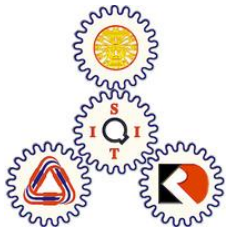


Capstone Project Technical Report

“ P2P Lending Dapp ”

Ms. Paurueta i Kobsahai 6322770064

Mr. Thanakrit Loetpricha 6322770114

Mr. Chanon Charuchinda 6322770692

DES484 Blockchain Development

Dr. Watthanasak Jeamwatthanachai

Sirindhorn International Institute of Technology Thammasat University

December 18, 2023

Table of Content

Part 1: Introduction

In recent years, Thailand has experienced a significant rise in the demand for borrowing funds. This trend is primarily driven by individuals and entities seeking capital to acquire various assets such as houses, vehicles, and for injecting funds into their businesses. Traditionally, the primary sources for such financial needs have been banks, credit service companies, and pawnshops. However, this conventional system poses several challenges that affect both borrowers and lenders.

1.1 The Problem Domain

1. **High Dependency on Traditional Financial Institutions:** The majority of loans are processed through banks and other financial institutions. While these entities provide a structured lending environment, their processes often involve extensive paperwork and procedural delays.
2. **Stringent Credit Checks:** A significant hurdle in securing loans from these institutions is the mandatory credit check. This requirement disqualifies many potential borrowers who do not meet the set credit standards, leaving them without viable options for obtaining loans.
3. **Alternative Lending Challenges:** Those unable to secure loans from formal institutions often resort to external, non-systemic sources. These sources, while not requiring credit history or other financial proof, come with their drawbacks. The most notable issues are exorbitant interest rates and lack of security for the lender, leading to a higher risk of default.
4. **The Dual Challenge of Security and Fairness:** Traditional lending systems often struggle to strike a balance between securing the lender's investment and providing fair terms to the borrower. This imbalance leads to either party being disadvantaged, fostering a non-conducive lending environment.

1.2 Proposed Solution

To address these challenges, our group proposes the development of a Peer-to-Peer (P2P) Lending and Borrowing System, leveraging the innovative capabilities of Blockchain technology

and Smart Contracts. This system aims to revolutionize the lending process by ensuring security, transparency, and fairness for all parties involved. The key features of this proposed system are:

1. **Decentralized Framework:** Utilizing Blockchain technology to create a decentralized lending platform, reducing dependency on traditional financial institutions.
2. **Smart Contracts for Enhanced Security:** Implementing Smart Contracts to automate the lending and borrowing process, ensuring the terms of the agreement are strictly followed and reducing the risk of default.
3. **Inclusive Lending:** By bypassing conventional credit checks, the system will enable a broader range of individuals and businesses to access loans, especially those previously marginalized by traditional credit systems.
4. **Fair and Transparent Transactions:** Blockchain technology ensures transparency in transactions, allowing both lenders and borrowers to engage in a fair and secure environment.

1.3 Identify the key requirements

1. **Blockchain Infrastructure:** The foundation of our P2P Lending and Borrowing System is a robust and scalable blockchain infrastructure. This platform is crucial for supporting the decentralized nature of the system and ensures the integrity and security of all transactions. Our blockchain infrastructure is designed to be highly efficient, capable of processing a high volume of transactions with speed and provides the necessary scalability as the user base grows. This infrastructure is the backbone of our system, ensuring that all activities are transparent, immutable, and tamper-proof.
2. **Smart Contract Development:** We have implemented sophisticated Smart Contracts, which are central to automating and securing the lending and borrowing processes. These contracts are self-executing with the terms of the agreement between the borrower and lender directly embedded in the code. The Smart Contracts govern critical aspects of a loan, such as the disbursal of funds, calculation of interest, and enforcement of repayment schedules. By leveraging Smart Contracts, we ensure that all agreements are executed precisely as agreed upon, thereby minimizing the risk of disputes and defaults.

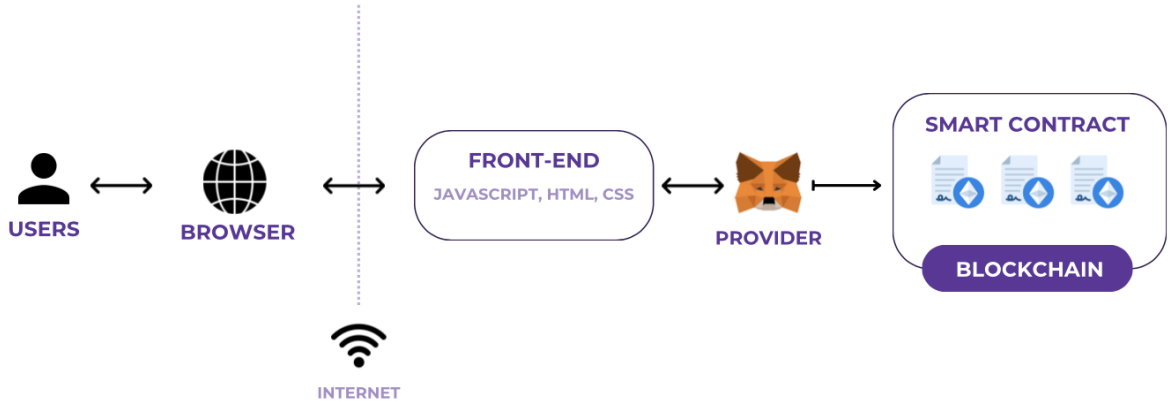
3. **User Verification and Security:** Our system prioritizes user security and authenticity. To achieve this, we have integrated MetaMask and Ganache for secure user authentication and interaction with the blockchain on a local host. This setup provides a secure environment for users to manage their digital identities and transactions. The use of MetaMask allows for a seamless and secure connection to the Ethereum blockchain, ensuring that all user data and financial transactions are protected through advanced encryption and security protocols.
4. **Credit Assessment Algorithm:** We've developed an innovative credit assessment algorithm that evaluates borrowers' creditworthiness using a broader set of criteria than traditional credit scores. This algorithm considers various factors such as transaction history, income stability, and digital footprints to provide a more comprehensive and fair assessment of a borrower's ability to repay. This approach allows us to extend credit services to a wider range of borrowers, including those who may not have a formal credit history.
5. **Decentralized Application (DApp) Interface:** The interface of our system is a user-friendly Decentralized Application (DApp). This interface is designed to be intuitive and accessible, enabling borrowers and lenders to easily navigate the platform. Users can apply for loans, set loan terms, view and accept loan offers, and monitor ongoing agreements with ease. The DApp interface enhances user experience by providing a seamless and transparent platform for all lending and borrowing activities.
6. **Legal Compliance and Contract Enforceability:** Our system is designed to operate within the legal frameworks of financial lending and borrowing. We ensure that all activities on our platform are compliant with relevant laws and regulations. This compliance extends to contract enforceability, where we have clear protocols in place for the resolution of disputes and enforcement of contract terms. By adhering to legal standards, we provide a secure and trustworthy environment for our users.
7. **Risk Management and Mitigation:** Risk management is a crucial aspect of our system. We have implemented strategies to assess and mitigate the risks associated with lending and borrowing. This includes provisions for loan collateral where applicable and mechanisms to handle default scenarios. Our approach to risk management ensures that

both lenders and borrowers are protected, maintaining the integrity and reliability of the platform.

8. **Monitoring and Reporting Tools:** To maintain transparency and enable effective decision-making, our system is equipped with advanced monitoring and reporting tools. These tools provide real-time insights into transactions and system performance. Users can access detailed reports to track their investments, returns, and loan repayments. This feature not only enhances user confidence in the platform but also aids in the continuous improvement and management of the system.

Part 2: The Architecture

Decentralized applications (dApps) are transforming the digital landscape by offering users autonomy, security, and transparency. Unlike traditional applications that rely on centralized servers, dApps are (partly or entirely) built on decentralized technology. So, designing the architecture for a peer-to-peer (P2P) lending decentralized application (DApp) involves several key components to ensure functionality, security, and user-friendliness. It consists of the following layers:



2.1 Smart Contracts Layer

This foundational layer includes various contracts that automate and enforce the rules of P2P lending:

1. **Borrowing Contract:** serves as a digital agreement between lenders and borrowers within a peer-to-peer lending platform. Its functions allow for the establishment of loan terms and the management of the lending process and enables setting and changing the lender, interest rate, and loan duration, as well as handling loan repayments and also provides mechanisms for claiming collateral for lenders in case of default.
2. **LoanManagement Contract:** can initiate lending pools and allocate funds by creating new lending contracts, while borrowers can establish borrowing contracts to receive loans.

3. **PoolLending Contract:** to facilitate the lending of funds and to provide a summary of the loan details.

2.1 Blockchain Layer

1. **Decentralization:** Utilizing platforms to deploy smart contracts for transparent and secure transactions.
2. **Transaction Management:** Managing the execution of blockchain transactions, including handling gas fees, and ensuring timeliness.

2.2 User Interface (UI) Layer / Frontend

1. **Web Interface:** A platform accessible across devices for a seamless user experience.

Part 3: Backend Development

3.1 Tools for develop the smart contract

Solidity: is a statically typed programming language designed for developing smart contracts that run on the Ethereum Virtual Machine (EVM). We use Solidity for writing smart contracts and are employed to create the smart contract code that defines the rules for lending and borrowing, handles the transfer of funds, enforces loan agreements, and maintains a ledger of all transactions on the blockchain.

Truffle: We use Truffle to set up environments with built-in functionalities for compiling contracts, managing deployment across various networks, running automated tests, and interacting with deployed contracts.

Chai: used for writing and executing automated tests to ensure that smart contracts function correctly to ensure the reliability and security of the smart contract code.

Node: are individual servers that participate in the blockchain network by validating and relaying transactions, support read and write operations. Example, if the application only allows reading data from the blockchain, no transaction fees need to be paid.

3.2 Smart Contract functionality

Smart Contracts	Function
BorrowingContract	- setLender - repayLoan - claimCollateral - setInterestRate - setLoanTerm - getSummary - getCollateralName - getCollateralValue - getCollateralOwner
LoanManagementContract	- **createLendingContract** (lender call this function to create new pool lending contract - constructor, send X Ethers as initial funds) - **createBorrowingContract** (borrower call this function to create new borrowing contracts) - constructor - borrowerRepayLoan - lenderClaimCollateralFromLoan - getLendingContractsByLender - getBorrowingContractByBorrwer - lendingContractSumamry - borrowingContractSummary - getProductDetailsFromBorrowingContract

	- getAllProductsOwnedBy
PoolLendingContract	- lendFunds - getSummary

3.2.1 BorrowingContract

The Borrower Contract is a sophisticated smart contract designed for seamless and secure transactions within the blockchain ecosystem. Aimed primarily at facilitating borrowing and lending processes, this contract introduces a range of functionalities to ensure transparency and efficiency. Key features include:

1) Constructor

```
constructor(
    address _borrower,
    uint _loanAmount,
    uint _ltv,
    uint _dueDate,
    address _lendingContractAddress,
    string memory _productName,
    uint _productValue
) public {
    borrowerAddress = address(uint160(_borrower));
    borrowingContractAddress = address(this);
    loanManagementContractAddress = msg.sender;
    lendingContractAddress = _lendingContractAddress;
    loanAmount = _loanAmount;
    interestRate = setInterestRate(_ltv);
    totalRepayableAmount = loanAmount + ((loanAmount * interestRate) / 100);
    dueDate = setLoanTerm(_dueDate);
    isFunded = false;
    isRepaid = false;

    collateral = CollateralProduct({
        name: _productName,
        value: _productValue,
        owner: _borrower
    });
    isCollateralClaimed = false;
}
```

2) Function

setLender: LoanManagement contract call this function during the creation of the borrowing contracts to assign the borrowing contract field of lenderAddress with the given address parameter.

```
function setLender(address _lenderAddress) external {
    require(
        _lenderAddress != borrowerAddress,
        "BC: Borrower cannot be the lender"
    );
    require(
        lenderAddress == address(0),
        "BC: Lender already set, cannot change lender address"
    );
    lenderAddress = address(uint160(_lenderAddress));
    isFunded = true;
}
```

repayLoan: Various conditions are checked to ensure the transaction is designated to the correct address. Only loanManagementContract can call this function. Only the borrower can repay the loan and sufficient loan amount with interest is needed to complete the transaction. Transfer the amount and mark contract as being repaid.

```
function repayLoan(address _borrower) external payable {
    require(
        msg.sender == loanManagementContractAddress,
        "BC: Unauthorized function call from LoanManagement"
    );
    require(
        _borrower == borrowerAddress,
        "BC: Only borrower can repay loan"
    );
    require(
        msg.value == totalRepayableAmount,
        "BC: Incorrect funding amount"
    );
    require(isFunded, "BC: Loan is not funded");

    require(
        block.timestamp <= dueDate,
        "BC: Loan term has already expired"
    );
    lenderAddress.transfer(msg.value);
    isRepaid = true;
}
```

claimCollateral: We take in the address of the lender as we need to check whether this transaction owner has the authority to claim the collateral (transfer the ownership or not). We need to make sure that the loan really is not being repaid and the loan is overdue, which we achieved using the requirement statement. Then we assign the collateral owner address to the lender address and mark the collaterals claimed.

```
function claimCollateral(address _lender) external {
    require(
        lenderAddress == _lender,
        "BC: Only lender can claim collateral"
    );
    require(!isRepaid, "BC: Loan is already repaid");
    require(block.timestamp > dueDate, "BC: Loan term has not expired yet");

    collateral.owner = lenderAddress;
    isCollateralClaimed = true;
}
```

setInterestRate: Our MVP allows only three values for loan-to-value. This function checks if the input is valid ltv terms or not. Then the function procedures to convert the ltv to its associated interest rate.

```
function setInterestRate(uint _ltv) internal pure returns (uint) {
    require(
        _ltv == 25 || _ltv == 50 || _ltv == 70,
        "BC: Invalid LTV option (only 25,50, or 70)"
    );
    return _ltv == 25 ? 5 : (_ltv == 50 ? 9 : 12);
}
```

setLoanTerm: Our MVP allows only three loan terms in days.The value block.timestamp is used to retrieved the current time. Then we find the due date by adding the specified loan term to the current time.

```
function setLoanTerm(uint loanTermDays) internal view returns (uint) {
    require(
        loanTermDays == 0 || loanTermDays == 30 || loanTermDays == 365,
        "BC: Invalid loan term (only 0, 30, 365 days)"
    );
    return
        loanTermDays == 0
            ? block.timestamp - (30 * 24 * 60 * 60) // 30 days ago
            : block.timestamp + (loanTermDays * 1 days);
}
```

getSummary: Return summary of a borrowing contract.

```
function getSummary()
    external
    view
    returns (
        address _borrower,
        address _lender,
        address _borrowingContractAddress,
        address _lendingContractAddress,
        uint _amount,
        uint _interestRate,
        uint _dueDate,
        uint _totalRepayableAmount,
        bool _isFunded,
        bool _isRepaid
    )
{
    return (
        borrowerAddress,
        lenderAddress,
        borrowingContractAddress,
        lendingContractAddress,
        loanAmount,
        interestRate,
        dueDate,
        totalRepayableAmount,
        isFunded,
        isRepaid
    );
}
```

getCollateralName: Get collateral name.

```
function getCollateralName() external view returns (string memory) {
    return collateral.name;
}
```

getCollateralValue : Get collateral value.

```
function getCollateralValue() external view returns (uint) {
    return collateral.value;
}
```

getCollateralOwner: Get address of the collateral owner.

```
function getCollateralOwner() external view returns (address) {
    return collateral.owner;
}
```

The Borrower Contract, with these integrated functions, stands as a testament to the potential of blockchain technology in revolutionizing the traditional borrowing and lending landscape, offering a higher degree of security, efficiency, and trust.

3.2.2 PoolLendingContract

The Pool Lending Contract is a streamlined smart contract developed for the blockchain platform, specifically designed to facilitate pooled lending activities. Its core functionalities include:

1) **Constructor**

```
constructor(address _lenderAddress) public payable {
    lenderAddress = _lenderAddress;
    totalFunds = msg.value;
    availableFunds = msg.value;
    numLoans = 0;
    loanManagementContractAddress = msg.sender;
    lendingContractAddress = address(this);
}
```

2) Function

lendFunds: This pivotal function allows participants to lend their funds into a collective pool. It is instrumental in aggregating resources from multiple lenders to provide substantial loan amounts.

```
function lendFunds(address _borrower, uint _amount) external {
    require(
        msg.sender == loanManagementContractAddress,
        "PLC: Unauthorized Loan Manager"
    );
    require(_amount <= availableFunds, "PLC: Not enough available funds");

    availableFunds = availableFunds - _amount;
    address payable borrowerPayable = address(uint160(_borrower));
    borrowerPayable.transfer(_amount);
    numLoans++;
}
```

getSummary: This feature provides a comprehensive overview of the contract's status, including total funds lent, active loan details, and overall pool performance.

```
function getSummary()
    public
    view
    returns (
        address _lenderAddress,
        address _lendingContractAddress,
        uint _totalFunds,
        uint _availableFunds
    )
{
    return (
        lenderAddress,
        lendingContractAddress,
        totalFunds,
        availableFunds
    );
}
```

3.2.3 LoanManagementContract

The Loan Management Contract is a dynamic smart contract on the blockchain tailored for efficient loan management.

1) **Constructor**

```
constructor() public {
    dappName = "BigBoy Lending Corp.";
}

event PoolLendingContractCreated(
    address lendingContractAddress,
    address lender,
    uint totalFunds
);

event BorrowingContractCreated(
    address borrowingContractAddress,
    address borrower,
    uint amount,
    string productName,
    uint productValue
);

event LoanRepaid(
    address borrowingContractAddress,
    address borrower,
    uint amountRepaid
);

event CollateralClaimed(
    address borrowingContractAddress,
    address lender,
    string collateralName
);
```

2) **Function**

createLendingContract: Lender calls this function to create a new pool lending contract - constructor, send X Ethers as initial funds. Using mapping to store the lending contract associated with the wallet address along with storing the address of the lending contract into another mapping using the integer as key for eases data retrieval.

```
function createLendingContract() external payable {
    PoolLendingContract newLendingContract = (new PoolLendingContract)
        .value(msg.value)(msg.sender);
    lendingContracts[msg.sender].push(address(newLendingContract));

    lendingContractCount++;
    lendingContractAddresses[lendingContractCount] = address(
        newLendingContract
    );

    emit PoolLendingContractCreated(
        address(newLendingContract),
        msg.sender,
        msg.value
    );
}
```

createBorrowingContract: Borrower calls this function to create new borrowing contracts. Firstly, we have to check if the lending contract exists or not. Then we check if the lending contract has enough funds for the borrower requested amount. We check the correctness of collateral information like valid name length and the collateral amount has to be greater than the borrowing amount to ensure that the lender doesn't get compromised if the borrower refuses to repay the loan. Mapping is used to store the borrowing contract associated with the wallet address into a list.

```
function createBorrowingContract(
    address _lendingContractAddress,
    uint _amount,
    uint _interestRate,
    uint _dueDate,
    string calldata _productName,
    uint _productValue
) external {
    PoolLendingContract lendingContract = PoolLendingContract(
        _lendingContractAddress
    );
    require(
        _amount <= lendingContract.availableFunds(),
        "LMC: Insufficient funds"
    );
    require(
        bytes(_productName).length > 0,
        "LMC: Product cannot have no name"
    );
    require(
        _productValue > _amount,
        "LMC: Collateral must be creater than loan amount"
    );

    BorrowingContract newBorrowingContract = new BorrowingContract(
        msg.sender,
        _amount,
        _interestRate,
        _dueDate,
        _lendingContractAddress,
        _productName,
        _productValue
    );
```

```
    borrowingContracts[msg.sender].push(address(newBorrowingContract));
    lendingContract.lendFunds(msg.sender, _amount);
    newBorrowingContract.setLender(lendingContract.lenderAddress());

    borrowingContractCount++;
    borrowingContractAddresses[borrowingContractCount] = address(
        newBorrowingContract
    );

    // Retrieve values from the new borrowing contract (for event)
    string memory retrievedProductName = newBorrowingContract
        .getCollateralName();
    uint retrievedProductValue = newBorrowingContract.getCollateralValue();
    address retrievedBorrower = newBorrowingContract.borrowerAddress();
    uint retrievedLoanAmount = newBorrowingContract.loanAmount();

    emit BorrowingContractCreated(
        address(newBorrowingContract),
        retrievedBorrower,
        retrievedLoanAmount,
        retrievedProductName,
        retrievedProductValue
    );
}
```

borrowerRepayLoan: Call the repayLoan functon from loanManagement contract

```
function borrowerRepayLoan(
    address _borrowingContractAddress
) external payable {
    BorrowingContract borrowingContract = BorrowingContract(
        _borrowingContractAddress
    );
    require(
        address(borrowingContract) != address(0),
        "LMC: Borrowing Contract does not exist"
    );
    require(
        msg.sender == borrowingContract.borrowerAddress(),
        "LMC: Only borrower can repay the loan"
    );

    borrowingContract.repayLoan.value(msg.value)(msg.sender);

    emit LoanRepaid(_borrowingContractAddress, msg.sender, msg.value);
}
```

lenderClaimCollateralFromLoan: Called claimCollaterla function form the loanManagement Contract.

```
function lenderClaimCollateralFromLoan(
    address _borrowingContractAddress
) external {
    BorrowingContract borrowingContract = BorrowingContract(
        _borrowingContractAddress
    );
    require(
        address(borrowingContract) != address(0),
        "LMC: Borrowing Contract does not exist"
    );
    require(
        msg.sender == borrowingContract.lenderAddress(),
        "LMC: Only lender can claimed borrower's collateral"
    );

    borrowingContract.claimCollateral(msg.sender);

    string memory collateralName = borrowingContract.getCollateralName();
    emit CollateralClaimed(
        _borrowingContractAddress,
        msg.sender,
        collateralName
    );
}
```

getLendingContractsByLender

```
function getLendingContractsByLender(
    address _lenderAddress
) external view returns (address[] memory) {
    return lendingContracts[_lenderAddress];
}
```

getBorrowingContractByBorrwer

```
function getBorrowingContractByBorrwer(
    address _borrowerAddress
) external view returns (address[] memory) {
    return borrowingContracts[_borrowerAddress];
}
```

lendingContractSumamry

```
function lendingContractSumamry(
    address _lendingContractAddress
)
    external
    view
    returns (
        address _lender,
        address _lendingContractAddressFromSelf,
        uint _totalFunds,
        uint _availableFunds
    )
{
    require(
        address(PoolLendingContract(_lendingContractAddress)) != address(0),
        "LMC: Lending contract doesn't exist"
    );
    PoolLendingContract lendingContract = PoolLendingContract(
        _lendingContractAddress
    );
    return lendingContract.getSummary();
}
```

borrowingContractSummary: Borrowing contract summary

```
function borrowingContractSummary(
    address _borrowingContractAddress
)
    external
    view
    returns (
        address _borrower,
        address _lender,
        address _borrowingContractAddressFromSelf,
        address _lendingContractAddress,
        uint _amount,
        uint _interestRate,
        uint _dueDate,
        uint _totalRepayableAmount,
        bool _isFunded,
        bool _isRepaid
    )
{
    require(
        address(BorrowingContract(_borrowingContractAddress)) != address(0),
        "LMC: Borrowing contract doesn't exist"
    );
    BorrowingContract borrowingContract = BorrowingContract(
        _borrowingContractAddress
    );
    return borrowingContract.getSummary();
}
```

getProductDetailsFromBorrowingContract: Fet collaterla products from the borrowing address.

```
function getProductDetailsFromBorrowingContract(
    address _borrowingContractAddress
) external view returns (string memory name, uint value, address owner) {
    require(
        address(BorrowingContract(_borrowingContractAddress)) != address(0),
        "LMC: Borrowing contract doesn't exist"
    );

    BorrowingContract borrowingContract = BorrowingContract(
        _borrowingContractAddress
    );
    return (
        borrowingContract.getCollateralName(),
        borrowingContract.getCollateralValue(),
        borrowingContract.getCollateralOwner()
    );
}
```

getAllProductsOwnedBy: Return products associated to an address.

```
function getAllProductsOwnedBy(
    address _owner
) external view returns (CollateralProduct[] memory) {
    address[] memory borrowerContracts = borrowingContracts[_owner];
    uint count = 0;

    for (uint i = 0; i < borrowerContracts.length; i++) {
        BorrowingContract borrowingContract = BorrowingContract(
            borrowerContracts[i]
        );
        if (borrowingContract.getCollateralOwner() == _owner) {
            count++;
        }
    }

    CollateralProduct[] memory ownedProducts = new CollateralProduct[](
        count
    );
    uint j = 0;

    for (uint i = 0; i /.length< borrowerContracts.length; i++) {
        BorrowingContract borrowingContract = BorrowingContract(
            borrowerContracts[i]
        );
        if (borrowingContract.getCollateralOwner() == _owner) {
            string memory name = borrowingContract.getCollateralName();
            uint value = borrowingContract.getCollateralValue();
            address owner = borrowingContract.getCollateralOwner();

            CollateralProduct memory product = CollateralProduct(
                name,
                value,
                owner
            );
            ownedProducts[j] = product;
            j++;
        }
    }
    return ownedProducts;
}
```

This Loan Management Contract serves as a versatile tool for both lenders and borrowers in the blockchain space, simplifying the creation and oversight of lending and borrowing activities while ensuring transparency and security.

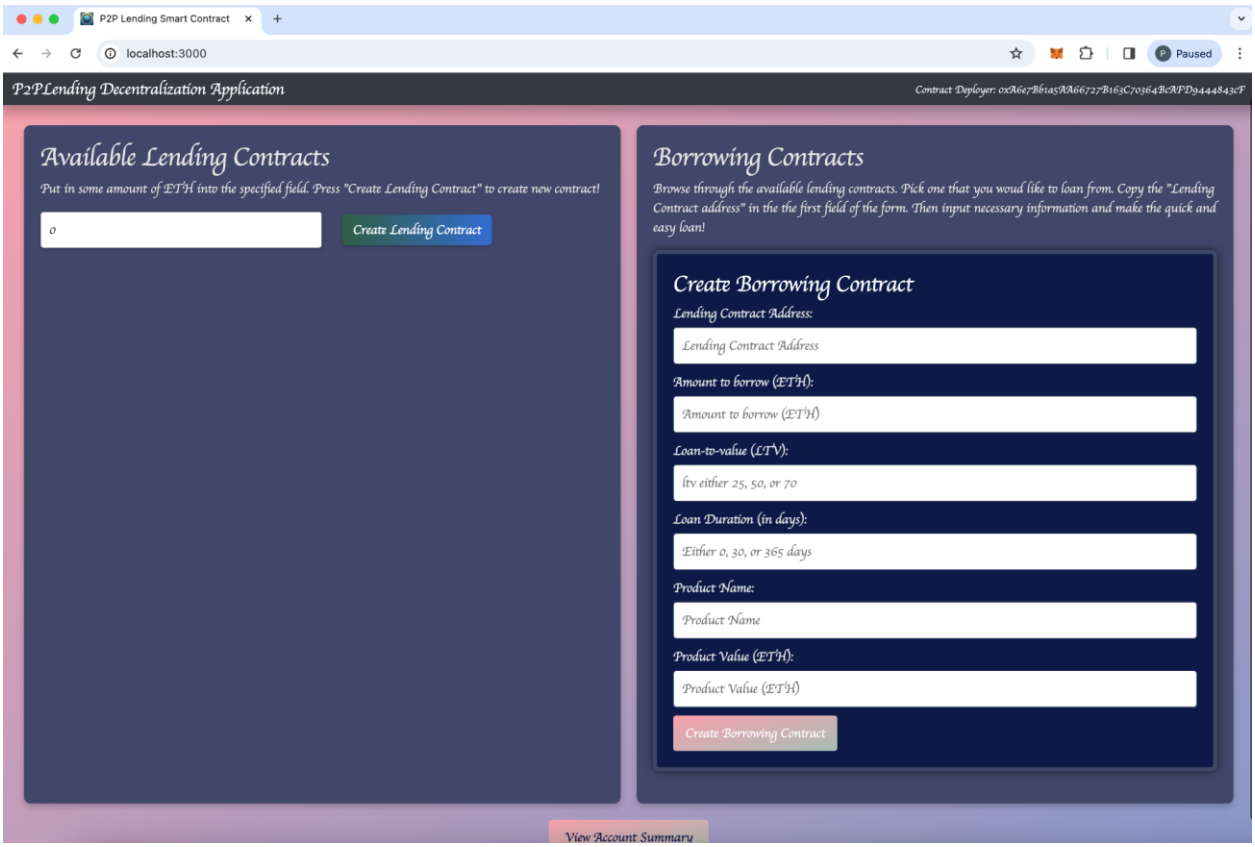
Part 4: Frontend Development

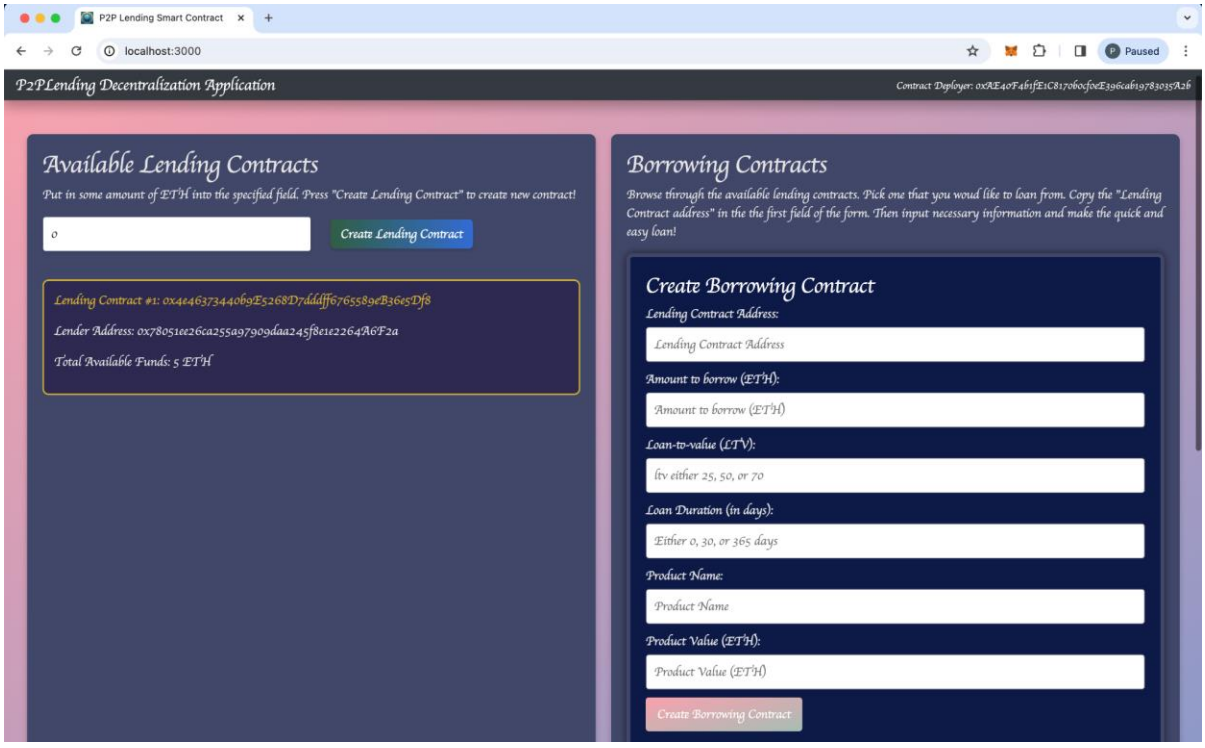
4.1 Tools for develop Frontend

We use web technologies **Web3, HTML, CSS, JavaScript**, Frameworks such as **React** to create responsive and dynamic user interface (UI) of a dApp for purpose a connection to a node is required and communication.

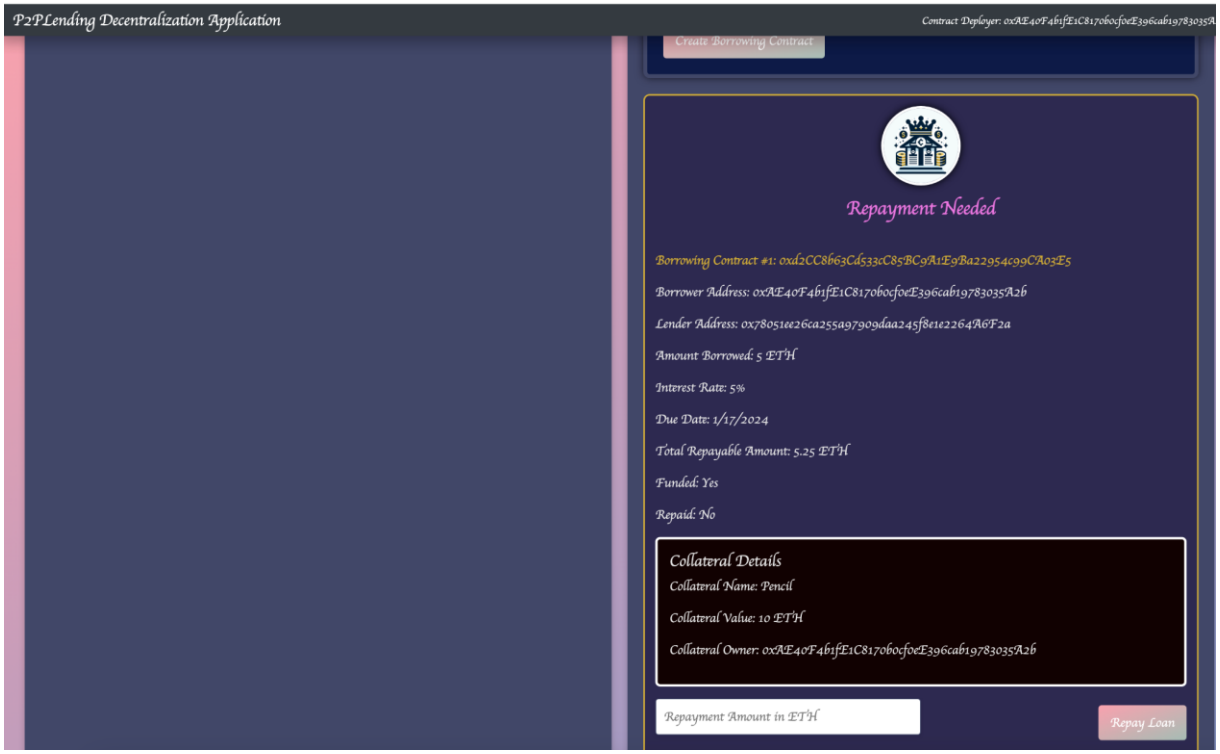
4.2 Website Demo

The process begins with the user creating a Lending Contract. Once the details of the lending terms are agreed upon, the user initiates the transaction which is then confirmed through MetaMask, a crypto wallet and gateway to blockchain apps. Upon confirmation, the platform reflects the updated lending contract, signaling the successful allocation of funds for potential borrowers.





Lending Contract created

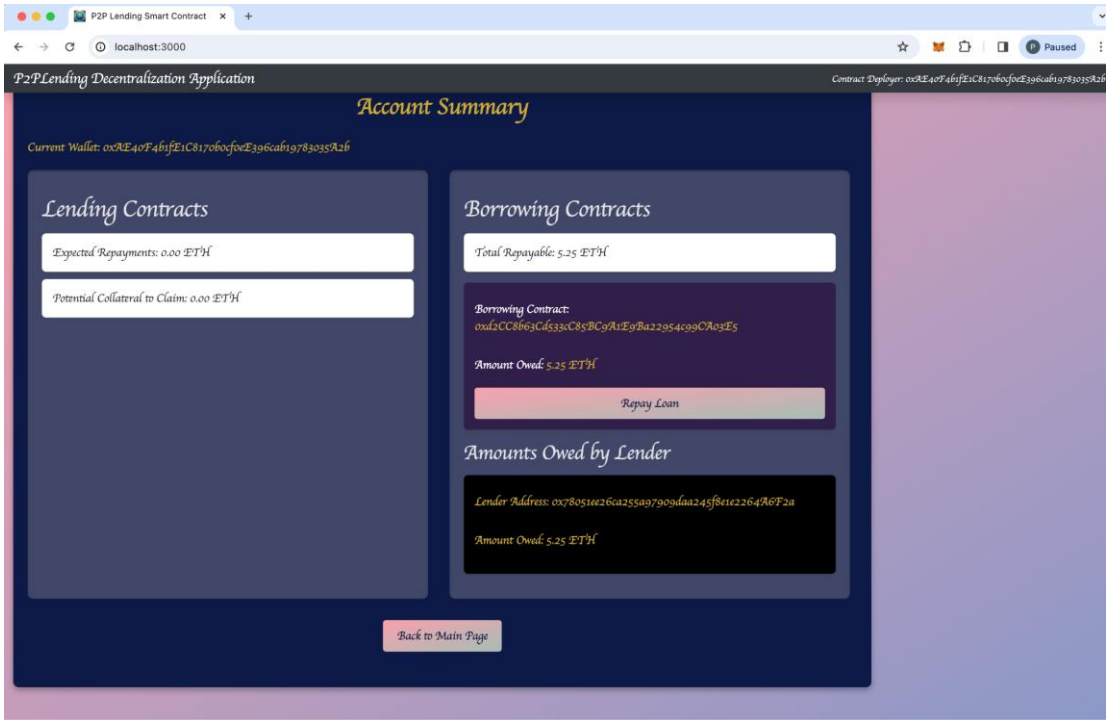


Borrowing Contract created

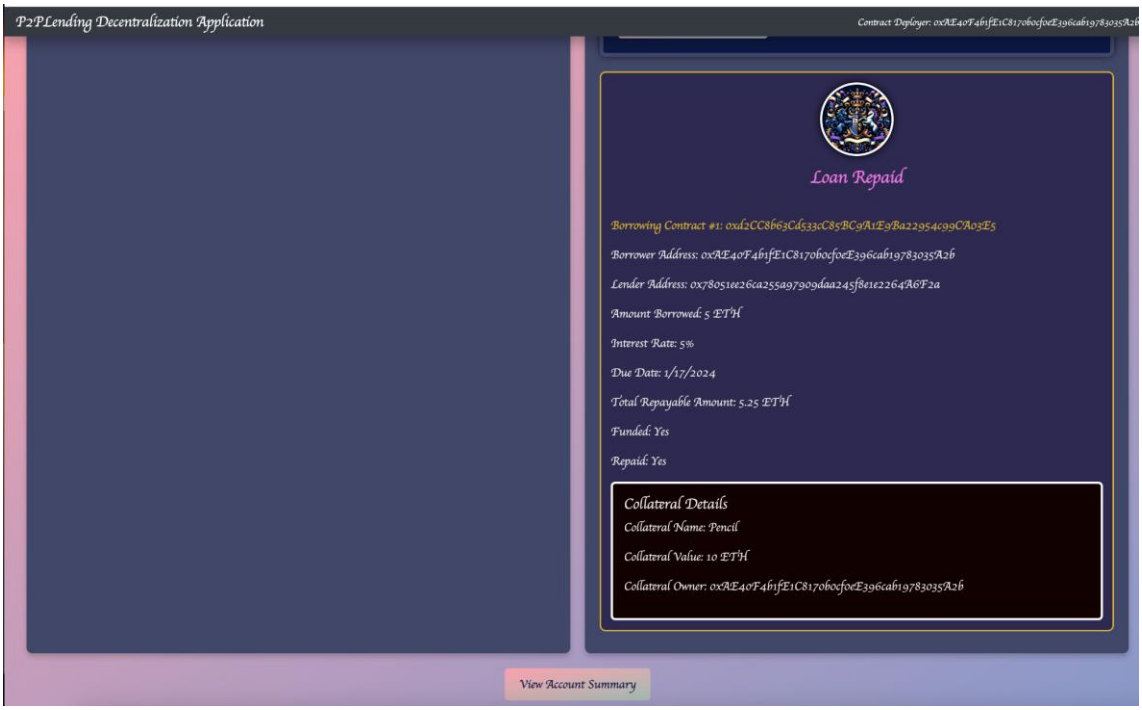
Subsequently, a Borrowing Contract is crafted. Here, a borrower peruses available loans and selects one that aligns with their needs. Similar to the lending process, this action is finalized through a MetaMask transaction. The blockchain is updated, and the borrowing contract is established, linking the borrower to the lender under agreed conditions.

As time progresses, the borrower is required to repay the loan. This crucial step is monitored by the platform to ensure timely execution.

In the event of a default, where the loan is not repaid in time, the lender is empowered to claim the collateral. The platform records this event and updates to indicate that the ownership of the collateral has been transferred to the lender, completing the risk mitigation cycle built into the lending agreement.



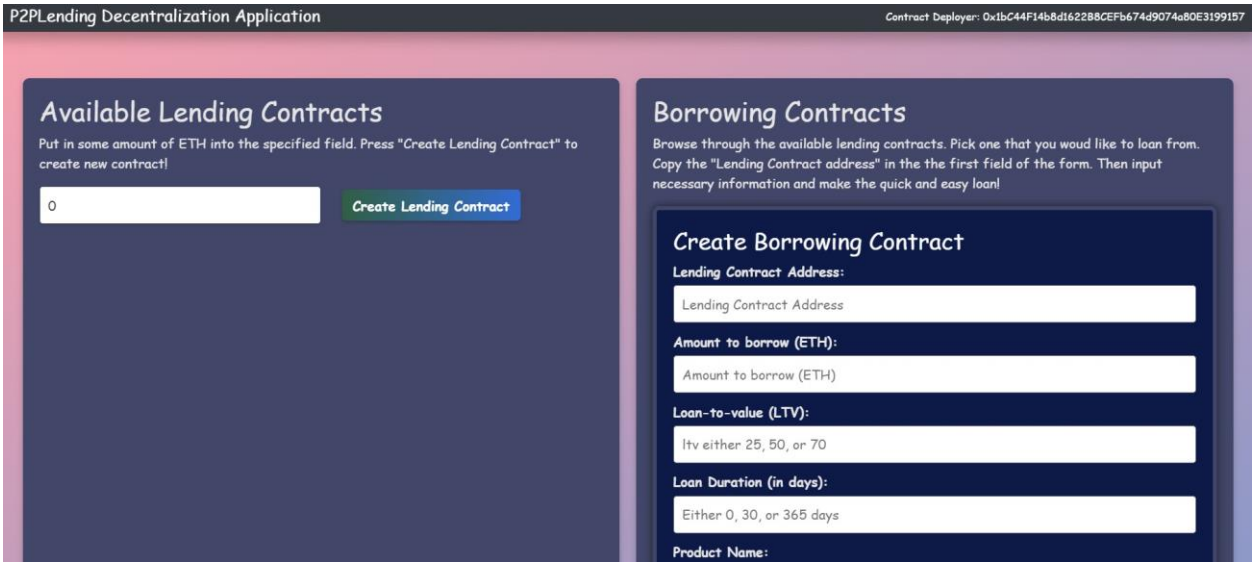
Account Summary


P2PLending Decentralization Application
Contract Deployer: 0xAE40F4b1fE1C8170b0cf0eE396cab19783035A2b
Loan Repaid
Borrowing Contract #1: 0xd2CC8b63Cd533cC85BC9A1E9Ba22954c99CA03E5
Borrower Address: 0xAE40F4b1fE1C8170b0cf0eE396cab19783035A2b
Lender Address: 0x78051ee26ca255a97909daa245f8e1e2264A6F2a
Amount Borrowed: 5 ETH
Interest Rate: 5%
Due Date: 1/17/2024
Total Repayable Amount: 5.25 ETH
Funded: Yes
Repaid: Yes
Collateral Details
Collateral Name: Pencil
Collateral Value: 10 ETH
Collateral Owner: 0xAE40F4b1fE1C8170b0cf0eE396cab19783035A2b
View Account Summary

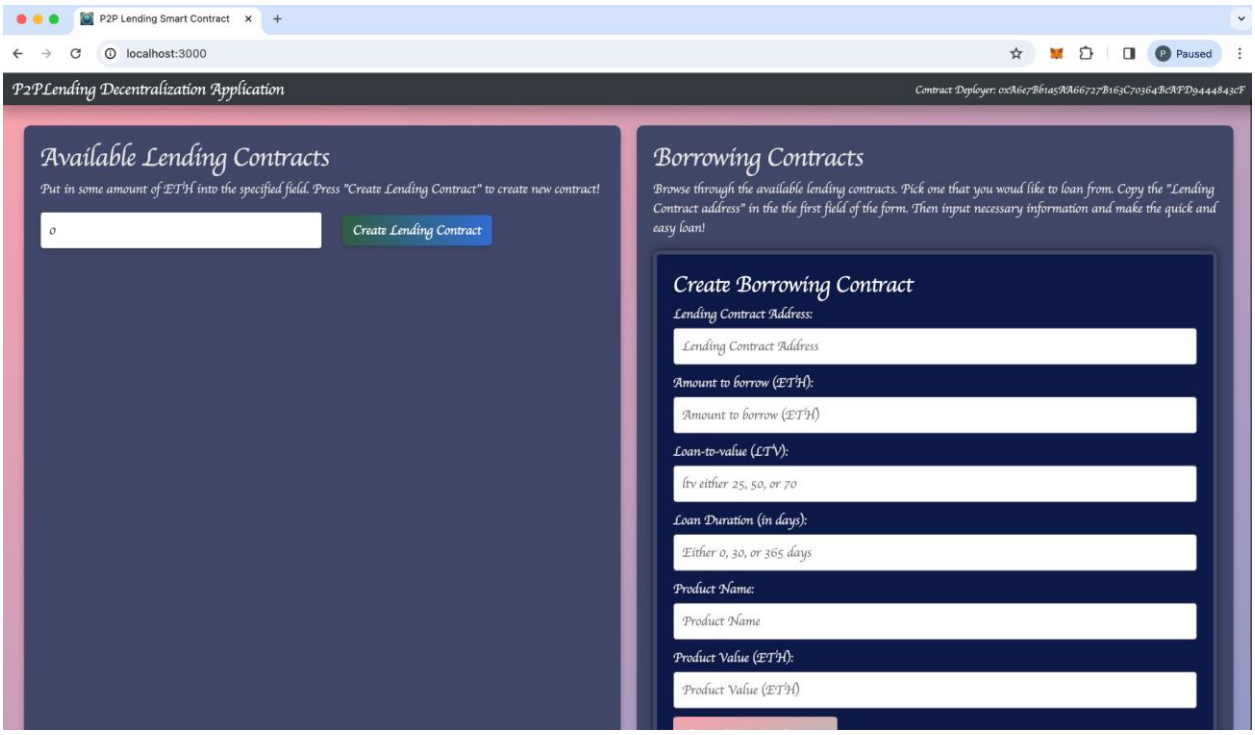
Loan Repaid

4.3 The Different of UI website on Windows and MAC OS

The font style is different for Mac and Window



Website on Windows



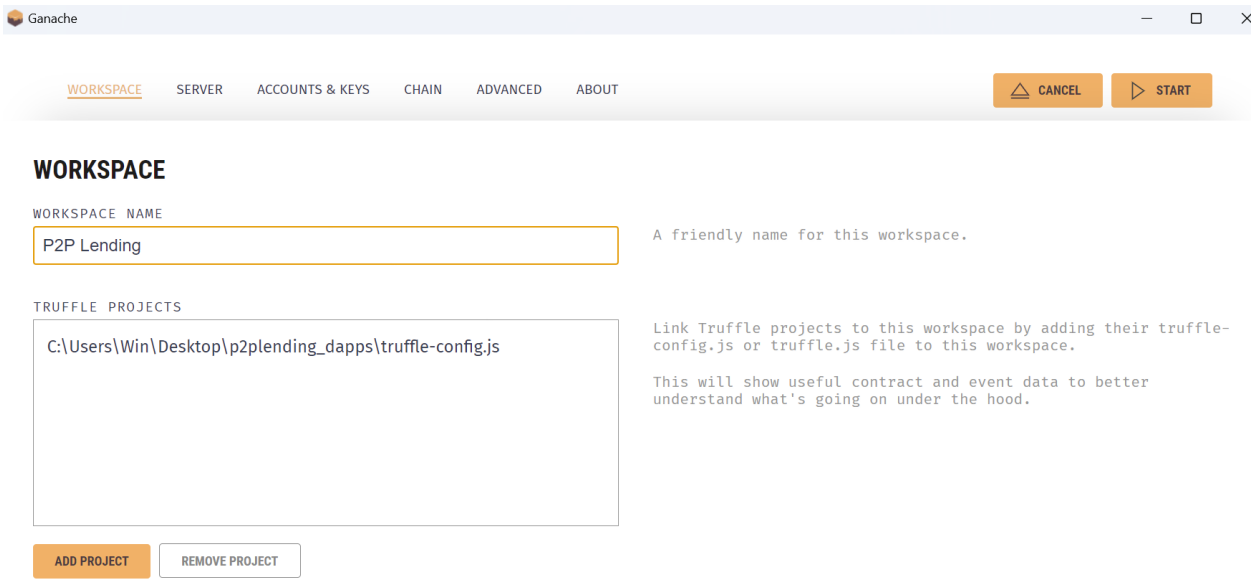
Website on MAC OS

Part 5: Test and Debug

5.1 Testing

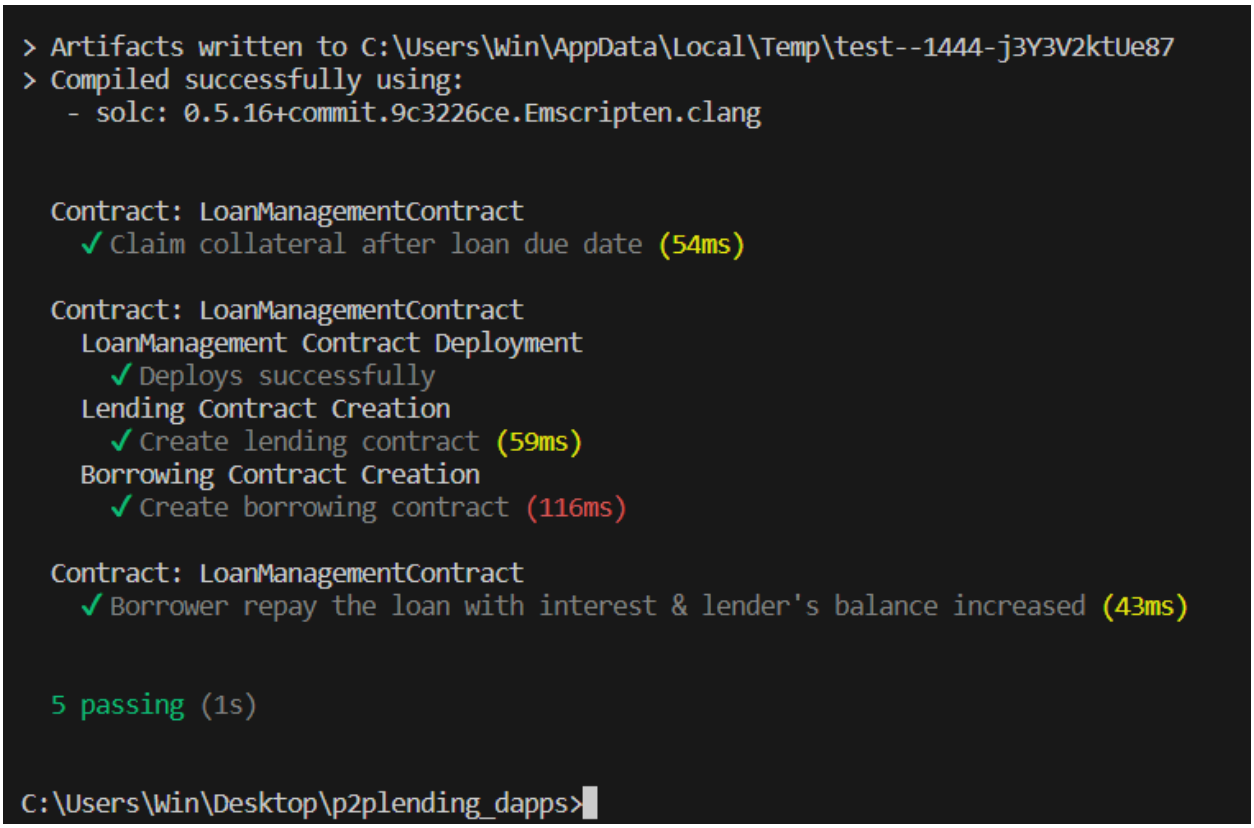
We started the development of our decentralized application based on the starter kit provided by the YouTube channel called “Dapp University”. We followed the installation process and project setup and used the provided source code from the video title “Intro to Blockchain Programming (Etherum, Web3.js & Solidity Smart Contracts) [FULL COURSE]”. Here are a few steps to get the project started.

1. Installed the required modules including Truffle, Node.js (node), npm, and Ganache for hosting and deploying the blockchain on local network.
2. Download our project source code.
3. Open a new workspace on Ganache, rename the workspace to something like “P2P Lending” and click “ADD PROJECT”. This will prompt a window popup. Then navigate to our source code, locate the “truffle-config.js” then click “START” or “SAVE WORKSPACE” (for the older version of Ganache) at the top right-hand corner.



4. Open the source code in some IDE like Vscode and open a new terminal to run the following command and note that the Ganache needs to be running in the background.
    a. truffle compile
    b. truffle migrate - - reset

5. To test the contracts, please type ‘truffle test’ into a separate command line or terminal



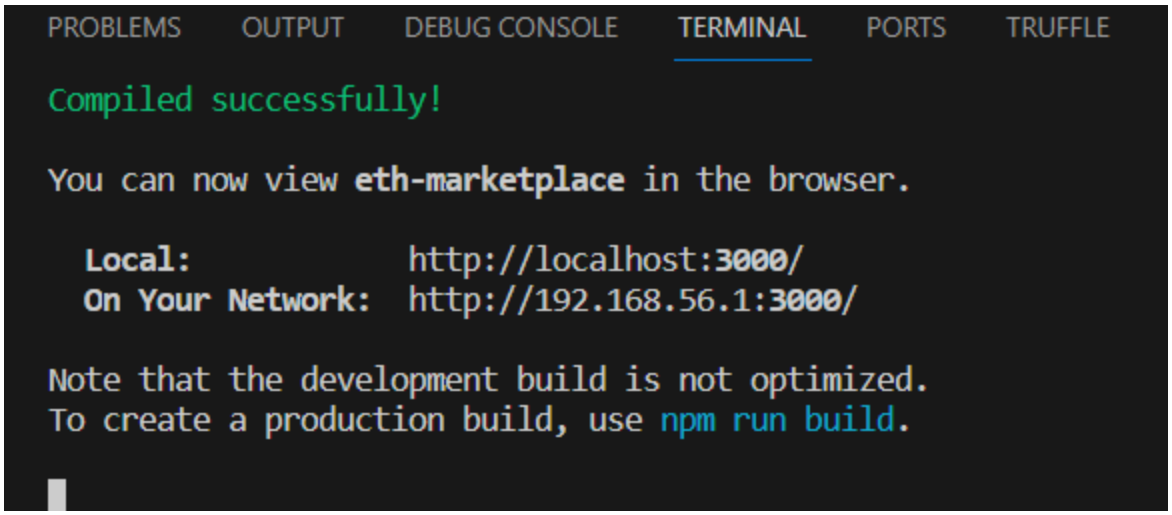
As the name of the test suggests, we test the main functionalities that peer-to-peer decentralized applications should be able to do.

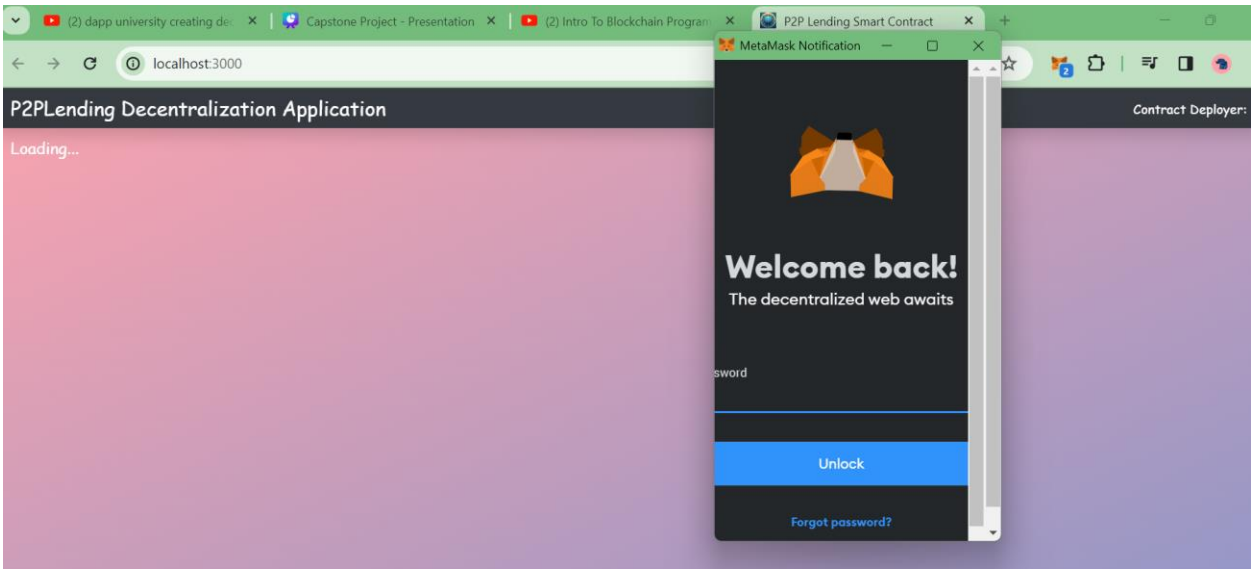
6. To start the application, we have to run the command “npm run start”. In the following image, we will show that there might be the following error due to a version of node mismatch. In order to solve this issue, run the following command depending on your operating system
    a. For Window: set NODE_OPTIONS=--openssl-legacy-provider
    b. For macOS: chmod +x node_modules/.bin/react-scripts

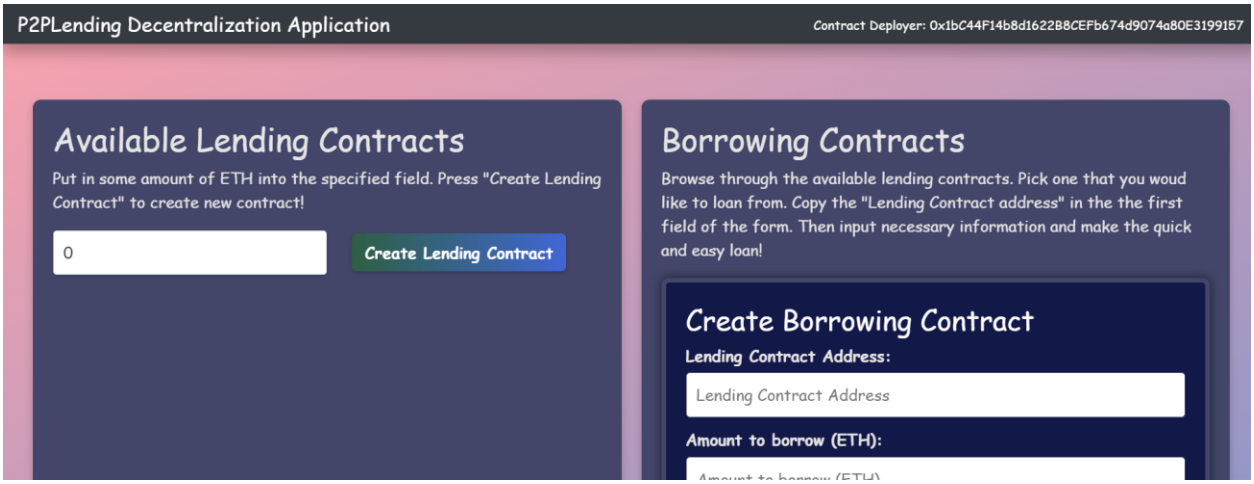
```
Error: error:0308010C:digital envelope routines::unsupported
    at new Hash (node:internal/crypto/hash:68:19)
    at Object.createHash (node:crypto:138:10)
    at module.exports (C:\Users\Win\Desktop\p2plending_dapps\node_modules\webpack\lib\util\createHash
    at NormalModule._initBuildHash (C:\Users\Win\Desktop\p2plending_dapps\node_modules\webpack\lib\No
    at handleParseError (C:\Users\Win\Desktop\p2plending_dapps\node_modules\webpack\lib\NormalModule.
    at C:\Users\Win\Desktop\p2plending_dapps\node_modules\webpack\lib\NormalModule.js:466:5
    at C:\Users\Win\Desktop\p2plending_dapps\node_modules\webpack\lib\NormalModule.js:327:12
    at C:\Users\Win\Desktop\p2plending_dapps\node_modules\loader-runner\lib\LoaderRunner.js:373:3
    at iterateNormalLoaders (C:\Users\Win\Desktop\p2plending_dapps\node_modules\loader-runner\lib\Loa
    at iterateNormalLoaders (C:\Users\Win\Desktop\p2plending_dapps\node_modules\loader-runner\lib\Loa
    at C:\Users\Win\Desktop\p2plending_dapps\node_modules\loader-runner\lib\LoaderRunner.js:236:3
    at runSyncOrAsync (C:\Users\Win\Desktop\p2plending_dapps\node_modules\loader-runner\lib\LoaderRun
    at iterateNormalLoaders (C:\Users\Win\Desktop\p2plending_dapps\node_modules\loader-runner\lib\Loa
    at Array.<anonymous> (C:\Users\Win\Desktop\p2plending_dapps\node_modules\loader-runner\lib\Loader
    at Storage.finished (C:\Users\Win\Desktop\p2plending_dapps\node_modules\enhanced-resolve\lib\Cach
    at C:\Users\Win\Desktop\p2plending_dapps\node_modules\enhanced-resolve\lib\CachedInputFileSystem.
C:\Users\Win\Desktop\p2plending_dapps\node_modules\react-scripts\scripts\start.js:19
  throw err;
  ^
```

```
C:\Users\Win\Desktop\p2plending_dapps\node_modules\react-scripts\scripts\start.js:19
  throw err;
  ^

Error: error:0308010C:digital envelope routines::unsupported
    at new Hash (node:internal/crypto/hash:68:19)
    at Object.createHash (node:crypto:138:10)
    at module.exports (C:\Users\Win\Desktop\p2plending_dapps\node_modules\webpack\lib\util\createHash.js:90:53)
    at NormalModule._initBuildHash (C:\Users\Win\Desktop\p2plending_dapps\node_modules\webpack\lib\NormalModule.js
    at C:\Users\Win\Desktop\p2plending_dapps\node_modules\webpack\lib\NormalModule.js:418:10
    at C:\Users\Win\Desktop\p2plending_dapps\node_modules\webpack\lib\NormalModule.js:293:13
    at C:\Users\Win\Desktop\p2plending_dapps\node_modules\loader-runner\lib\LoaderRunner.js:367:11
    at C:\Users\Win\Desktop\p2plending_dapps\node_modules\loader-runner\lib\LoaderRunner.js:233:18
    at context.callback (C:\Users\Win\Desktop\p2plending_dapps\node_modules\loader-runner\lib\LoaderRunner.js:111:
    at C:\Users\Win\Desktop\p2plending_dapps\node_modules\babel-loader\lib\index.js:51:103 {
  opensslErrorStack: [ 'error:03000086:digital envelope routines::initialization error' ],
  library: 'digital envelope routines',
  reason: 'unsupported',
  code: 'ERR_OSSL_EVP_UNSUPPORTED'
}

Node.js v20.9.0
```

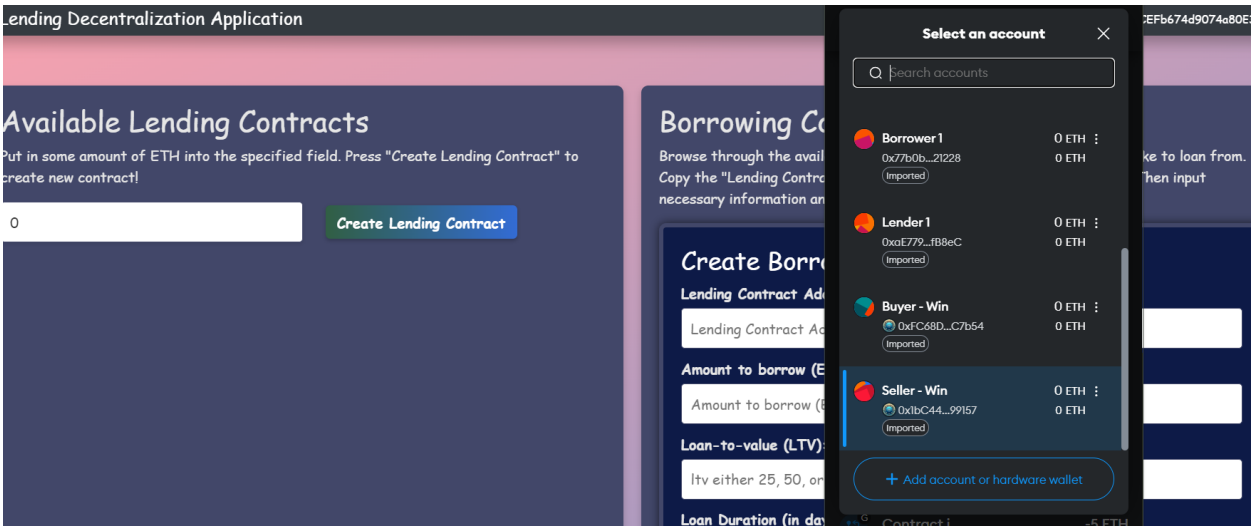
7. The program should run successfully, and you will be greeted with the following page. Please run the program on a browser with MetaMask Google Chrome extension installed. A password for MetaMask should be provided as well. Afterward, the application homepage will be displayed.


PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
TRUFFLE
Compiled successfully!
You can now view eth-marketplace in the browser.
Local: http://localhost:3000/
On Your Network: http://192.168.56.1:3000/
Note that the development build is not optimized.
To create a production build, use npm run build.

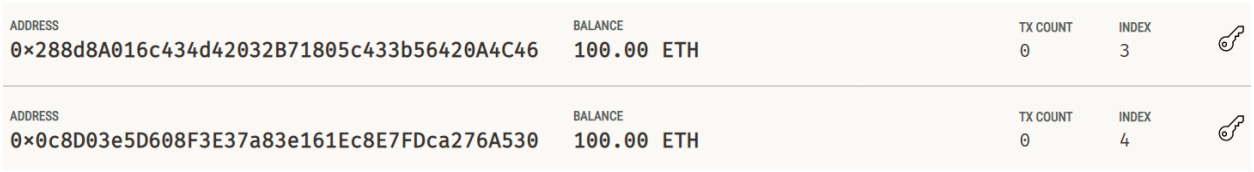

P2P Lending Smart Contract
MetaMask Notification
localhost:3000
P2PLending Decentralization Application
Contract Deployer:
Loading...
Welcome back!
The decentralized web awaits
sword
Unlock
Forgot password?


P2PLending Decentralization Application
Contract Deployer: 0x1bC44F14b8d1622B8CEFb674d9074a80E3199157
Available Lending Contracts
Put in some amount of ETH into the specified field. Press "Create Lending Contract" to create new contract!
0
Create Lending Contract
Borrowing Contracts
Browse through the available lending contracts. Pick one that you woud like to loan from. Copy the "Lending Contract address" in the the first field of the form. Then input necessary information and make the quick and easy loan!
Create Borrowing Contract
Lending Contract Address:
Lending Contract Address
Amount to borrow (ETH):

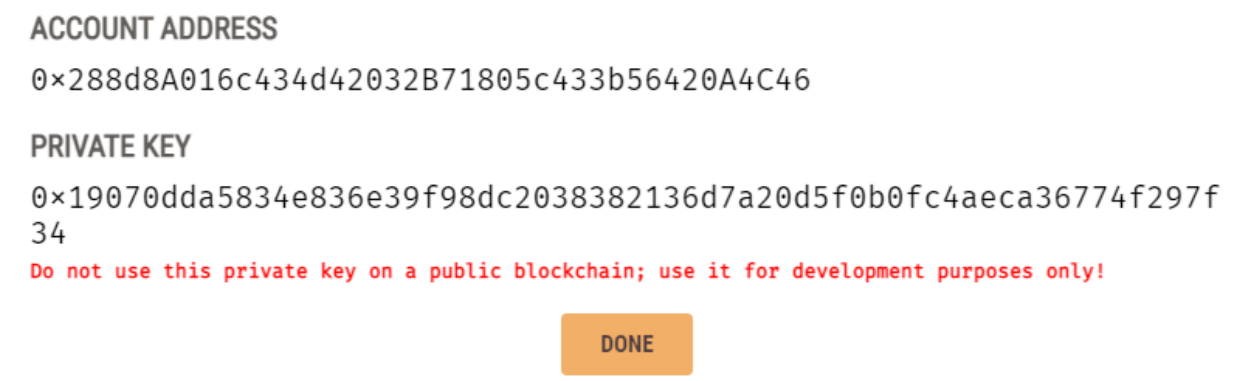
8. In order to interact with the decentralized application, we have to link the account from Ganache to MetaMask first. In order to do so, we follow the following:

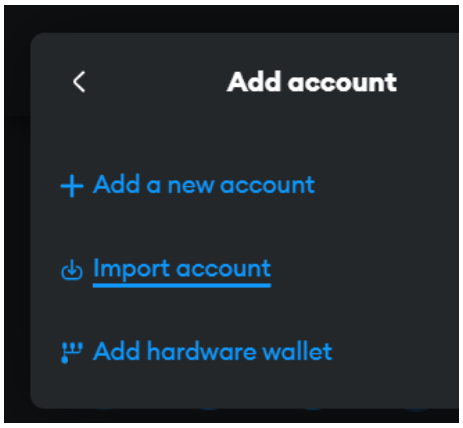


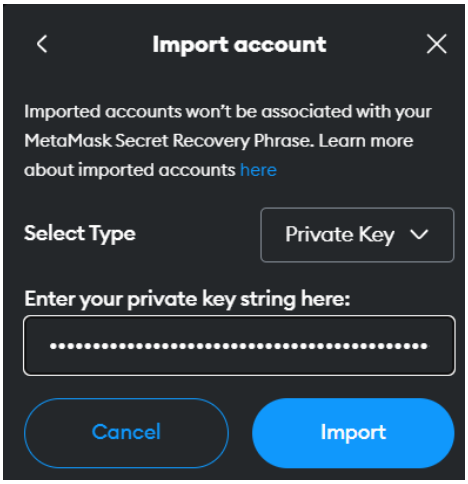
Click on the MetaMask extension and click the account. Find “Add account or hardware wallet”

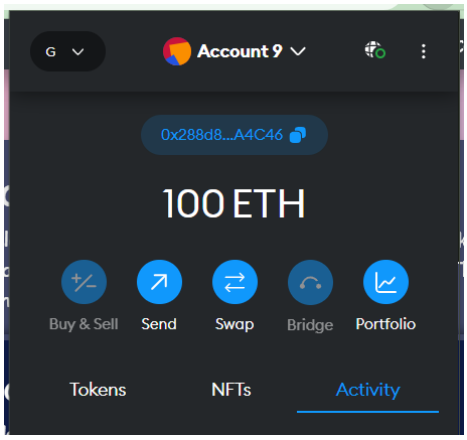


Open Ganache and select two addresses. Then click on the key at the right-hand corner to review the private key of this account. WARNING: Private keys should never be exposed to the public because your account can be compromised by a malicious actor gaining access into your account.





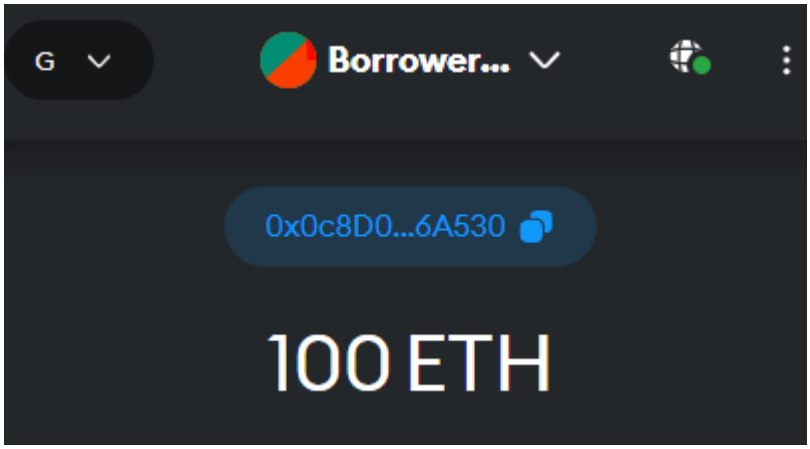


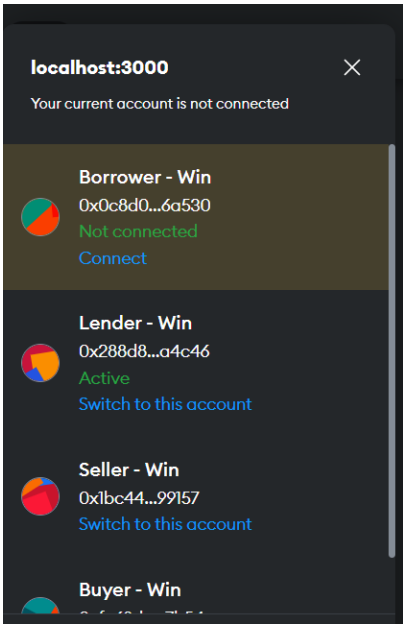


Copy the private key for each account from Ganache and open MetaMask. Find the Import account and place the private key and click Import. New account should now be accessible through MetaMask. The name of the account could be changed to something like Lender. Repeat the same

process but for a different wallet address and name it Borrower. We will interact with blockchain through these accounts.

9. Please make sure that these accounts are connected to the application. The small globe icon at the top should be green. If it isn't, click on the globe icon to connect the account.





10. That is all you need to do to get started interacting with our application

5.2 Debugging

1. Decreasing ETH in Lender address after borrowing took the loan and reappear after another transaction

This is really a concerning problem since our application deals with money directly. We couldn't find the reason as we checked all the addresses and the transferring script and every check box. We decided to create another version of the smart contract, thinking of it as a revamp period. We redesigned how contracts interact and optimized the contract by removing unnecessary variable storages. The problem disappears and every transaction is now correct.

Fortunately, this proves to be a blessing in disguise as we were able to create better structure contracts that minimize gas usage as well.

2. Connecting to the MetaMask:

Sometimes even after switching from one wallet account to another, the transaction is still associated with the previous wallet. Nonetheless, the transaction would be rejected because our smart contract uses various required statements to check the correct address before every transaction.

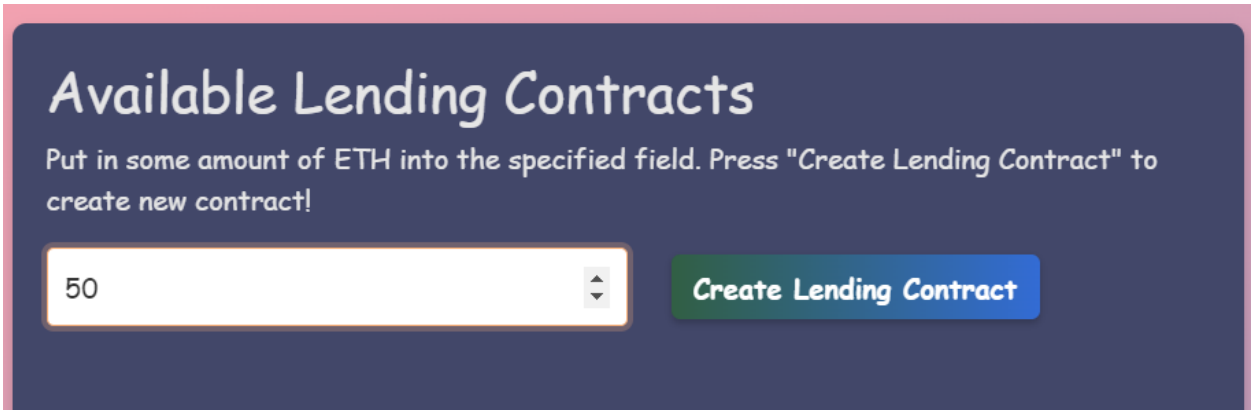
We decided to add a few automatic reloading statements to the application to make sure that the application fetches the correct address and current state of the blockchain. After this implementation, the problem with the mismatched MetaMask problem disappears.

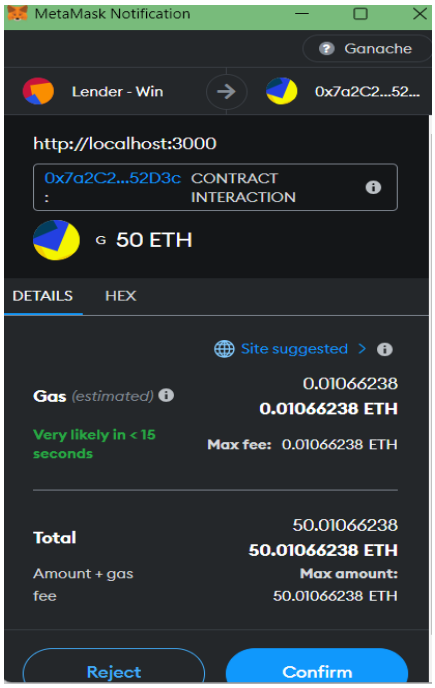
3. Front-end Development

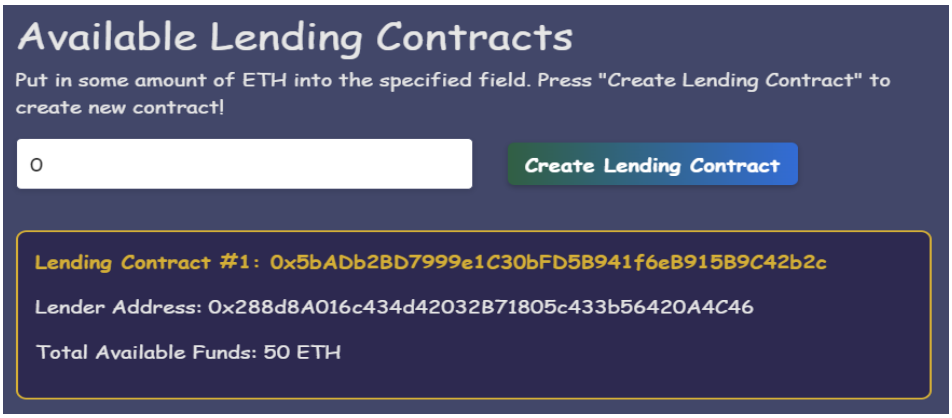
Front-end development isn't our forte. It took a lot of effort to connect to the blockchain correctly. Getting the initial setup of the application has been the most challenging aspect of the application. Fortunately, we were able to overcome the challenges and bugs to show the desired web application.

Part 6: Deploy and Maintain

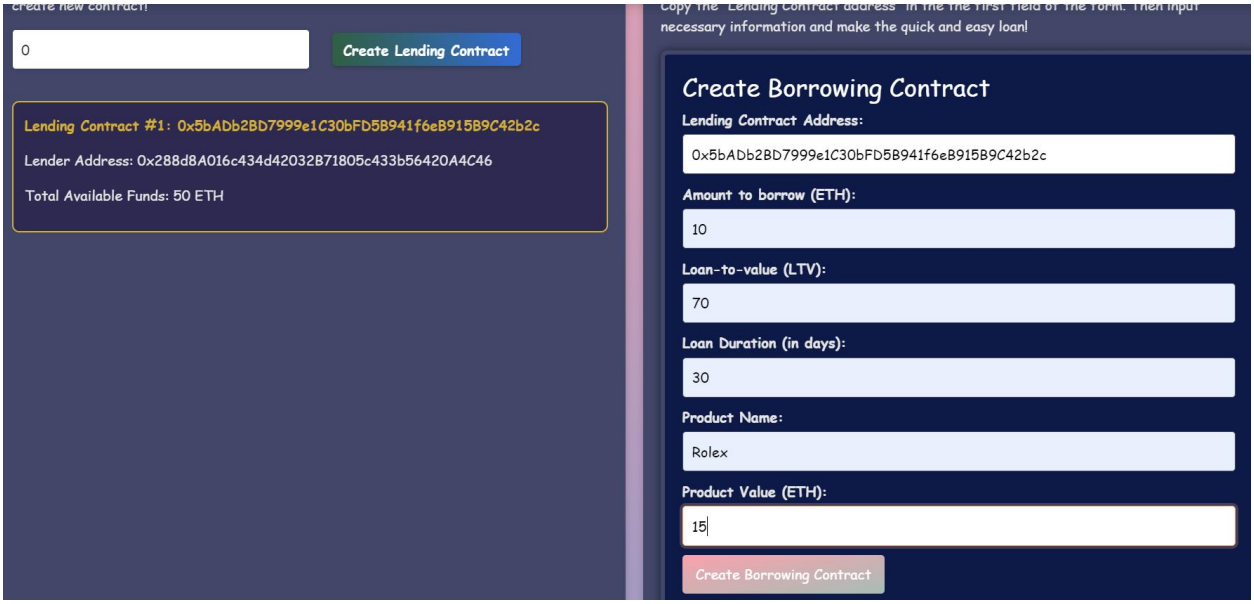
6.1 Creating Lending Contract



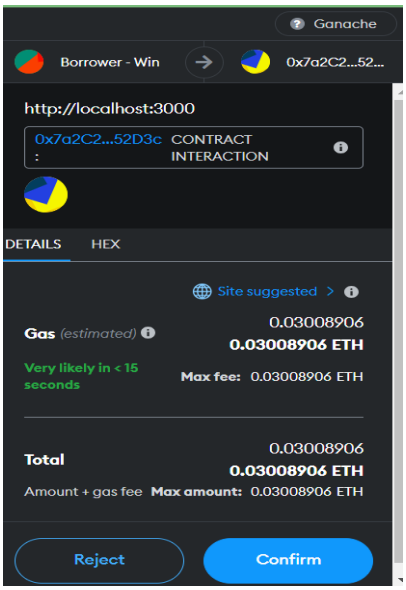




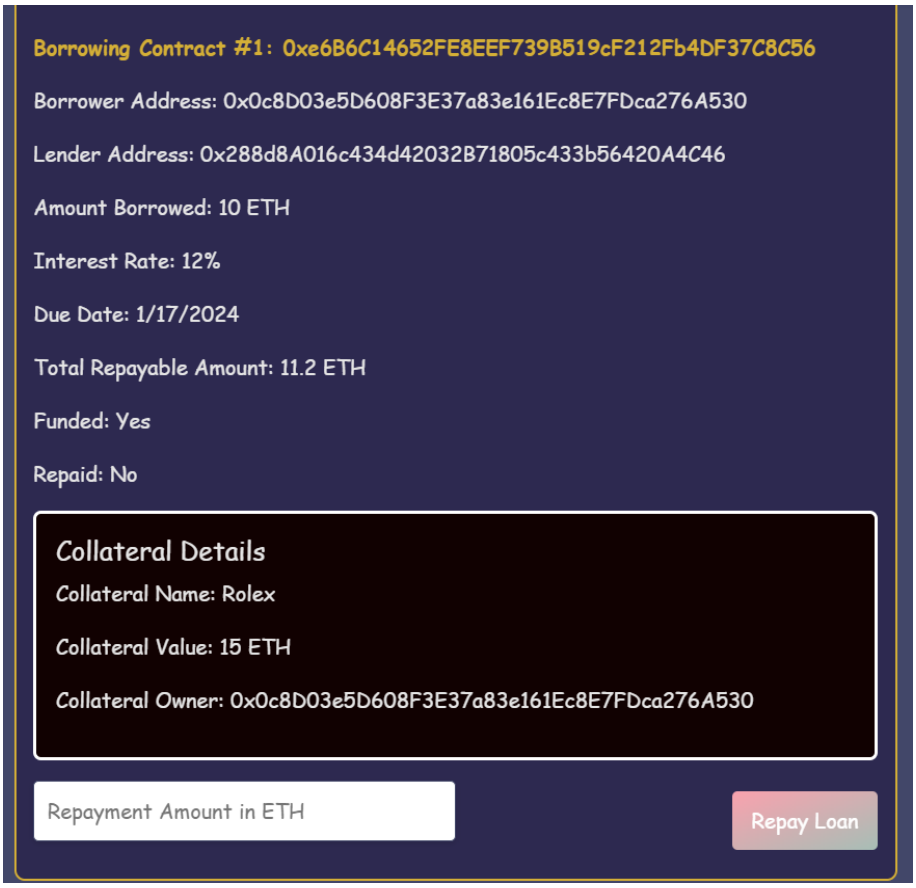
6.2 Creating Borrowing Contract



Borrower input necessary information into the form in order to create the Borrowing Contract.

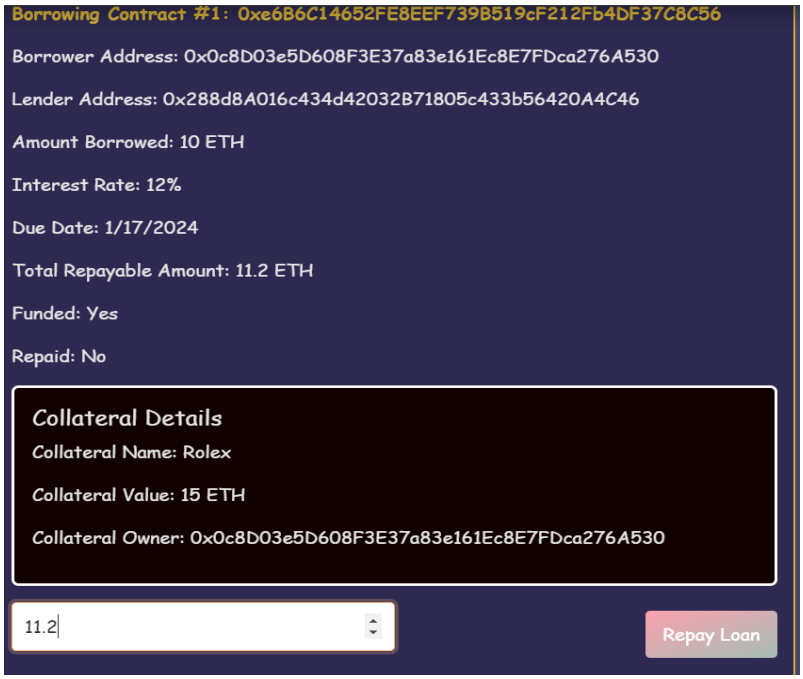


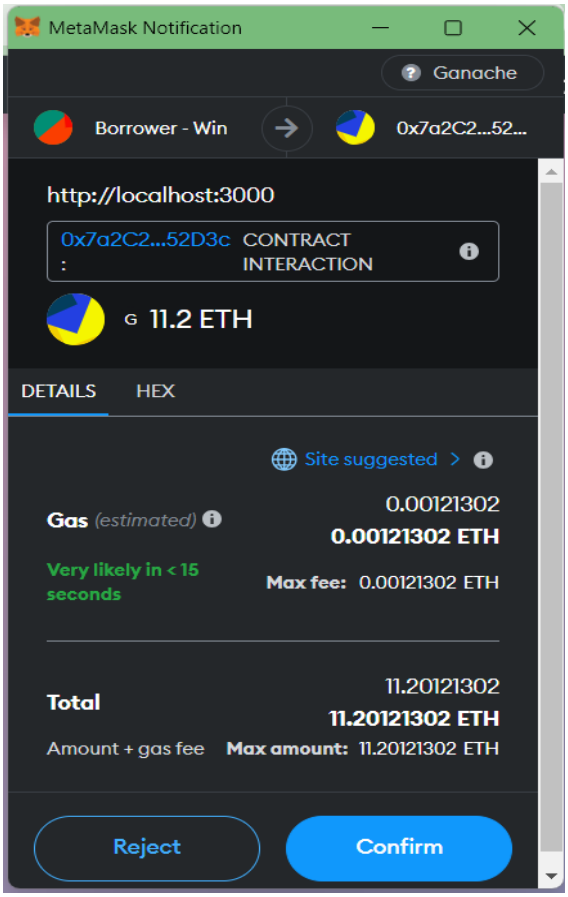
MetaMask popup will be displayed, and user have to click “Confirm” to confirm the transaction.

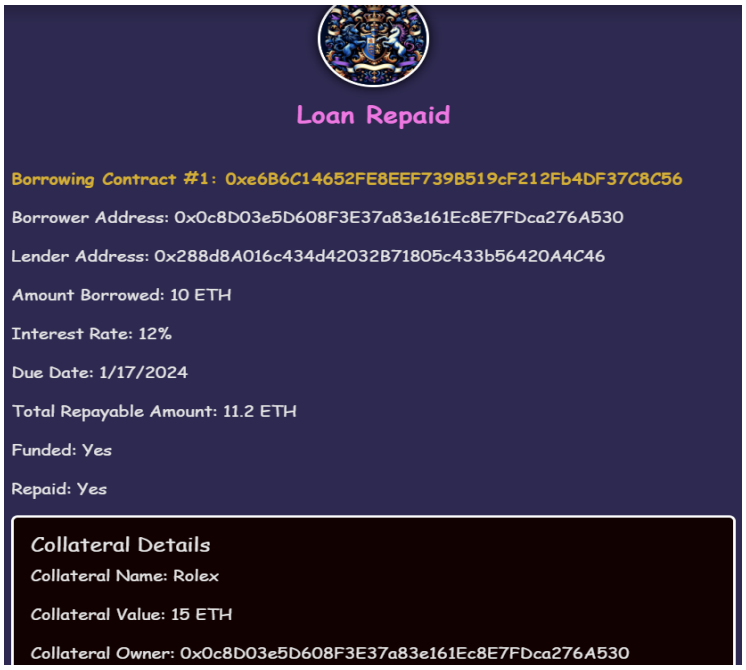


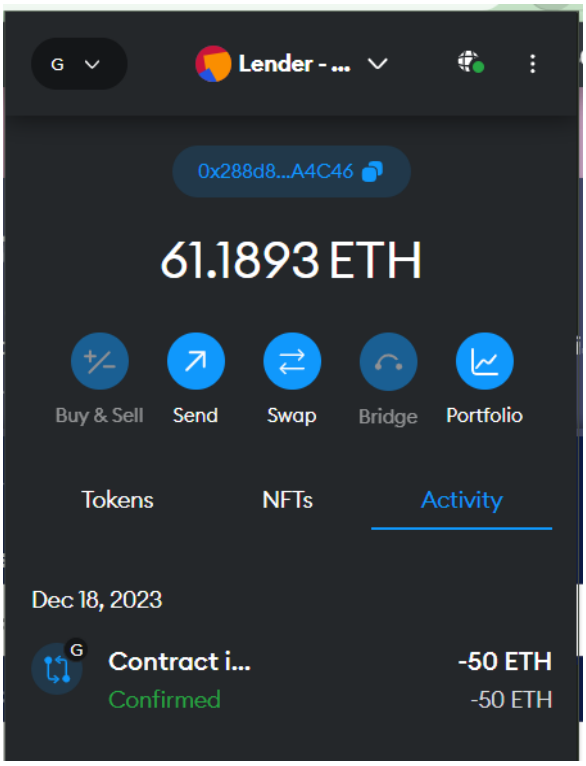
New Borrowing Contract has been created. Each borrowing contract has its own section with information relating to the contract. It also includes the Collateral details as well. The button at the bottom allows the borrower to input the value to repay the loan.

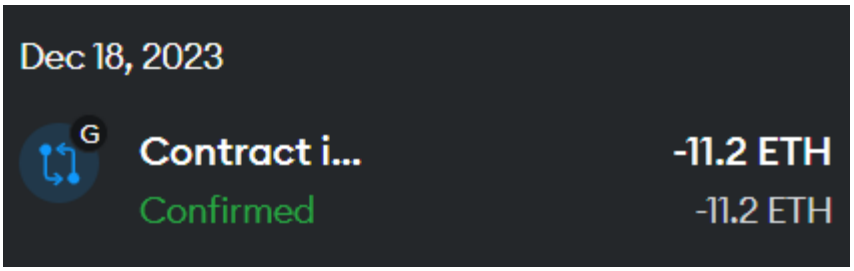
Repay the Loan: Borrower transfer money back to the lender


Borrowing Contract #1: 0xe6B6C14652FE8EEF739B519cF212Fb4DF37C8C56
Borrower Address: 0x0c8D03e5D608F3E37a83e161Ec8E7FDca276A530
Lender Address: 0x288d8A016c434d42032B71805c433b56420A4C46
Amount Borrowed: 10 ETH
Interest Rate: 12%
Due Date: 1/17/2024
Total Repayable Amount: 11.2 ETH
Funded: Yes
Repaid: No
Collateral Details
Collateral Name: Rolex
Collateral Value: 15 ETH
Collateral Owner: 0x0c8D03e5D608F3E37a83e161Ec8E7FDca276A530
11.2
Repay Loan


MetaMask Notification
Ganache
Borrower - Win
0x7a2C2...52...
http://localhost:3000
0x7a2C2...52D3c : CONTRACT INTERACTION
G 11.2 ETH
DETAILS
HEX
Site suggested
Gas (estimated)
0.00121302
0.00121302 ETH
Very likely in < 15 seconds
Max fee: 0.00121302 ETH
Total
11.20121302
11.20121302 ETH
Amount + gas fee
Max amount: 11.20121302 ETH
Reject
Confirm


Loan Repaid
Borrowing Contract #1: 0xe6B6C14652FE8EEF739B519cF212Fb4DF37C8C56
Borrower Address: 0x0c8D03e5D608F3E37a83e161Ec8E7FDca276A530
Lender Address: 0x288d8A016c434d42032B71805c433b56420A4C46
Amount Borrowed: 10 ETH
Interest Rate: 12%
Due Date: 1/17/2024
Total Repayable Amount: 11.2 ETH
Funded: Yes
Repaid: Yes
Collateral Details
Collateral Name: Rolex
Collateral Value: 15 ETH
Collateral Owner: 0x0c8D03e5D608F3E37a83e161Ec8E7FDca276A530


G
Lender - ...
0x288d8...A4C46
61.1893 ETH
Buy & Sell
Send
Swap
Bridge
Portfolio
Tokens
NFTs
Activity
Dec 18, 2023
Contract i...
Confirmed
-50 ETH
-50 ETH

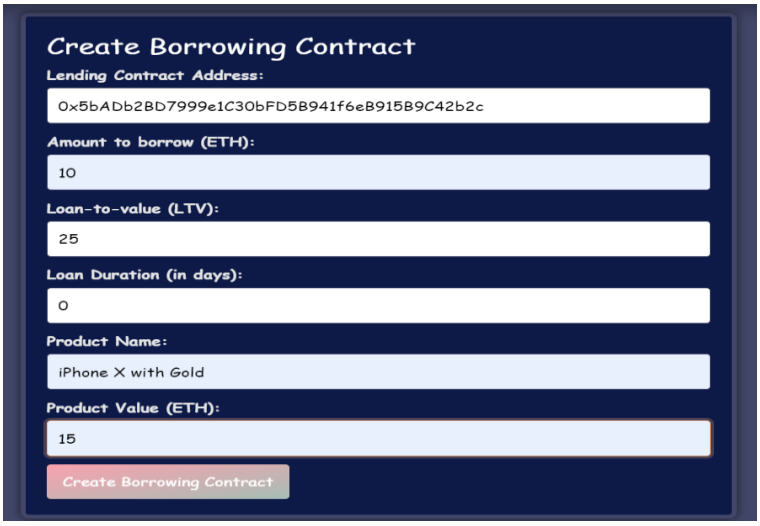


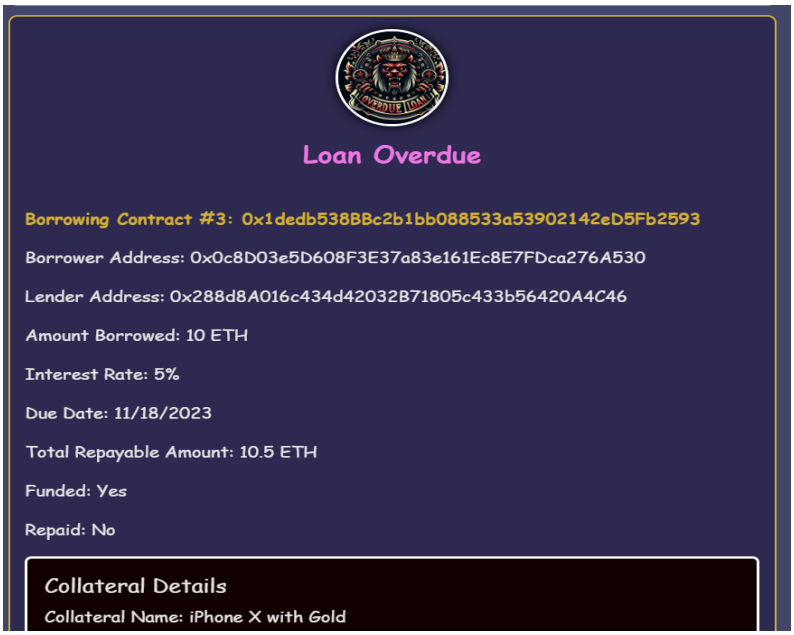
The lender made a lending contract worth 50 Ether. Hence the Account balance would be approximately less than 50 ETH because the gas fees have been included during the initial contract generation. Now, the borrower repays the loan 11.2 ETH. The new balance of the lender should be around <50 + 11.2 ~ 61.2 ETH, which aligns with the image shown above.

Special case when the Loan dueDate has already expired:

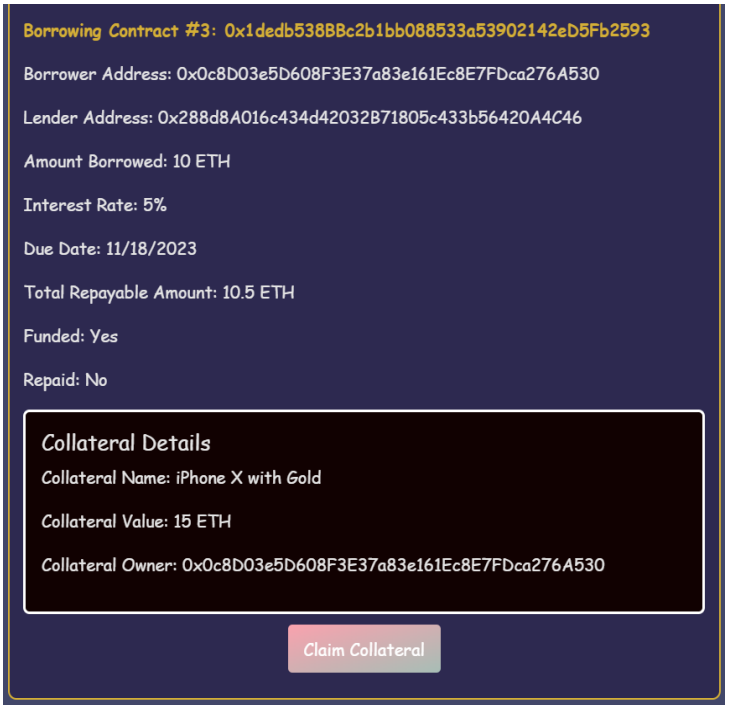
The lender can claim the collateral

This can be done by putting value 0 into the Loan Duration in days. Of course, in real production, this feature will have to be removed since no one wants to create a borrowing contract that expires the moment it is created. The only reason this feature was included is because we have to test whether the application interaction with collateral works as intended or not.




Loan Overdue
Borrowing Contract #3: 0x1dedb538BBc2b1bb088533a53902142eD5Fb2593
Borrower Address: 0x0c8D03e5D608F3E37a83e161Ec8E7FDca276A530
Lender Address: 0x288d8A016c434d42032B71805c433b56420A4C46
Amount Borrowed: 10 ETH
Interest Rate: 5%
Due Date: 11/18/2023
Total Repayable Amount: 10.5 ETH
Funded: Yes
Repaid: No
Collateral Details
Collateral Name: iPhone X with Gold


Collateral Details
Collateral Name: iPhone X with Gold
Collateral Value: 15 ETH
Collateral Owner: 0x0c8D03e5D608F3E37a83e161Ec8E7FDca276A530

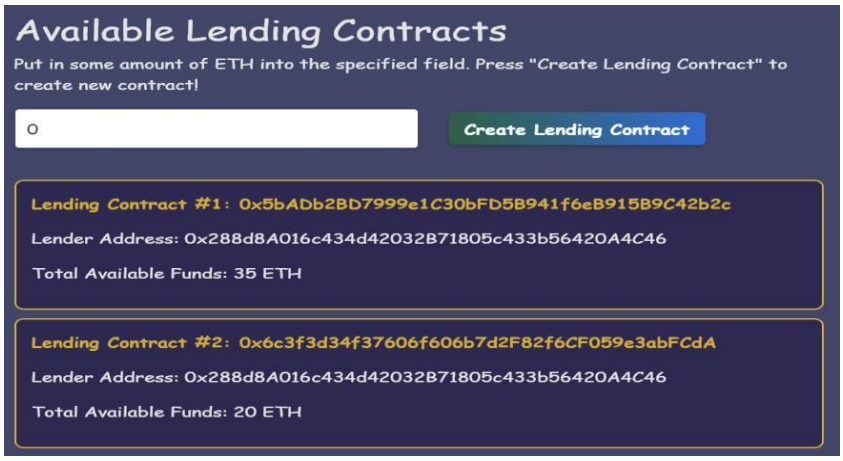

Borrowing Contract #3: 0x1dedb538BBc2b1bb088533a53902142eD5Fb2593
Borrower Address: 0x0c8D03e5D608F3E37a83e161Ec8E7FDca276A530
Lender Address: 0x288d8A016c434d42032B71805c433b56420A4C46
Amount Borrowed: 10 ETH
Interest Rate: 5%
Due Date: 11/18/2023
Total Repayable Amount: 10.5 ETH
Funded: Yes
Repaid: No
Collateral Details
Collateral Name: iPhone X with Gold
Collateral Value: 15 ETH
Collateral Owner: 0x0c8D03e5D608F3E37a83e161Ec8E7FDca276A530
Claim Collateral

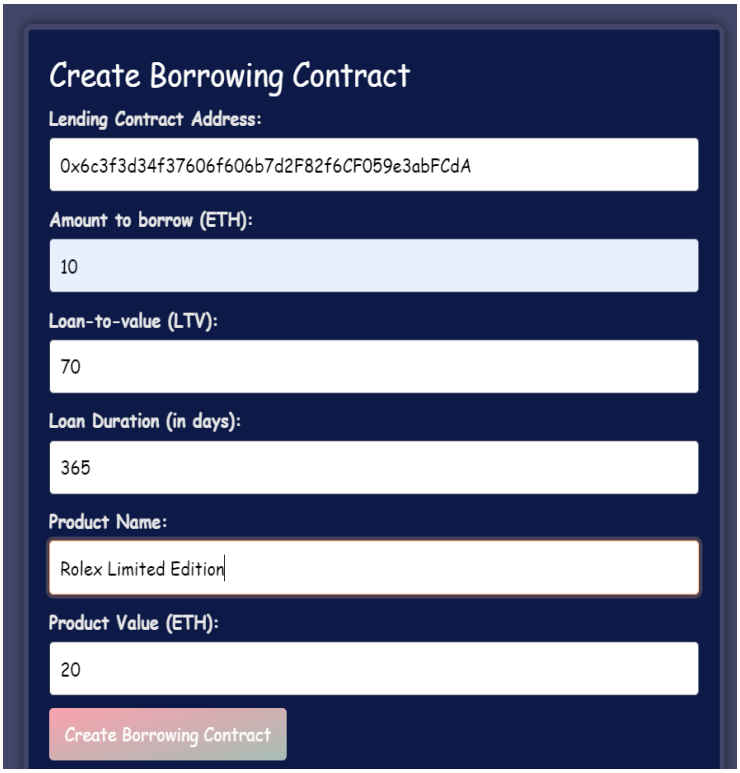
From the Lender side, there will be a button “Claim Collateral” visible. This button allows the lender to claim the collateral because the lending contract due date has already been passed. Please take a note of the current address for the Collateral Owner. It is the same address as the Borrower address. This ownership will change once the Lender clicked the button.

After clicking the button, the ownership of the collateral has been transferred to the lender. Please pay attention to the Lender Address and Collateral Owner address, it is the same address.

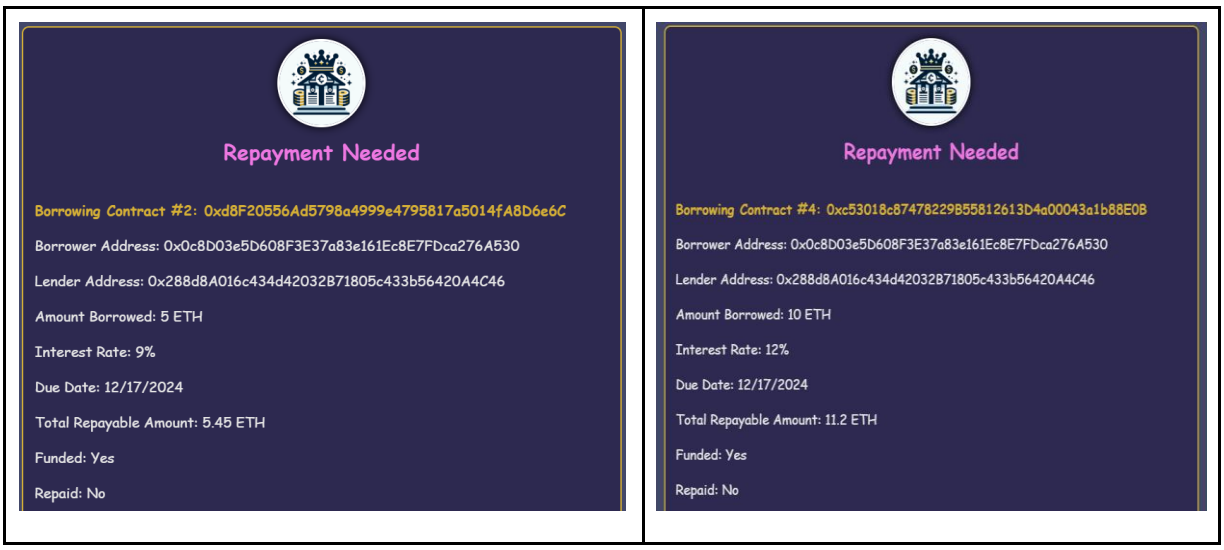
View the account information regarding the amount of borrowing contracts (the expected ETH to payback and to which lender), the amount of lending contracts (the expected ETH to receive back and from which borrower), along with collateral to claimed if the borrower failed to repay the loan

Let’s creating new lending contract to show the how the summary page works.

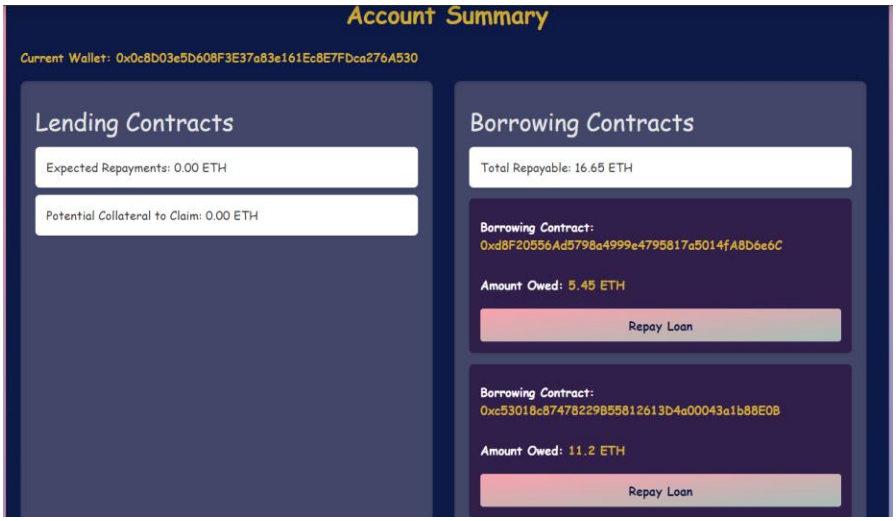




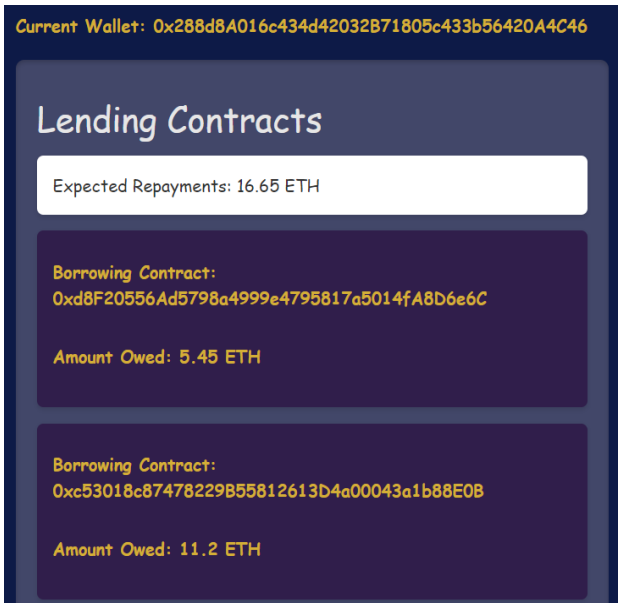
Let the borrower make two more loans to really visualize the full capability of the summary page.

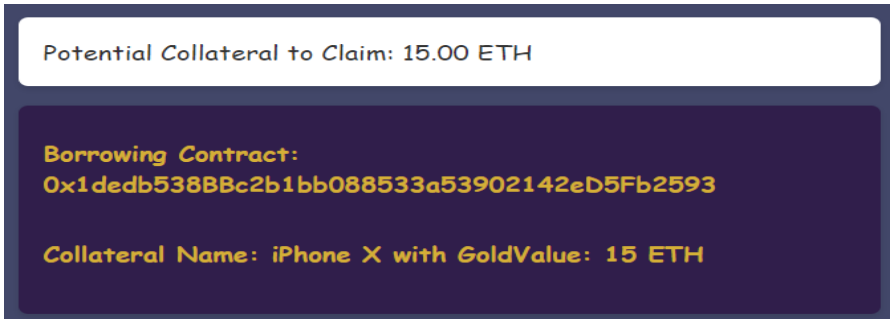


Let’s navigate to the summary page website by clicking on the “View Account Summary” at the bottom of the homepage.



The two images above showcase the Account Summary page from the point of view of the borrower. In our example, the borrower has two unpaid loans. The accumulated loan amount is shown in the upper section of the Borrowing Contracts column. There exists the Amounts Owed by Lender section that summarized the information on how much the borrower owns to this particular lender.





Now looking from the perspective of the lender, lending can see the total expected repayments at the top. This number aligns with the point of view of borrowers that own the same amount to this lender. There is also a section that mentioned Potential Collateral to Claim. This is where the lender can get the information on the collateral item that they have or haven't claimed as theirs due to the fact that the borrowing contract due date has expired. From an earlier demonstration, we saw that the Collateral Value of the borrowing contract which is overdue is 15 ETH, which aligns with what is displayed here.

Nonetheless if the lender becomes the borrower and vice versa, the information will be shown on both sides of the summary page.

Part 7: Future Project Improvement

7.1 Collateral Management

minted the real-world asset into NFT, which can be converted into our platform ERC20 token that is used for collateral. We would add the fractionalization functionality to the collateral which allows the borrower to split the collateral into smaller units that can be used for borrowing. This can prove to be more practical and user-friendly as some users might want to make multiple loans, but they don't have enough resources to use. Nonetheless, this poses a challenging question regarding how to manage the real-world asset such as how can we ensure that the asset is safe guarded in order to not compromise the lender in the case of overdue loan payment. There is a particular blockchain solution that implements real-world assets as collateral called Collateral Network (https://collateralnetwork.io/). Unfortunately, we couldn't find how exactly they manage the collateral.

7.2 Using Openzeppelin for ERC20 and SafeMath

Current development setup caused the Openzeppelin to crash due to a mismatched environment. When we initially developed the smart contract on Remix IDE, we had no issue with the versioning. We made our own Token to use as collateral as well and manage the transaction correctly. However, this is a prominent problem in local development that hindered our utilization of the great open-source tools for developing secured smart contracts and dApps.

7.3 Improve Functionality

The search for individual contracts on the homepage and blur out the borrowing contract that doesn't belong to the current account. Mainly the reason why the current version of the application shows so much information is because we really want to show that our application made the correct transaction. This is our approach to creating minimum viable products, or MVP, to show enough significant functions that peer-to-peer lending should have and that they existed in our application.

References

Boonthavornthavee, A. (2020). *Crypto Lending with Smart Contracts* (Thesis Project for Master Degree; pp. 7–78). University of Southampton Faculty of Engineering and Physical Sciences, Electronics and Computer Science.

Cointelegraph Bitcoin & Ethereum Blockchain News. (n.d.). Retrieved from Cointelegraph website: https://cointelegraph.com/learn/metamask-tutorial-for-beginners-how-to-set-up-a-metamask-wallet

Intro To Blockchain Programming (Etherum, Web3.js & Solidity Smart Contracts) [FULL COURSE]. (n.d.). Retrieved from www.youtube.com website: https://youtu.be/VH9Q2lf2mNo?si=G-VrY3QKEbMzMlmG

Kagan, J. (2020, May 11). Peer-to-Peer Lending: The Lowdown. Retrieved from Investopedia website: https://www.investopedia.com/terms/p/peer-to-peer-lending.asp

McCubbin, G. (n.d.). Solidity for Beginners · Smart Contract Development Crash Course. Retrieved from Dapp University website: https://www.dappuniversity.com/articles/solidity-tutorial

Solidity Tutorial. (n.d.). Retrieved from www.tutorialspoint.com website: https://www.tutorialspoint.com/solidity/index.htm

Web3 Tutorial Project | Build a web3js dApp for a smart contract. (n.d.). Retrieved from www.youtube.com website: https://www.youtube.com/watch?v=Qu6GloG0dQk