

My project consists of 2 smart contracts. The first smart contract contains an NFT collection named NFTLoanCoin.

In this program I use solidity version 0.8.20.

The NFT runs on the ERC721 token base, specifically here we are using the ERC721URIStorage library which will allow us to store the NFT metadata on the blockchain, specifically the uri of the given nft.

We will be able to store IPFS URLs with metadata for a given NFT in the uri.

We also use the ownable library here, which will ensure that only the owner can perform certain functions

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.20;
3
4  import "@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";
5  import "@openzeppelin/contracts/access/Ownable.sol";
6
7  contract NFTLoanCoin is ERC721URIStorage, Ownable {
8      constructor()
9          ERC721("Loan NFT", "LNFT")
10         Ownable(0x7Db9C84846809362E37cBBfb61f4c9e63fD0c992)
11     {}
12
13     function mint(address _to, uint256 _tokenId, string calldata _uri)
14         external
15         onlyOwner
16     {
17         _mint(_to, _tokenId);
18         _setTokenURI(_tokenId, _uri);
19     }
20 }
21
```

So, this is what the contract itself looks like. We can see that in the constructor we are creating an ERC721 token with the name Loan NFT and the symbol LNFT.

In the constructor, we also set my wallet as the owner.

Next, we see the mint function, which allows us to mint the NFT and save the uri. As I mentioned, we will store the IPFS URL with metadata in the uri.

Only the owner will be able to call the mint function. This is to prevent users from adding NFTs themselves as they please.

NFTLoanCoin is deployed at this address:

0xB553Da1cA2B67e2EF86b05e6c2a7a68e0867E17f

As a second smart contract I use LoanContract, where the whole logic is implemented.

This smart contract runs based on an ERC20 token named LoanToken with the LT symbol. This token is used here for all payments.

In the constructor we initialize my wallet as the owner, we also store the address of this contract and initialize the initial token value to 10 million. In the constructor we can see that the initial value is multiplied by the decimals() function, their value is set to 18 by default. So this means that our currency will contain 18 decimal places. Solidity doesn't allow you to store float numbers, so it's handled this way. These tokens are forwarded to the owner's account. Money to other accounts is then added from the owner's wallet.

Next, we create an instance of the first contract in the constructor - NFTLoanCoin and store a reference to it. We will then use these three stored values.

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.20;
3
4  import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
5  import "@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";
6  import "@openzeppelin/contracts/access/Ownable.sol";
7  import "./NFTLoanCoin.sol";
8
9  //1000000000000000000
10 //5000000000000000000
11 //2500000000000000000
12
13 //1713015426
14
15
16 contract LoanContract is ERC20 {
17     address public owner;
18     address public contractAddress;
19     NFTLoanCoin public nftLoanCoin;
20
21     constructor() ERC20("LoanToken", "LT"){
22         owner = 0x7Db9C84846809362E37cBBfb61f4c9e63fD0c992;
23         contractAddress = address(this);
24         _mint(owner, 1000000 * 10 ** decimals());
25         nftLoanCoin = new NFTLoanCoin();
26     }
```

LoanContract is deployed at this address:
0x094193ba58942A4623EbC074f63AD2C60c9Fa59b

I have created two structures next. A Loan structure, which will store the loan information, and an OfferedNFT structure, which will store the information about the offered NFT as a repayment of the loan.

The Loan Structure contains the addresses of the Loan Participants, the amount to be lent, the date by which the Loan must be repaid and the total interest including the interest of the Guarantor and the Lender

It also contains 4 boolean values that indicate the states I am in. This allows us to run the application correctly and provides security.

The OfferedNFT structure contains information about which NFT is offered, the price and the id of the loan to which the offer relates. It also contains information if the offer is still active. Again, for security reasons and to ensure proper operation.

```
28     struct Loan{
29         address borrower;
30         address guarantor;
31         address lender;
32         uint256 amount;
33         uint256 dueDate;
34         uint256 interest;
35         uint256 guarantorInterest;
36         uint256 lenderInterest;
37         bool requestActive;
38         bool guarantorFound;
39         bool guarantorAccepted;
40         bool lenderFound;
41     }
42
43     struct OfferedNFT{
44         uint256 loanId;
45         uint256 tokenId;
46         uint256 price;
47         bool offerActive;
48     }
```

Next, we map 2 numbered values to these 2 structures. We can consider these values as their ids. The id will be indexed from zero and incremented by 1 with each new instance.

```
mapping(uint256 => Loan) public loans;
mapping(uint256 => OfferedNFT) public nfts;
uint256 public loanCounter = 0;
uint256 public nftCounter = 0;
```

The addMoney function is used to add money. This is achieved by sending money from the owner's account, this money was sent when the contract was created.

Of course, this function can only be called by the owner and cannot add more money than he has in his account.

```
function addMoney(address toAddress, uint amount) external {
    require(msg.sender == owner);
    require(amount <= balanceOf(msg.sender));
    transfer(toAddress, amount);
}
```

Functions to request for a loan. The applicant must apply for an amount greater than 0, must offer interest greater than 0 and the latest repayment date must be in the future. If the conditions are met. A Loan structure is created and populated with the given information. If the information cannot be found, it is set to 0. We are in a state where the loan request is active, so we set this value to true, the others false.

```
function requestLoan(uint256 amount, uint256 dueDate, uint256 interest) external {
    require(amount > 0, "Amount must be greater than 0");
    require(interest > 0, "Interest must be greater than 0");
    require(dueDate > block.timestamp, "Due date must be in the future");

    loans[loanCounter++] = Loan({
        borrower: msg.sender,
        amount: amount,
        dueDate: dueDate,
        interest: interest,
        requestActive: true,

        guarantor: address(0),
        lender: address(0),
        guarantorInterest: 0,
        lenderInterest: 0,
        guarantorFound: false,
        guarantorAccepted: false,
        lenderFound: false
    });
}
```

placeGuarantee function. At first function checks if it is in the correct state - i.e. if the request is created, but the guarantee is not already placed, or even accepted, etc. Next we check if the person is not giving the guarantee to himself (he is not both the borrower and the guarantor), if he has enough funds in his account. We also check if the interest for the guarantor is not zero or higher than the borrower offers. Finally the transaction is sent to the smart contract itself.

```
function placeGuarantee(uint256 loanId, uint256 guarantorInterest) external{
    require(loans[loanId].requestActive, "Loan request not found or not active");
    require(!loans[loanId].guarantorFound, "Guarantor already found");
    require(!loans[loanId].guarantorAccepted, "Guarantor already found and accepted");
    require(!loans[loanId].lenderFound, "Loan is already done");
    require(msg.sender != loans[loanId].borrower, "Borrower cannot be the guarantor");
    require(balanceOf(msg.sender) >= loans[loanId].amount, "You dont have enough money");
    require(loans[loanId].interest > guarantorInterest, "Your interest can not exceed borrowers interest amount");
    require(guarantorInterest > 0, "You can not add negative interest");

    loans[loanId].guarantor = msg.sender;
    loans[loanId].guarantorInterest = guarantorInterest;
    loans[loanId].lenderInterest = loans[loanId].interest - guarantorInterest;
    loans[loanId].guarantorFound = true;

    transfer(contractAddress, loans[loanId].amount);
}
```

acceptGuarantee. This function again monitors whether we are in the correct state. We also make sure that only the borrower can accept the guarantee. If the conditions are met, we just set the corresponding state to true.

```
function acceptGuarantee(uint256 loanId) external {
    require(loans[loanId].requestActive, "Loan request not found or not active");
    require(loans[loanId].guarantorFound, "Guarantee still not found");
    require(!loans[loanId].guarantorAccepted, "Guarantee already accepted");
    require(!loans[loanId].lenderFound, "Loan is already done");
    require(msg.sender == loans[loanId].borrower, "You are not the borrower");

    loans[loanId].guarantorAccepted = true;
}
```

rejectGuarantee. The same conditions as in the previous case. In case of acceptance, we will send the money back to the guarantor and delete the data. We switch to the state where the guarantor has not submitted the guarantee.

```
function rejectGuarantee(uint256 loanId) external {
    require(loans[loanId].requestActive, "Loan request not found or not active");
    require(loans[loanId].guarantorFound, "Guarantee still not found");
    require(!loans[loanId].guarantorAccepted, "Guarantee already accepted");
    require(!loans[loanId].lenderFound, "Loan is already done");
    require(msg.sender == loans[loanId].borrower, "You are not the borrower");

    loans[loanId].guarantorFound = false;
    loans[loanId].guarantorInterest = 0;
    loans[loanId].lenderInterest = 0;

    ERC20(contractAddress).transfer(loans[loanId].guarantor, loans[loanId].amount);

    // Remove the guarantee
    loans[loanId].guarantor = address(0);
}
```

function getAllLoans. The function returns an array of all loans.

```
function getAllLoans() external view returns (Loan[] memory) {
    Loan[] memory allLoans = new Loan[](loanCounter);

    for (uint256 i = 0; i < loanCounter; i++) {
        allLoans[i] = loans[i];
    }

    return allLoans;
}
```


loanMoney - function checks the correct status again. In addition, the lender must also not be a guarantor or borrower. He must also have enough money in the account. Then the lender's information is entered and the money is transferred

```
function loanMoney(uint256 loanId) external {
    require(loans[loanId].requestActive, "Loan request not found or not active");
    require(loans[loanId].guarantorFound, "Guarantee still not found");
    require(loans[loanId].guarantorAccepted, "Guarantee not accepted");
    require(!loans[loanId].lenderFound, "Lending is already done");
    require(msg.sender != loans[loanId].borrower, "You cannot loan money to yourself");
    require(msg.sender != loans[loanId].guarantor, "You are guarantor already");
    require(balanceOf(msg.sender) >= loans[loanId].amount, "You dont have enough money for this loan");

    loans[loanId].lender = msg.sender;
    loans[loanId].lenderFound = true;

    transfer(loans[loanId].borrower, loans[loanId].amount);
}
```

withdrawFromGuarantor - function must be called only by the lender, be in the correct state and be after the redemption date. After calling this function, the lender gets the money from the guarantor, the loan is marked as inactive and cannot be interacted with further. Of course, neither the guarantor nor the lender will receive any interest.

```
function withdrawFromGuarantor(uint256 loanId) external {
    require(loans[loanId].requestActive, "Loan request not found or not active");
    require(loans[loanId].guarantorFound, "Guarantee still not found");
    require(loans[loanId].guarantorAccepted, "Guarantee not accepted");
    require(loans[loanId].lenderFound, "Lender still not found");
    require(msg.sender == loans[loanId].lender, "You are not lender");
    require(loans[loanId].dueDate < block.timestamp, "Due date must be in the past");

    loans[loanId].requestActive = false;

    ERC20(contractAddress).transfer(loans[loanId].lender, loans[loanId].amount);
}
```

payLoan - again we check the correct status, and the function can only be called by the borrower, furthermore the borrower must have sufficient funds to pay the amount borrowed + interest. The loan is marked as inactive, the money is transferred, and the loan can no longer be interacted with.

```
172 function payLoan(uint256 loanId) external {
173     require(loans[loanId].requestActive, "Loan request not found or not active");
174     require(loans[loanId].guarantorAccepted, "Guarantee not accepted");
175     require(loans[loanId].guarantorFound, "Guarantee still not found");
176     require(loans[loanId].lenderFound, "Lender still not found");
177     require(msg.sender == loans[loanId].borrower, "You are not borrower");
178     require(balanceOf(msg.sender) >= loans[loanId].amount + loans[loanId].interest, "You dont have enough money to pay amount + interest value");
179
180     loans[loanId].requestActive = false;
181
182     transfer(contractAddress, loans[loanId].amount + loans[loanId].interest);
183     ERC20(contractAddress).transfer(loans[loanId].guarantor, loans[loanId].amount + loans[loanId].guarantorInterest);
184     ERC20(contractAddress).transfer(loans[loanId].lender, loans[loanId].amount + loans[loanId].lenderInterest);
185 }
186
```

offerNFT - the function checks the correct state, can only be called by the borrower, checks if the borrower offers its own NFT and also the NFT owner must first approve the contract to be allowed to work with the NFT. This confirmation must be confirmed on the second contract with the NFT collection. The price the borrower asks for the NFT must not exceed the amount of the loan. Therefore, if the price of the NFT is equal to the amount of the loan, the borrower must still pay the interest. Finally, a structure is created that contains the information submitted.

```
function offerNFT(uint256 loanId, uint256 _tokenId, uint256 price) external {
    require(nftLoanCoin.ownerOf(_tokenId) == msg.sender, "You are not the owner of this NFT");
    require(nftLoanCoin.isApprovedForAll(msg.sender, contractAddress), "First you have to gain rights");
    require(loans[loanId].requestActive, "Loan request not found or not active");
    require(loans[loanId].guarantorAccepted, "Guarantee not accepted");
    require(loans[loanId].guarantorFound, "Guarantee still not found");
    require(loans[loanId].lenderFound, "Lender still not found");
    require(loans[loanId].amount >= price, "Price must be smaller or equal to borrowed money");
    require(msg.sender == loans[loanId].borrower, "You are not borrower");

    nfts[nftCounter++] = OfferedNFT({
        loanId: loanId,
        tokenId: _tokenId,
        price: price,
        offerActive: true
    });
}
```

acceptNFT - then, based on the NFT offer id, the lender can accept the offer. Again we check the correct state and whether the function is called by the lender. If the conditions are satisfied, we move the NFT and delete the corresponding structure.

```
function acceptNFT(uint256 nftOfferId) external {
    uint256 loanId = nfts[nftOfferId].loanId;
    require(loans[loanId].requestActive, "Loan request not found or not active");
    require(loans[loanId].guarantorAccepted, "Guarantee not accepted");
    require(loans[loanId].guarantorFound, "Guarantee still not found");
    require(loans[loanId].lenderFound, "Lender still not found");
    require(nfts[nftOfferId].offerActive, "NFT not offered");
    require(msg.sender == loans[loanId].lender, "You are not lender");

    nftLoanCoin.transferFrom(loans[loanId].borrower, loans[loanId].lender, nfts[nftOfferId].tokenId);

    loans[loanId].amount = loans[loanId].amount - nfts[nftOfferId].price;
    nfts[nftOfferId].loanId = 0;
    nfts[nftOfferId].tokenId = 0;
    nfts[nftOfferId].price = 0;
    nfts[nftOfferId].offerActive = false;
}
```

rejectNFT - very similar functionality in case of rejection

```
function rejectNFT(uint256 nftOfferId) external {
    uint256 loanId = nfts[nftOfferId].loanId;
    require(loans[loanId].requestActive, "Loan request not found or not active");
    require(loans[loanId].guarantorAccepted, "Guarantee not accepted");
    require(loans[loanId].guarantorFound, "Guarantee still not found");
    require(loans[loanId].lenderFound, "Lender still not found");
    require(nfts[nftOfferId].offerActive, "NFT not offered");
    require(msg.sender == loans[loanId].lender, "You are not lender");

    nfts[nftOfferId].loanId = 0;
    nfts[nftOfferId].tokenId = 0;
    nfts[nftOfferId].price = 0;
    nfts[nftOfferId].offerActive = false;
}
```

(ii) Tests

Next, I created tests. For each of the functions needed in the first section of the assignment I created 3 test states. The first one tests the branch where everything goes fine, the two other tests the negative branches (the state where the function is reverted). There would be very many test functions in our case, I hope this is enough for the purpose of our project.

For the tests, it is worth mentioning that it is important to understand what all must precede the functions. For example, if we want to test the guaranteed acceptance functionality, we need to create the loan first, then add money to the guarantor, then create the guarantee, and then test the guarantee acceptance functionality itself.

Example of two test cases. The rest of the tests are included in the project:

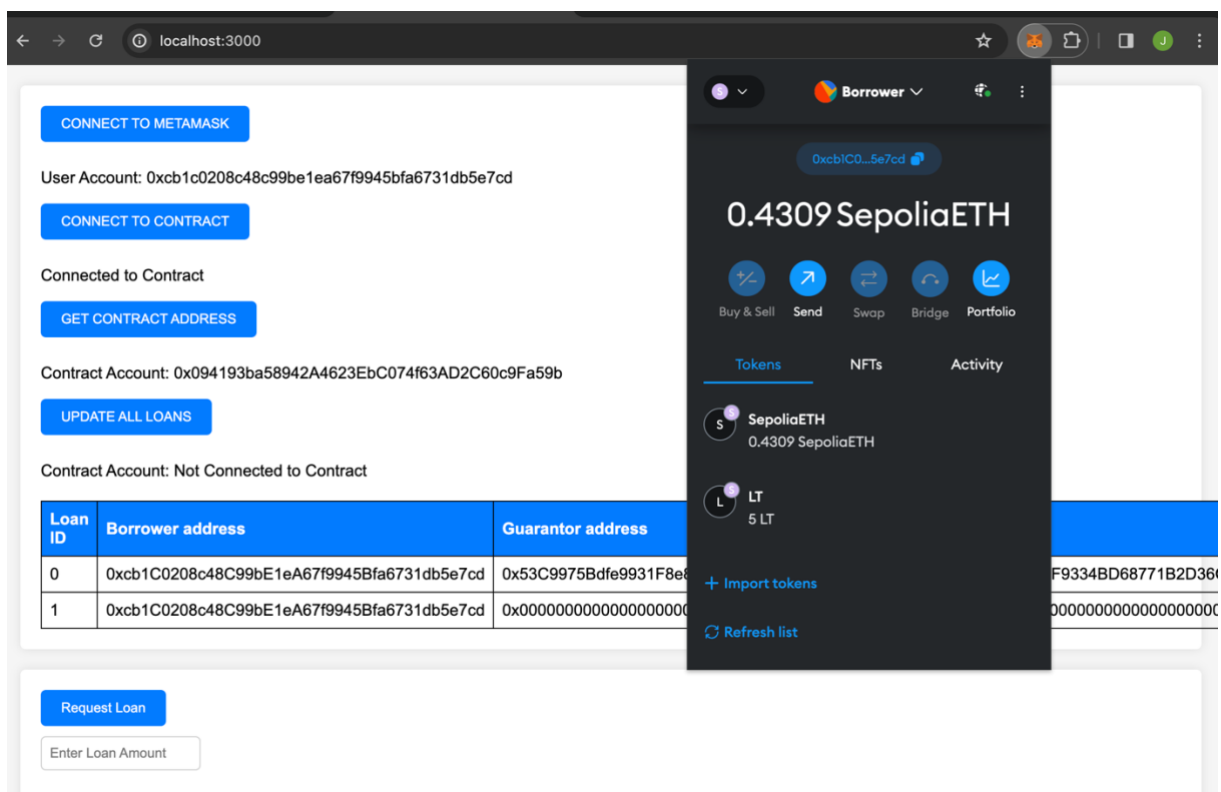
```
1  const {
2    time,
3    loadFixture,
4  } = require("@nomicfoundation/hardhat-toolbox/network-helpers");
5  const { anyValue } = require("@nomicfoundation/hardhat-chai-matchers/withArgs");
6  const { expect } = require("chai");
7
8  describe("LoanContract", function () {
9    async function deploy() {
10
11      // Contracts are deployed using the first signer/account by default
12      const [borrower, guarantor, lender] = await ethers.getSigners();
13      const owner = await ethers.getSigners("0x7Db9C84846809362E37cBBfb61f4c9e63fD0c992");
14      const LoanContract = await ethers.getContractFactory("LoanContract");
15      const loanContract = await LoanContract.deploy();
16
17      return { loanContract, owner, borrower, guarantor, lender };
18    }
19
20    //const { lock, unlockTime } = await loadFixture(deployOneYearLockFixture);
21
22    describe("addMoney", function () {
23
24      it("Should allow adding money", async function () {
25        const { loanContract, owner, guarantor } = await loadFixture(deploy);
26
27        await expect(
28          loanContract.connect(owner).addMoney(guarantor.address, 10000)
29        ).to.not.be.reverted;
30      });
31
32      it("Should not allow adding money because not owner", async function () {
33        const { loanContract, owner, guarantor } = await loadFixture(deploy);
34
35        await expect(
36          loanContract.connect(guarantor).addMoney(guarantor.address, 10000)
37        ).to.be.reverted;
38      });
39
40    });
41
42
43
44    describe("LoanRequest", function () {
45
```

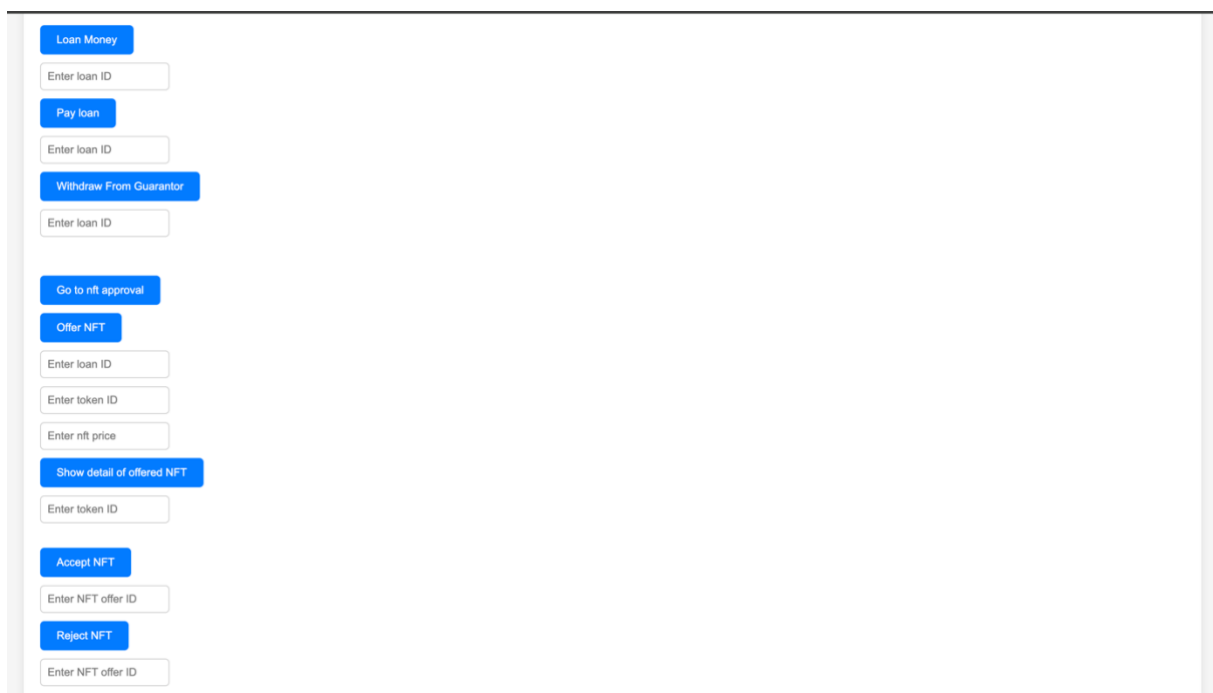
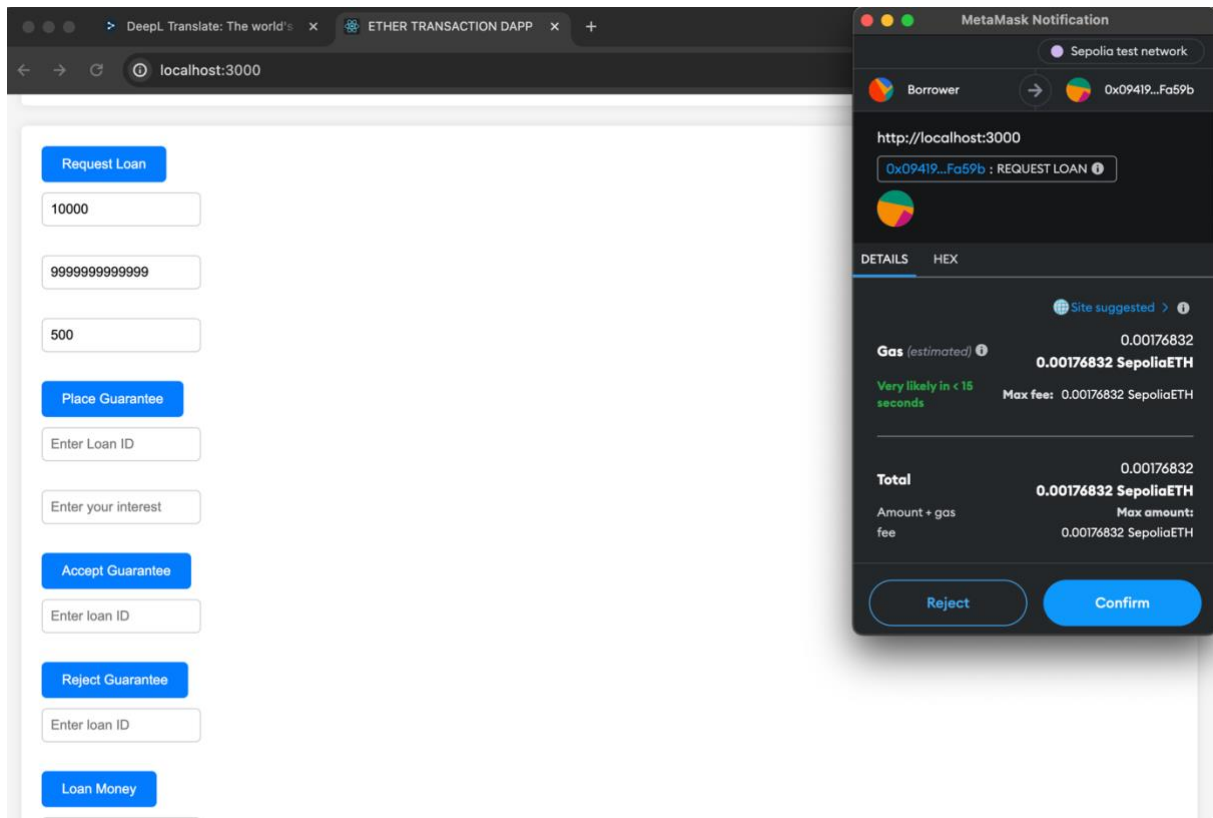

(iii) User interface

The express package is installed in the project. It contains the server.js file that runs the server. After entering the "npm start" command in the terminal, the server starts on port 300 where we can find our frontend. Within the frontend I have created 4 pages. 1 for the user to work with loans, 1 for the user to work with NFT, 1 for the owner to add money and 1 for the owner to add NFT.

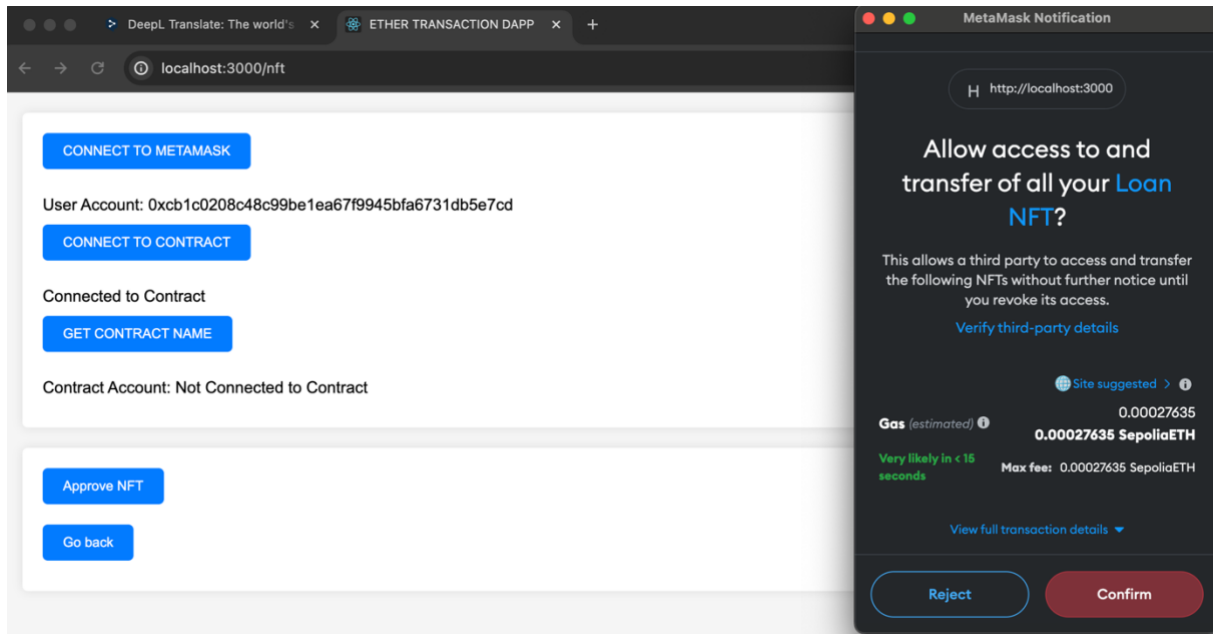
The user page is shared by the borrower, the lender, and the guarantor. This is because the guarantor can also apply for loans or lend etc. Specific issues, such as the guarantor not also being a lender, etc are taken care of by the contract itself.

The page contains a button to connect to the metamask, then to the contract itself and then all the functions for borrowing etc. In case the function needs to enter parameters, there are also textboxes next to the button.

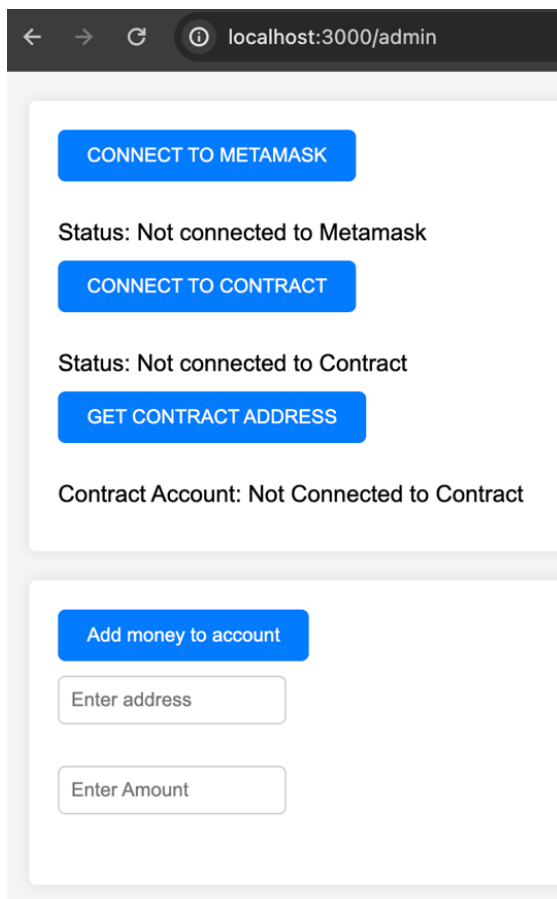




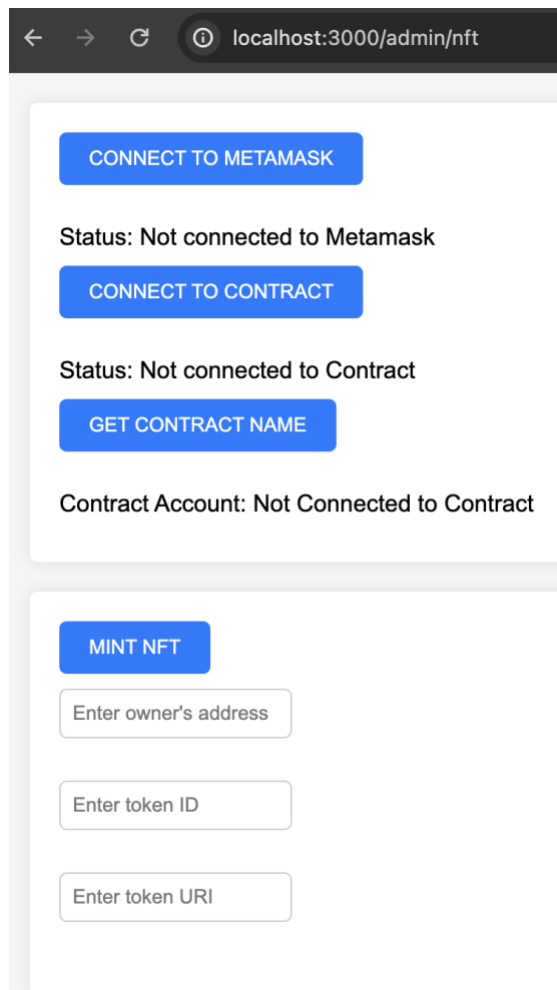
If needed, the user can click on the "Go to nft approval" button to get to the next page where he grants permission to the contract to manipulate his NFT.



In addition, our frontend includes an admin panel where you can add money to accounts. This is good to have separately, firstly for security reasons and secondly because a normal user will never be able to use this feature anyway and it would take up space unnecessarily.



It also includes a site for adding NFTs



localhost:3000/admin/nft

CONNECT TO METAMASK

Status: Not connected to Metamask

CONNECT TO CONTRACT

Status: Not connected to Contract

GET CONTRACT NAME

Contract Account: Not Connected to Contract

MINT NFT

Enter owner's address

Enter token ID

Enter token URI

The code for the frontend is of course also included in the project. Specifically, the files: server.js, index.html, userNFT.html, admin.html, adminNFT.html

Finally, I would like to discuss potential security issues. Most, hopefully all, of the most important security threats are secured by "require" functions. As I mentioned with each function, these methods ensure that the functions can only be called in the right situations, by the right users, and so on. However, we may run into a few problems. One of the most basic problems is that after a given loan is completed, the loan remains in contract, this was originally intentional to make the loans traceable, but in the case of a very very large number of loans, the loan id may already be such a large number that it exceeds the limits. Since this number is of type uint256, this is unlikely. On the other hand, our contract does allow you to create loans with a value of 1, for example, and spam loans into our contract. But since there is a fee for each request, this is also quite unlikely. In any case, it would be useful to create a blacklist of blocked users that make similar inappropriate requests.

Similar situations may arise, for example, where the guarantor deliberately gives very high interest rates when it is not worthwhile for any lender to make the loan. Of course, there is then a function to accept or reject the guarantee. However, the guarantor may give his guarantee repeatedly, again a blacklist could prevent this.