

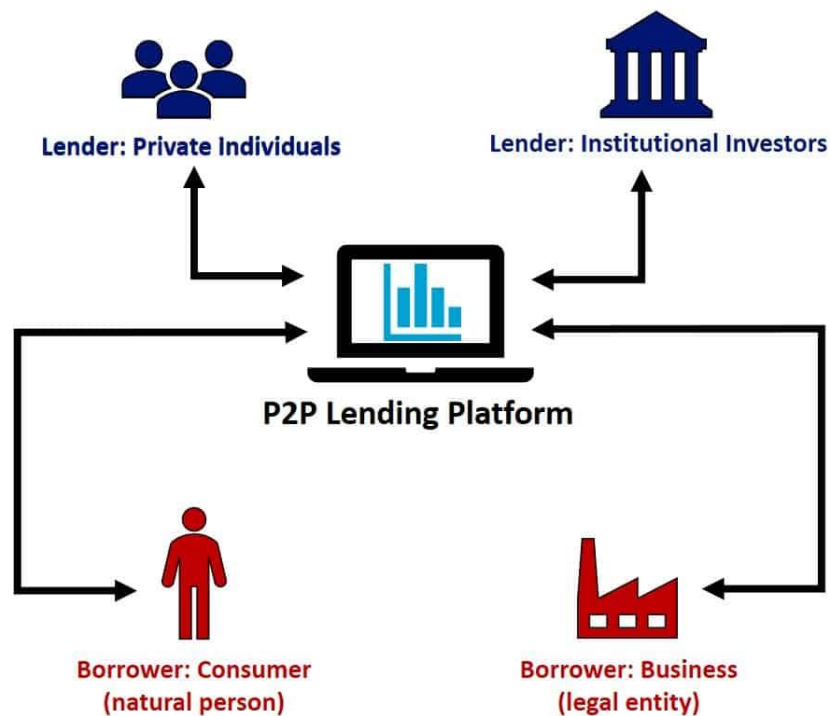
Project: Peer-to-Peer Lending Platform

Introduction

Welcome to the Peer-to-Peer Lending Platform built on the Ethereum blockchain network.

This modular project aims to revolutionize the lending industry by introducing transparency, fairness, and efficiency through decentralized technology.

Unlike traditional banking systems, our focus, by efficiently utilizing the knowledge we have gained about smart contracts, is on empowering both borrowers and investors, ensuring mutually beneficial interactions without hidden fees or unfair practices.



1. Overview

Problem Statement

The traditional banking industry is often criticized for prioritizing profit over customer welfare. Borrowers face exorbitant interest rates, while investors experience subpar returns.

Additionally, **hidden fees** erode the trust between banks and their customers.

Solution Overview

Our Peer-to-Peer Lending Platform leverages the Ethereum blockchain to create a transparent, equitable, and community-oriented lending environment.

By embracing blockchain's immutable ledger and smart contracts, we aim to revolutionize lending by enabling:

- Borrowers to request funding with fair **interest rates**.
- Lenders to **invest** in projects/credits with improved returns.
- **Voting mechanisms** for contract revocation and fraud detection.

2. Features

a) Borrowing Funds

Requesting Funding

Borrowers can create funding requests, specifying the amount required, purpose, and proposed interest rate. This request is then verified and added to the platform.

Providing Funding Details

Lenders can review the funding requests and associated details provided by borrowers. Transparent information allows lenders to make informed decisions about investing.

Withdrawal of Funding

Once a funding request reaches its goal, the borrower can withdraw the funds. Smart contracts ensure that funds are released only upon successful completion of the funding goal.

Repayment Installments

Borrowers are required to adhere to a predefined repayment schedule. The platform facilitates automated repayment installments, enhancing convenience for both borrowers and lenders.

b) Lending Funds

Investing in Project/Credit

Lenders can invest in various funding requests based on their preferences. By diversifying their investments across multiple projects, lenders can manage risk effectively.

Voting for Contract Revocation

The community of lenders holds the power to vote for the revocation of a contract in case of suspected fraudulent activities or violations. This democratic approach ensures platform integrity.

3. Project Structure

The modular project we are going to build is upon a **well-structured architecture** that leverages various smart contracts to ensure security, modularity, and efficient functionality.

Below is a brief overview of the project's structure:

a) Libraries

SafeMath.sol

The SafeMath library is a foundational component of the project, responsible for **preventing integer overflow** and underflow vulnerabilities in arithmetic operations.

It provides secure mathematical operations for integers, safeguarding the project against potential vulnerabilities.

b) Contracts

Ownable.sol

The Ownable contract establishes a basic access **control mechanism**, ensuring that certain actions or functions within the system can only be executed by the owner of the contract.

This is a common pattern in Ethereum smart contracts to manage **permissions and administrative actions**.

Credit.sol

The Credit contract extends the functionality of the '**Ownable**' contract. It represents a credit request made by a borrower on the platform.

The contract defines details about the credit, including the **requested amount**, **repayment plan**, and **any collateral offered**.

By inheriting from '**Ownable**', the contract can manage ownership and access control for credit-related actions.

PeerToPeerLending.sol

The PeerToPeerLending contract is the core component of the platform. It inherits from the '**Ownable**' contract and serves as the bridge between borrowers and investors.

This contract contains essential functions for borrowing funds, investing in projects/credit, voting on contract revocation, and refunding investments.

It creates **instances** of the '**Credit**' contract to facilitate individual credit requests and maintain the lending ecosystem.

c) Interactions

'**Credit.sol**' and '**PeerToPeerLending.sol**' both inherit from '**Ownable.sol**', which means they share the ownership management logic. The '**Ownable**' contract's functions enable only authorized parties to perform administrative tasks.

'**PeerToPeerLending.sol**' creates instances of the '**Credit**' contract, enabling the creation and management of individual credit requests.

This contract also provides functions for investing in projects/credit and voting on contract-related actions.

SafeMath Library

- The SafeMath library is used to prevent common vulnerabilities like integer **overflow** and **underflow** that can occur in smart contracts due to the limited size of data types like uint256.
- By providing secure arithmetic operations, this library enhances the safety and reliability of smart contracts that handle **numerical calculations**.

1) Multiplication Function:

Purpose: Performs secure multiplication of two uint256 numbers.

Explanation: This function prevents overflow by checking that the result of the multiplication divided by the first number **(a)** is equal to the second number **(b)**.

If this check fails, it indicates that an overflow occurred.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

library SafeMath {
    function mul(uint256 a, uint256 b) internal pure returns (uint256) {
        if (a == 0) {
            return 0;
        }
        uint256 c = a * b;
        assert(c / a == b);
        return c;
    }
}
```

2) Division Function:

Purpose: Performs secure division of two uint256 numbers.

Explanation: This function divides the first number **(a)** by the second number **(b)** and returns the result.

No overflow or underflow checks are needed for division because division doesn't cause these issues.

```

function div(uint256 a, uint256 b) internal pure returns (uint256) {
    //there is no case where this function can overflow/underflow
    uint256 c = a / b;
    return c;
}

```

3) Subtraction Function:

Purpose: Performs secure subtraction of one uint256 number from another.

Explanation: This function ensures that the result of the subtraction is greater than or equal to zero by asserting that the second number **(b)** is less than or equal to the first number **(a)**.

This prevents underflow.

```

function sub(uint256 a, uint256 b) internal pure returns (uint256) {
    assert(b <= a);
    return a - b;
}

```

4) Addition Function:

Purpose: Performs secure addition of two uint256 numbers.

Explanation: This function adds the two input numbers **(a and b)** and ensures that the result is greater than or equal to the original value of the first number **(a)**.

This prevents overflow.

```

function add(uint256 a, uint256 b) internal pure returns (uint256) {
    uint256 c = a + b;
    assert(c >= a);
    return c;
}

```

Ownable Contract

- The '**Ownable**' contract has been introduced further, which includes ownership-related functionality.
- It allows only the owner to execute specific actions through the `onlyOwner` modifier and emits an event to record ownership transfers.
- The contract is initialized with the deployer's address as the initial owner, creating a foundation for **secure access control** in Ethereum smart contracts.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

contract Ownable {
    address public owner;

    // This event is emitted when ownership of the contract is transferred, providing
    // transparency to ownership changes.
    event LogOwnershipTransferred(address indexed _currentOwner, address indexed _newOwner);
}
```

Modifier:

- A modifier in Solidity is like a **rule** that you create and can apply to functions in your smart contract.
- It's a way to make sure that **certain conditions are met** before a function can be executed. If the conditions in the modifier are not satisfied, the function won't be allowed to run.
- This helps you control who can do what in your smart contract and keeps things **secure and organized**.
- To implement a modifier in a function, the modifier name is written at the end of function declaration.

Syntax:

```
modifier modifier_name(){
    modifier_body
    _;
}

function function_name() visibility modifier_name{
    function_body
}
```


Modifier `onlyOwner()` :

- Includes a modifier named **'onlyOwner'**, which is used to **restrict certain functions** to be callable only by the contract's owner.
- It enforces that the sender of the transaction matches the owner address.

```
modifier onlyOwner() {
    require(msg.sender == owner);
    _;
}
```

Constructor() :

- Initialize the owner variable with the address of the sender who deployed the contract.
- This establishes the creator of the contract as its initial owner.

```
constructor(){  73765 gas 49400 gas
    owner = msg.sender;
}
```


Credit Contract:

Introduction

The Credit contract is a Solidity smart contract that facilitates lending and borrowing activities. It allows borrowers to request a credit amount, and lenders to invest in the credit. The contract tracks repayments, interest, and credit states. The contract employs various modifiers to control access and conditions for different functions.

Contract Structure

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

import './SafeMath.sol';
import './Ownable.sol';

contract Credit is Ownable {
    using SafeMath for uint;

    // State variables, enums, events, modifiers, constructor, and functions
}
```

The contract inherits from the Ownable contract and uses the SafeMath library for safe arithmetic operations. The SafeMath library consists of **uint** type arithmetic operations so it is used with **uint** datatype. **Keep in mind that all addition, subtraction, multiplication and division operations in this contract will be done using 'SafeMath' library functions.**

State Variables

The state variables included in the contract are as follows:

borrower (address type): The address of the borrower who initiated the credit request.

```
address borrower;
```

requestedAmount (uint type): The amount of credit requested by the borrower.

returnAmount (uint type): The total amount to be returned by the borrower (including interest).

repaidAmount (uint type): The amount already repaid by the borrower.

Interest (uint type): The interest rate for the credit.

```
uint requestedAmount;  
  
uint returnAmount;  
  
uint repaidAmount;  
  
uint interest;
```

requestedRepayments (uint type): The number of requested repayments.

remainingRepayments (uint type): The number of remaining repayments.

repaymentInstallment (uint type): The amount to be paid in each repayment installment.

requestedDate (uint type): The timestamp when the credit was requested.

lastRepaymentDate (uint type): The timestamp of the last repayment made.

```
uint requestedRepayments;  
  
uint remainingRepayments;  
  
uint repaymentInstallment;  
  
uint requestedDate;  
  
uint lastRepaymentDate;
```

Description (bytes type): A bytes array describing the credit.

Active (bool type): A boolean indicating whether the credit is active, which is initialized to **true**.

```
bytes description;  
  
bool active = true;
```

lendersCount (uint type): The total count of lenders participating in the credit.

revokeVotes (uint type): The count of votes to revoke the credit.

```
uint lendersCount = 0;  
  
uint revokeVotes = 0;
```

revokeTimeNeeded (uint type): The timestamp required to reach for revocation.

```
uint revokeTimeNeeded = block.timestamp + 1 seconds;
```

fraudVotes (uint type): The count of votes for considering the borrower as a fraudster.

```
uint fraudVotes = 0;
```

lenders: A mapping of lender addresses to their participation status.

```
mapping(address => bool) public lenders;
```

lendersInvestedAmount: A mapping of lender addresses to their invested amounts.

```
mapping(address => uint) lendersInvestedAmount;
```

revokeVoters: A mapping of lender addresses to their revoking votes.

```
mapping(address => bool) revokeVoters;
```

fraudVoters: A mapping of lender addresses to their fraud votes.

```
mapping(address => bool) fraudVoters;
```

Enums

The contract defines the following **enum** and **enum** state variable:

```
enum State {  
    investment,  
    repayment,  
    interestReturns,  
    expired,  
    revoked,  
    fraud  
}
```

```
State state;
```

The possible states of the enum includes: **investment**, **repayment**, **interestReturns**, **expired**, **revoked**, and **fraud**.

Events

The contract emits various events to log different activities and state changes. The events include:

LogCreditInitialized: Logged when the credit is initialized.

```
event LogCreditInitialized(address indexed _address, uint indexed timestamp);
```

LogCreditStateChanged: Logged when the credit state changes.

```
event LogCreditStateChanged(State indexed state, uint indexed timestamp);
```

LogCreditStateActiveChanged: Logged when the active state of the credit changes.

```
event LogCreditStateActiveChanged(bool indexed active, uint indexed timestamp);
```

LogBorrowerWithdrawal: Logged when the borrower withdraws funds.

```
event LogBorrowerWithdrawal(address indexed _address, uint indexed _amount, uint indexed timestamp);
```

LogBorrowerRepaymentInstallment: Logged when the borrower makes a repayment installment.

```
event LogBorrowerRepaymentInstallment(address indexed _address, uint indexed _amount, uint indexed timestamp);
```

LogBorrowerRepaymentFinished: Logged when the borrower completes repayment.

```
event LogBorrowerRepaymentFinished(address indexed _address, uint indexed timestamp);
```

LogBorrowerChangeReturned: Logged when excess funds are returned to the borrower.

```
event LogBorrowerChangeReturned(address indexed _address, uint indexed _amount, uint indexed timestamp);
```

LogLenderInvestment: Logged when a lender makes an investment.

```
event LogLenderInvestment(address indexed _address, uint indexed _amount, uint indexed timestamp);
```

LogLenderWithdrawal: Logged when a lender withdraws funds.

```
event LogLenderWithdrawal(address indexed _address, uint indexed _amount, uint indexed timestamp);
```

LogLenderChangeReturned: Logged when excess funds are returned to a lender.

```
event LogLenderChangeReturned(address indexed _address, uint indexed _amount, uint indexed timestamp);
```

LogLenderVoteForRevoking: Logged when a lender votes to revoke the credit.

```
event LogLenderVoteForRevoking(address indexed _address, uint indexed timestamp);
```

LogLenderVoteForFraud: Logged when a lender votes for the borrower as a fraudster.

```
event LogLenderVoteForFraud(address indexed _address, uint indexed timestamp);
```

LogLenderRefunded: Logged when a lender is refunded.

```
event LogLenderRefunded(address indexed _address, uint indexed _amount, uint indexed timestamp);
```

Modifiers

The contract defines several modifiers to control access and conditions for functions:

isActive: Ensures that the credit is active.

```
modifier isActive() {  
    require(active == true);  
    _;  
}
```

onlyBorrower: Restricts access to the borrower.

```
modifier onlyBorrower() {  
    require(msg.sender == borrower);  
    _;  
}
```

onlyLender: Restricts access to registered lenders.

```
modifier onlyLender() {  
    require(lenders[msg.sender] == true);  
    _;  
}
```

canAskForInterest: Allows lenders to request interest returns.

```
modifier canAskForInterest() {  
    require(state == State.interestReturns);  
    require(lendersInvestedAmount[msg.sender] > 0);  
    _;  
}
```

canInvest: Allows lenders to invest in the credit.

```
modifier canInvest() {  
    require(state == State.investment);  
    _;  
}
```

canRepay: Allows the borrower to make repayment installments.

```
modifier canRepay() {  
    require(state == State.repayment);  
    _;  
}
```

canWithdraw: Ensures enough balance to allow withdrawal by the borrower.

```

modifier canWithdraw() {
    require(address(this).balance >= requestedAmount);
    _;
}

```

isNotFraud: Ensures the credit is not marked as fraud.

```

modifier isNotFraud() {
    require(state != State.fraud);
    _;
}

```

isRevokable: Ensures the credit is in the "investment" state and can be revoked.

```

modifier isRevokable() {
    require(block.timestamp >= revokeTimeNeeded);
    require(state == State.investment);
    _;
}

```

isRevoked: Ensures the credit is in the "revoked" state.

```

modifier isRevoked() {
    require(state == State.revoked);
    _;
}

```

Functions

1. Constructor()

The constructor function is marked with the constructor keyword and is executed only during the contract deployment. The constructor's purpose is to set up the initial configuration of the credit contract when it is deployed, initializing important parameters like **loan amount**, **interest rate**, **repayment** details, and more.

- It accepts four parameters:
 - **_requestedAmount:** The requested loan amount by the borrower.
 - **_requestedRepayments:** The number of repayments the borrower intends to make.
 - **_interest:** The interest rate associated with the credit.
 - **_description:** A description of the credit.

```

constructor(uint _requestedAmount,
uint _requestedRepayments,
uint _interest,
bytes memory _description)

```

- Set the **borrower's** address to the original sender of the deployment transaction (**tx.origin**).
- Assign the provided **_interest** value to the **interest** state variable.
- Assign the provided **_requestedAmount** value to the **requestedAmount** state variable.
- Assign the provided **_requestedRepayments** value to the **requestedRepayments** state variable.
- Set the **remainingRepayments** state variable to the same value as **_requestedRepayments** initially.

```

    borrower = tx.origin;

    interest = _interest;

    requestedAmount = _requestedAmount;

    requestedRepayments = _requestedRepayments;

    remainingRepayments = _requestedRepayments;

```

- Calculate the **returnAmount** by adding the **requestedAmount** and **interest**, using the 'SafeMath' library's 'add()' function.

```

    returnAmount = requestedAmount.add(interest);

```

- Calculate the **repaymentInstallment** by dividing **returnAmount** by **requestedRepayments**, using the 'SafeMath' library's 'div()' function.
- Assign the provided **_description** to the **description** state variable.
- Set the **requestedDate** state variable to the current timestamp using **block.timestamp**.

```

    repaymentInstallment = returnAmount.div(requestedRepayments);

    description = _description;

    requestedDate = block.timestamp;

```

- **Emit an event (LogCreditInitialized)** to record the initialization of the credit contract, capturing the **borrower's** address and the **current timestamp**.

```

emit LogCreditInitialized(borrower, block.timestamp);

```

2. getBalance()

The 'getBalance()' function allows anyone to query the current balance of the credit contract. This balance represents the total amount of ether held by the contract. It returns the current **balance** of the 'Credit' contract.

```

function getBalance() public view returns (uint256)

```

- You can use 'address(this)' to access the **address** of the 'Credit' contract.
- You can use '.balance' to access the **balance**. It will give the balance of whatever **address** is written before it in **uint** form.

```

    function getBalance() public view returns (uint256) {
    | |   return address(this).balance;
    }

```

3. changeState()

The 'changeState()' function allows only the owner of the contract (due to 'onlyOwner' **modifier**) to change the state of the 'state' (**enum**) variable. This is important as a lot of the **functions** in the contract have **modifiers** that restrict the accessibility of the function depending on the state of the 'state' variable.

```

function changeState(State _state) external onlyOwner

```

- Assign the provided **_state** value to the **state** state variable.
- **Emit an event (LogCreditStateChanged)** capturing the new state assigned to the **state** variable and the **current timestamp**.


```

function changeState(State _state) external onlyOwner{
    state = _state;

    // Log state change.
    emit LogCreditStateChanged(state, block.timestamp);
}

```


4. toggleActive()

The 'toggleActive()' function allows only the owner of the contract (due to 'onlyOwner' **modifier**) to toggle the status of the **active** state variable (changes **true** to **false** or vice versa). The **active** state variable indicates whether the credit is active, so using this function the owner of the contract can change whether the status of the credit is active or not.

```
function toggleActive() external onlyOwner returns (bool)
```

- Toggle the status of the **active** state variable and store it as the new value of **active** state variable. (active = !active)
- **Emit an event (LogCreditStateActiveChanged)** capturing the toggled value assigned to the **active** variable and the **current timestamp**.
- **Return** the new value of the **active** state variable.

```

function toggleActive() external onlyOwner returns (bool){  34796 gas
    active = !active;

    // Log active status change.
    emit LogCreditStateActiveChanged(active, block.timestamp);

    return active;
}

```

5. Invest()

The 'invest()' function allows lenders to invest their funds into the credit contract, becoming potential creditors to the borrower. Lenders can invest an amount equal to or greater than the requested loan amount. If enough funds are invested, the credit **state** transitions to the repayment phase.

The **function** will be made **payable** as ether is being used in it. The function also has the 'canInvest' **modifier** attached to it, that checks if the current state of the **state** variable is 'investment'.

```
function invest() public canInvest payable
```

- Declare a **uint** local variable named 'extraMoney' and initialize it to zero.

```
uint extraMoney = 0;
```

- Check if the contract's current balance is greater than or equal to the requested credit amount (**requestedAmount**). If true then:
- (Start of **if** body) Calculate the **extraMoney** sent by the lender by subtracting the **requestedAmount** from the **contract's balance**, using the 'SafeMath' library's 'sub()' function.

```
if (address(this).balance >= requestedAmount) {  
    |  
    |   extraMoney = address(this).balance.sub(requestedAmount);  
    |  
}
```

- Using **assert**, give an 'error' if the **requestedAmount** is equal to the subtraction result of the **balance** of the contract address to the **extraMoney**.

```
address(this).balance.sub(extraMoney)
```

- Use another **assert**, to give an 'error' if the amount of **extraMoney** is greater than the amount sent in this function (msg.value).

```
assert(extraMoney <= msg.value);
```

- Using **if** conditional, check if extra money is present, if **true** then:
 - (Start of **if** body) Return the extra money to the lender using the transfer function.

```
// Return the extra money to the sender.  
payable(msg.sender).transfer(extraMoney);
```

- Emit an event (**LogLenderChangeReturned**) to log the return of the extra money. (End of **if** body)

```
if (extraMoney > 0) {  
    |  
    |   // Return the extra money to the sender.  
    |   payable(msg.sender).transfer(extraMoney);  
    |  
    |   emit LogLenderChangeReturned(msg.sender, extraMoney, block.timestamp);  
    |  
}
```

- Change the credit **state** to `State.repayment` since the requested amount is now met.
- Emit an event(**LogCreditStateChanged**) to log the change in credit **state**. (End of **if** body)

```

    state = State.repayment;

    emit LogCreditStateChanged(state, block.timestamp);
}

```

- Mark the lender as invested by updating the **lenders mapping** with the lender's **address** to **true**.
- **Increment** the **lendersCount** variable to keep track of the number of lenders.

```
lenders[msg.sender] = true;
```

```
lendersCount++;
```

- Update the lender's invested amount in the **lendersInvestedAmount** mapping.

```

// Add the amount invested to the amount mapping.
lendersInvestedAmount[msg.sender] = lendersInvestedAmount[msg.sender].add(msg.value.sub(extraMoney));

```

- Emit an event (**LogLenderInvestment**) to log the lender's investment.

```

// Log lender invested amount.
emit LogLenderInvestment(msg.sender, msg.value.sub(extraMoney), block.timestamp);
}

```

PeerToPeerLending Contract

- The '**PeerToPeerLending**' contract encapsulates essential functionalities required for borrowers and investors to **interact** in a fair and transparent lending ecosystem.
- The '**PeerToPeerLending**' contract establishes a **decentralized** lending platform on the Ethereum blockchain, redefining the traditional lending landscape by increasing fairness, transparency, and inclusivity.
- Using blockchain technology, this contract enables **borrowers to seek funding** for their projects and **investors to support projects** of their choice.

1) Inherits the Ownable contract: Implementing a secure ownership management system. This ensures that administrative functions are accessible only to **authorized owners**.

2) SafeMath Integration: Utilizes the '**SafeMath**' library to perform arithmetic operations securely, guarding against potential vulnerabilities arising from integer overflow and underflow.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

import './SafeMath.sol';
import './Credit.sol';

contract PeerToPeerLending is Ownable {
    using SafeMath for uint;
```

3) User Management

Defines a comprehensive '**User**' structure to manage individual user profiles.

Tracks whether a user has an

- 1) Is the user currently credited? (**bool**)
- 2) The address of the Active Credit (**address**)
- 3) Is the user marked as fraudulent? (**bool**)
- 4) All user credits. (**address []**)

```
// User structure
struct User {
    bool credited;

    address activeCredit;

    bool fraudStatus;

    address[] allCredits;
}
```

4) Storing Data

Now, we need to create a **mapping** that associates each user's Ethereum address with their corresponding **'User'** struct.

Also, declare an **array** to store the addresses of all credit contracts created on the platform.

```
// We store all users in a mapping.
mapping(address => User) public users;

// Array of all credits addresses.
address[] public credits;
```

5) Events

Now, we are going to introduce events. By emitting these events, the contract provides stakeholders with **valuable information** regarding credit creation, state changes, active status toggles, and updates to user fraud statuses.

A. LogCreditCreated():

- This event is emitted when a new credit is created on the platform.
- Provides a transparent record of credit **creation instances**, capturing relevant addresses and timestamps.

B. LogCreditStateChanged():

- This event signifies a **change in the state** of a credit contract.
- Offers insights into the evolving lifecycle of credit contracts, recording state transitions alongside timestamps.

C. LogCreditActiveChanged():

- Emitted when the active status of a **credit contract is toggled** (enabled or disabled).
- Provides a clear record of changes in the active status of credit contracts, allowing stakeholders to track their availability.

D. LogUserSetFraud():

- This event is emitted when a user's **fraud status** is updated.
- Maintains an audit trail of fraud status changes for users, ensuring transparency in addressing potential fraudulent activities.

```
event LogCreditCreated(address indexed _address, address indexed _borrower, uint indexed timestamp);
event LogCreditStateChanged(address indexed _address, Credit.State indexed state, uint indexed timestamp);
event LogCreditActiveChanged(address indexed _address, bool indexed active, uint indexed timestamp);
event LogUserSetFraud(address indexed _address, bool fraudStatus, uint timestamp);
```

6) Applying for Credit - applyForCredit Function

The "applyForCredit" function empowers users to request credit by submitting essential parameters, such as the requested amount, repayment terms, interest rate, and credit description. The function meticulously orchestrates the following steps:

A. User Eligibility Checks:

The function starts by validating user eligibility. It ensures that the user has not been credited previously and is not marked as fraudulent.

```
function applyForCredit(uint requestedAmount,uint repaymentsCount,uint interest,bytes memory creditDescription)
    public
    returns(address _credit) {
    // The user should not have been credited;
    require(users[msg.sender].credited == false);

    // The user should not be marked as fraudulent.
    require(users[msg.sender].fraudStatus == false);
```

B. Active Credit Validation:

The function verifies that the user does not have an active credit contract. This step prevents users from initiating multiple concurrent credits.

```
// Assert there is no active credit for the user.
assert(users[msg.sender].activeCredit == address(0));
```

C. Mark User as Credited:

Upon successful eligibility checks, the user's "credited" status is marked as true. This guards against reentrancy attacks by preventing multiple requests.

```
// Mark the user as credited. Prevent from reentrancy.  
users[msg.sender].credited = true;
```

D. Credit Contract Creation:

A new credit contract is created using the provided parameters: requested amount, repayment count, interest rate, and credit description. This new credit contract serves as the mechanism through which funds will be managed and repaid.

```
// Create a new credit contract with the given parameters.  
Credit credit = new Credit(requestedAmount, repaymentsCount, interest, creditDescription);
```

E. User Profile Update:

The user's "activeCredit" is set to the address of the newly created credit contract. This link establishes the association between the user and the applied credit.

```
// Set the user's active credit contract.  
users[msg.sender].activeCredit = address(credit);
```

F. Credit Tracking:

The address of the newly created credit contract is added to the "credits" array. This array maintains a comprehensive list of all active credit contracts within the platform.

```
// Add the credit contract to our list with contracts.  
credits.push(address(credit));
```


G. User Credit Record Update:

The address of the newly created credit contract is appended to the user's "allCredits" array.

This individualized list captures all credit contracts associated with the user.

```
// Add the credit to the user's profile.  
users[msg.sender].allCredits.push(address(credit));
```

H. Event Emission:

The function emits the "LogCreditCreated" event, documenting the credit creation instance. This event captures the credit contract's address, the borrower's address, and the exact timestamp.

```
// Log the credit creation event.  
emit LogCreditCreated(address(credit), msg.sender, block.timestamp);
```

I. Return Credit Contract Address:

Finally, the function returns the address of the newly created credit contract to the user, enabling them to track and manage their credit request.

```
    // Return the address of the newly created credit contract.  
    return address(credit);  
}
```

7) Retrieving Credit Addresses: getCredits Function

This function simplifies the process of obtaining a comprehensive **list of ongoing credit** instances.

```
function getCredits() public view returns (address[] memory) {  
    return credits;  
}
```

8) Retrieving User's Credit Addresses: getUserCredits Function


- Will be used to provide a streamlined method to **retrieve addresses** associated with their individual credit contracts.
- This function grants immediate access to a **personalized list** of their credit instances.

```
function getUserCredits() public view returns (address[] memory) {  
    return users[msg.sender].allCredits;  
}
```

9) Setting Fraud Status: setFraudStatus Function

- The '**setFraudStatus**' function allows to mark a specific user's account as fraudulent.
- Upon execution, the user's "**fraudStatus**" is set to true, signifying that their account is deemed potentially **fraudulent**.

- To ensure transparency and record the action, the "**LogUserSetFraud**" event is

```
function setFraudStatus(address _borrower) external returns (bool) {  infinite gas
    // Update user fraud status.
    users[_borrower].fraudStatus = true;

    // Log fraud status.
    emit LogUserSetFraud(_borrower, users[_borrower].fraudStatus, block.timestamp);

    return users[_borrower].fraudStatus;
}
```

emitted.

10) Changing Credit State: changeCreditState Function

- Due to the '**onlyOwner**' modifier, only accessible to the platform owner.
- Facilitates the alteration of the state of a specific credit contract.
- Upon invocation, the function **instantiates** the specified credit contract using the provided address.
- The "**changeState**" function within the credit contract is called, initiating the transition to the specified new state.
- To record the state change, the function emits the "**LogCreditStateChanged**" event.

```
function changeCreditState (Credit _credit, Credit.State state) public onlyOwner {
    // Call credit contract changeStage.
    Credit credit = Credit(_credit);
    credit.changeState(state);

    // Log state change.
    emit LogCreditStateChanged(address(credit), state, block.timestamp);
}
```

11) Toggling Credit Activity: changeCreditState Function

- Due to the '**onlyOwner**' modifier, exclusively accessible to the platform owner.

- Facilitates the toggling of the active status of a specific credit contract.
- Upon invocation, the function creates an interaction with the specified credit contract using the **provided address**.
- The "**toggleActive**" function within the credit contract is called. This function negates the credit contract's active status and returns a boolean value indicating the updated active state.
- To store the function transaction, the function emits the "**LogCreditActiveChanged**" event.

```
function changeCreditState (Credit _credit) public onlyOwner {  undefined gas
    // Call credit contract toggleActive method.
    Credit credit = Credit(_credit);
    bool active = credit.toggleActive();

    // Log state change.
    emit LogCreditActiveChanged(address(credit), active, block.timestamp);
}
```