Technical Report SmartContract

Boris Chan Yip Hon Bilel Zaghdoudi

August 2024

Funding

This project has been funded by the DistriTraca project contract no. 2023290944 AID - Sorbonne University.

1 Introduction

1.1 Purpose of the report

The purpose of this report is to provide technical support for Bachelor's and Master's students who have no prior knowledge of smart contracts and blockchain technologies. The aim is for them to be able to draw on this document to acquire the technical skills needed to implement smart contracts independently. This report is designed to be a comprehensive, self-sufficient guide, providing all the knowledge needed to deploy a smart-contract on an EVM (Ethereum Virtual Machine) compatible blockchain.

1.2 Context

This report was written as part of a research project on blockchain technologies. One of the project's objectives was the implementation of smart contracts on EVM-compatible blockchains. The aim of this report is to document the steps and concepts involved in deploying a smart contract, in order to make it easier for undergraduate and graduate students who will be working on this project in the future to get to grips with it.

1.3 Preface

1. Explanation of tools and technical terms
2. Prerequisite installation
   (a) MetaMask (sub sub section 3.1)
   (b) ChainList (sub sub section 3.2)
   (c) Remix (sub sub section 3.3)

(d) VisualStudio Code (sub sub section 3.4)

3. Writing a smart contract
   (a) Methodology (sub section 4)
   (b) Remix (sub section 4.3)
      i. Compilation solidity (sub sub section 4.3)
      ii. Transaction deployment and execution (sub sub section 4.4)
   (c) Action execution order (sub section 4.5)
4. Automate calls using node JS
   (a) Private Key (sub section 5.1)
   (b) ABI (sub section 5.2)
   (c) Contract address (sub section 5.2)
   (d) Node JS code (sub section 5.4)
5. Appendix
   - Narrative game data structure
   - Narrative game smart contract
   - Node JS code

2 Explanation of tools and technical terms

- MetaMask: MetaMask is a browser extension that serves as a digital wallet for managing cryptocurrencies such as Ether. It enables easy interaction with decentralized applications (dApps) on the Ethereum blockchain. In concrete terms, MetaMask can be used to store, send and receive cryptocurrencies. It can also be used to sign transactions to prove that I'm the one authorizing them.
- ChainList: ChainList is a website that helps users easily connect their MetaMask wallet to different blockchains, also known as "networks". Specifically, when a blockchain other than Ethereum is used, information from that blockchain must be manually added to MetaMask. ChainList simplifies this process by providing a list of compatible networks that can be added to MetaMask with a single click. It's a handy tool for quickly accessing multiple blockchains without having to manually enter parameters.
- Remix: Remix is a tool that is widely used to write, test and deploy smart contracts on the Ethereum blockchain. This software, accessible directly via a web browser, enables developers to code in Solidity, the programming language used to create smart contracts. In practice, Remix is used to

write the code for a smart contract, test it by simulating transactions, and finally deploy it on a blockchain. Remix's interface is designed to simplify these steps, offering integrated debugging and simulation tools. As a result, developers can ensure that their code works properly before making it operational on the blockchain.

- Testnet: A testnet is an alternative version of a blockchain used primarily to test applications, smart contracts or updates before they are deployed on the main blockchain, known as the mainnet. On a testnet, " fake " versions of cryptocurrencies are used, enabling developers to experiment without risking the loss of real cryptocurrencies. The testnet is often used to check that everything is working properly, and that any bugs or errors are corrected, before deploying changes on the mainnet.

- RPC: An RPC (Remote Procedure Call) is a method used to enable a program to communicate with a blockchain remotely. In practice, an RPC is used to send commands or requests from an application, such as a wallet or development software, to a node on the blockchain. For example, when a MetaMask user wishes to check his balance or send a transaction, MetaMask sends an RPC request to the blockchain node to obtain the necessary information or execute the transaction. RPC is therefore essential to the interaction between an application and a blockchain.

- Faucet: A faucet is a tool that allows users to receive small amounts of cryptocurrency for free, usually on a testnet. A faucet is mainly used to test applications or smart contracts on a testnet. As the cryptocurrencies on a testnet have no real value, a faucet distributes these fictitious funds so that developers can run tests without risking their own assets. Users can request funds from a faucet, which then sends them to their wallet, enabling them to interact with the test network.

- ABI: The ABI (Application Binary Interface) is an essential element for interacting with a smart contract on a blockchain. The ABI describes the functions and data structures of a smart contract in a standardized way. When an application or user wishes to interact with a smart contract, the ABI is used to understand how to call the contract's functions, what parameters are required, and how to interpret the data returned. In short, the ABI serves as an "instruction manual" that enables other applications to communicate correctly with a smart contract.

Here's how the different concepts are interconnected:

- MetaMask is interconnected with :
  - ChainList: for easily adding new blockchain networks to MetaMask
  - RPC: to enable MetaMask to communicate with different blockchain nodes.

  - Testnet: MetaMask can be configured to interact with testnets to test smart contracts.
  - Faucet: used to obtain cryptocurrencies on a testnet via MetaMask.

- Remix is interconnected with :
  - Testnet: to test the smart contracts developed in Remix before deploying them on the main network.
  - MetaMask: to sign and deploy smart contracts directly from Remix.

- Testnet is interconnected with :
  - Faucet: to obtain test cryptocurrencies for use on the testnet.

- RPC is interconnected with :
  - Remix: for deploying smart contracts via RPC calls to the blockchain.

- ABI is interconnected with :
  - Remix: to generate the ABI of a smart contract that will be used to interact with this contract.
  - MetaMask: to understand and interact with the functions of a deployed smart contract.

The link between MetaMask, RPC, Testnet, Faucet, Remix and ChainList can be explained as follows:

- ChainList and MetaMask : MetaMask is a digital wallet that enables users to manage their cryptocurrencies and interact with decentralized applications on different blockchains. However, to interact with a specific blockchain via MetaMask, one must first add the parameters of that blockchain (such as the RPC node address, network ID, etc.). ChainList, on the other hand, makes this task easier. It provides a list of compatible blockchains that users can add directly to MetaMask with a single click. In this way, ChainList simplifies the process of configuring new networks on MetaMask, enabling the user to quickly access different blockchains without having to manually enter the necessary technical information.

- MetaMask and RPC: MetaMask uses RPCs (Remote Procedure Calls) to communicate with different blockchain nodes. When a user wishes to send a transaction, check a balance or interact with a smart contract via MetaMask, an RPC request is sent to the blockchain node to execute this action. RPC thus enables MetaMask to connect to the blockchain, send instructions and receive data in return.

- MetaMask and Testnet: MetaMask can be configured to connect to a testnet, which is a version of the blockchain used for testing. By using

MetaMask on a testnet, developers and users can experiment with transactions or smart contracts without using real cryptocurrencies. This allows them to check that everything is working properly before deploying on the main blockchain (mainnet).

- MetaMask and Faucet: When a user wishes to test on a testnet via MetaMask, he needs cryptocurrencies specific to this testnet, which have no real value. A faucet is used to obtain these test cryptocurrencies free of charge. The user can request funds via a faucet, which sends these cryptocurrencies directly to his MetaMask wallet, enabling him to test transactions or smart contracts on the testnet.
- RPC and Remix: Remix, as a development environment for smart contracts, also uses RPC to deploy and test contracts on a blockchain. When a developer deploys a smart contract from Remix, an RPC request is sent to a node on the blockchain to register the contract on the network. In addition, actions such as reading or executing a smart contract's functions are also performed via RPC calls.
- Interaction between MetaMask and Remix via RPC: When a developer uses Remix to deploy a smart contract, he may choose to sign the transaction via MetaMask. In this case, Remix generates an RPC call to deploy the contract, and MetaMask steps in to sign and authorize this request before it is sent to the blockchain node. MetaMask thus ensures that actions initiated in Remix are secured and validated by the user via RPC.

3 Prerequisite installation

3.1 MetaMask

MetaMask plugins extension, install directly on browser. Install MetaMask on browser, via an extension (figure 1) :
https://chromewebstore.google.com/detail/metamask/nkbihfbeogaeaoehlefnkodbefgpgknn?hl=fr

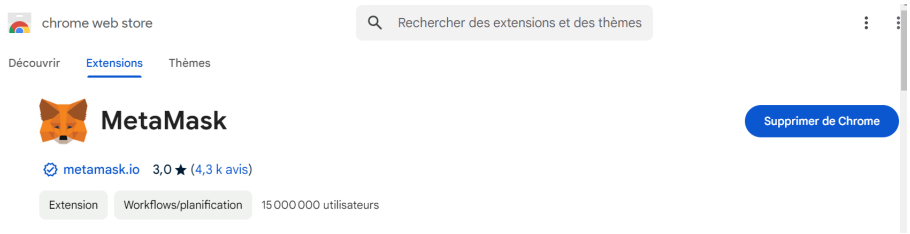

Figure 1: Chrome extension

Note: instead of "Remove from Chrome", it should say "Add to Chrome" (or something similar). (I've already installed MetaMask, hence the removal

message) or use the following link, depending on your media (figure 2).
https://metamask.io/download/

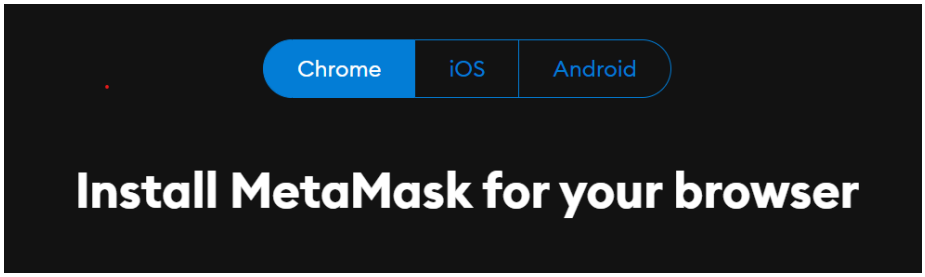

Figure 2: Various installation supports

Once MetaMask has been installed on your browser, it appears by default in the top right-hand corner. MetaMask is represented by a fox head (figure 3).

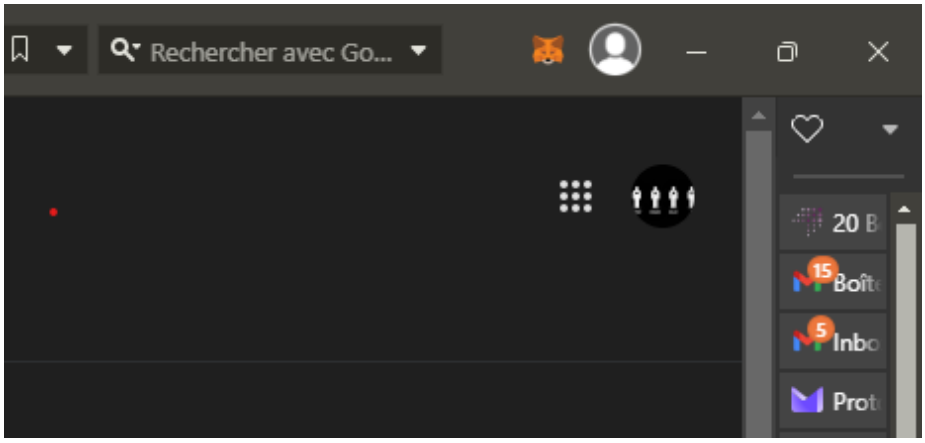

Figure 3: MetaMask Icone

Here's the window, once deployed. Follow the instructions to create an account. Official documentation
https://support.metamask.io/getting-started/getting-started-with-metamask/
or follow the steps to create an account directly from the window above.(figure 4).

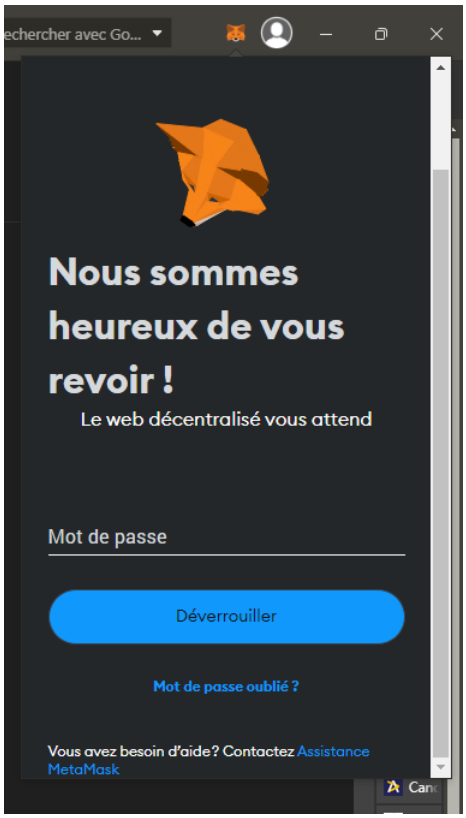

Figure 4: MetaMask window

The MetaMask opening window displays the following information:

- current name of the mainet or testnet
- selected wallet
- current wallet address
- the amount of cryptocurrency in the account
- some features :
  - buy
  - send
  - swap
  - bridge
  - wallet history

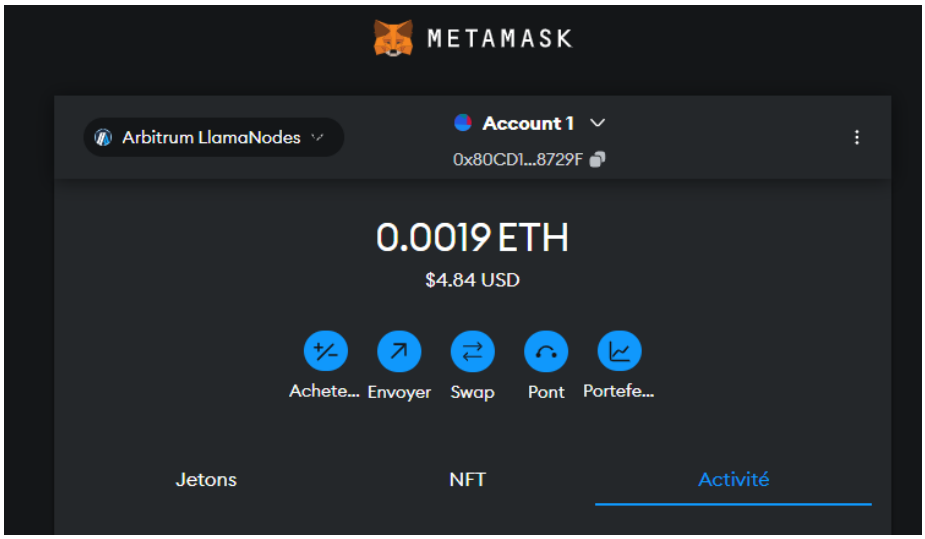

Figure 5: MetaMask page setup

3.2 ChainList

Add blockchains and their respective testnet, if possible have cryptocurrency on it or test swabs. Link to ChainList : https://chainlist.org/

Once on the ChainList page, a search bar allows you to target the blockchains you wish to add to your MetaMask account. (figure 6).

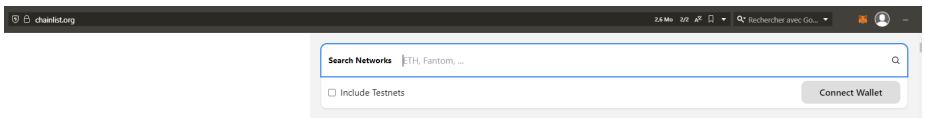

Figure 6: ChainList search bar

Click on “Connect Wallet” to connect your MetaMask account. The MetaMask confirmation windows will appear, allowing you to connect to your MetaMask account.

In the following screenshot, I would like to add a chain from the Arbitrum blockchain to my MetaMask account, if possible, the mainet of the blockchain and the associated testnet. (figure 7). In the present case, where several chains are presented, it is first necessary to research the blockchain you wish to add. This being said, there are several chains because they may be an earlier version

before a merge, or the chains use different consensus algorithms, or a version change or some other reason. In this case, it's necessary to do some research to identify the one you need. Mainets require the blockchain's crypto-currency to be able to carry out transactions on them.

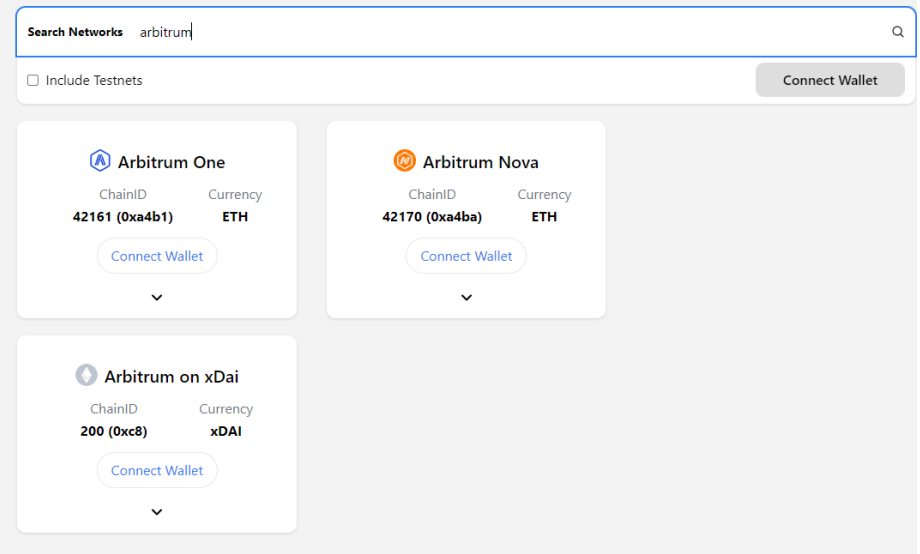

Figure 7: Searching for Arbitrum without Testnets

3.2.1 Testnet

There's a free alternative: use public blockchain test chains, Testnets ,in the following screenshot, I've checked the "Include Testnets" box. Testnets use faucets for testing, not crypto-currencies. Similarly, there are several Testnets available, so you'll need to do some research to find out which mainet a testnet is attached to. (figure 8).

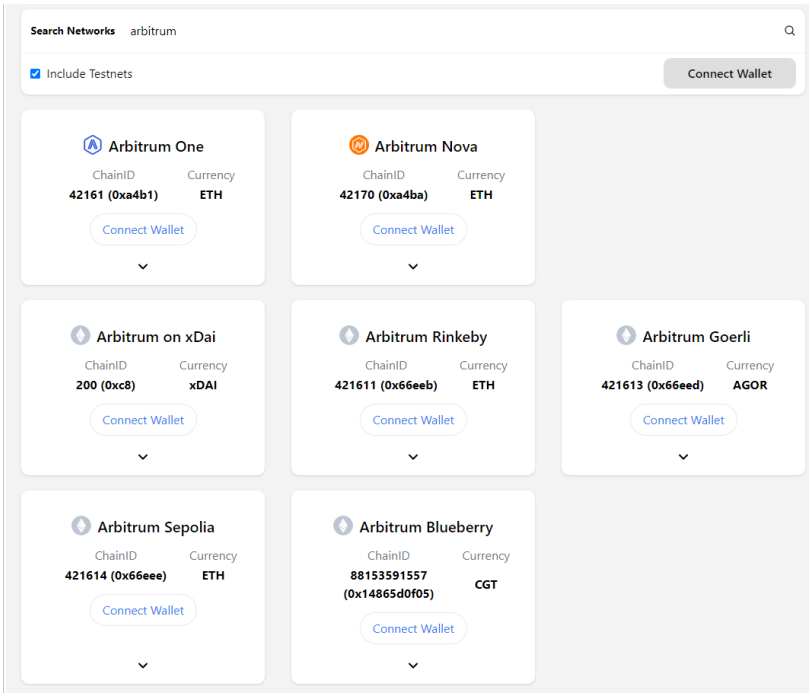

Figure 8: Searching for Arbitrum with Testnets

In both cases, mainets or testnets, click on “Connect Wallet” to add the channel to your MetaMask account. If you have not already done so, you will need to request a connection to MetaMask. A window appears, requesting confirmation of the network change. A window similar to the following screenshot (figure 9). Authorize, to finalize the chain change.

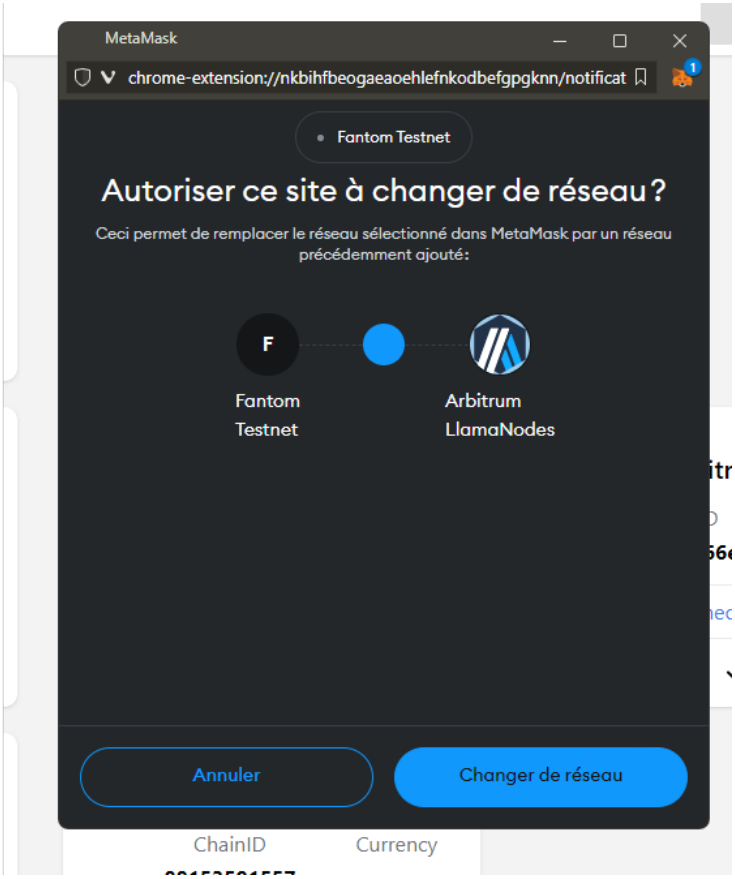

Figure 9: Chain change request

It's also a good idea to keep the RPC you're using and link it to the JS file when calls are automated. The choice of RPC depends on its performance in terms of latency and privacy. To find the RPC, click on the downward-pointing arrow in the square of the desired chaine - for example, on the Arbitrum One chaine.

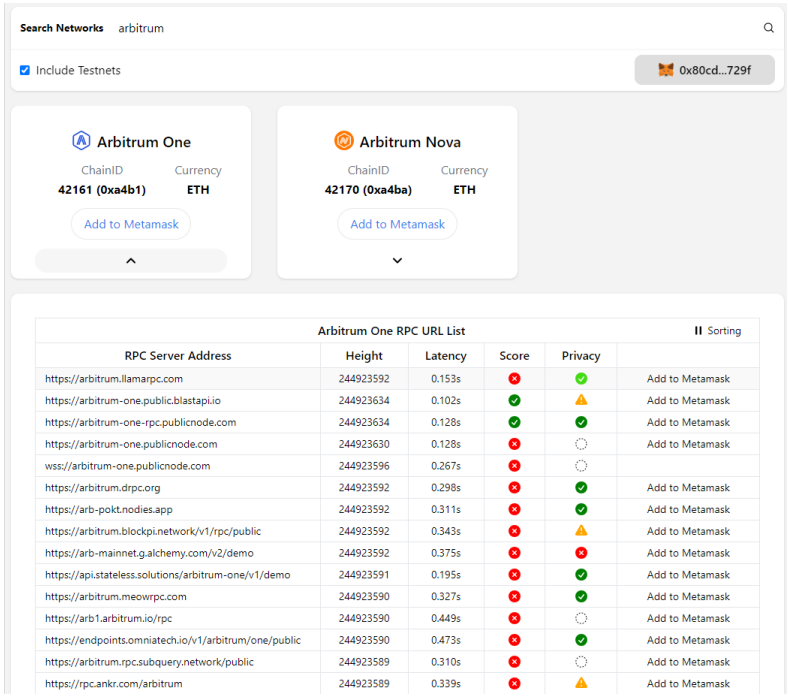

Figure 10: Arbiturm RPC ChainList

The same applies to all other chains accessible on ChainList. Sometimes, not all RPC links are reachable. In this case, use Google or Infura search engines. https://www.infura.io/networks

Please note: you may receive an error message describing that the node cannot be reached when executing the node file on VisualStudioCode. In this case, changing the RPC address may solve the problem. I'll explain where to change the RPC later, in the code.
To change the chain, first make sure that the chain has been added to MetaMask. If this is not the case, go back to the ChainList step. Then simply select another chain from the MetaMask menu. To unfold all the chains added to MetaMask, click on the arrow next to the name of the chain and pointing downwards to unfold the menu.

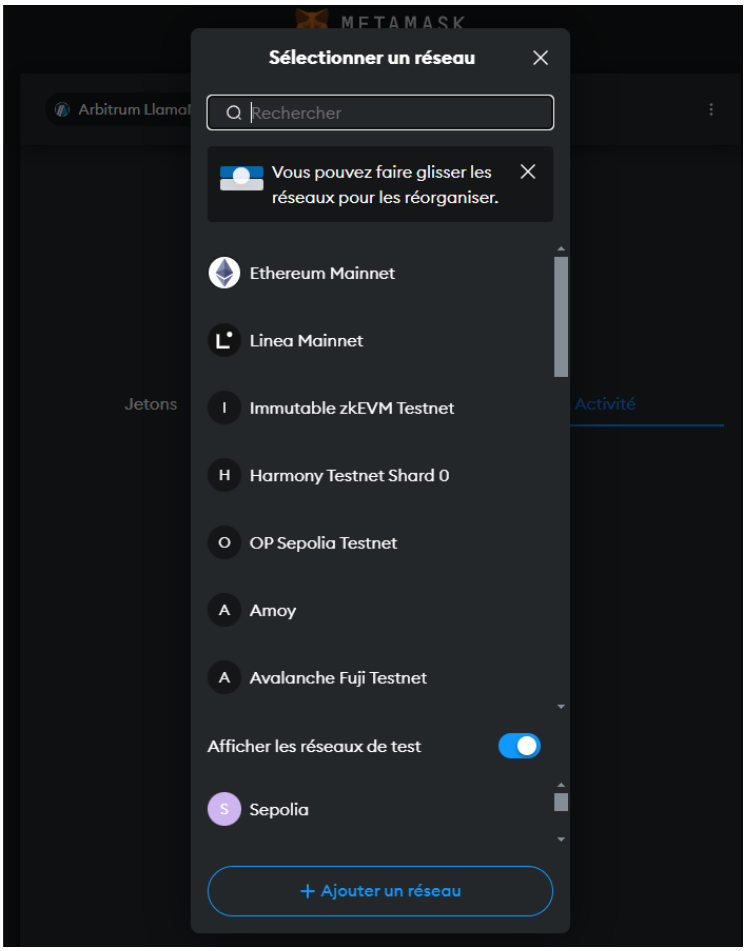

Figure 11: Chaine change

Select the chaine to be used for the substitution, and confirm the MetaMask confirmation windows, a window similar to the one for setting up a chaine on MetaMask.

3.2.2 Faucet

Know your wallet address as it is needed to make faucet requests. Google search: ¡blockchain name¿ faucet
Some useful links, grouping together several chains :
https://www.alchemy.com/faucets/ethereum-sepolia
https://faucet.quicknode.com/drip
https://www.infura.io/faucet/sepolia

https://faucets.chain.link/sepolia

3.3 Remix

Remix offers an interactive environment where you can write and test code in real time. Open your web browser and go to the following link:
https://remix.ethereum.org/
Remix is directly usable, with no need to install libraries. The programming language is Solidity, which is similar in syntax to JavaScript. You can create several dedicated workspaces for different projects, or even all the smart-contracts in the game space. The smart-contracts are in the contracts folder, 'narrative_game_state.sol' is the name of the contract I used for my data structure. (figure 12).

Important: smart-contracts written directly to Remix via the web browser are stored in the browser cache. We strongly recommend that you systematically save them in a separate text file.

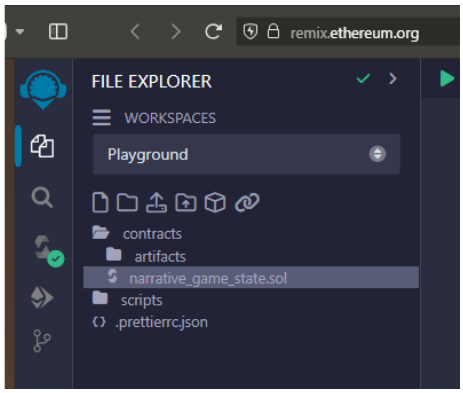

Figure 12: Remix file explorer

To learn Solidity in a more technical and fun way, I recommend following the CryptoZombies tutorials.
https://cryptozombies.io/fr

3.3.1 Compilation

To be warned of any errors during compilation, check the 'Auto-compile' box. For the choice of compiler, the most recent will be chosen by default. It may happen that some chains do not work with a compiler version that is too recent, in which case you need to select an older supportable version or change EVM version of Remix

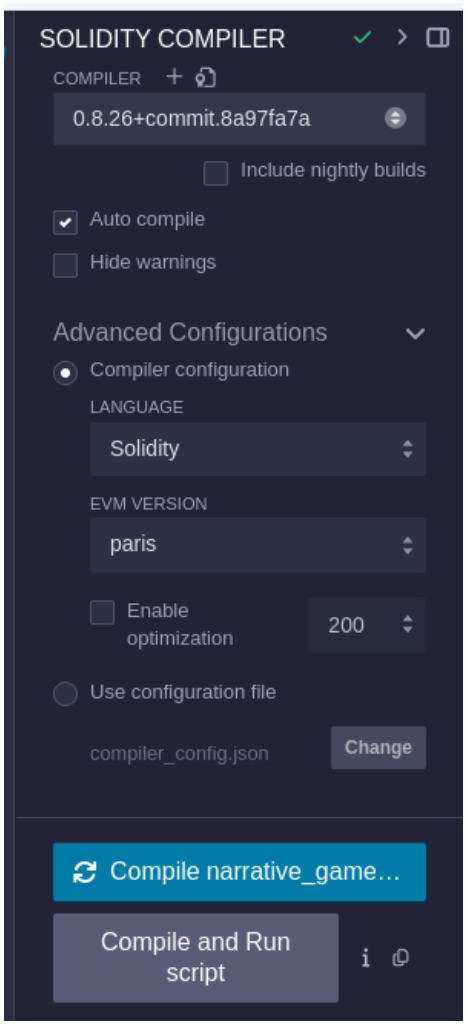

Figure 13: Auto Compile Remix

3.4 Visual Studio Code

VisualStudioCode est efficace et simple de prise en main. https://code.visualstudio.com/ Using an IDE simplifies programming APIs and installing the Node.js and NPM libraries. To install Node.js and npm on your system, here are the specific commands for your operating system:

- For Windows
  - Go to the official Node.js website : https://nodejs.org/
  - Download the recommended LTS (Long Term Support) installer for Windows.

  - Run the downloaded installation program.
  - Follow the instructions on the screen. This will install both Node.js and npm automatically.

- For Linux
  - Open a terminal and run the following commands:

```
$ curl -fsSL https://deb.nodesource.com/setup\_lts.x | sudo -E bash -
$ sudo apt-get install -y nodejs
```

After installation, you can check that everything is working correctly by running the following commands in your terminal:

```
$ node -v
$ npm -v
```

4 Writing a smart contract

4.1 Analysis of the data structure

The first step is to analyse the data on paper. This analysis identifies the elements needed to form the essence of the smart-contract's data structure. This involves understanding the project requirements and clearly defining the variables, data types and functions that will be implemented in the smart-contract.

4.2 Programming

After describing the data structure with the variables and structures on paper (or in a text file), the smart-contract can be coded in Solidity. To code the smart-contract, the simplest way is to launch a request to ChatGPT to obtain the Solidity code associated with the data structure we identified earlier:
" Write the smart-contract associated with the following data structure: "copy and paste the data structure" "
Let's take a smart contract as an example, which takes a name as input and returns "Hello" + name + ", it's a pleasure to meet you !". The data structure is as follows.

```
struct Greeting {
    string name;
}
```

The associated smart contract, generated by ChatGPT.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

```

```
contract Greeting {

    struct Name {
        string name;
    }

    function greet(string memory _name) public pure returns (string
         memory) {
        Name memory person = Name({
            name: _name
        });

        string memory greeting = string(abi.encodePacked("Hello ",
            person.name, ", it's a pleasure to meet you"));

        return greeting;
    }
}
```

Listing 1: Smart contract en Solidity

This code must be structured in such a way as to correctly implement the functionalities defined during the analysis. If this is not the case, modify the code using ChatGPT.

To learn how to translate data structures into smart contracts, it is important to become familiar with several basic concepts in Solidity programming, as well as to acquire a solid understanding of smart contracts and the blockchain in general. Here are some teaching resources that can help you master this skill :
https://cryptozombies.io/
https://solidity-by-example.org/
https://docs.soliditylang.org/

4.3 Compilation

"Injector Provider" is directly linked to the current blockchain used on MetaMask. This is where you specify which chain will be used by remix to deploy the smart-contract. If the chain changes in the meantime, the environment needs to be updated.

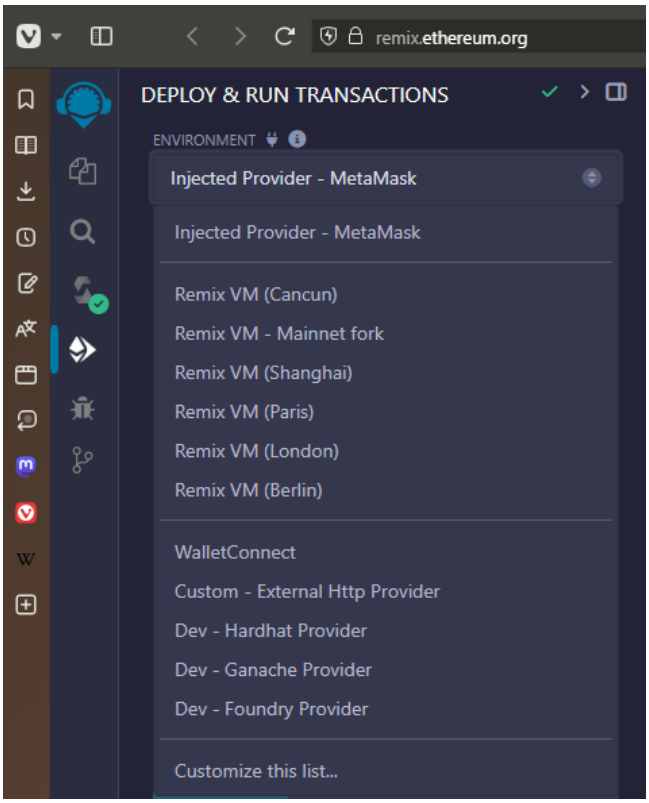

Figure 14: Injector Remix

4.4 Déploiement

Here is a screenshot of the Remix deployment taskbar. You need to check that :

- smart-contract compiles correctly
- the injector provider has selected MetaMask
- the information specified in account is correct: wallet address on MetaMask and the amount of crypto-currency or faucets in the account.

If the information is not correct, update the environment information again. Once checked, click on "Deploy". (figure 15).

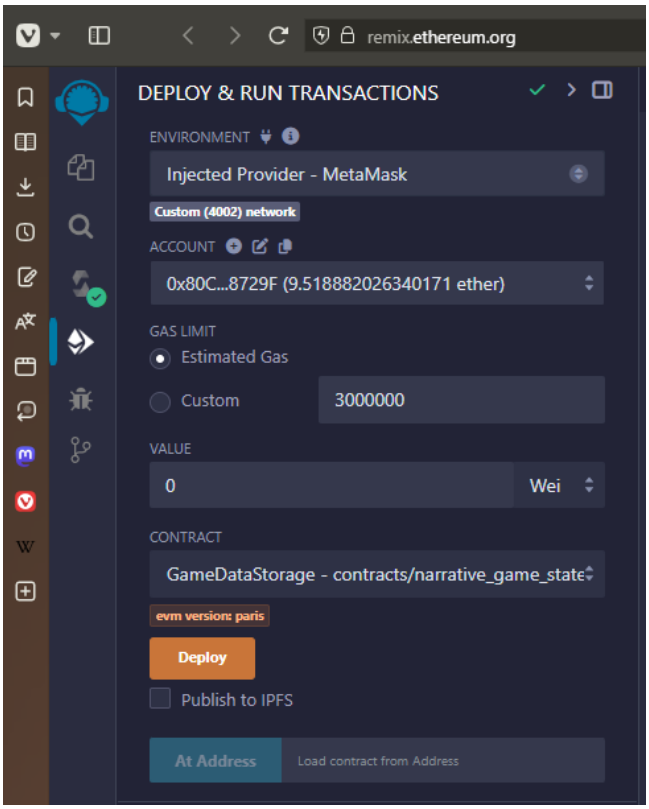

Figure 15: Deploy Remix

A confirmation of the deployment of smart-contract on the chain and at what cost must be validated to approve the deployment. (figure 16).

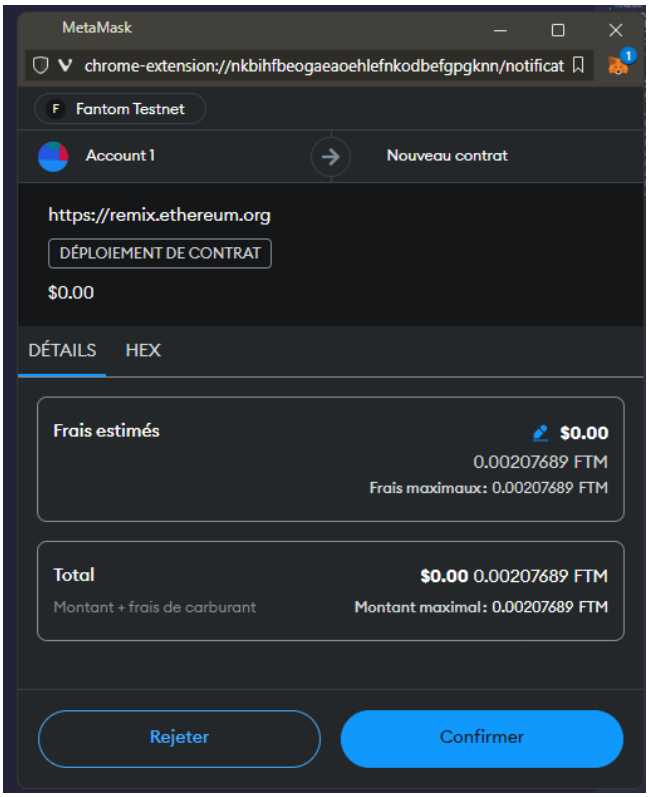

Figure 16: Deploy request Remix

In the Greeting example, Figure 17 describes the response to the smart-contract Greeting deployment request.

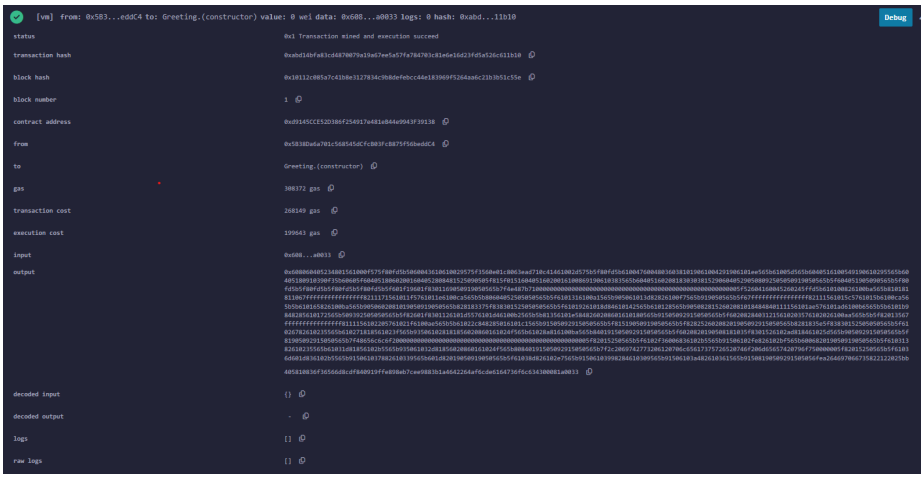

Figure 17: Greeting Deploy

Once the contract has been deployed, it can be interacted with directly via Remix(figure 18). The parameters are entered manually, which is a way of checking that the contract executes correctly before running it on an API.

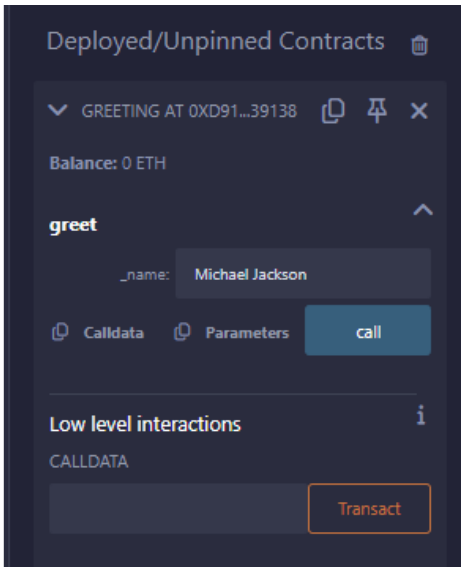

Figure 18: Passing settings manually

In this example, I enter the name "Michael Jackson" as a parameter, then call the smart-contract by clicking on "call". Finally, the output of the smart-

contract (figure 19) :

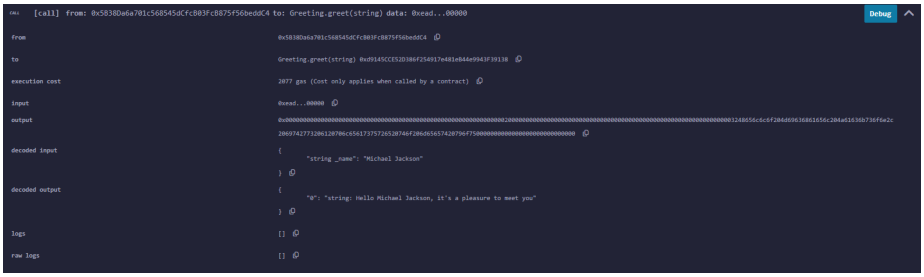

Figure 19: Greeting Micheal Jackson

4.5 Action execution order

To summarise the steps:

1. write the entire data structure to be evaluated (variable, structure, enum, etc) on a sheet of paper
2. code the smart contract yourself using the examples or have ChatGPT code the smart contract associated with the data structure
3. open a Remix window
4. copy/paste the code into the Remix project created for this purpose
5. activate Remix auto-compilation
6. choose the supported compiler (by default, the compiler will be the most recent)
7. log in to your MetaMask account
8. add the chain to be tested to MetaMask via ChainList
9. change chain if necessary, to have the right chain to test
10. set up the right environment, i.e. the MetaMask injector, or choose one of the environments proposed locally to run the tests
11. deployment of smart-contract
12. interact with the smart contract by entering parameter values to ensure that the smart contract executes as required.

5 Automate calls using node JS

5.1 Private Key

Account details are available in "Account details", after scrolling down the 3-dot bar. The wallet address and private key are included.

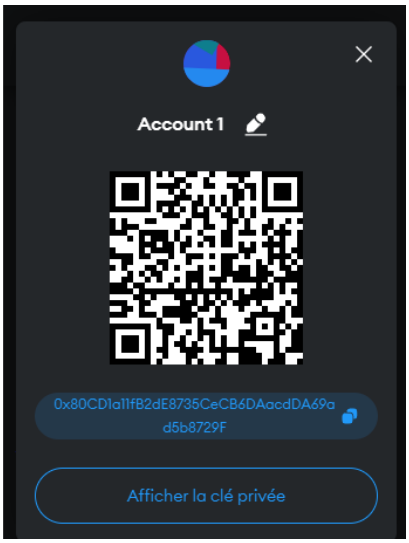

Figure 20: Private Key

Once the smart-contract has been deployed, Remix provides the ABI (Application Binary Interface) and the address of the deployed contract. These elements are crucial for interacting with the smart-contract from external applications. The ABI defines the interface for communicating with the smart-contract, while the address is used to locate the smart-contract on the blockchain.

5.2 ABI

The ABI code is directly available by clicking on 'ABI', a copy of the ABI is made and is ready to be pasted as shown in the figure 21.

To paste, the ABI replaces ["content"] with the copy of the ABI. By default, the copy includes the opening and closing square brackets. Important: a common mistake is not to select the square brackets when pasting the ABI. As a result, the ABI looks like this: [["content"]] which can lead to compilation errors.

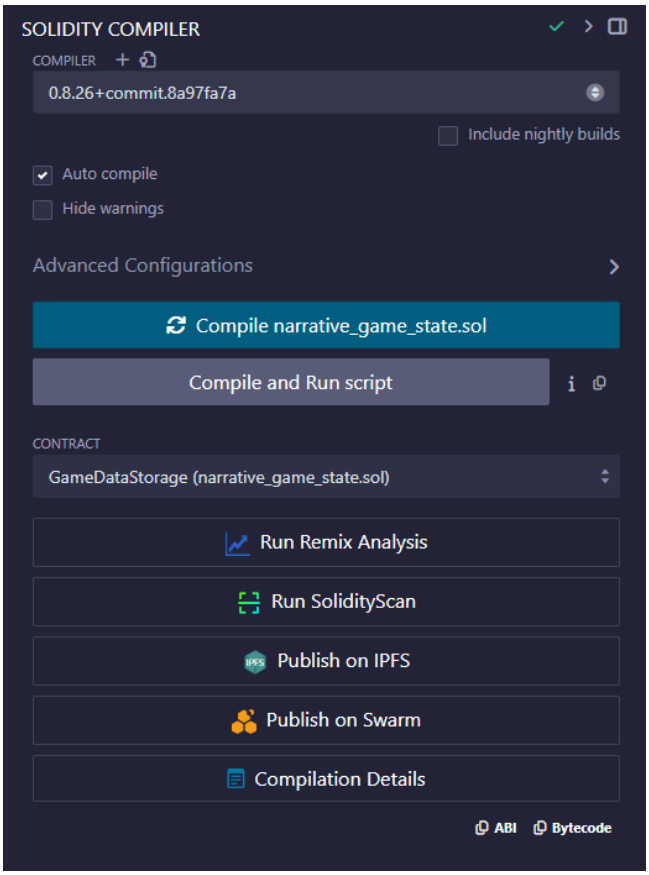

Figure 21: ABI Remix

5.3 Adresse du contract

Deployed smart-contracts are referenced in "Deployed/Unpinned Contracts", where you can copy/paste the addresses.

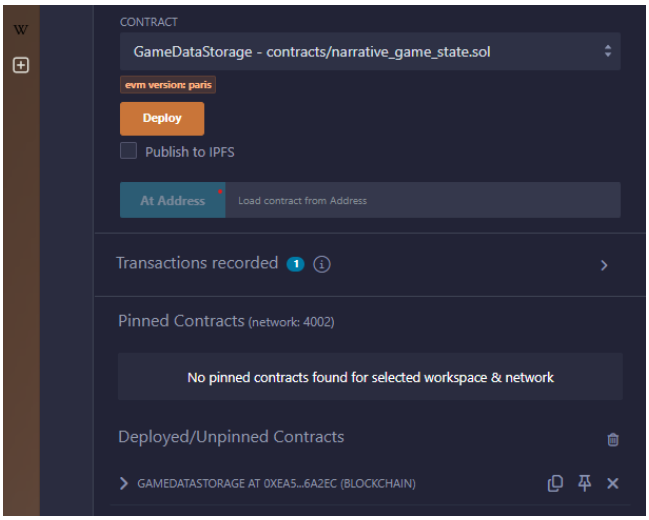

Figure 22: Deployed Unpinned Contracts

5.4 Node JS code

The associated JS code for automating requests is below. The elements required are the private key of the MetaMask account wallet, which will be used for smart-contracts, the RPC address and the address of the smart-contract once deployed. The code in question is a JavaScript script that uses the Web3 library to interact with a smart contract deployed on an Ethereum-compatible blockchain. The script performs several actions, including signing messages, sending transactions to a smart contract and verifying signatures. Here is a detailed explanation of the code:

```
const { Web3 } = require("web3");
const fs = require("fs");
const util = require("util");
```

Listing 2: Code JS utilisé pour les rêquetes

Importing modules

- ‘web3‘: Library used to interact with the Ethereum blockchain.
- ‘fs‘: File system module for reading and writing files.
- ‘util‘: Utility module for formatting outputs.

```
// Initialize Web3
const web3 = new Web3( "put the RPC link here");
```

Initialising Web3

- The script initialises a Web3 object by connecting to an RPC (Remote Procedure Call) provider on the blockchain. You need to replace 'put the RPC link here' with the URL of the RPC provider (such as Infura, Alchemy, etc.).

```
// Your account"s private key (Ensure to keep it secure)
const privateKey = "mettre sa clee privee ici";
const account = web3.eth.accounts.privateKeyToAccount(privateKey);
web3.eth.accounts.wallet.add(account);
web3.eth.defaultAccount = account.address;
```

Managing accounts

- The user's account is recovered from a private key that is converted into a Web3 account. The private key must be secure and never exposed to the public.
- The account is then added to the Web3 wallet so that it can sign transactions, and is set as the default account.

```

// ABI of your smart contract
const contractABI = [ "copy/paste the ABI here, be careful with the
    []. "];

// The address of your deployed contract
const contractAddress = "put the contract address here";
```

Smart Contract

- The smart contract's ABI (Application Binary Interface) is a JSON array which describes the contract's interface (its functions, events, etc.). It should be copied instead of "copy/paste the ABI here, watch out for the []".
- The address of the deployed contract must be specified instead of "put the contract address here".

```
const contract = new web3.eth.Contract(contractABI, contractAddress
    );
```

Create Ethereum Smart Contract Instance:

- "web3.eth.Contract" is a part of "web3.js", which is a library used to interact with the Ethereum blockchain.
- "contractABI": The ABI (Application Binary Interface) defines the methods and events of the Ethereum smart contract.
- "contractAddress": The address of the deployed smart contract on the blockchain.

- This line creates a contract instance, which allows you to interact with the smart contract's functions, events, and data.

```
const logFile = fs.createWriteStream("name_file.log", { flags: "a"
    });
const logStdout = process.stdout;
```

Logging to File and Console:

- "fs.createWriteStream": Opens a file called "name_file.log" to write logs to. The " flags: "a" " means the file is opened in append mode (so new logs are added to the end of the file, rather than overwriting it).
- "process.stdout": This is the default output (usually the terminal) for "console.log"

```
console.log = function () {
    logFile.write(util.format.apply(null, arguments) + "\n");
    logStdout.write(util.format.apply(null, arguments) + "\n");
}
```

This redefines the behavior of 'console.log'

- "util.format.apply(null, arguments)": Takes the arguments passed to "console.log" and formats them as a string.
- This formatted string is written to both the log file ("logFile.write(...)") and the terminal ("logStdout.write(...)"), so the logs are recorded in both places

```
console.error = function () {
    logFile.write(util.format.apply(null, arguments) + "\n");
    logStdout.write(util.format.apply(null, arguments) + "\n");
}
```

Similarly, 'console.error' is redefined in the same way as 'console.log', ensuring that errors are also logged to both the file and the terminal.

```
async function createEvents() {
    for (let i = 0; i < 100; i++) {
```

The "createEvents" function generates and sends transactions to the smart contract in a loop (100 times in this example).

```

        const UUId = `example-uuid-playerID-${i}`;
        const _ids = {
            player_id: `playerID-${i}`,
            game_id: "OMFG",
            reported_player_id: "no_id_player",
            reported_game_id: "OMFG"
        };

```

```
        const _gamestates = {
            current_chapter: "chapter 1",
            current_level: `level ${i}`,
            current_choices: "a, b ou c ",
            previous_chapter: "chapter 1",
            previous_level: `level ${i - 1}`,
            previous_choices: "a, b ou c ",
            target_chapter: "chapter 1",
            target_level: `level ${i + 1}`,
            target_choices: "a, b ou c "
        };

        const _decisions = {
            orchestration: 3,
            membership: 0,
            verification: 2,
            conflict: 3,
            choice: 4,
            recovery: 5,
            misbehaviour: 6,
            anomalie: 1
        };
```

Simulated data: game data (_ids, _gamestates, _decisions) is generated for each iteration with specific values. This data simulates the state of a player in a game.

```
        let start, end;

        try {
            // Start time measurement for the entire process
            start = Date.now();
```

Time measurement: the time taken for each transaction is measured using the "Date.now()" functions before and after the transaction is sent.

```

            // Generate a message to sign
            const message = web3.utils.soliditySha3(UUId, JSON.
                stringify(_ids), JSON.stringify(_gamestates), JSON.
                stringify(_decisions));
            const signature = await web3.eth.accounts.sign(message,
                 privateKey);
```

Message signature: before sending a transaction, the script generates a message from the game data using "solidity Sha3" and signs it with the user's private key. This signature guarantees the integrity and authenticity of the data.

```
            // Sending the transaction with signature included
            const gasEstimate = await contract.methods.setGameState
                (UUId, _ids, _gamestates, _decisions)
                .estimateGas({ from: web3.eth.defaultAccount });
```

Gas estimate: the script estimates the quantity of gas needed to execute the smart contract's "setGameState" function and adds a small margin (10,000 units of gas).

```
            const result = await contract.methods.setGameState(UUId
                , _ids, _gamestates, _decisions)
                .send({ from: web3.eth.defaultAccount, gas: Number(
                    gasEstimate) + 10000, signature: signature.
                    signature });
```

Transaction sent: the transaction is sent to the smart contract, including the signature. The result of the transaction is logged.

```
            console.log(`Transaction ${i + 1} successful:`, result)
                ;

            // Verify the signature after the transaction
            const isValid = await verifySignature(UUId, _ids,
                _gamestates, _decisions, signature.signature);
            console.log(`Signature verification result: ${isValid ?
                 "Valid" : "Invalid"}`);
```

Signature verification: after the transaction has been sent, the script checks that the signature is valid by comparing it with the address of the account that signed the message.

```
            // End time measurement for the entire process
            end = Date.now();
            console.log("Total Time for Transaction, Signature, and
                 Verification:", end - start, "ms");
```

Time measurement: the time taken for each transaction is measured using the "Date.now()" functions before and after the transaction is sent.

```
        } catch (error) {
            console.error(`Error in transaction ${i + 1}:`, error);
        }
    }
}
```

Error handling: if an error occurs during the transaction or any other stage, it is captured and logged.

```

// Function to verify signature from the blockchain
async function verifySignature(UUId, _ids, _gamestates, _decisions,
     signature) {
    const message = web3.utils.soliditySha3(UUId, JSON.stringify(
        _ids), JSON.stringify(_gamestates), JSON.stringify(
        _decisions));
    const recoveredAddress = await web3.eth.accounts.recover(
        message, signature);

    return recoveredAddress.toLowerCase() === web3.eth.
        defaultAccount.toLowerCase();
}

createEvents();
```

"VerifySignature" function: this function is used to verify that a specific signature corresponds to the user's address:

- Message recreation: the message is recreated identically to the one used for signing.
- Address recovery: the address that signed the message is recovered from the signature using "web3.eth.accounts.recover".
- Comparison: the recovered address is compared with the default account address to check the validity of the signature.
- Output logging: the script redirects "console.log" and "console.error" output to a log file ("name_file.log") as well as to standard output, allowing you to keep track of events in a file.

This script is typically used to automate the sending of transactions on a smart contract by simulating gaming events. It also includes mechanisms for verifying security via digital signatures. This is an example of integration between off-chain systems (JavaScript, Web3) and smart contracts on a blockchain. To launch the script, type the following command in a VSCode terminal:

```
1    $ node <nom_du_fichier>.js
```

6 Appendix

Structure de donnée utiliser pour modéliser un état de jeu dans un jeu narratif.

```
struct ID {
    string player_id;
    string game_id;
    string reported_player_id;
    string reported_game_id;
    }

struct GameState {
    string current_chapter;
    string current_level;
    string current_choices;

    string previous_chapter;
    string previous_level;
    string previous_choices;

    string target_chapter;
    string target_level;
    string target_choices;
    }

    enum Orchestration { Created, Locked, Inactive, Writing,
        Reading, Abort, Drop }
    enum Membership { Created, Locked, Inactive, Writing, Reading,
        Abort, Drop }
    enum Verification { Created, Locked, Inactive, Writing, Reading
        , Abort, Drop }
    enum Conflict { Created, Locked, Inactive, Writing, Reading,
        Abort, Drop }
    enum Choice { Created, Locked, Inactive, Writing, Reading,
        Abort, Drop }
    enum Recovery { Created, Locked, Inactive, Writing, Reading,
        Abort, Drop }
    enum Misbehaviour { Created, Locked, Inactive, Writing, Reading
        , Abort, Drop }
    enum Anomalie { None, Critical, Major, Minor }

struct Decision {
    Orchestration orchestration;
    Membership membership;
    Verification verification;
    Conflict conflict;
    Choice choice;
    Recovery recovery;
    Misbehaviour misbehaviour;
    Anomalie anomalie;
    }

struct InitializeGameState {
    ID ids;
    GameState gameStates;
    Decision decisions;
    }
```

Listing 3: Structure de donnée

Le smart-contract utilisé pour la sauvegarde d"un état de jeu.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract GameDataStorage {
    struct ID {
        string player_id;
        string game_id;
        string reported_player_id;
        string reported_game_id;
    }

    struct GameState {
        string current_chapter;
        string current_level;
        string current_choices;

        string previous_chapter;
        string previous_level;
        string previous_choices;

        string target_chapter;
        string target_level;
        string target_choices;
    }

    enum Orchestration { Created, Locked, Inactive, Writing,
        Reading, Abort, Drop }
    enum Membership { Created, Locked, Inactive, Writing, Reading,
        Abort, Drop }
    enum Verification { Created, Locked, Inactive, Writing, Reading
        , Abort, Drop }
    enum Conflict { Created, Locked, Inactive, Writing, Reading,
        Abort, Drop }
    enum Choice { Created, Locked, Inactive, Writing, Reading,
        Abort, Drop }
    enum Recovery { Created, Locked, Inactive, Writing, Reading,
        Abort, Drop }
    enum Misbehaviour { Created, Locked, Inactive, Writing, Reading
        , Abort, Drop }
    enum Anomalie { None, Critical, Major, Minor }

    struct Decision {
        Orchestration orchestration;
        Membership membership;
        Verification verification;
        Conflict conflict;
        Choice choice;
        Recovery recovery;
        Misbehaviour misbehaviour;
        Anomalie anomalie;
    }

```

```
    struct InitializeGameState {
        ID ids;
        GameState gameStates;
        Decision decisions;
    }

    // Mapping to store InitializeGameState by player_id
    mapping(string => InitializeGameState) public
        initializeGameStates;

    function setGameState(
        string memory player_id,
        ID memory _ids,
        GameState memory _gameStates,
        Decision memory _decisions
    ) public {
        InitializeGameState memory newGameState =
            InitializeGameState(
            _ids,
            _gameStates,
            _decisions
        );
        initializeGameStates[player_id] = newGameState;
    }

    function setOrchestrationState(string memory player_id,
        Orchestration state) public {
        initializeGameStates[player_id].decisions.orchestration =
            state;
    }

    function setMembershipState(string memory player_id, Membership
         state) public {
        initializeGameStates[player_id].decisions.membership =
            state;
    }

    function setVerificationState(string memory player_id,
        Verification state) public {
        initializeGameStates[player_id].decisions.verification =
            state;
    }

    function setConflictState(string memory player_id, Conflict
        state) public {
        initializeGameStates[player_id].decisions.conflict = state;
    }

    function setChoiceState(string memory player_id, Choice state)
        public {
        initializeGameStates[player_id].decisions.choice = state;
    }

    function setRecoveryState(string memory player_id, Recovery
        state) public {
        initializeGameStates[player_id].decisions.recovery = state;
    }
```

```
92
93      function setMisbehaviourState(string memory player_id,
            Misbehaviour state) public {
94          initializeGameStates[player_id].decisions.misbehaviour =
                state;
95      }
96
97      function setAnomalieState(string memory player_id, Anomalie
            state) public {
98          initializeGameStates[player_id].decisions.anomalie = state;
99      }
100 }
```

Listing 4: Smart contract en Solidity

```
1  const { Web3 } = require('web3');
2  const fs = require('fs');
3  const util = require('util');
4
5  // Initialize Web3
6  const web3 = new Web3('addresse RPC ici ');
7
8  // Your account's private key (Ensure to keep it secure)
9  const privateKey = 'clee privee  ici ';
10 const account = web3.eth.accounts.privateKeyToAccount(privateKey);
11 web3.eth.accounts.wallet.add(account);
12 web3.eth.defaultAccount = account.address;
13
14 // ABI of your smart contract
15 const contractABI = ['ABI a coller ici ici'];
16
17 // The address of your deployed contract
18 const contractAddress = 'addresse du contract a mettre ici';
19
20 // Create contract instance
21 const contract = new web3.eth.Contract(contractABI, contractAddress
       );
22
23 const logFile = fs.createWriteStream('transactions_01.log', { flags
       : 'a' });
24 const logStdout = process.stdout;
25
26 console.log = function () {
27     logFile.write(util.format.apply(null, arguments) + '\n');
28     logStdout.write(util.format.apply(null, arguments) + '\n');
29 }
30
31 console.error = function () {
32     logFile.write(util.format.apply(null, arguments) + '\n');
33     logStdout.write(util.format.apply(null, arguments) + '\n');
34 }
35
36 async function createEvents() {
37     for (let i = 0; i < 100; i++) {
38         const UUId = `example-uuid-playerID-${i}`;
39         const _ids = {
40             player_id: `playerID-${i}`,
```

```
            game_id: 'OMFG',
            reported_player_id: 'no_id_player',
            reported_game_id: 'OMFG'
        };

        const _gamestates = {
            current_chapter: 'chapter 1',
            current_level: `level ${i}`,
            current_choices: 'a, b ou c ',
            previous_chapter: 'chapter 1',
            previous_level: `level ${i - 1}`,
            previous_choices: 'a, b ou c ',
            target_chapter: 'chapter 1',
            target_level: `level ${i + 1}`,
            target_choices: 'a, b ou c '
        };

        const _decisions = {
            orchestration: 3,
            membership: 0,
            verification: 2,
            conflict: 3,
            choice: 4,
            recovery: 5,
            misbehaviour: 6,
            anomalie: 1
        };

        let start, end;

        try {
            // Start time measurement for the entire process
            start = Date.now();

            // Sending the transaction without signature included
            const gasEstimate = await contract.methods.setGameState
                (UUId, _ids, _gamestates, _decisions)
                .estimateGas({ from: web3.eth.defaultAccount });

            const result = await contract.methods.setGameState(UUId
                , _ids, _gamestates, _decisions)
                .send({ from: web3.eth.defaultAccount, gas: Number(
                    gasEstimate) + 10000 });

            console.log(`Transaction ${i + 1} successful:`, result)
                ;

            // End time measurement for the entire process
            end = Date.now();
            console.log('EventStorageTime', end - start);

        } catch (error) {
            console.error(`Error in transaction ${i + 1}:`, error);
        }
    }
}
```

```
createEvents();
```

Listing 5: Node JS code