# BiblioChain

A Blockchain-based library management system for libraries

Ray Rafael S. Abenido
John Gilbert Mundin

Submitted to:
Christian E. Pulmano

# Executive Summary

# Problem Definition

BiblioChain aims to resolve the problem of potentially inaccurate and tampered records, which can hinder effective bookkeeping and management of book inventory, and help librarians and administrators more easily determine who borrowed what book at what time by implementing a blockchain-based library management system. More specifically, it aims to

The objectives of BiblioChain are:

1. Ensure immutability of all records entered into BiblioChain.
2. Promote transparency and allow librarians to trace book borrowing history and user history.
3. Take advantage of the autonomous nature of smart contracts to reduce administrative burden, costs, and streamline library operations.
4. Provide libraries the ability to carry out basic library operations in a blockchain-based environment.

# Discussion of Blockchain Solution

In its current state, BiblioChain is designed to interact with three categories of users: the administrator (or superuser) who deployed and is the owner of an smart contract instance of BiblioChain, the librarian who helps manages an instance of BiblioChain, and the regular library user who borrows and returns books. These three categories reflect the usual structure of a real-life library management system: library users who borrow and return books, librarians who manage library inventory, and an administrator who oversees all library operations. The administrator is the highest-ranked person in BiblioChain, followed by the librarian, then by the library user.

BiblioChain gives each category of users a set of permissions according to their position, which are summarized in the table below. These permissions reflect many of the functions their counterparts in real-life library management systems perform:

| User Category | Permissions |
|---|---|
| **Administrator** | Enroll and unenroll librarians, enroll and unenroll users, and add, modify, and remove books |

| | |
|---|---|
| **Librarian** | Enroll and unenroll users, and add, modify, and remove books |
| **Library User** | Borrow and return books, renew books, and pay penalty if any |

The administrator is automatically set to the address of the user who deployed the BiblioChain. However, for one to become a librarian or library user, their address must be enrolled first. To be enrolled in this context means that BiblioChain has granted privileges to the user to interact with it in some way. If the user's address is not enrolled as either librarian or user, then they cannot interact with BiblioChain at all. Librarians and library users may also be unenrolled by the appropriate user.

Each category of users also comes with several inherent limitations. Enrolled library users cannot become librarians, and enrolled librarians cannot become library users. This limitation was designed to prevent library users with librarian privileges to remove their own hold orders, remove their late penalty fee, and general abuse of power within the smart contract. The administrator also cannot become librarians or users as they already have full access to the entire smart contract. Once the smart contact is deployed, the address of the Administrator **cannot be changed**. The administrator will always be the address of the user who deployed the smart contract.

Library users can borrow any book in the library, provided it hasn't been borrowed by someone else yet. They are given a two-week borrowing period after which they must return the book or can renew their two-week borrowing period. However, if the user returns the book after the two-week borrowing period has passed, they are automatically given a hold order and barred from borrowing books until they pay a penalty fee. This penalty fee is proportional to the amount of time passed since the two-week borrowing period has expired. Users also cannot renew their borrowing period if it has already expired.

# Business Benefits

**Reduced Administrative Work:**
BiblioChain automates key administrative tasks such as user enrollment, book borrowing, and penalty management. This greatly reduces the workload that the staff has to do.

**Enhanced User Experience:**
The system's automation simplifies book checkouts, renewals, and returns, leading to higher user satisfaction.

**Transparency:**
Its immutable nature ensures transparent and auditable records of all book-related transactions. This means that libraries can easily trace the history of each book and who borrowed it.

**Penalty Automation and Fairness:**
BiblioChain enforces penalty rules, which eliminates potential disputes and fosters a sense of fairness among users or those that are borrowing books. In the current BiblioChain, the only penalty it has is when the user/borrower does not return the book within the allotted borrowing period.

**Optimized Book Inventory Management:**
BiblioChain provides tools for efficient book inventory management, allowing administrators to add, modify, and remove books from the system without trouble which leads to a well-organized library catalog.

**Decentralized Security:**
Decentralization ensures the integrity and availability of book and user information, even in the face of external threats. Furthermore, it solves the problem of the risk of inaccuracies or tampering of data.

**Reduced Labor and Operation Cost:**
By automating repetitive tasks and reducing the potential for errors, BiblioChain contributes to reducing the cost in terms of labor and operations. Therefore, libraries can allocate their budget more efficiently.

# Conclusion

BiblioChain effectively follows the principles of blockchain technology to address challenges within traditional library management systems. By leveraging the immutability of blockchain, the contract ensures that once transactions are recorded, they remain tamper-proof and maintain data integrity. Transparency is a key feature, allowing authorized users, particularly administrators and librarians, to trace and verify book borrowing history and user interactions. While the decentralization aspect is present in the underlying blockchain infrastructure, the current implementation of governance remains somewhat centralized, with the administrator holding significant control. The contract incorporates security measures such as cryptographic hashing and digital signatures to safeguard against data tampering and unauthorized access. Automation of administrative tasks through smart contracts, exemplified by the penalty system, enhances operational efficiency.

**Future Considerations:**

1. Tokenization for Incentives
    a. Introduce a token-based incentive system to reward positive user behavior, fostering engagement and loyalty within the library community
2. Interoperability with External Systems
    a. Facilitate seamless data exchange by enabling interoperability with external library systems, promoting collaboration and data consistency across platforms
3. Decentralized Identity Integration
    a. Explore incorporating decentralized identity standards to enhance user privacy and security, aligning BiblioChain with evolving digital identity trends

# Appendix A
# Source Code

Source code is also hosted at GitHub and can be accessed through this link.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.2 <0.9.0;

contract BiblioChain {
    address public administrator; // the owner of the contract
    uint private nextBookID = 1;
    uint public fineAmount = 50000 wei; // 50,000 Wei fine for late book
return
    uint public leaseDuration = 2 weeks; // Two weeks lease duration

    constructor() {
        administrator = msg.sender;
    }

    // Bibliochain is user-centered
    // Three categories of users: administator, librarian, and the user
    // 1) Administrators can enroll and unenroll librarians & users. They
can also add, modify,
    // and remove books
    // 2) Librarians have the same permissions as the above, except for
enrolling and unerolling
    // librarians
    // 3) Users can borrow books, return them, and renew the borrowing
period. They will pay
    // a pentaly if they have exceeded their borrowing period and have not
returned their book

    struct Book {
        string bookName;
        string bookAuthor;
        string genre;
        string description;
        bool isBorrowed;
```

```
        bool doesExist;
    }

    struct User {
        bool isEnrolled;
        bool hasBorrowedBook;
        bool hasHoldOrder;
        uint borrowedBookID;
        uint dateBorrowed;
        uint dateDue;
        uint penaltyAmount; // Penalty amount for the user
    }

    mapping(address => User) public users;
    mapping(uint => Book) public books;
    mapping(address => bool) public isEnrolledAsLibrarian;

    // Array to hold user addresses
    address[] private userAddresses;

    /* ==================================
     * MODIFIERS
     ================================== */

    modifier isSuperUser() {
        require(msg.sender == administrator, "Request rejected. You are
not the administrator.");
        _;
    }

    modifier isLibrarian() {
        require(isEnrolledAsLibrarian[msg.sender] == true, "Request
rejected. You are not the librarian.");
        _;
    }

    modifier isSuperUserOrLibrarian() {
        require(
            msg.sender == administrator ||
isEnrolledAsLibrarian[msg.sender] == true,
```

```
            "Request rejected. You are not the administrator or
librarian."
        );
        _;
    }

    modifier isUserEnrolled(address _address) {
        require(users[_address].isEnrolled == true, "Request rejected. The
user is not enrolled.");
        _;
    }

    modifier doesBookExist(uint _bookID) {
        require(books[_bookID].doesExist == true, "Request rejected. The
book does not exist.");
        _;
    }

    modifier isBookBorrowed(uint _bookID) {
        require(books[_bookID].isBorrowed == true, "Request rejected. The
book is not borrowed by any user.");
        _;
    }

    modifier hasNoPenalty(address _userAddress) {
        require(users[_userAddress].penaltyAmount == 0, "Request rejected.
User has an existing penalty.");
        _;
    }

    /* ==================================
     * FUNCTIONS AVAILABLE ONLY TO ADMINISTRATOR
     ================================== */

    function enrollLibrarian(address _address) external isSuperUser() {
        require(users[_address].isEnrolled == false, "Enrolled users can't
become librarians.");
        require(administrator != _address, "Administrator need not apply
to be librarian or user.");
        isEnrolledAsLibrarian[_address] = true;
```

```
    }

    function unenrollLibrarian(address _address) external isSuperUser() {
        isEnrolledAsLibrarian[_address] = false;
    }

    /* ==================================
     * FUNCTIONS AVAILABLE TO BOTH ADMINISTRATOR AND LIBRARIAN
     ================================== */

    // FOR (UN)ENROLLING USERS
    function enrollUser(address _userAddress) external
isSuperUserOrLibrarian() {
        require(isEnrolledAsLibrarian[_userAddress] == false, "Requested
rejected. Librarians cannot become users.");
        require(administrator != _userAddress, "Administrator need not
apply to be librarian or user.");
        users[_userAddress].isEnrolled = true;
        userAddresses.push(_userAddress); // Add the user address to the
array
    }

    function unenrollUser(address _userAddress) external
isSuperUserOrLibrarian() isUserEnrolled(_userAddress) {
        require(users[_userAddress].hasBorrowedBook == false, "Request
rejected. User has borrowed a book");
        require(users[_userAddress].hasHoldOrder == false, "Request
rejected. User has a pending hold order.");

        // If the user has a penalty, we deduct it before unenrolling the
user
        if (users[_userAddress].penaltyAmount > 0) {
            deductPenalty(_userAddress);
        }

        users[_userAddress].isEnrolled = false;

        // Remove the user address from the array
        for (uint i = 0; i < userAddresses.length; i++) {
            if (userAddresses[i] == _userAddress) {
```

```
                // Swap with the last element and pop
                userAddresses[i] = userAddresses[userAddresses.length -
1];
                userAddresses.pop();
                break;
            }
        }
    }

    // FOR MANAGING BOOK INVENTORY
    function incrementBookNumber() private {
        nextBookID += 1;
    }

    function addBook(
        string memory _bookName,
        string memory _bookAuthor,
        string memory _genre,
        string memory _description
    ) external isSuperUserOrLibrarian() {
        books[nextBookID].doesExist = true;
        books[nextBookID].bookName = _bookName;
        books[nextBookID].bookAuthor = _bookAuthor;
        books[nextBookID].genre = _genre;
        books[nextBookID].description = _description;
        incrementBookNumber();
    }

    function removeBook(uint _bookID) external isSuperUserOrLibrarian()
doesBookExist(_bookID) {
        require(!books[_bookID].isBorrowed, "Request rejected. Book is
currently borrowed.");
        books[_bookID].doesExist = false;
        books[_bookID].bookName = "";
        books[_bookID].bookAuthor = "";
        books[_bookID].genre = "";
        books[_bookID].description = "";
    }
```

```
    function modifyBookName(uint _bookID, string memory _bookName)
external isSuperUserOrLibrarian() doesBookExist(_bookID) {
        require(books[_bookID].isBorrowed == false, "Request rejected.
Books cannot be modified when borrowed by a user.");
        books[_bookID].bookName = _bookName;
    }

    function modifyBookAuthor(uint _bookID, string memory _bookAuthor)
external isSuperUserOrLibrarian() doesBookExist(_bookID) {
        require(books[_bookID].isBorrowed == false, "Request rejected.
Books cannot be modified when borrowed by a user.");
        books[_bookID].bookAuthor = _bookAuthor;
    }

    function modifyDescription(uint _bookID, string memory _description)
external isSuperUserOrLibrarian() doesBookExist(_bookID) {
        require(books[_bookID].isBorrowed == false, "Request rejected.
Books cannot be modified when borrowed by a user.");
        books[_bookID].description = _description;
    }

    function modifyGenre(uint _bookID, string memory _genre) external
isSuperUserOrLibrarian() doesBookExist(_bookID) {
        require(books[_bookID].isBorrowed == false, "Request rejected.
Books cannot be modified when borrowed by a user.");
        books[_bookID].genre = _genre;
    }

    // FUNCTION TO CHARGE PENALTY FOR LATE BOOK RETURN
    function chargePenalty(address _userAddress) private {
        //require(users[_userAddress].hasBorrowedBook, "Request rejected.
User has not borrowed any book.");
        uint currentDate = block.timestamp;
        //require(currentDate > users[_userAddress].dateDue, "Request
rejected. Book is not overdue yet.");
        uint overdueDays = (currentDate - users[_userAddress].dateDue) / 1
days;
        uint penaltyAmount = fineAmount * overdueDays;
```

```
        // Deduct penalty amount from user's balance or take other
appropriate action
        // For simplicity, we'll store the penalty amount in the user's
struct
        users[_userAddress].penaltyAmount += penaltyAmount;

        // Emit an event with penalty details
        emit PenaltyCharged(_userAddress,
users[_userAddress].borrowedBookID, penaltyAmount);
    }

    function deductPenalty(address _userAddress) private {
        // Implement the logic to deduct the penalty amount from the
user's balance
        // In a real-world scenario, you would need to implement a payment
system or use a token transfer function.
        // For simplicity, we'll set the penalty amount to 0 in the user's
struct.
        users[_userAddress].penaltyAmount = 0;
        users[_userAddress].hasHoldOrder = false;
    }

    event PenaltyCharged(address indexed userAddress, uint indexed
borrowedBookID, uint penaltyAmount);

    // FUNCTION TO ITERATE OVER USER ADDRESSES
    function getAllUserAddresses() external view returns (address[]
memory) {
        return userAddresses;
    }

    /* ==================================
     * FUNCTIONS AVAILABLE TO THE USER
     ================================== */

    // FUNCTION FOR USER TO PAY PENALTY AND LIFT THE PENALTY
    function payPenalty() external isUserEnrolled(msg.sender) {
        require(users[msg.sender].penaltyAmount > 0, "Request rejected.
User does not have any pending penalty.");
```

```
        // Implement the logic for the user to pay the penalty amount
        // For simplicity, we'll call the deductPenalty function to set
the penalty amount to 0.
        deductPenalty(msg.sender);
    }

    // FUNCTION FOR USER TO BORROW A BOOK
    function borrowBook(uint _bookID) external isUserEnrolled(msg.sender)
doesBookExist(_bookID) hasNoPenalty(msg.sender) {
        require(!books[_bookID].isBorrowed, "Request rejected. The book is
already borrowed.");
        require(!users[msg.sender].hasHoldOrder, "Request rejected. You
cannot borrow books if you have a hold order.");

        // Implement the logic for the user to borrow a book
        // For simplicity, we'll set the book status to borrowed and
update user's information.
        books[_bookID].isBorrowed = true;
        users[msg.sender].hasBorrowedBook = true;
        users[msg.sender].borrowedBookID = _bookID;
        users[msg.sender].dateBorrowed = block.timestamp;
        users[msg.sender].dateDue = block.timestamp + leaseDuration;
    }

    // if the user has borrowed a book and is not late/overdue for
returning it,
    // they can renew it again for another two weeks
    function renewLease() external isUserEnrolled(msg.sender) {
        require(users[msg.sender].hasBorrowedBook, "Request rejected. User
has not borrowed any book.");
        require(!isBookOverdue(msg.sender), "Request rejected. Book is
overdue.");

        // Extend the lease for another two weeks
        users[msg.sender].dateDue += leaseDuration;
    }

    // Function to check if the borrowed book is overdue
    function isBookOverdue(address _userAddress) private view returns
(bool) {
```

```
        require(users[_userAddress].hasBorrowedBook, "User has not
borrowed any book.");

        uint currentDate = block.timestamp;

        return currentDate > users[_userAddress].dateDue;
    }

    // return book
    function returnBook() external isUserEnrolled(msg.sender) {
        require(users[msg.sender].hasBorrowedBook, "User has not borrowed
any book.");
        if (block.timestamp > users[msg.sender].dateDue) {
            users[msg.sender].hasHoldOrder = true;
            chargePenalty(msg.sender);
        }

        users[msg.sender].hasBorrowedBook = false;
        books[users[msg.sender].borrowedBookID].isBorrowed = false;
        users[msg.sender].borrowedBookID = 0;

        // no need to change dateBorrowed and dateDue
        // they are no longer used anyways
    }

    // this is a tester function meant to simulate a situation
    // where the user is late in returning their book on time
    /* function setBookAsLate() external isUserEnrolled(msg.sender) {
        users[msg.sender].dateDue = 0;
    } */
}
```

# Appendix B
# Documentation

**How to Deploy and Use Bibliochain on Remix IDE:**

1. Upload BiblioChain.sol on Remix IDE
2. Select an address to serve as the administrator of a Bibliochain instance then compile and deploy BiblioChain.sol
3. Once the smart contract has been deployed, first choose one address to serve as the Librarian then using the administrator address enroll the librarian address as librarian via enrollLibrarian()
4. Verify if the librarian address has been enrolled via the isEnrolledAsLibrarian variable. The variable is set to public and should be accessible.
5. Add a book via the addBook() and choose an address to serve as user address using either the librarian or administrator addresses to carry out the next step. Verify that the book exists and user address is enrolled via the books and users public variables respectively.
    a. Books are assigned a book number starting from one, which then increments as books are added to the smart contract. Since the book you added is the first book added in the smart contract, its book number is 1.
6. [Optional] Modify the added book's details if you want.
7. Using the user address, borrow the book you added. Verify the book has been borrowed via the books and users public variables.
8. Using the user address, return the book you borrowed. Verify the book has been returned via the books and users public variables.

*Simulating a Late Book Scenario*

1. To simulate a situation where the user is late in returning a book, uncomment setBookAsLate(). This function has been commented out, but not deleted from the smart contract for testing purposes.
2. Execute the first eight steps again, then borrow a book using the user address and click setBookAsLate().
3. Return the book you borrowed using the user address. Verify that the user address has a hold order via the users public variable.
4. Click payPenalty(). This will remove the hold order. Verify that the user address has no hold order via the users public variable.