Race Attack in Blockchain Systems

A blockchain is a distributed, immutable ledger that records transactions in a secure and transparent manner. This decentralized system relies on cryptography to verify and validate each block, ensuring data integrity and preventing unauthorized modifications. The blockchain's structure, which comprises blocks linked together in a chronological chain, provides a robust foundation for secure and reliable data management.

Overview of the blockchain structure

Blockchain Structure

A blockchain is a chain of blocks, each containing transactions and a timestamp. Each block is linked to the previous block using a cryptographic hash, creating an immutable and chronological record of transactions.

Importance of Block Validation

Each block is validated by a network of nodes, ensuring data integrity and preventing fraudulent transactions. This consensus mechanism, often using proof-of-work or proof-of-stake, secures the blockchain and maintains its reliability.

Overview of the blockchain structure

Blocks

Blocks are the fundamental units of a blockchain. Each block contains a set of transactions, a timestamp, and a hash of the previous block. This creates a chain of blocks, where each block is linked to the previous one.

Hashing

Each block is cryptographically hashed, generating a unique identifier. This hash is also included in the subsequent block, ensuring the integrity of the chain. Any attempt to alter a block's data will result in a different hash, making the change immediately apparent.

Importance of block validation and security

Data Integrity

Block validation ensures that the data within each block remains unchanged and tamper-proof. This safeguards the integrity of the blockchain, ensuring reliable and trustworthy transactions.

Fraud Prevention

The consensus mechanism involved in block validation prevents fraudulent transactions from being added to the blockchain, maintaining its security and trust.

Transparency and Auditability

The public and verifiable nature of blockchain transactions provides transparency and auditability, allowing anyone to trace the history of transactions and verify their legitimacy.

What is a Race Attack?

Attacker's Goal

An attacker aims to manipulate the blockchain by submitting transactions faster than legitimate users, potentially causing double-spending or other malicious actions.

Exploiting Vulnerabilities

Race attacks exploit weaknesses in blockchain consensus mechanisms, such as slow block confirmation times or vulnerabilities in the network's communication protocols.

Potential Consequences

Successful race attacks can lead to financial losses, compromised transactions, and a breach of trust in the blockchain system.

What is a Race Attack?

1

Double-Spending

An attacker attempts to spend the same cryptocurrency twice.

2

Transaction Race

The attacker tries to submit a transaction before the legitimate user.

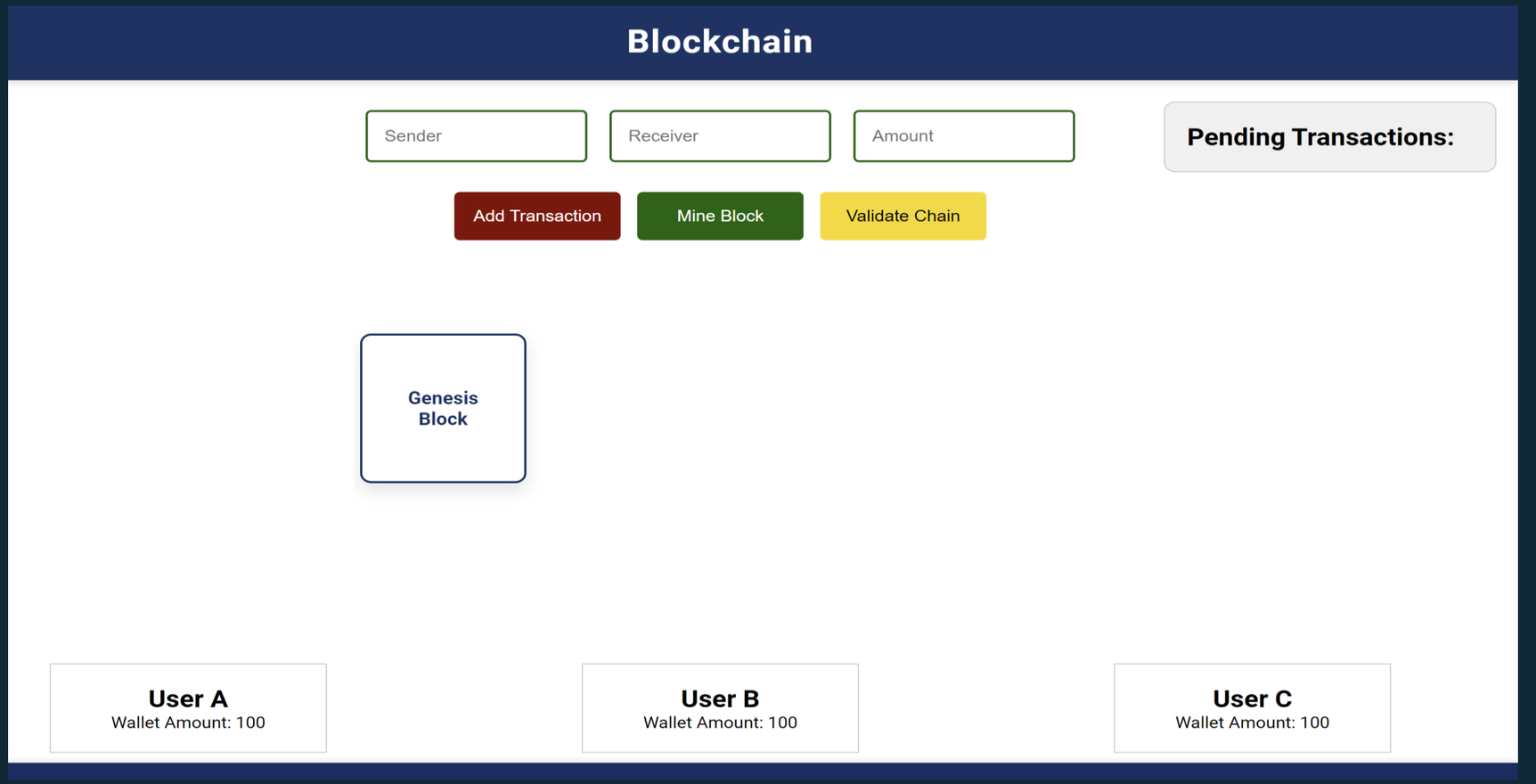
3

Blockchain Manipulation

The attacker aims to deceive the blockchain network.

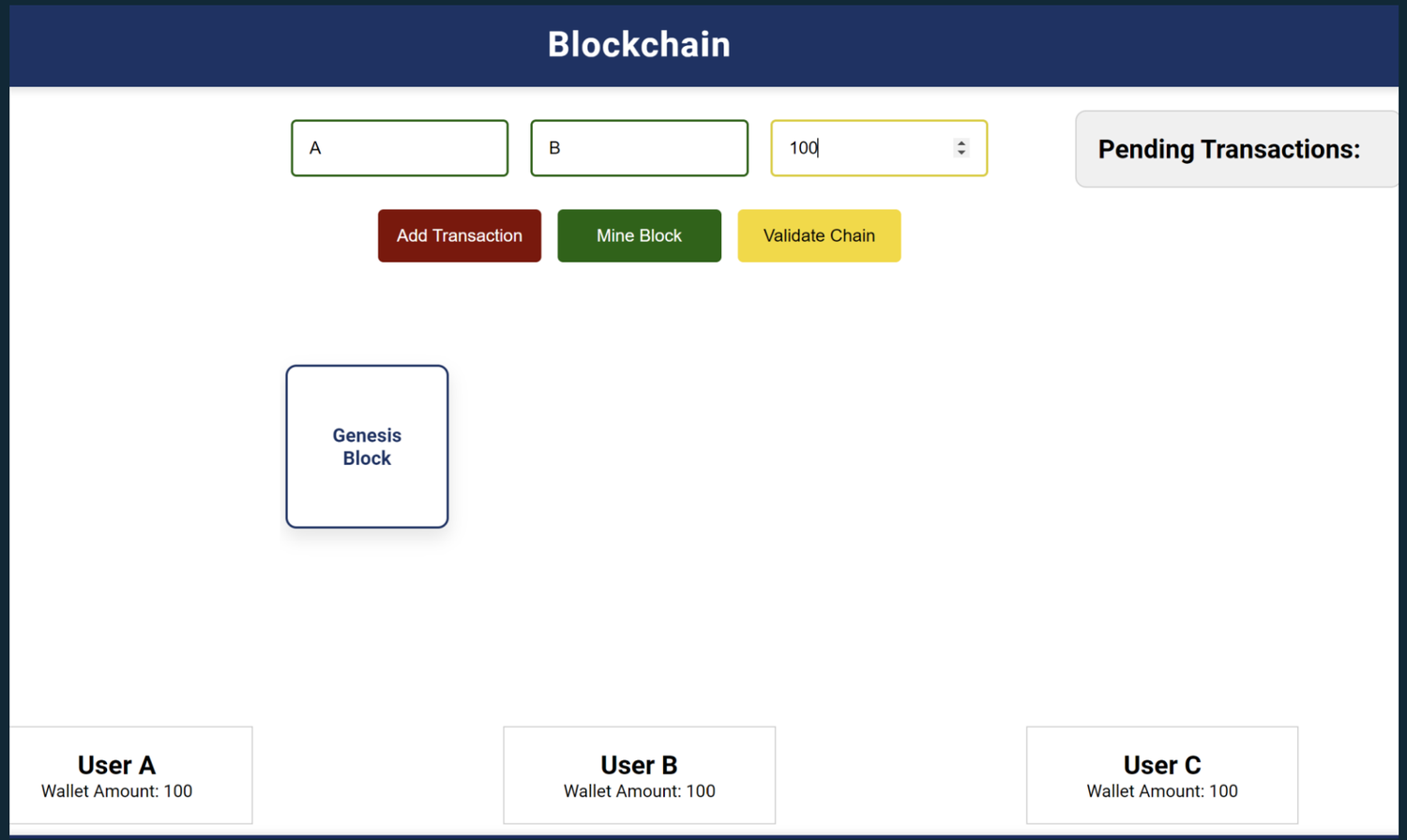
Working

Race Attack is demonstrated with initial state of 3 users A,B,C , having initial amount of 100 coins
In their accounts



Working

Now the attacker A initialized first transaction to user B amounting to 100 coins.

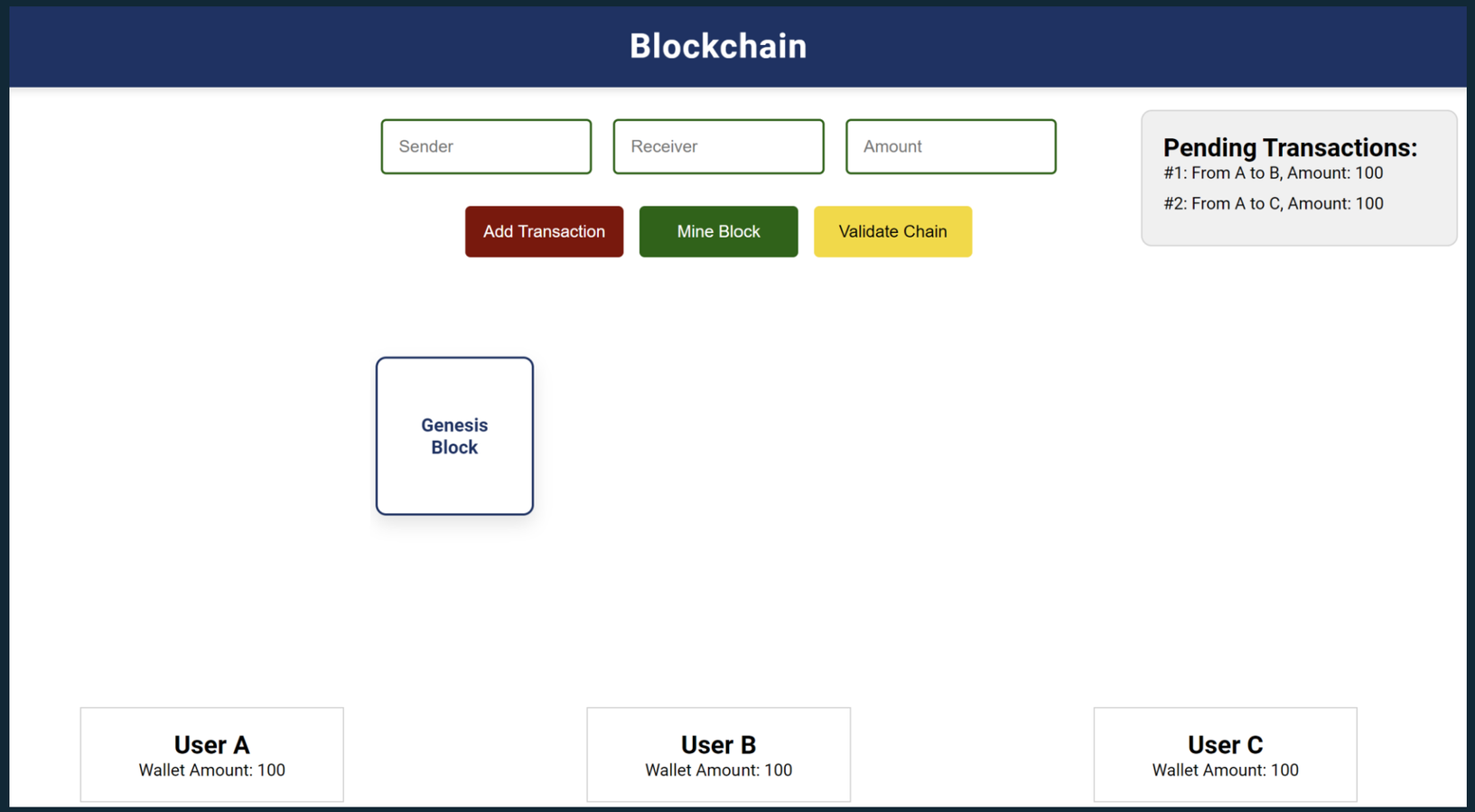


Working

A initiated another transaction to C using same coins .

Both of transaction are stored in pending transactions pool called as mempool.

Competing Transactions: The system shows a queue of "Pending Transactions" that have been added but not yet mined into a block. If two transactions from User A are in the pending list (one to User B and another to User C), but User A only has sufficient balance for one, only one transaction can succeed.



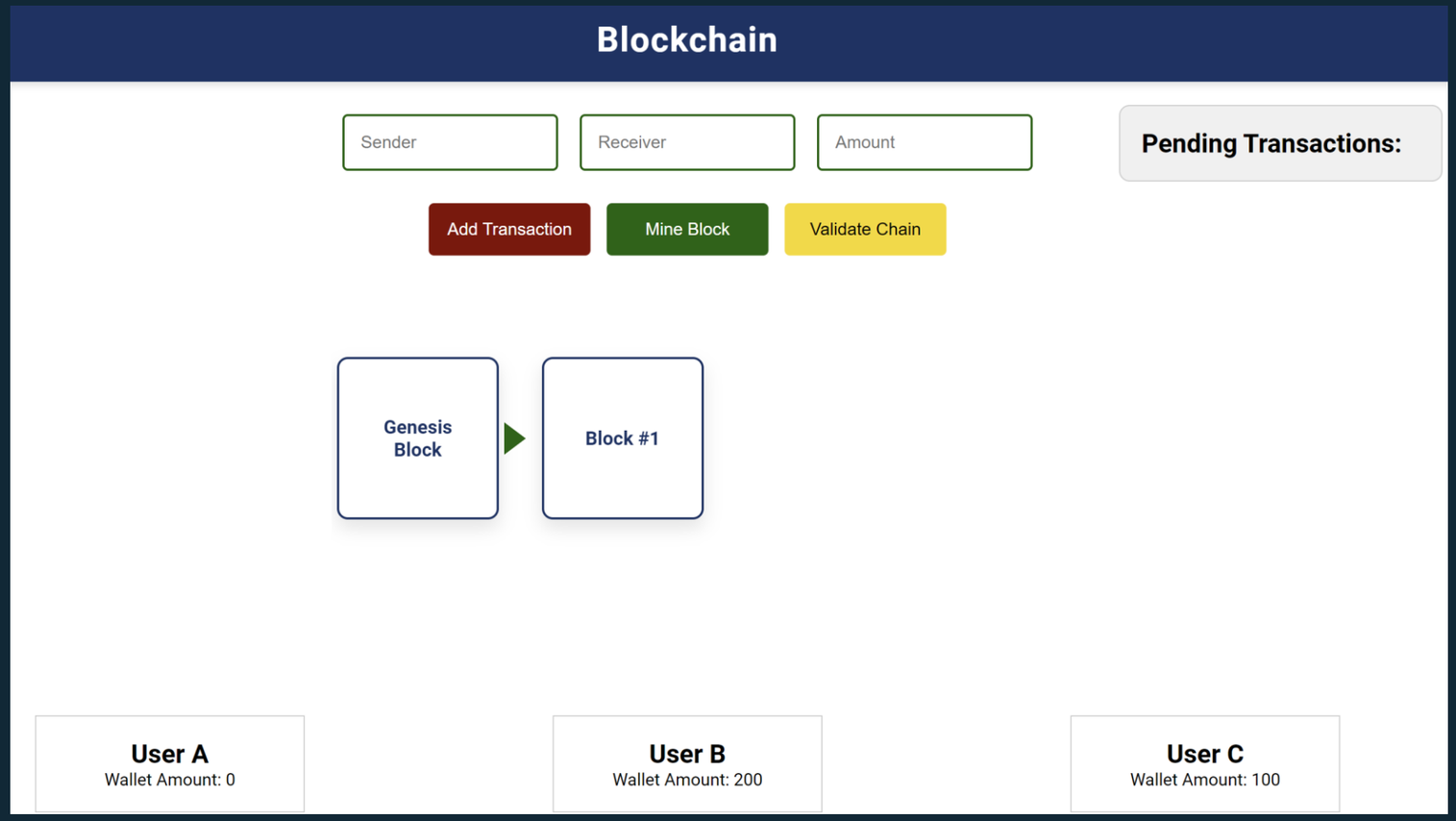
Working

Now when miners mine the block , selecting a block of pending transaction to get validated
1.A---→B
2.A--→C

Transaction from A to B is validated where B is another Address of attacker A , incrementing B accounts balance to 200

While Tranaction from A to receiver C does not get validated .

Mining a Block: When the "Mine Block" button is pressed, one of the pending transactions will be added to the blockchain based on the block miner's view of the transaction order, leaving the other in a conflicting state, which can be rejected later during validation.



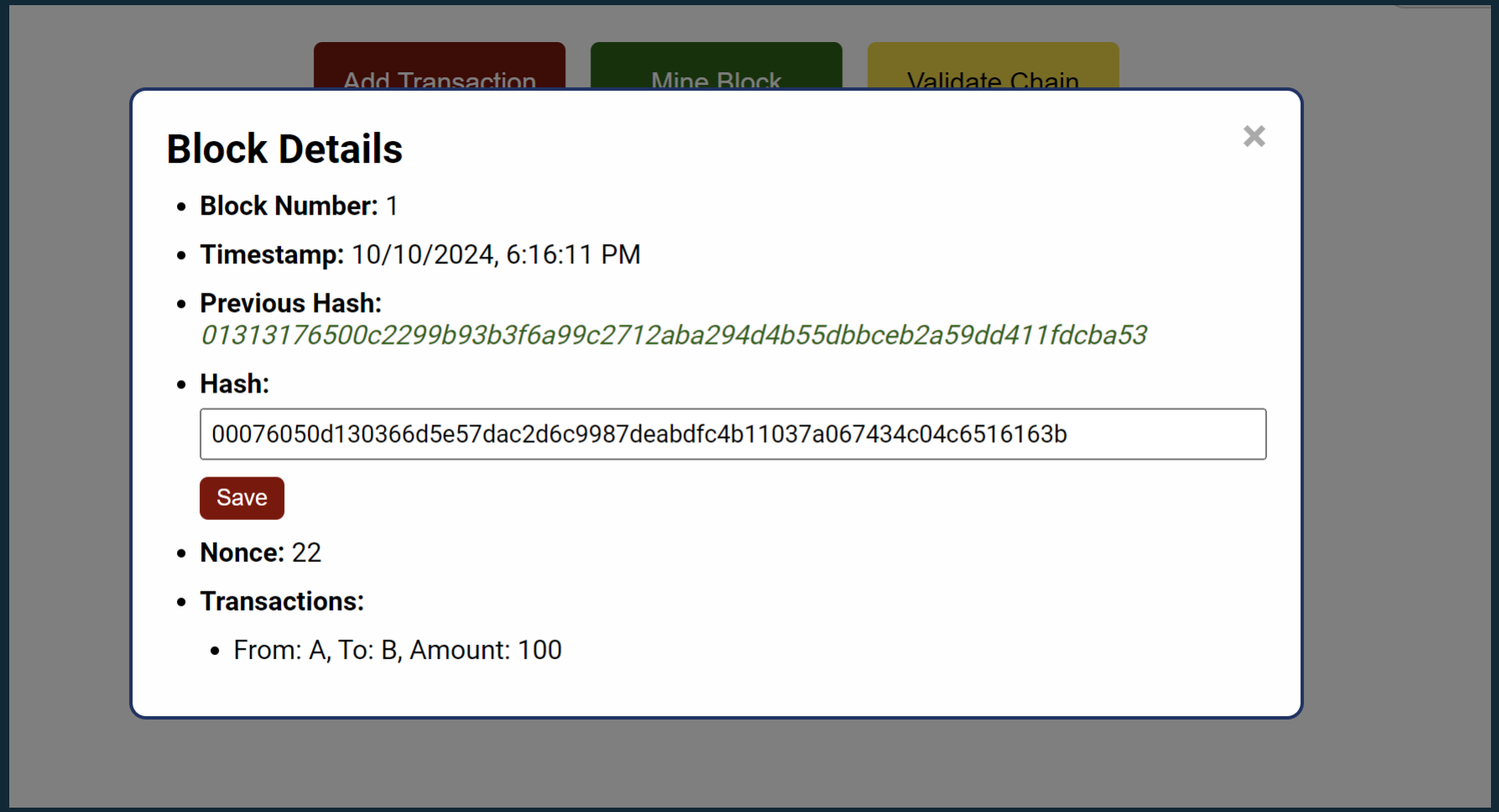
Working

Only transaction from A to B is added to blockchain after validation from other nodes,

Leaving A to C transaction in inconsistent state And later discarding from blockchain.

Due to the decentralized nature of blockchain, both transactions may be propagated to different parts of the network before a consensus is reached.

The transaction from User A to User C becomes inconsistent and gets discarded because User A's balance is insufficient after the A to B transaction is confirmed, leading to a double-spend conflict.



Code Snippets

Modal Display Function

This function is responsible for opening a modal. It takes the **modalId** as a parameter, which refers to the specific modal that should be opened. It first locates the modal using **document.getElementById(modalId).** If the modal exists, it changes its **style.display** property to **"flex",** which makes the modal visible and enables the flexbox layout.

Code Snippet

```
const openModal = (modalId) => {
    const modal = document.gtElementById(modalId);
    if (modal) {
        modal.style.display = "flex";
    }
}
```

Code Snippets

Block Class

Block Class

•Represents a single block in the blockchain.

- •index: The position of the block in the blockchain.
- •timestamp: When the block was created.
- •transactions: A list of transactions associated with the block.
- •previousHash: The hash of the previous block to maintain the chain's integrity.
- •nonce: A value incremented during mining to achieve the required hash.
- •hash: The calculated hash of the block based on its content and nonce, generated using the calculateHash method.

Code Snippet

```
class Block {
    constructor(index, timestamp, transactions, previousHash = '') {
        this.index = index;
        this.timestamp = timestamp;
        this.transactions = transactions;
        this.previousHash = previousHash;
        this.nonce = 0;
        this.hash = this.calculateHash();
    }

    calculateHash() {
        return CryptoJS.SHA256(
            this.index +
            this.timestamp +
            JSON.stringify(this.transactions) +
            this.previousHash +
            this.nonce
        ).toString();
    }

    mineBlock(difficulty) {
        const target = Array(difficulty + 1).join("0");
        while (this.hash.substring(0, difficulty) !== target) {
            this.nonce++;
            this.hash = this.calculateHash();
        }
    }
}
```

Code Snippets

BlockChain Class Definition

- Initializes the blockchain with:
  - A genesis block (the first block) created by the createGenesisBlock method.
- Creates the first block (genesis block) in the chain with predefined data. No previous hash since it is the first block.

•**etLatestBlock()**:

- **Method: addBlock(newBlock)**:
  - Adds a new block to the chain by:
    - Setting its previousHash to the latest block's hash.
    - Mining the block (solving the hash puzzle).
    - Clearing the pendingTransactions once the block is successfully mined**Method: isChainValid()**:
  - Validates the integrity of the blockchain by checking:
    - The recalculated hash of each block matches its current hash.
    - The previousHash field of each block correctly points to the hash of the previous block.
    - Each block's hash satisfies the mining difficulty (starts with enough leading zeros).
- **Method: addTransaction(sender, receiver, amount)**:
  - Adds a new transaction to the pendingTransactions pool.
  - Simplified here by skipping the balance validation to ensure the transaction is always added.

Code Snippet

```
class Blockchain {
    constructor() {
        this.chain = [this.createGenesisBlock()];
        this.difficulty = 2; // Number of leading zeros required
        this.pendingTransactions = [];
        this.users = {
            'A': {
                publicKey: 'A',
                privateKey: 'A',
                balance: 100
            },
            'B': {
                publicKey: 'B',
                privateKey: 'B',
                balance: 100
            },
            'C': {
                publicKey: 'C',
                privateKey: 'C',
                balance: 100
            }
        };
    }

    createGenesisBlock() {
        return new Block(0, new Date().toLocaleString(), ["Genesis Block"], "0");
    }

    getLatestBlock() {
        return this.chain[this.chain.length - 1];
    }

    addBlock(newBlock) {
        newBlock.previousHash = this.getLatestBlock().hash;
        newBlock.mineBlock(this.difficulty);
        this.chain.push(newBlock);
        this.pendingTransactions = [];
    }

    isChainValid() {
        for (let i = 1; i < this.chain.length; i++) {
            const currentBlock = this.chain[i];
            const previousBlock = this.chain[i - 1];

            if (currentBlock.hash !== currentBlock.calculateHash()) {
                return false;
            }

            if (currentBlock.previousHash !== previousBlock.hash) {
                return false;
            }

            if (currentBlock.hash.substring(0, this.difficulty) !== Array(this.difficulty + 1).join("0")) {
                return false;
            }
        }
        return true;
    }

    addTransaction(sender, receiver, amount) {
        // if (this.users[sender].balance >= amount) {
            // this.users[sender].balance -= amount;
            // this.users[receiver].balance += amount;
            this.pendingTransactions.push({ sender, receiver, amount });
            return true;
        // }
        // return false;
    }
}
```

Code Snippets

Handle Transaction Input

This function handles the submission of transaction input.It gets the transaction input from the input field using **querySelector().** The **value.trim()** method ensures any leading or trailing spaces are removed. It displays a message based on whether the input is valid. If the input is empty, the message shows an error in red, and if the input is valid, it displays the submitted transaction in green. The function also calls **addTransaction()** (explained later) to process the transaction, and clears the input field after submission.

Code Snippet

```
const handleTransactionInput = () => {
    const input =
document.querySelector('.transaction-input
input');
    const value = input.value.trim();
    const message =
document.getElementById('message');

    if (value === "") {
        message.textContent = "Transaction input
cannot be empty!";
        message.style.color = "red";
    } else {
        message.textContent = `Transaction
"${value}" has been submitted.`;
        message.style.color = "green";
        addTransaction(value);
        input.value = ""; // Clear input field
    }
}
```

Code Snippets

Add Transaction Function

This function adds the new transaction to a list of pending transactions. It selects the unordered list **(ul)** inside the **.pending-transactions** section and creates a new **li** (list item) element. The **transaction** parameter is set as the text content of the new **li**, and it's appended to the **ul**, which visually updates the pending transactions list.

Code Snippet

```
const addTransaction = (transaction) => {
    const pendingList =
document.querySelector('.pending-transactions
ul');
    const li = document.createElement('li');
    li.textContent = transaction;
    pendingList.appendChild(li);
}
```

Code Snippets

Blockchain Block Click Event

This code adds a click event listener to each block in the blockchain.The **querySelectorAll()** method selects all elements with the class **.block,** and for each one, it attaches a click event listener. When a block is clicked, it retrieves the block's ID **(blockId)** from the **data-blockId** attribute of the clicked block **(e.target.dataset.blockId).** The function then opens the corresponding modal by calling **openModal()** and passing the block-specific modal ID (e.g., **modal-1**).

Code Snippet

```
document.querySelectorAll('.block').forEach(block
=> {
    block.addEventListener('click', (e) => {
        const blockId = e.target.dataset.blockId;
        openModal(`modal-${blockId}`);
    });
});
```

Code Snippets

Save Editable Hash Function

This function allows the user to save an editable hash within a modal and update the corresponding blockchain block. It first selects the editable input field **(editableHash)** within the modal using the block's ID. The entered value is trimmed of any extra spaces **(newHash).** If it's not empty, the text content of the block corresponding to the blockId is updated with the new hash. After saving the hash, the modal is closed by calling **closeModal().** If the new hash is empty, an alert is shown.

Code Snippet

```
const saveHash = (blockId) => {
    const editableHash =
document.querySelector(`#modal-${blockId}
.editable-hash`);
    const newHash = editableHash.value.trim();

    if (newHash) {
        document.querySelector(`.block[data-block-
id="${blockId}"]`).textContent = newHash;
        closeModal(`modal-${blockId}`);
    } else {
        alert("Hash cannot be empty.");
    }
}
```