraApiKey}`),
    network_id: '*',
    gas: 5500000,
    confirmations: 2,
    timeoutBlocks: 200,
    skipDryRun: true,
  },
 },
```

And then leave the remaining code as it is in the truffle-config.js file and save it.

4. Now compile your contract using **truffle compile**
5. Now then deploy your contract on Sepolia Network using **truffle migrate --network sepolia**
6. Now copy the deployed smart contract address and then paste it in the address in the app.js file

**Deployment of Front End on GCP**

1. Created an Instance on GCP and then uploaded the file in the cloud.
2. Then when we run **npm install && npm start**
3. Then our app will be running on the port 3010.
4. We can view our Daap on the web address http://34.16.146.191:3010

**13) Code implementation of the digital assets-token smart contracts.**

We have defined a smart contract with the name MuseumExhibits and have defined a ERC721 token with name MuseumExhibitNFT with ME as its symbol. We can keep track of address of Museum Authority who is the deployer of the smart contract and the one who verifies the authenticity of Museum Exhibits. Every NFT is identified by means of exhibitTokenId. The MuseumExhibitss structure stores the details of the exhibit details that are being requested for minting by the exhibit owners. The MuseumExhibit structure has the year, owner name, exhibit name and history of the minted exhibits.

The Sale structure has the price and tells whether the exhibit is on sale or not and can use it when we want to sale an exhibit. The Rating structure is used for calculating the ratings of an exhibit. We have mappings from the exhibitTokenId to the above three structures with names exhibitDetails, saleDetails, ratings. We have a mapping from exhibitTokenId to the address list of all owners of an exhibit named allOwnersOfExhibit. We have defined some modifiers to check certain conditions.

**mintMuseumExhibit:** This function can only be accessed by the Museum Authority to mint the exhibit. It increments the exhibitTokenID and mints that token to the owner of the exhibit. All the exhibitDetails like year, name, exhibit name, history are updated for the token.

**requestMintingAnExhibit:** This function requests the minting of an exhibit giving year, owner name, exhibit name and history of the exhibit.

**getExhibitDetails:** This function is view only function and have a modifier that checks whether exhibitTokenId exists and can be accessed by anyone and returns all the details of a particular Exhibit given its exhibitTokenId.

**getRatingsOfExhibit:** This function is view only function and have a modifier that checks whether exhibitTokenId exists and can be accessed by anyone and returns the rating of the exhibit given its exhibitTokenId.

**keepExhibitOnSale:** This function is used to keep an exhibit on sale. We have modifiers that check whether the token exists or not and is invoked by the owner of the exhibit.

**viewAllExhibits:** This function is view only function and returns all the exhibits that have been minted the Museum Authority.

**viewAllExhibitsOnSale:** This function is view only function and returns all the exhibits that have been kept on sale by the exhibit owners.

**removeExhibitFromSale:** This function is to remove an exhibit from sale given its tokenID and have modifiers that check whether the given exhibitTokenId exists or not, and the function caller is the owner of the given exhibit and whether the exhibit is on sale or not to further proceed with removing it.

**updateSalePrice :** This function is used to update the price of Exhibit that has been kept on sale given its exhibitTokenID and the newPrice that has to be updated. It has modifiers which verify whether the tokenId exists and the exhibit is on sale or not.

**buyExhibit :** This function is used to buy an exhibit that has been put on sale given the exhibitTokenID. It has modifiers that check whether the tokenId exists or not and the exhibit is on sale or not to buy it and the buyer is transferring enough to buy it. We also verify that the owner of the exhibit themselves cannot buy that exhibit again. We then transfer the price of the exhibit to the owner. And the token is also transferred to the buyer and we remove the exhibit from sale.

**listAllOwners:** This function is view only and returns the list of all the owners of that exhibit from the time the exhibit has been minted given the tokenId.

**rateAnExhibit:** This function is used to give the rating for an exhibit given its tokenId and rating. It ahs modifier that checks whether the given tokenId exists or not.

**getUnmintedExhibitDetails**: This function returns an array of MuseumExhibitss structs that represent unminted exhibits. It iterates through the requests array and collects all the exhibit requests that have not yet been authenticated or minted (authenticOrNot set to false**).** This function is useful for the MuseumAuthority to see the pending exhibit requests that need to be minted.

**appreciateToken**: This function tells about the exhibit token to increase the price of an exhibit by a given percentage. It checks if the caller is the owner of the token and if the token exists, then calculates the increased price based on the given percentage and updates the price in the saleDetails mapping for the given tokenId.

**depreciateToken:** This function tells about the exhibit token to decrease the price of an exhibit by a given percentage**.** It checks if the caller is the owner of the token and if the token exists, then calculates the decreased price based on the given percentage and updates the price in the saleDetails mapping for the given tokenId.

**getExhibitPrice:** This function returns the current price of an exhibit for a given tokenId. It checks if the token exists and then retrieves the price from the saleDetails. It checks if the token exists and then retrieves the price from the saleDetails mapping. This is useful for potential buyers who want to check the price of an exhibit before purchasing it

**getTokenId:** This function returns the tokenId of an exhibit based on its properties: yearOfCreation, owner, nameOfExhibit, and history. It iterates through the exhibitDetails mapping and compares the exhibit properties with the given input. If a match is found, the function returns the tokenId of that exhibit. This is useful for querying specific exhibits based on their attributes.

**getRequestId:** This function returns the index of a specific exhibit request in the requests array based on its properties: yearOfCreations, owners, nameOfExhibits, and historys. It iterates through the requests array and compares the exhibit request properties with the given input. If a match is found, the function returns the index of that exhibit request. This is useful for querying specific exhibit requests based on their attributes, especially for the MuseumAuthority, who needs to identify the request to be minted.

## 14) References

Solidity — Solidity 0.8.19 documentation (soliditylang.org)

ERC721 - OpenZeppelin Docs

https://www.manning.com/books/blockchain-in-action

## 15) Approved by

Ram Kashyap Cherukumilli