Université Claude Bernard Lyon1
M2 Informatique/Data & Intelligence for Smart Systems

Wedding Contract Project

Blockchain Block

Supervised by: Prof. Jean Patrick GELAS

Released by: Marie Antoinette KILO
(Student ID :12311005)

2024

https://github.com/marie-kilo/Wedding-Contract-Project-BlockChain-Ethereum-

1-Introduction:

My project utilizes Ethereum smart contracts, specifically written in Solidity (version 0.5.7). Two contracts were developed: **WeddingCertificate** and **WeddingCertificateFactory**.

- **WeddingCertificate**: This contract stores the names of the partners. It includes a constructor for initializing partner names and a function to emit an event with partner details.
- **WeddingCertificateFactory**: this factory contract creates instances of **WeddingCertificate**. Factory patterns are a common design pattern in smart contract development for managing multiple contract instances.

1.1Web3.js Integration:

Web3.js is a collection of libraries that allow to interact with a local or remote Ethereum node using HTTP, IPC, or WebSocket. In my project, Web3.js is used to connect the web interface with the Ethereum blockchain.

- **Connecting to MetaMask**: The script checks for Ethereum capabilities in the user's browser and requests account access using MetaMask. This is a common approach to integrating Ethereum functionality into web applications, enabling users to perform transactions and interact with smart contracts.

```
document.addEventListener('DOMContentLoaded', async () => {
    let web3;

    if (typeof window.ethereum !== 'undefined') {
        web3 = new Web3(window.ethereum);

        try {
            await window.ethereum.request({ method: 'eth_requestAccounts' });
        } catch (error) {
            console.error('User denied account access:', error);
            return;
        }
    } else {
        console.log('Non-Ethereum browser detected. You should consider trying MetaMask!');
        return;
    }
```

2-Front-End Interaction:

The front-end, built with HTML and JavaScript, interacts with the smart contracts using the Web3.js library. This interaction is facilitated through:

- **Contract ABI and Address**: The ABI (Application Binary Interface) and the address of the **WeddingCertificateFactory** contract are used to instantiate a Web3.js contract object, allowing for interactions with the contract's functions.

```
const FactoryABI = [ ···
];
const FactorytAddress = '0xd89BB9958E49451dAE641e3177500bD5639B7A9D'; // Factoy address

const contract = new web3.eth.Contract(FactoryABI, FactorytAddress);
```

A- Creating Wedding Certificates:

Detailed Functionality:

- **Event Listener**: The script adds an event listener to the **weddingForm** form that triggers when the form is submitted.
- **Preventing Default Form Submission**: **event.preventDefault()** is called to prevent the form from submitting in the traditional manner, allowing the script to handle the submission.
- **Retrieving User Input**: It collects the values entered by the user in the **partner1** and **partner2** input fields, representing the names of the partners.
- **Transaction Status Element**: A reference to the **transactionResult** element is obtained to display transaction status messages to the user.
- **Initiating Transaction**: The script retrieves the user's Ethereum accounts using **web3.eth.getAccounts()**. It then initiates a transaction by calling the **createCertificate** method of the smart contract, passing in the partner names and specifying the transaction's sender as the first account retrieved.
- **Transaction Feedback**:
- **Transaction Hash**: When the transaction hash is available, indicating the transaction was sent to the network, a link to the transaction on Goerli Etherscan is displayed, allowing the user to track the transaction status.
- **Transaction Receipt**: Upon receiving the transaction receipt, the script displays confirmation information, including a newly assigned certificate number, the block number of the transaction, and the address of the newly created **WeddingCertificate** contract.
- **Error Handling**: In case of errors during transaction processing (e.g., transaction rejected or failed), an error message is displayed to the user.
- **Error Handling for Account Retrieval**: If there's an error in retrieving the user's accounts (e.g., user not logged into MetaMask), an error message is displayed.

```javascript
document.getElementById('weddingForm').addEventListener('submit', async (event) => {
    event.preventDefault();

    const partner1 = document.getElementById('partner1').value;
    const partner2 = document.getElementById('partner2').value;
    const transactionResultElement = document.getElementById('transactionResult');

    transactionResultElement.innerHTML = 'Status: Please wait... Confirm With MetaMask!!!';

    try {
        const accounts = await web3.eth.getAccounts();

        contract.methods.createCertificate(partner1, partner2).send({ from: accounts[0] })
        .on('transactionHash', (hash) => {
            const txHashLink = `https://goerli.etherscan.io/tx/${hash}`;
            transactionResultElement.innerHTML = `Transaction sent! <a href="${txHashLink}" target="_blank">${hash}</a><br/>Waiting for confirmation..
        })
        .on('receipt', (receipt) => {
            const certificateNumber = parseInt(receipt.events.ContractCreated.returnValues.certificateNumber, 10) + 1;
            const newContractAddress = receipt.events.ContractCreated.returnValues.contractAddress;
            const blockNumber = receipt.blockNumber;
            const txHashLink = `https://goerli.etherscan.io/tx/${receipt.transactionHash}`;

            transactionResultElement.innerHTML = `
                Transaction sent! <a href="${txHashLink}" target="_blank">${receipt.transactionHash}</a><br/>Transaction confirmed!
                <br/>Certificate #: ${certificateNumber}
                <br/>Block #: ${blockNumber}
                <br/>New Certificate Address: ${newContractAddress}
            `;
        })
        .on('error', (error) => {
            console.error('Transaction failed:', error);
            transactionResultElement.innerHTML = `Transaction failed: ${error.message}`;
        });
    } catch (error) {
        console.error('Error:', error);
        transactionResultElement.textContent = 'Transaction failed: ' + error.message;
    }
});
```

B- Verification Process Overview

1. **Event Listener:** The script adds an event listener to a button with the id verify. When clicked, it triggers the verification process.
2. **Retrieving Contract Address**: It fetches the value entered by the user in the input field with the id contractAddress, which should be the Ethereum address of a specific WeddingCertificate contract instance the user wishes to verify.
3. **Setting Up the Contract Instance:** Using the provided contract address and the ABI (Application Binary Interface) of the WeddingCertificate contract, it creates a new contract instance with web3.eth.Contract. This ABI outlines the functions and events within the WeddingCertificate contract, enabling interaction with the contract's methods.
4. **Retrieving Partner Names:**
   - The script calls the partner1() and partner2() functions of the contract instance, which are marked as constant and view, meaning they don't modify the blockchain state and only return data.
   - These functions return the names of the partners stored in the smart contract, representing the parties involved in the wedding certificate.
5. **Displaying Verification Results**: The retrieved names of **the partners** are displayed in an element with the id verificationResult, along with the **current block number** of the Ethereum blockchain. This block number serves as a reference to the state of the blockchain at the time of verification.
6. **Error Handling:** If there are any issues in executing the contract calls (e.g., incorrect contract address, network errors, etc.), an error message is displayed in the verificationResult element.

Key Components

Web3.js: This library is used for interacting with Ethereum nodes, enabling the script to communicate with the blockchain and smart contracts.

Smart Contract Interaction: The script uses the ABI of the **WeddingCertificate** contract to interact with its deployed instance, specifically calling functions to read data.

Asynchronous JavaScript: The use of **async/await** syntax allows for asynchronous blockchain calls to be made in a more readable and manageable way.

```
:ument.getElementById('verify').addEventListener('click', async function() {
  const contractAddress = document.getElementById('contractAddress').value;

  // Assuming you have the ABI of WeddingCertificate
> const certificateAbi =  [···
  ];
  const certificateContract = new web3.eth.Contract(certificateAbi, contractAddress);

  try {
      const partner1Name = await certificateContract.methods.partner1().call();
      const partner2Name = await certificateContract.methods.partner2().call();

      const currentBlockNumber = await web3.eth.getBlockNumber();

      document.getElementById('verificationResult').innerText = `Partner 1: ${partner1Name}, Partner 2: ${partner2Name}, Block#: ${currentBlockNumber}`;
  } catch (error) {
      console.error('Error:', error);
      document.getElementById('verificationResult').innerText = 'Error: ' + error.message;
  }
```

3-Technical Stack :

- **Solidity**: For writing smart contracts deployed on the Ethereum blockchain.
- **Web3.js**: For interacting with Ethereum nodes from the web interface.
- **MetaMask**: For enabling users to approve transactions and connect to the Ethereum blockchain.
- **HTML/JavaScript**: For building the user interface.

4-Deployment:

Smart Contracts via Remix IDE

1. **Development**: Use Remix IDE, a powerful open-source tool for writing Solidity contracts in a browser-based IDE, to develop my smart contracts (**WeddingCertificate** and potentially **WeddingCertificateFactory**).
2. **Compilation**: Compile the contracts within Remix IDE, which provides a user-friendly interface for compiling Solidity code, managing compiler versions, and optimizing the output.
3. **Deployment to Testnet**:
    - Connect Remix IDE to the Goerli testnet by selecting the "Injected Web3" environment in the "Deploy & Run Transactions" plugin. This requires having a web3 provider like MetaMask installed and configured for the Goerli testnet.
    - Deploy the contracts using Remix, which will prompt a transaction in MetaMask. Confirm the transaction to deploy the contract.
    - Once deployed, Remix provides the contract address and ABI, which are needed for the web interface to interact with the contract.

Web Interface with **server.js**

1. **Server Setup**: my **server.js** file indicates the use of Express.js to serve static files from a **public** directory. Ensure my web interface files (**index.html**, **app.js**, etc.) are correctly placed in this directory.
2. **Running the Server**: Execute **node server.js (or npm start)** to start the Express server. By default, it listens on port 3000, making my web interface accessible at **http://localhost:3000**.
3. **Configuration**: Update **app.js** in my web interface to use the deployed contract addresses and ABIs obtained from Remix IDE.

5-Testing

Smart Contracts :

1. **Remix IDE Testing**: Use Remix's built-in testing features and plugins to write and run tests for my smart contracts. I can simulate transactions and inspect state changes directly within the IDE.
2. **Manual Interaction**: Use Remix to manually interact with the deployed contracts on Goerli, testing functions like **createCertificate** and **emitPartnerDetails** to ensure they behave as expected.

Web Interface :

1. **Local Testing**: With my Express server running, visit **http://localhost:3000** to manually test the web interface. Ensure that form submissions for creating and verifying certificates interact correctly with the smart contracts on Goerli.
2. **Network and Console Logs**: Use browser developer tools to monitor network requests and console logs for any errors or warnings that might indicate issues with the web interface or its interaction with the blockchain.

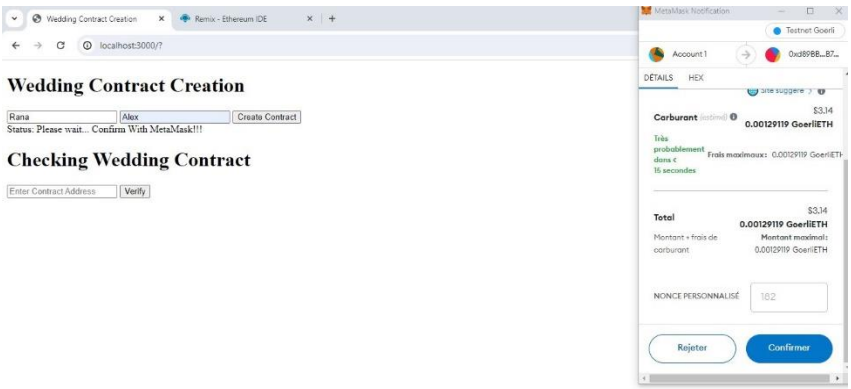
6-Conclusion

The Wedding Contract Project successfully combines blockchain's immutability with a user-friendly interface to create and verify eternal wedding certificates on the Ethereum blockchain. It demonstrates the innovative application of smart contracts for personal milestones, offering a novel way to celebrate and permanently record the union of couples. This project not only showcases the potential of blockchain technology in non-financial domains but also opens up new avenues for its use in personal and legal documentation.

7- illustrative pictures :

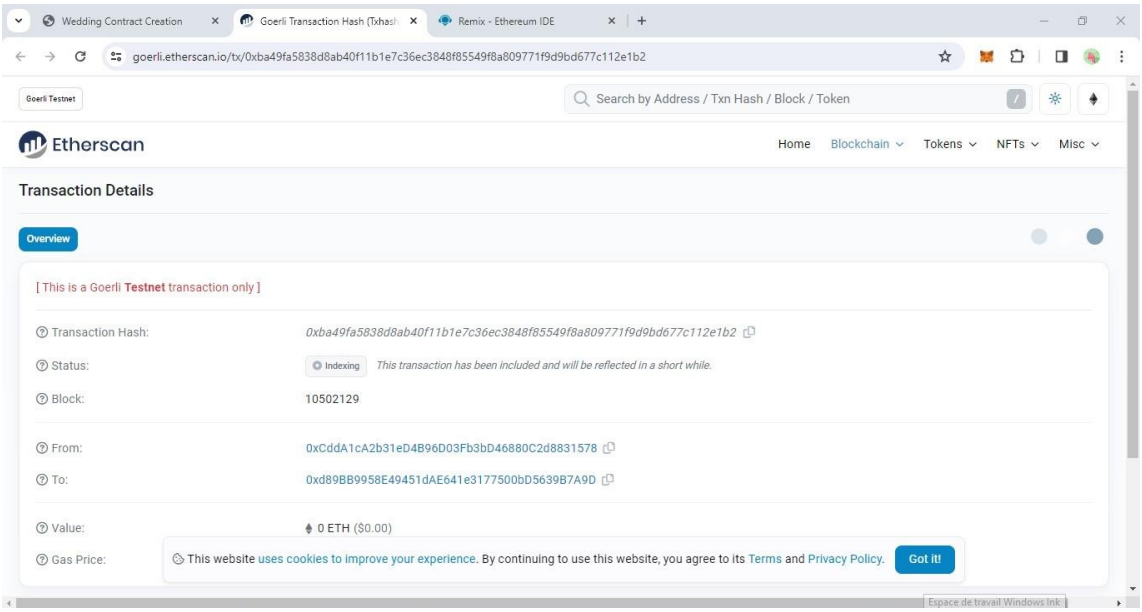
1-My interface :

2-Connecting with MetaMask & Confirming :

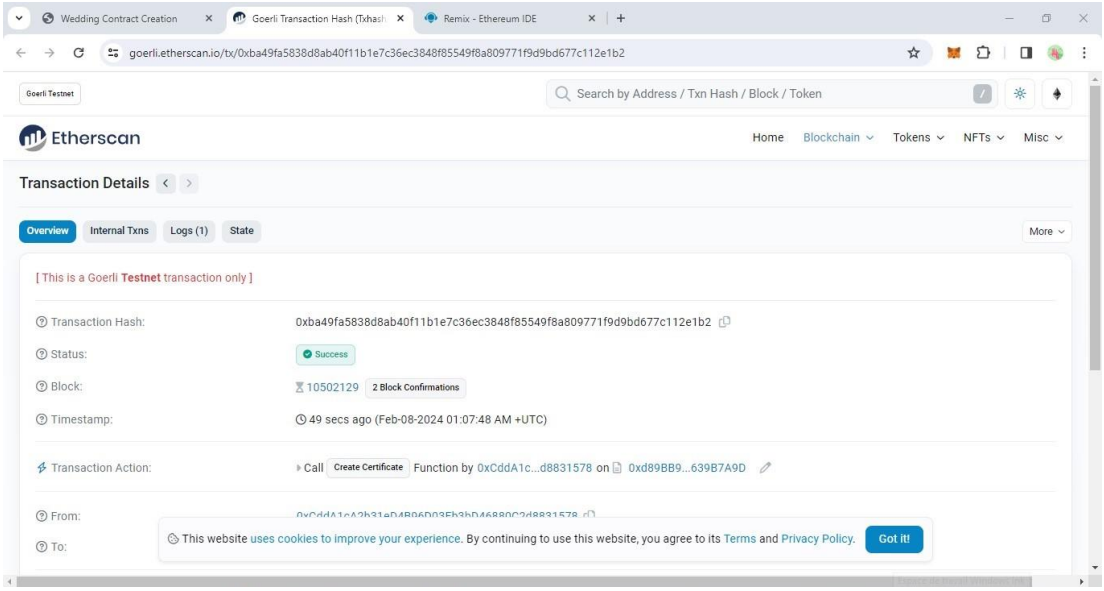


3-Hash tx (as link with Goerly.etherscan.io/tx) & status of tx:

4- Pending Transaction: (@Goerly.etherscan.com/tx/)



5-Succes transaction: (@Goerly.etherscan.com/tx/)



6-Result of Creation:Certificate #,Block#, New Certificate Adress:

7-Checking Wedding Contract; Partners, Block #: