Overview

Pre course

Pre course - recap : Slide 27

Dear Participants

Please make sure to accurately complete steps 1 and 2 before our training program commences on July 26th. If you have any queries or issues, feel free to reach out to support@settlemint.com.

Congratulations 🥳🎉 You have been assigned to team **{ Cloud provider - Region }**

We're thrilled to welcome you to our Blockchain Training Program.This is a fantastic opportunity to elevate your skills amidst the exponential growth of blockchain adoption across various industries.

Before you delve into the training program, we've created some important steps for you to get you started.

Getting Started:

Step 1: Sign Up

Please refer to the attached PDF named "Registration" to complete your registration following the provided guide. It is important that you fill out every detail correctly to avoid any errors.

Step 2: Prepare Your Environment

As part of this training program, you have been assigned to teams. Each team is associated with a specific cloud provider, on which all your blockchain services will be deployed.

You have been selected for Team **{ Cloud provider - Region }**, which means you will deploy all your

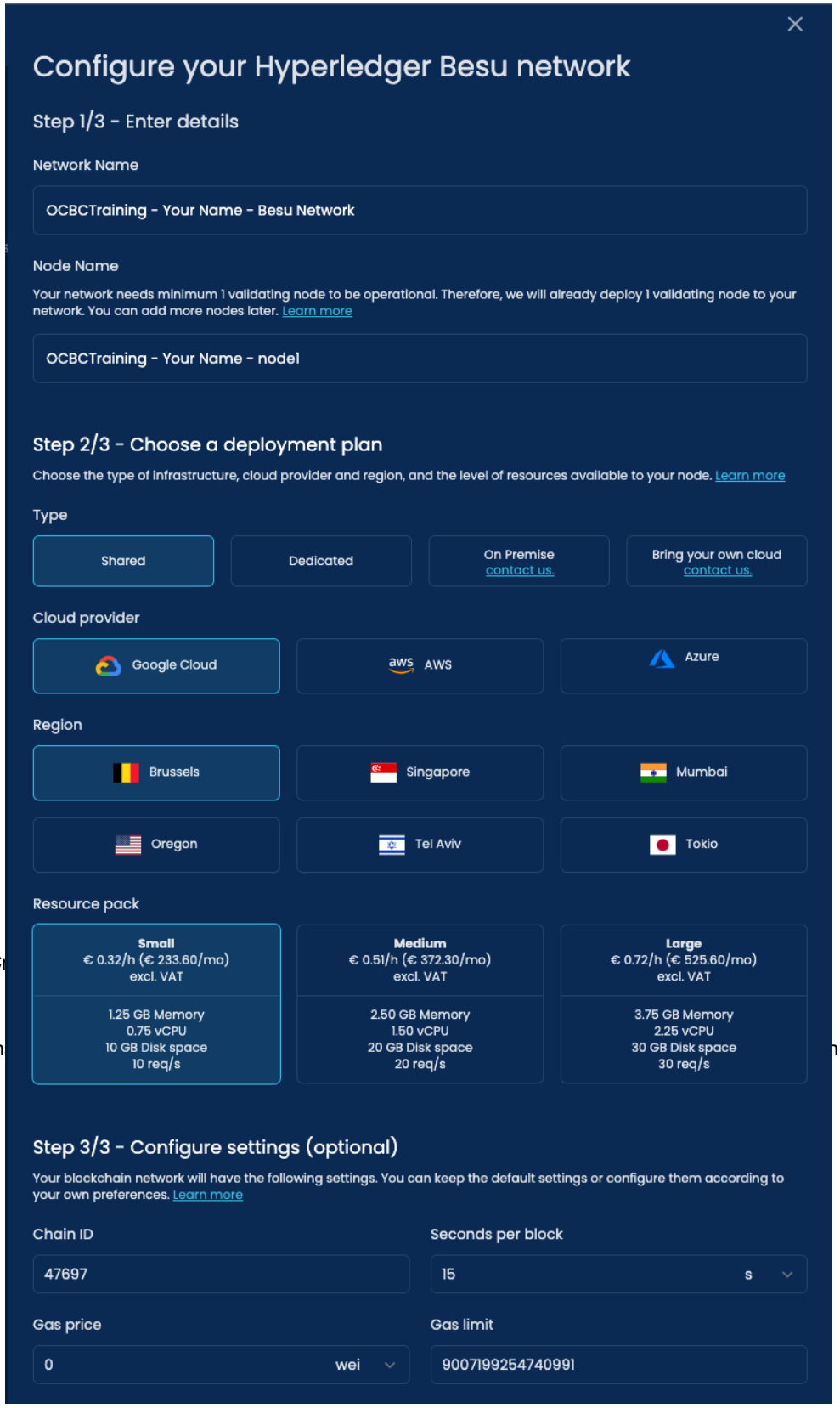
Every service that you deploy Must have the prefix "OCBCTraining"

services using the following deployment plan (refer to step 3/2 in the image below):

Network name: **OCBCTraining-{your name}-Besu Network**
Node name: **OCBCTraining-{your name}-Node1**
Type: **Shared**
Cloud Provider: **{Cloud provider }**
Region: **{ Region }**

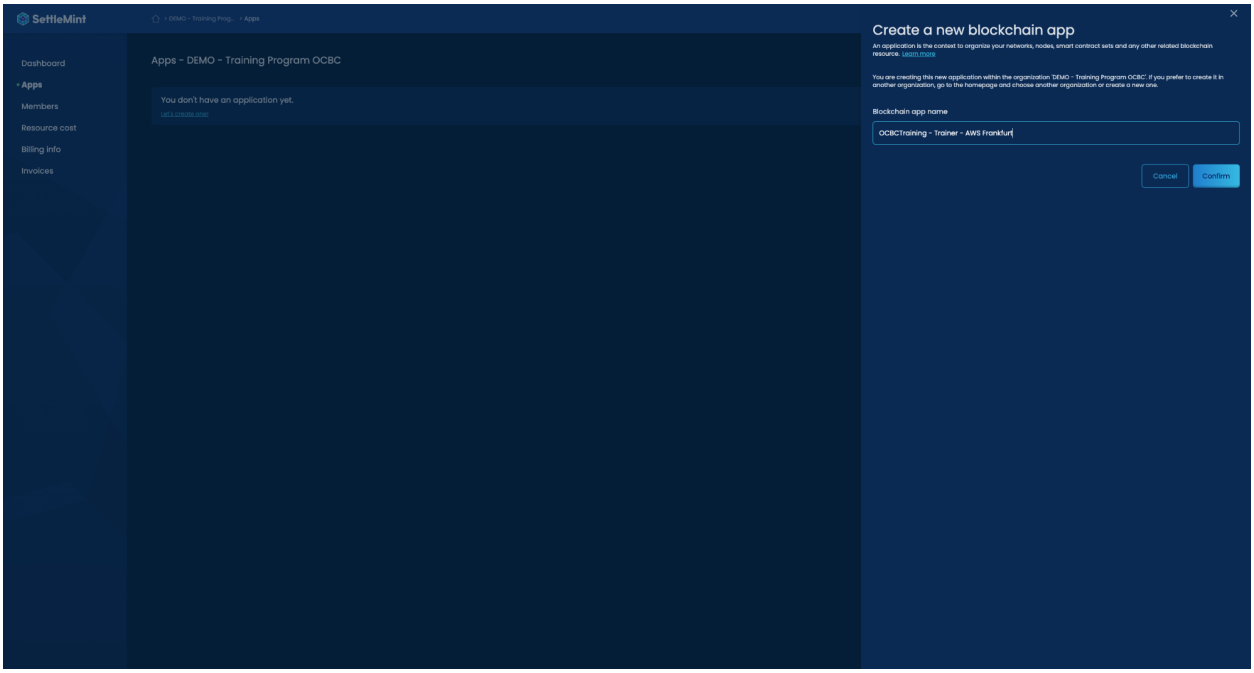
1. Cr

In n.

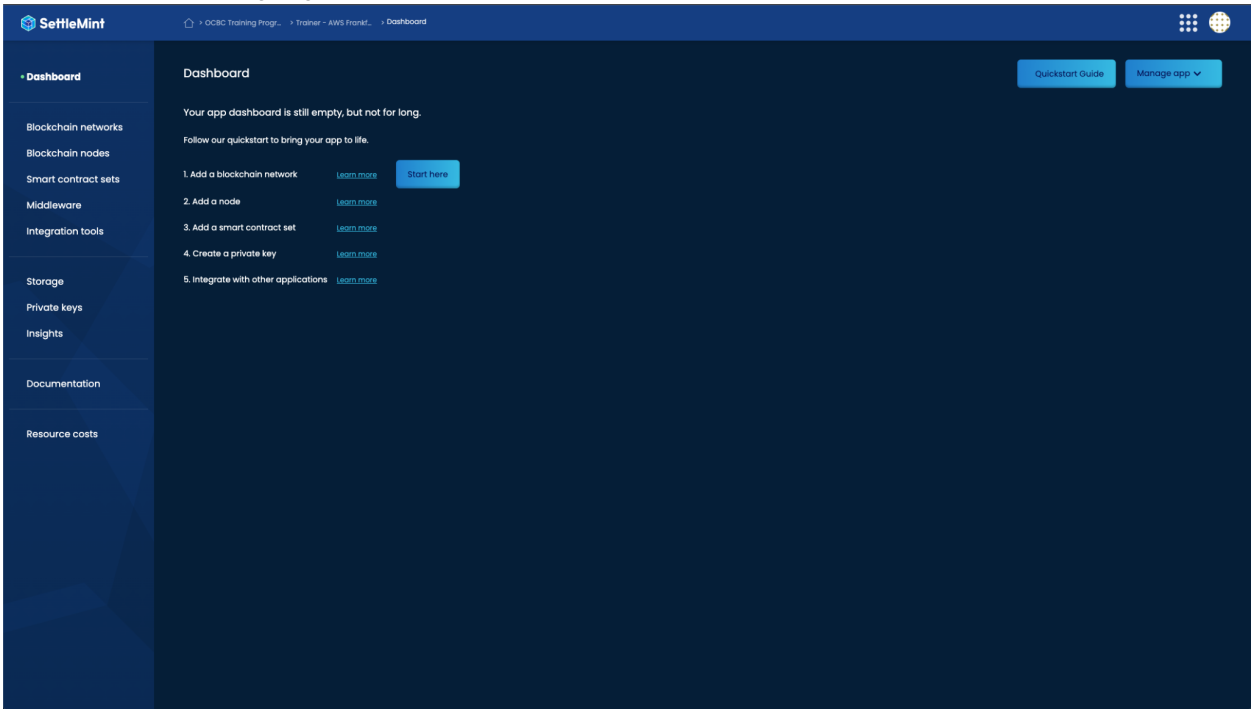

Configure your Hyperledger Besu network
Step 1/3 - Enter details
Network Name
OCBCTraining - Your Name - Besu Network
Node Name
Your network needs minimum 1 validating node to be operational. Therefore, we will already deploy 1 validating node to your network. You can add more nodes later. Learn more
OCBCTraining - Your Name - node1
Step 2/3 - Choose a deployment plan
Choose the type of infrastructure, cloud provider and region, and the level of resources available to your node. Learn more
Type
Shared
Dedicated
On Premise contact us.
Bring your own cloud contact us.
Cloud provider
Google Cloud
AWS
Azure
Region
Brussels
Singapore
Mumbai
Oregon
Tel Aviv
Tokio
Resource pack
Small
€ 0.32/h (€ 233.60/mo)
excl. VAT
1.25 GB Memory
0.75 vCPU
10 GB Disk space
10 req/s
Medium
€ 0.51/h (€ 372.30/mo)
excl. VAT
2.50 GB Memory
1.50 vCPU
20 GB Disk space
20 req/s
Large
€ 0.72/h (€ 525.60/mo)
excl. VAT
3.75 GB Memory
2.25 vCPU
30 GB Disk space
30 req/s
Step 3/3 - Configure settings (optional)
Your blockchain network will have the following settings. You can keep the default settings or configure them according to your own preferences. Learn more
Chain ID
47697
Seconds per block
15
s
Gas price
0
wei
Gas limit
9007199254740991


SettleMint
> DEMO - Training Prog... > Apps
Dashboard
Apps
Members
Resource cost
Billing info
Invoices
Create a new blockchain app

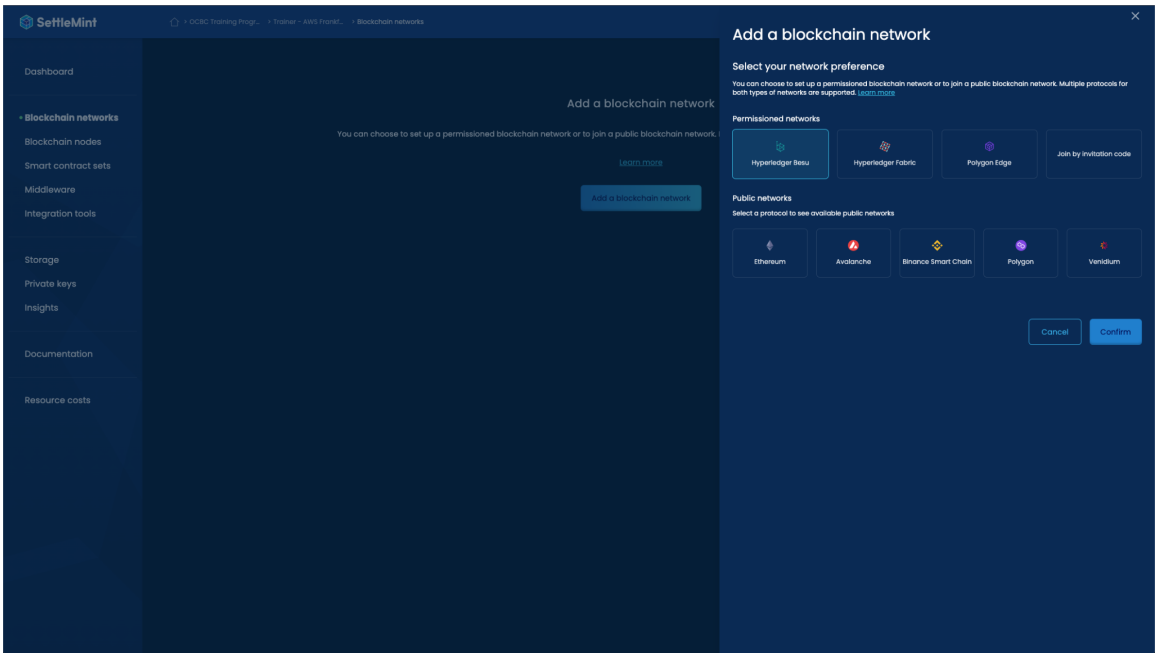
2. Configure your application with the prefix “OCBCTraining” - your name and the assigned cloud provider.



3. You will see the following page



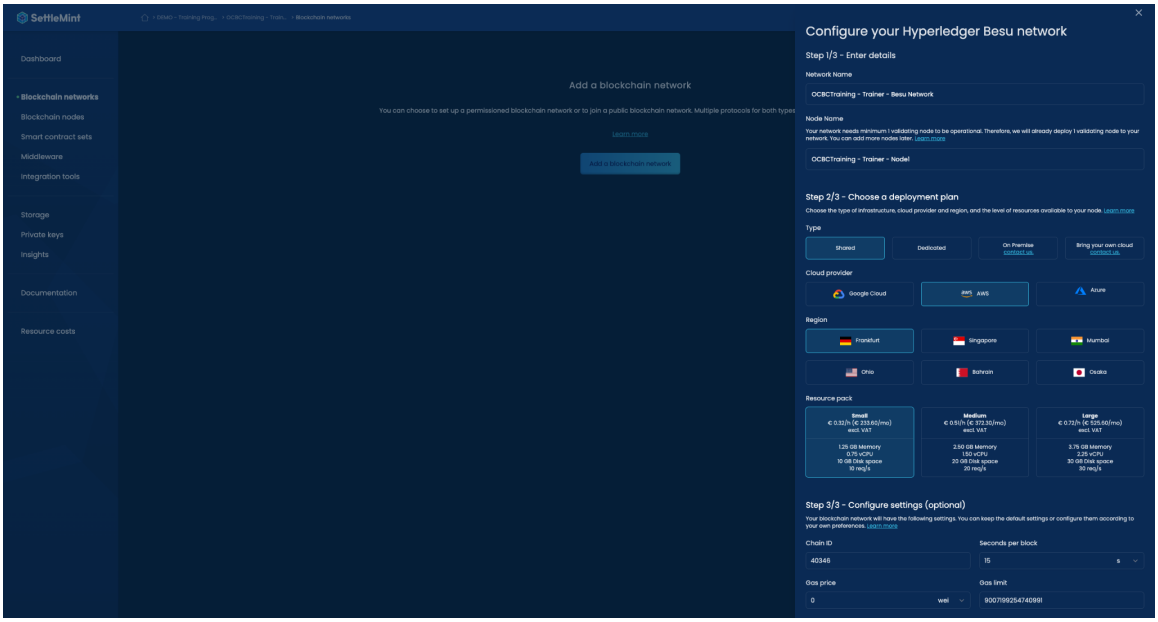
4. From there, navigate to "Blockchain Networks" and click on "Add Blockchain Network". Select "Hyperledger Besu" from the options provided.



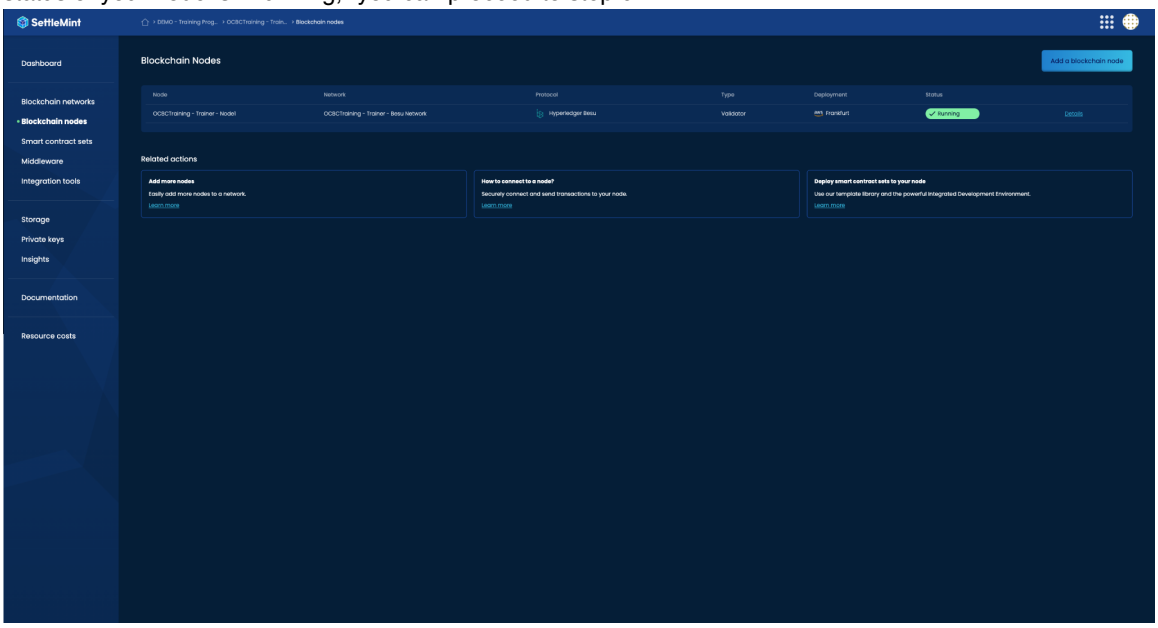
5. Choose the cloud deployment option according to your assigned team. For this example, participant “Trainer” is assigned to "AWS Frankfurt," so he will deploy all the services with the following settings:

    Network name: OCBCTraining - {your name} - Besu Network
    Node name: OCBCTraining - {your name} - Node1
    Type: Shared
    Region: Frankfurt
    Resource Pack: Small
    Cloud Provider: AWS

In the screenshots below, we are using 'AWS-Frankfurt' to deploy. Please use the cloud provider and location you have been assigned. This is found on page 1 and in the email you received. If you have any questions, contact support@settlemint.com



You can now view the blockchain node that you just deployed in the "Blockchain Nodes" tab. Once the status of your node is "Running," you can proceed to step 6.

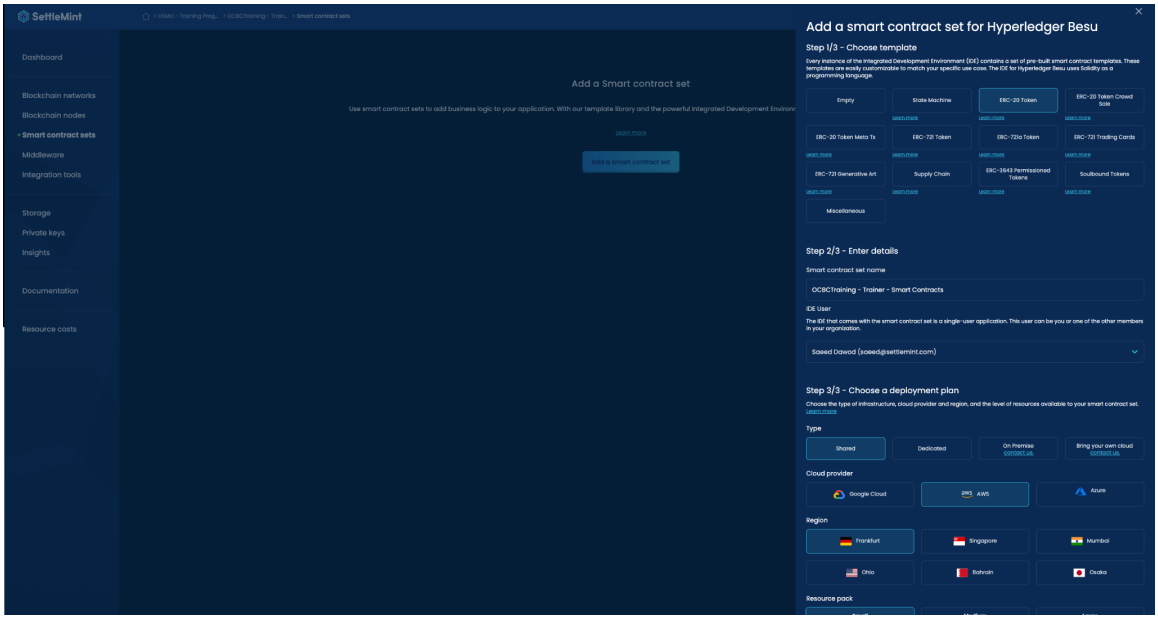


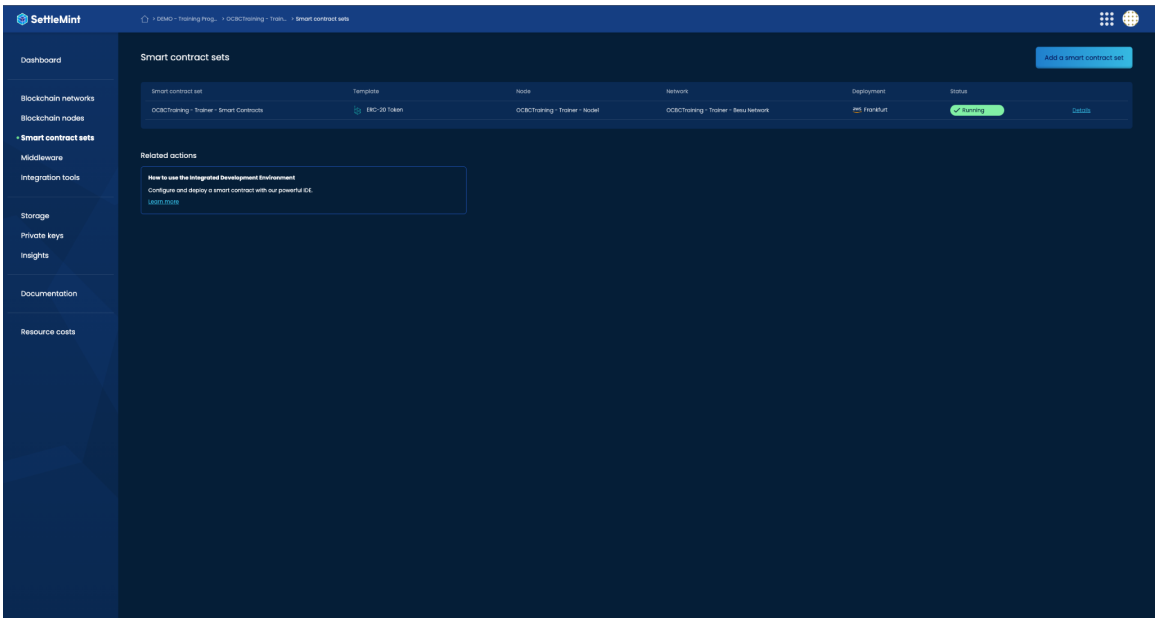
6. Next, let's deploy a smart contract set. The process is very similar to step 5:

   A. Navigate to the "Smart Contract Sets" tab in your left menu.
   B. Click on the "Add Smart Contract Set" button.
   C. In the first section, "Choose Template," select the "ERC-20 Token" template.
   For the remaining settings, choose the same options as you did in step 5:

   Type: Shared
   Smart Contract Set Name: OCBCTraining - {your name} - smart contracts
   Cloud Provider: Your assigned cloud provider

Region: Your assigned region
Resource Pack: Small

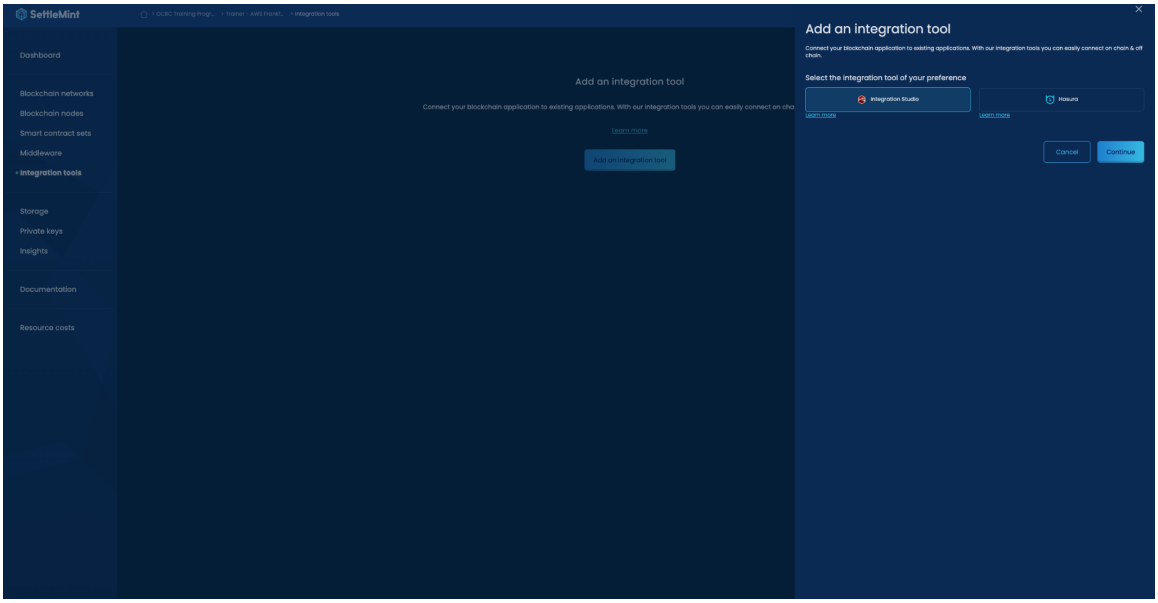




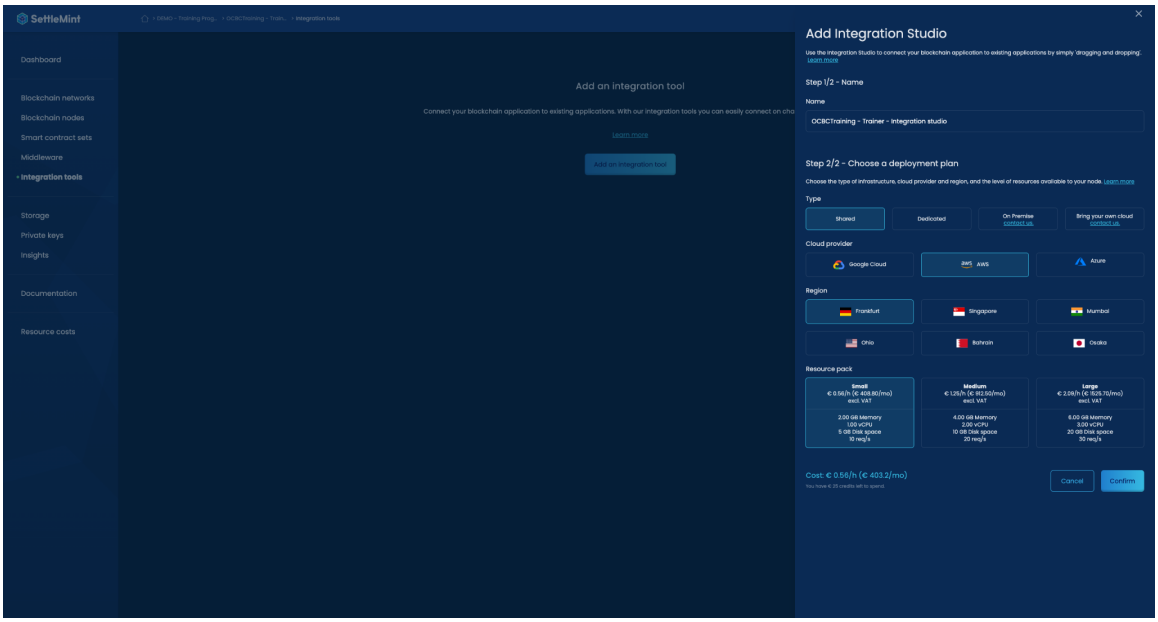
Click "Confirm". Once your smart contract set is in the "Running" status, you are ready to proceed to step

7. To deploy the Integration Studio, please follow these steps:

   A. In the left menu, select "Integration Tools."
   By now, you should have become a blockchain deployment expert!

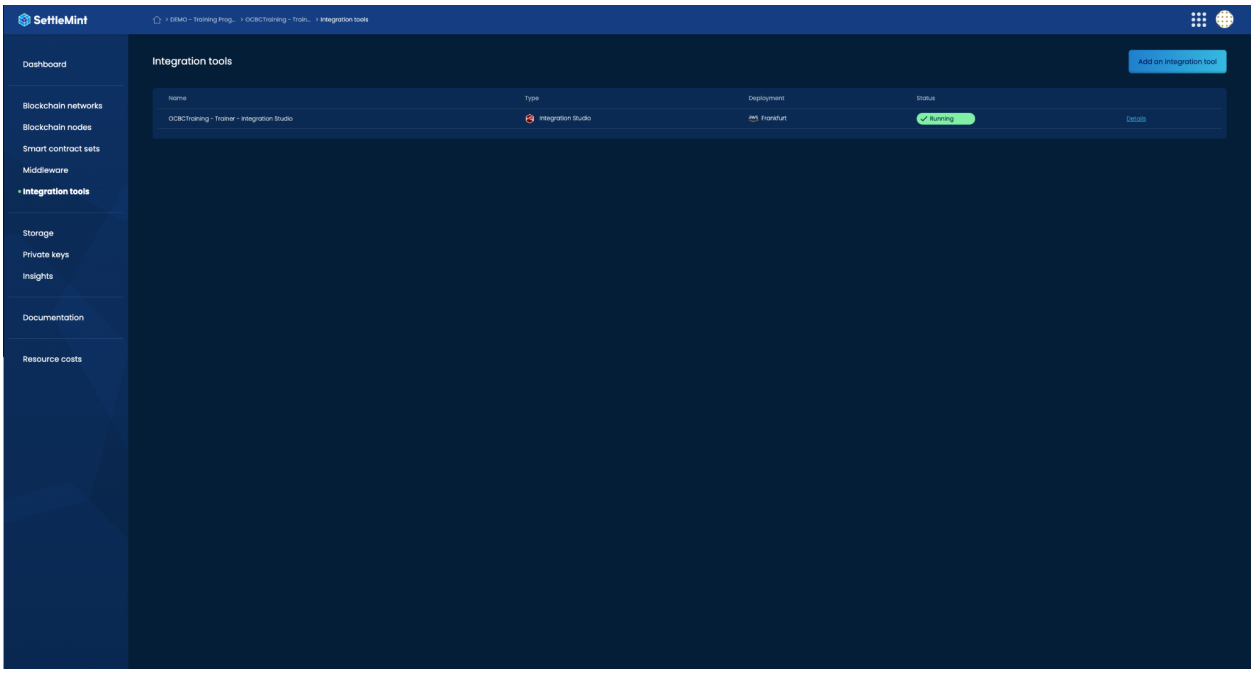


B. Select integration studio
C. Deploy it according to the team settings that you have been assigned to.

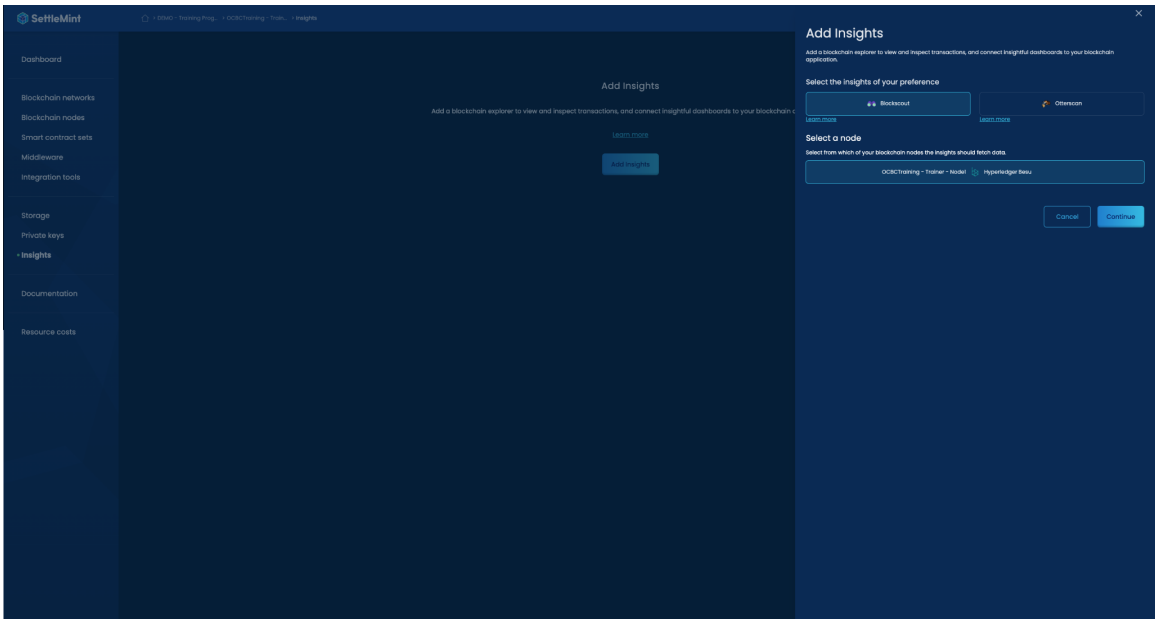


Once it is ready, we will proceed to deploy Blockscout. Blockscout will provide you with a graphical user interface (GUI) to view the blockchain that you have deployed.

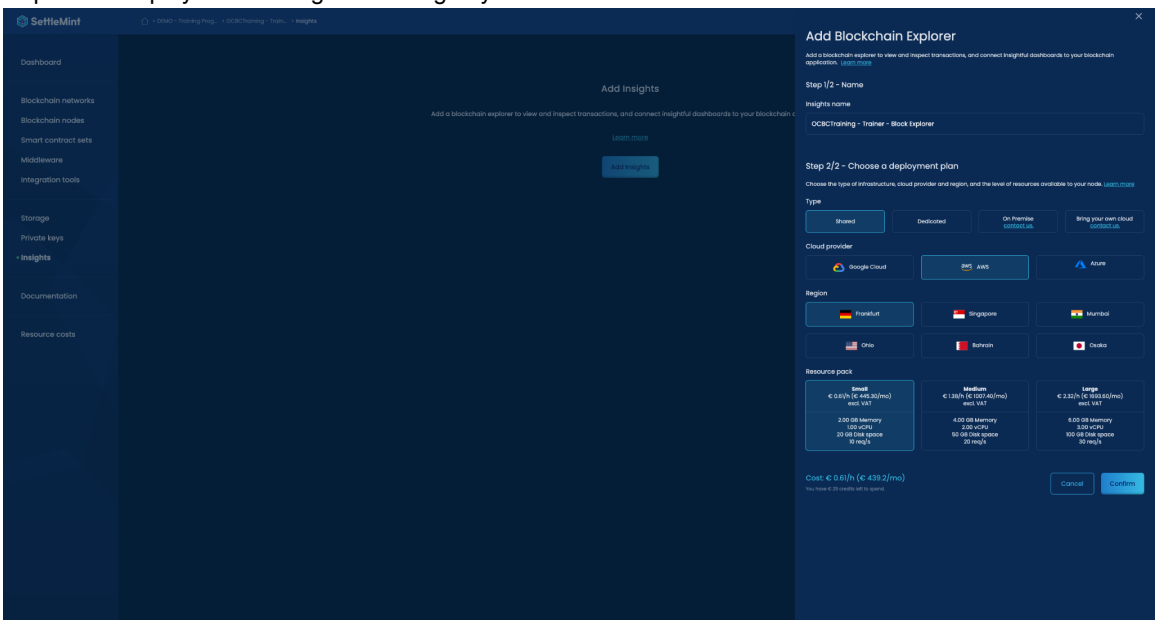
Please follow the next steps to deploy Blockscout.



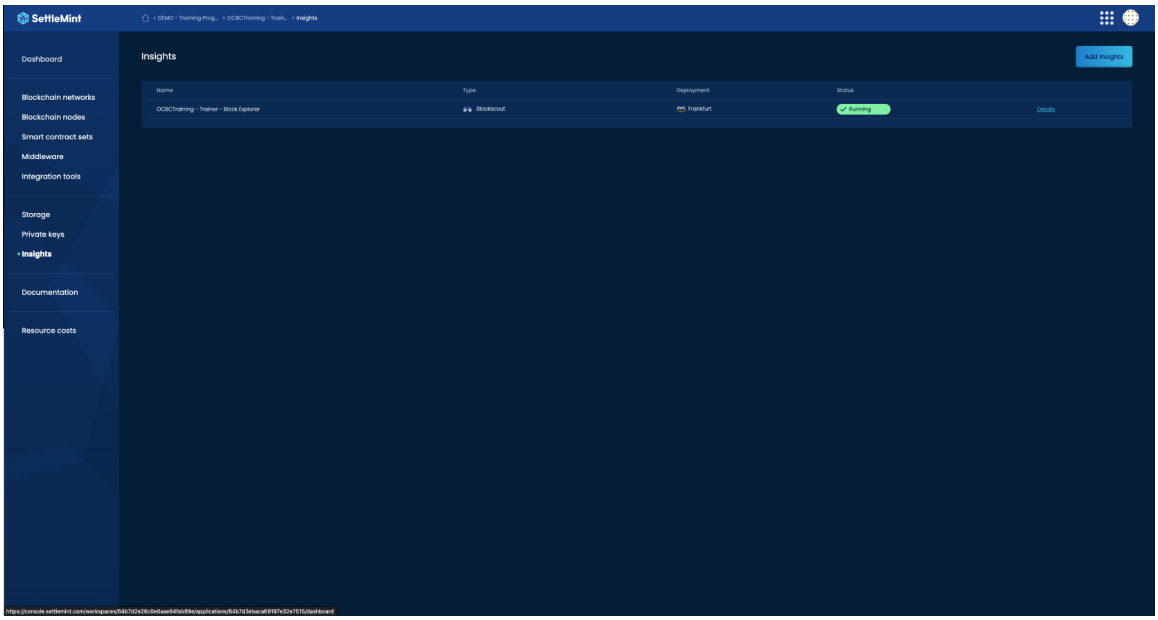
8. To deploy Blockscout, please follow these steps:
   A. Click on "Insights" in the left menu.
   B. Choose "Blockscout" and select the node that you have deployed.
   By following these steps, you will successfully deploy Blockscout and be able to access the insights and information about your deployed blockchain node.



C. pick the deployment settings according to your team



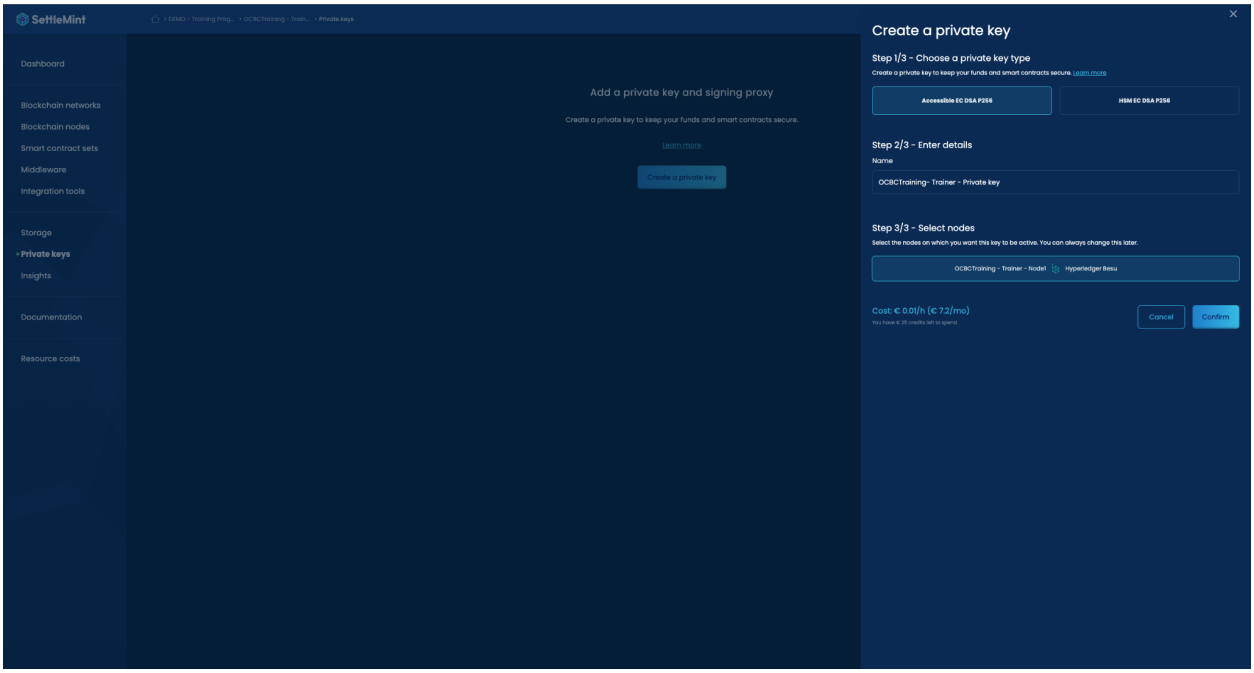
Once the Blockscout status changes to "Running," you can proceed to the final step, which is deploying a private key.



9. DeployTo deploy a private key, please follow these steps:
   A. Navigate to the "Private Keys" section in the left menu.
   B. Click on the "Add a Private Key" button.
   C. Choose the "Accessible" option and select the Besu node that you have deployed.

   By following these steps, you will successfully deploy a private key associated with the Besu node you have deployed

That's it! You are now ready to embark on your blockchain journey!



Course 2 - Part 1

Slide 28: Deploying a smart contract set (ERC 20)

https://github.com/SaeeDawod/training-program/blob/master/course2-part1/contractEditorErc20-intergration-studio-

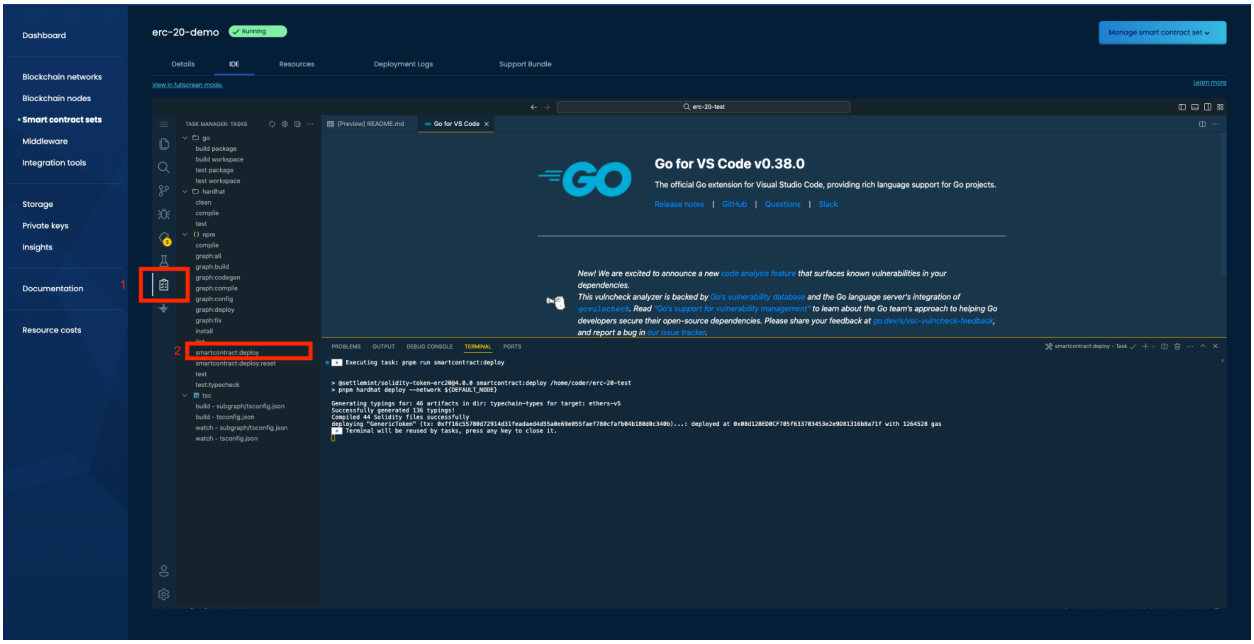
flow.json

Objective: Gain a full, rapid understanding of the platform by deploying a block and performing a simple task.

Step 1: Open your application in the Settlemint platform.

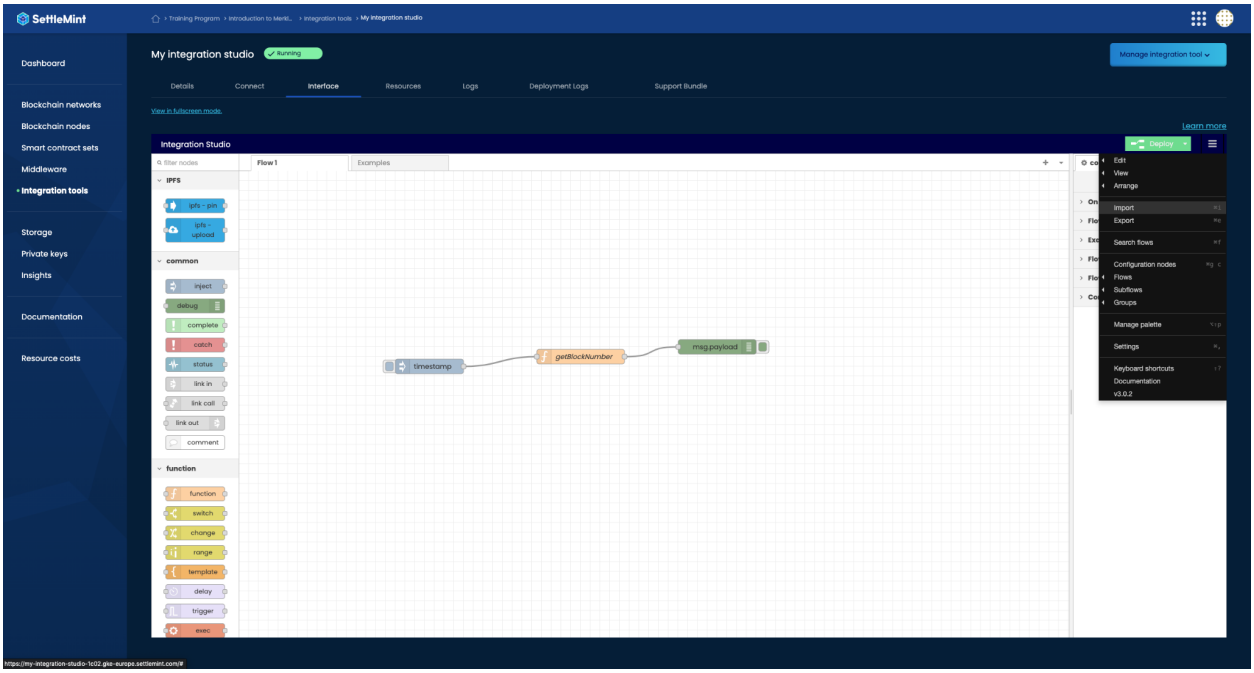
Step 2: Navigate to the IDE and press the Tasks button.

Step 3: Click Smartcontracts:Deploy, after your token is successfully deployed copy the smart contract address form the terminal



Congratulations you have deployed your very own token!

Step 4: Open Integration Studio and navigate to the Interface page



Step 5: Click the Burger Menu in the top right corner and select Import,

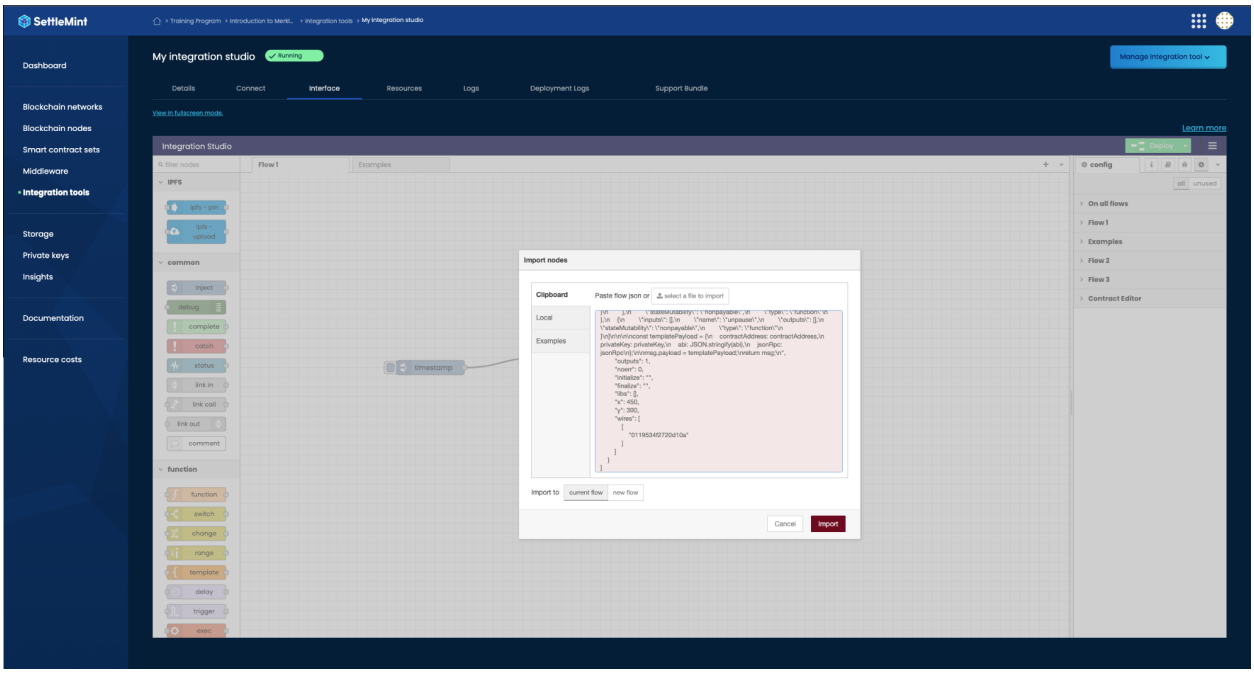
Step 6: Paste the JSON file into the import dialog and click Open

```
[
  {
    "id": "cfc3eb139dc03dc5",
    "type": "tab",
    "label": "Contract Editor",
    "disabled": false,
    "info": "",
    "env": []
  },
  {
    "id": "0119534f2720d10a",
    "type": "template",
    "z": "cfc3eb139dc03dc5",
    "name": "ContractEditor",
    "field": "payload",
    "fieldType": "msg",
    "format": "html",
    "syntax": "mustache",
    "template": "<!DOCTYPE html>\n<html>\n<head>\n    <title>Smart Contract Interface</title>\n    <script src=\"https://cdnjs.cloudflare.com/ajax/libs/ethers/6.6.0/ethers.umd.min.js\"></script>\n    <style>\n        body {\n            background-color: #f7f7f7;\n            font-family: Arial, sans-serif;\n            margin: 20px;\n            padding: 0;\n            overflow-x: hidden;\n        }\n\n        h1 {\n            text-align: center;\n            margin-top: 20px;\n        }\n\n        .contract-section {\n            margin-top: 30px;\n        }\n\n        .contract-title {\n            color: #333;\n            font-size: 24px;\n            margin-bottom: 15px;\n            padding: 0;\n        }\n\n        .contract-functions {\n
```

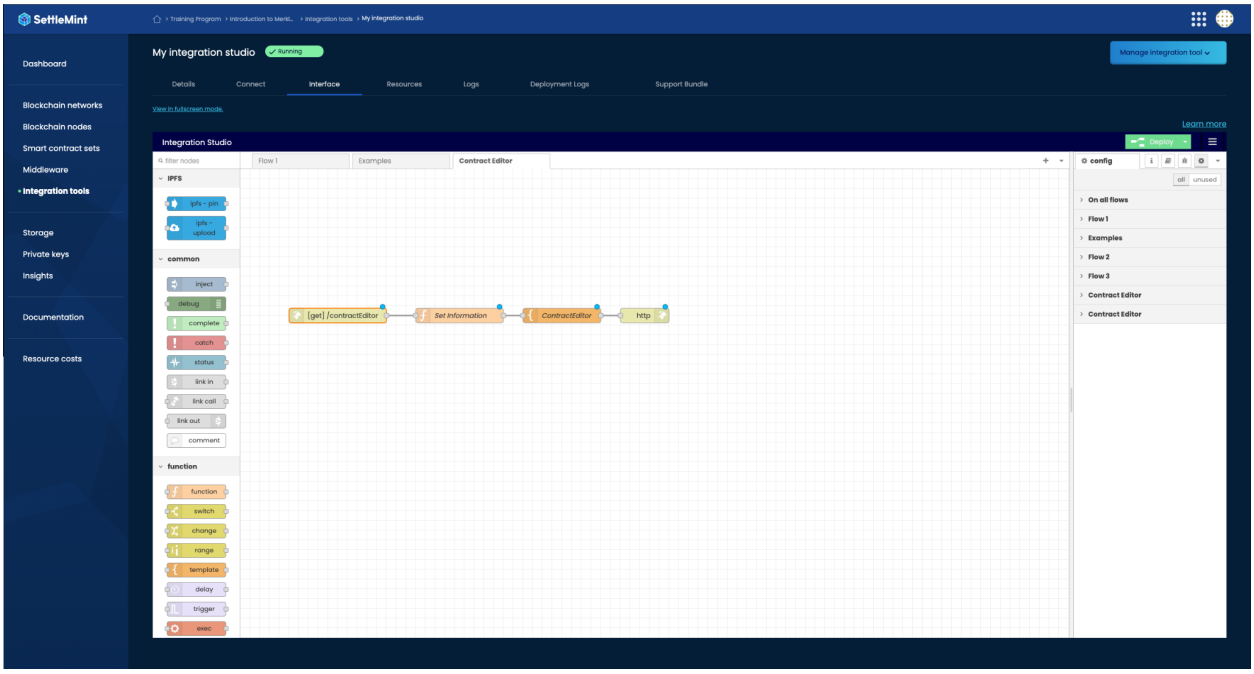
```
display: grid;\n            grid-template-columns: repeat(auto-fit, minmax(320px, 1fr));\n            gap: 20px;\n            margin-top: 10px;\n        }\n\n        .function-card {\n            background-color: #fff;\n            border-radius: 8px;\n            box-shadow: 0 2px 6px rgba(0, 0, 0, 0.1);\n            padding: 20px;\n        }\n\n        .function-title {\n            font-size: 20px;\n            font-weight: bold;\n            margin: 0 0 10px;\n            padding: 0;\n        }\n\n        .function-description {\n            color: #555;\n            margin-bottom: 10px;\n        }\n\n        .function-input {\n            display: flex;\n            align-items: center;\n            margin-bottom: 10px;\n        }\n\n        .input-label {\n            width: 120px;\n            font-weight: bold;\n            margin-right: 10px;\n        }\n\n        .input-field {\n            flex: 1;\n            border: 1px solid #ccc;\n            border-radius: 4px;\n            box-sizing: border-box;\n            font-size: 14px;\n            padding: 8px;\n        }\n\n        .function-result {\n            margin-top: 10px;\n            font-size: 14px;\n            white-space: pre-wrap;\n            overflow-y: auto;\n        }\n\n        .function-result-label {\n            margin-bottom: 5px;\n        }\n\n        .function-button {\n            background-color: #007bff;\n            border: none;\n            border-radius: 4px;\n            color: #fff;\n            cursor: pointer;\n            font-size: 14px;\n            padding: 10px 20px;\n            margin-top: 10px;\n        }\n\n        .function-button:hover {\n            background-color: #0056b3;\n        }\n\n        .logs-div {\n            background-color: #222;\n            color: #fff;\n            margin: 20px auto;\n            padding: 20px;\n            border-radius: 8px;\n            position: sticky;\n            bottom: 0px;\n        }\n\n        .logs-div p {\n            margin: 5px;\n            padding: 5px;\n            line-height: 1.5;\n            overflow-x: auto;\n        }\n\n        .error-message {\n            background-color: #f44336;\n            border-radius: 4px;\n            color: white;\n            margin-top: 10px;\n            padding: 10px;\n            overflow-x: auto;\n        }\n\n        .error-message button {\n            background-color: transparent;\n            border: none;\n            color: white;\n            cursor: pointer;\n            font-size: 16px;\n            line-height: 1;\n            padding: 0;\n            position: absolute;\n            top: 5px;\n            right: 5px;\n        }\n    </style>\n</head>\n\n<body>\n    <h1>Contract Editor</h1>\n<h2>Contract : {{payload.contractAddress}}</h2>\n    <div class=\"contract-section\">\n        <h2 class=\"contract-title\">Read-only Functions</h2>\n        <div class=\"contract-functions\" id=\"readFunctions\"></div>\n    </div>\n\n    <div class=\"contract-section\">\n        <h2 class=\"contract-title\">Write Functions</h2>\n        <div class=\"contract-functions\" id=\"writeFunctions\"></div>\n    </div>\n\n    <div class=\"logs-div\" id=\"logsDiv\"></div>\n\n    <script>\n        \n\n        window.onload = async function () {\n            \n            const contractAddress = '{{{payload.contractAddress}}}'; // Contract address placeholder\n            const privateKey = '{{{payload.privateKey}}}'; // Private key placeholder\n            const abi = JSON.parse('{{{payload.abi}}}'); // ABI placeholder\n\n            const provider = new ethers.JsonRpcProvider('{{{payload.jsonRpc}}}');\n            const wallet = new ethers.Wallet(privateKey, provider);\n    const contract = new ethers.Contract(contractAddress, abi, wallet);\n\n    const readFunctionsDiv = document.getElementById('readFunctions');\n    const writeFunctionsDiv = document.getElementById('writeFunctions');\n    const logsDiv = document.getElementById('logsDiv');\n\n    function formatResult(result) {\n    if (typeof result === 'bigint') {\n        return result.toString();\n    }\n    if (Array.isArray(result)) {\n        return result.map(formatResult);\n    }\n    if (typeof result === 'object') {\n        const formattedResult = {};\n        for (const key in result) {\n            formattedResult[key] = formatResult(result[key]);\n        }\n        return formattedResult;\n    }\n    return result;\n}\n\n    abi.forEach((func) => {\n        if (func.type === \"function\" || func.type === \"variable\") {\n            const functionDiv = document.createElement('div');\n            functionDiv.className = 'function-card';\n\n            const functionName = document.createElement('h3');\n            functionName.className = 'function-title';\n            functionName.textContent = func.name;\n            functionDiv.appendChild(functionName);\n\n            const functionDescription = document.createElement('p');\n            functionDescription.className = 'function-description';\n            functionDescription.textContent = func.stateMutability === 'view' ? 'Read-only function' : 'Write function';\n            functionDiv.appendChild(functionDescription);\n\n            const inputFields = [];\n            if (func.inputs && func.inputs.length > 0) {\n                func.inputs.forEach((input) => {\n                    const inputContainer = document.createElement('div');\n                    inputContainer.className = 'function-input';\n\n                    const inputLabel = document.createElement('label');\n                    inputLabel.className = 'input-label';\n                    inputLabel.textContent = input.name + ' (' + input.type + '): ';\n                    inputContainer.appendChild(inputLabel);\n\n                    const inputField = document.createElement('input');\n                    inputField.className = 'input-field';\n                    inputField.setAttribute('name', func.name + '_' + input.name);\n                    inputContainer.appendChild(inputField);\n\n                    inputFields.push(inputField);\n\n                    functionDiv.appendChild(inputContainer);\n                });\n            }\n\n            const resultMessage = document.createElement('div');\n            resultMessage.className = 'function-result';\n            resultMessage.id = func.name + '_result';\n            functionDiv.appendChild(resultMessage);\n\n            const callButton =
```

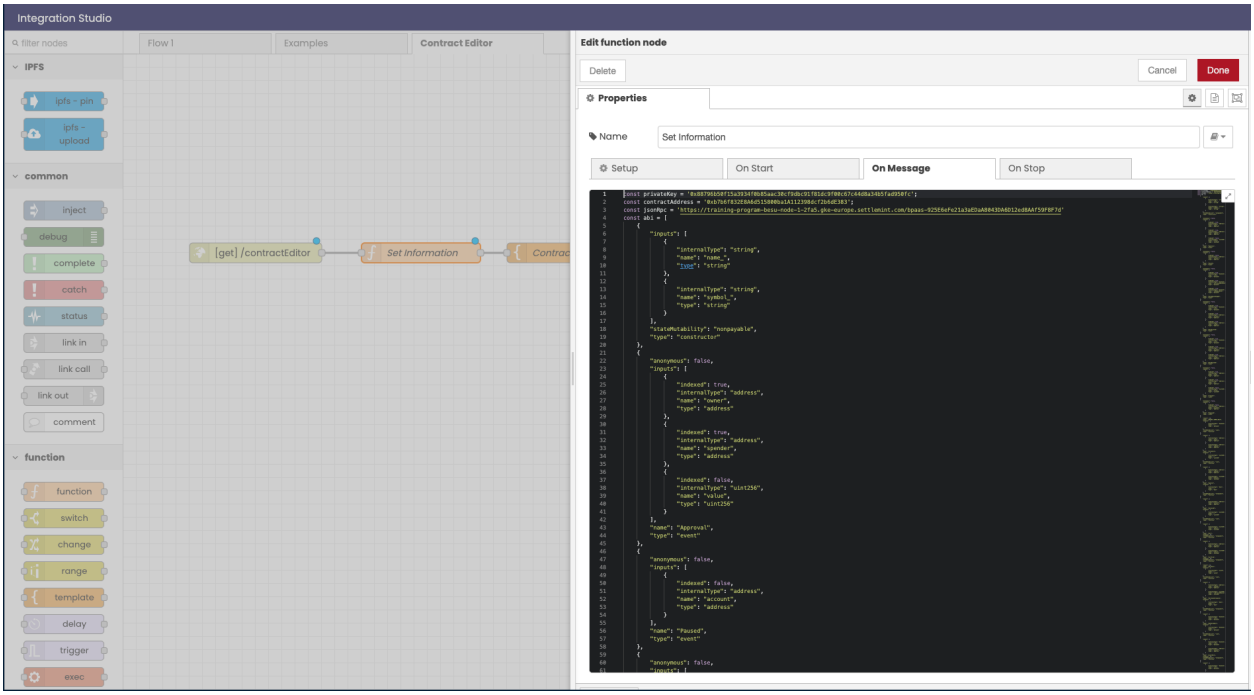
document.createElement('button');\n callButton.className = 'function-button';\n callButton.textContent = 'Call';\n callButton.onclick = async function () {\n const args = inputFields.map(inputField => inputField.value);\n try {\n let result;\n if (func.stateMutability === 'view') {\n resultMessage.textContent = 'Loading...';\n result = await contract[func.name](...args);\n resultMessage.textContent = 'Result:\\n' + formatResult(result);\n } else {\n resultMessage.textContent = 'Loading...';\n const tx = await contract[func.name](...args);\n const logMessage = document.createElement('p');\n logMessage.textContent = 'Transaction hash: ' + tx.hash + ', Args: ' + JSON.stringify(args) + ', Function: ' + func.name;\n logsDiv.appendChild(logMessage);\n resultMessage.textContent = 'Transaction sent. Waiting for confirmation...';\n await tx.wait();\n resultMessage.textContent = 'Transaction confirmed for ' + func.name;\n\n // Display complete transaction object in logs\n const txObjectMessage = document.createElement('p');\n txObjectMessage.textContent = 'Transaction object: ' + JSON.stringify(tx);\n logsDiv.appendChild(txObjectMessage);\n }\n } catch (err) {\n const errorMessage = document.createElement('div');\n errorMessage.className = 'error-message';\n errorMessage.textContent = 'Error for ' + func.name + ': ' + err.message;\n const dismissButton = document.createElement('button');\n dismissButton.textContent = 'X';\n dismissButton.onclick = function () {\n functionDiv.removeChild(errorMessage);\n };\n errorMessage.appendChild(dismissButton);\n functionDiv.appendChild(errorMessage);\n }\n };\n\n functionDiv.appendChild(callButton);\n\n if (func.stateMutability === 'view') {\n readFunctionsDiv.appendChild(functionDiv);\n } else {\n writeFunctionsDiv.appendChild(functionDiv);\n }\n }\n });\n};\n </script>\n</body>\n\n</html>",
    "output": "str",
    "x": 660,
    "y": 300,
    "wires": [
      [
        "a52e534fe124b0d5"
      ]
    ]
  },
  {
    "id": "a52e534fe124b0d5",
    "type": "http response",
    "z": "cfc3eb139dc03dc5",
    "name": "",
    "statusCode": "",
    "headers": {},
    "x": 830,
    "y": 300,
    "wires": []
  },
  {
    "id": "3944e3fbedb2ac27",
    "type": "http in",
    "z": "cfc3eb139dc03dc5",
    "name": "",
    "url": "/contractEditor",
    "method": "get",
    "upload": false,
    "swaggerDoc": "",
    "x": 200,

```
        "y": 300,
        "wires": [
            [
                "15e052d0b38acb4f"
            ]
        ]
    },
    {
        "id": "15e052d0b38acb4f",
        "type": "function",
        "z": "cfc3eb139dc03dc5",
        "name": "Set Information",
        "func": "const privateKey = '0x88796b50f15a3934f0b85aac30cf9dbc91f81dc9f00c67c44d8a34b5fad950fc';
\nconst contractAddress = '0xf4Df742D502A294eA0B2712B5b61D575E3DAe788';\nconst jsonRpc = 'https://training-
program-besu-node-1-2fa5.gke-europe.settlemint.com/
bpaas-925E6eFe21a3aEDaA8043DA6D12ed8AAf59F8F7d'\nconst abi = [\n    {\n        \"inputs\": [],\n
\"stateMutability\": \"nonpayable\",\n        \"type\": \"constructor\"\n    },\n    {\n        \"inputs\": [],\n        \"name\":
\"greetingMessage\",\n        \"outputs\": [\n            {\n                \"internalType\": \"string\",\n                \"name\": \"\",\n
\"type\": \"string\"\n            }\n        ],\n        \"stateMutability\": \"view\",\n        \"type\": \"function\"\n    },\n    {\n
\"inputs\": [\n            {\n                \"internalType\": \"string\",\n                \"name\": \"name\",\n                \"type\":
\"string\"\n            }\n        ],\n        \"name\": \"setGreetings\",\n        \"outputs\": [\n            {\n                \"internalType\":
\"string\",\n                \"name\": \"\",\n                \"type\": \"string\"\n            }\n        ],\n        \"stateMutability\":
\"nonpayable\",\n        \"type\": \"function\"\n    }\n]\n\nconst templatePayload = {\n    contractAddress:
contractAddress,\n    privateKey: privateKey,\n    abi: JSON.stringify(abi),\n    jsonRpc: jsonRpc\n};\n\nmsg.payload =
templatePayload;\nreturn msg;",
        "outputs": 1,
        "noerr": 0,
        "initialize": "",
        "finalize": "",
        "libs": [],
        "x": 450,
        "y": 300,
        "wires": [
            [
                "0119534f2720d10a"
            ]
        ]
    }
]
```



Step 7: Double-click the Set Information function node.

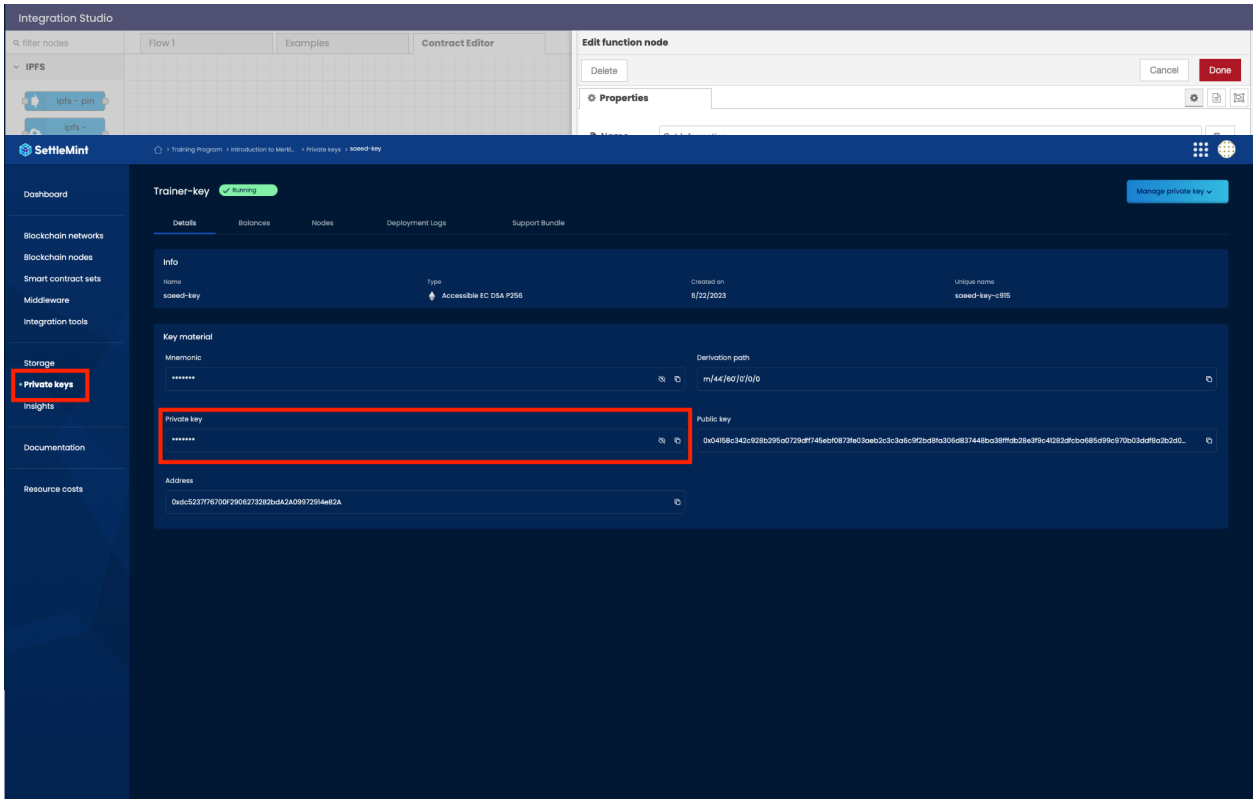




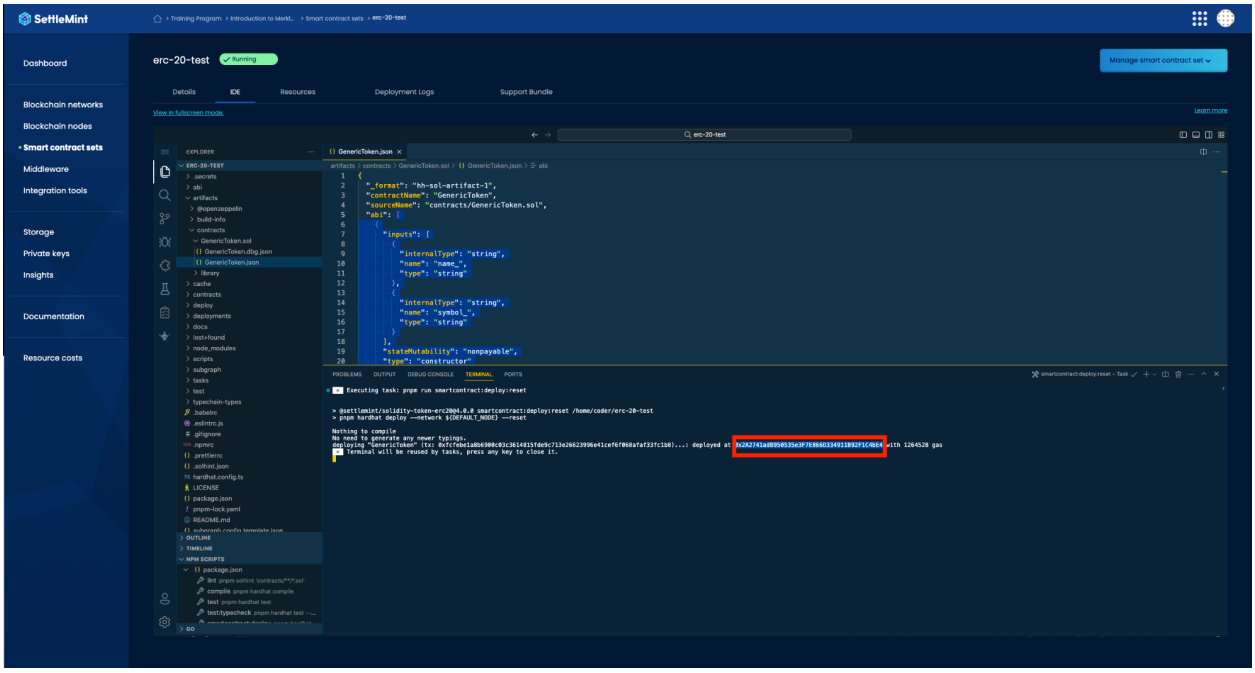
Step 8: Update the following fields:

a. private key:

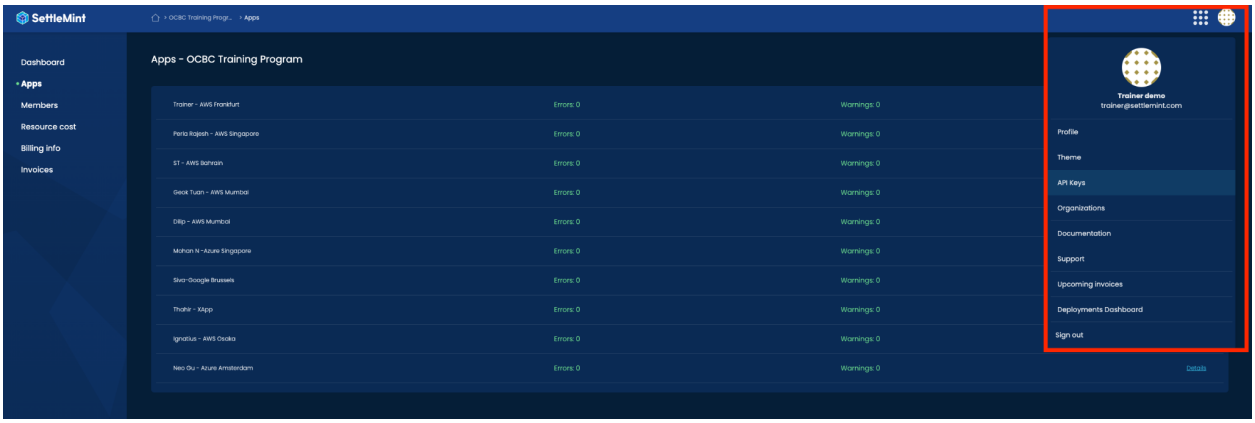
To get the private key navigate in the left menu to “private keys” , copy the private key and paste as the value of private key in the “Set information” function node.



b. Contract Address:
Paste the contract address that you obtained in step 3.



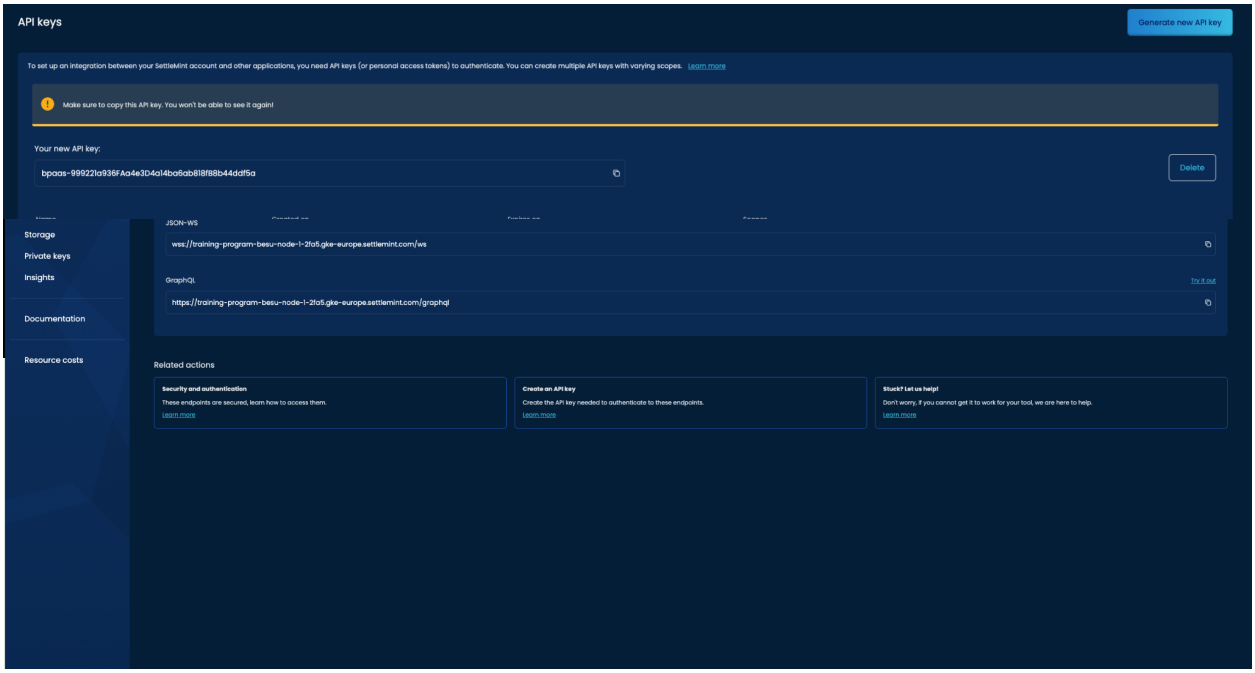
c. JSON-RPC: Create an API key by clicking your user icon in the top right corner and selecting API Keys



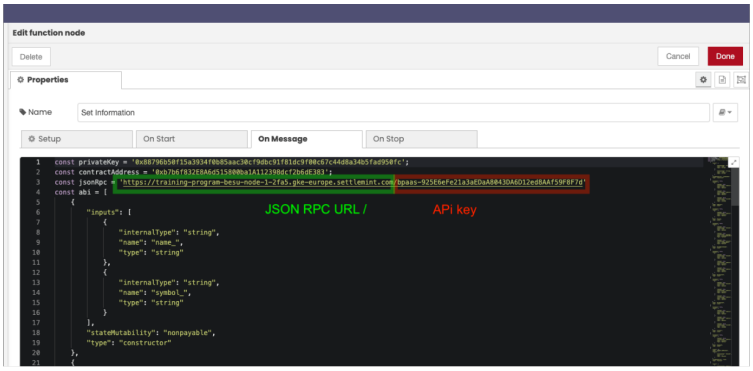
i. Enter a key name and check the Blockchain Nodes service.

Copy the JSON-RPC URL and paste it into the JSON-RPC field, followed by your API key.

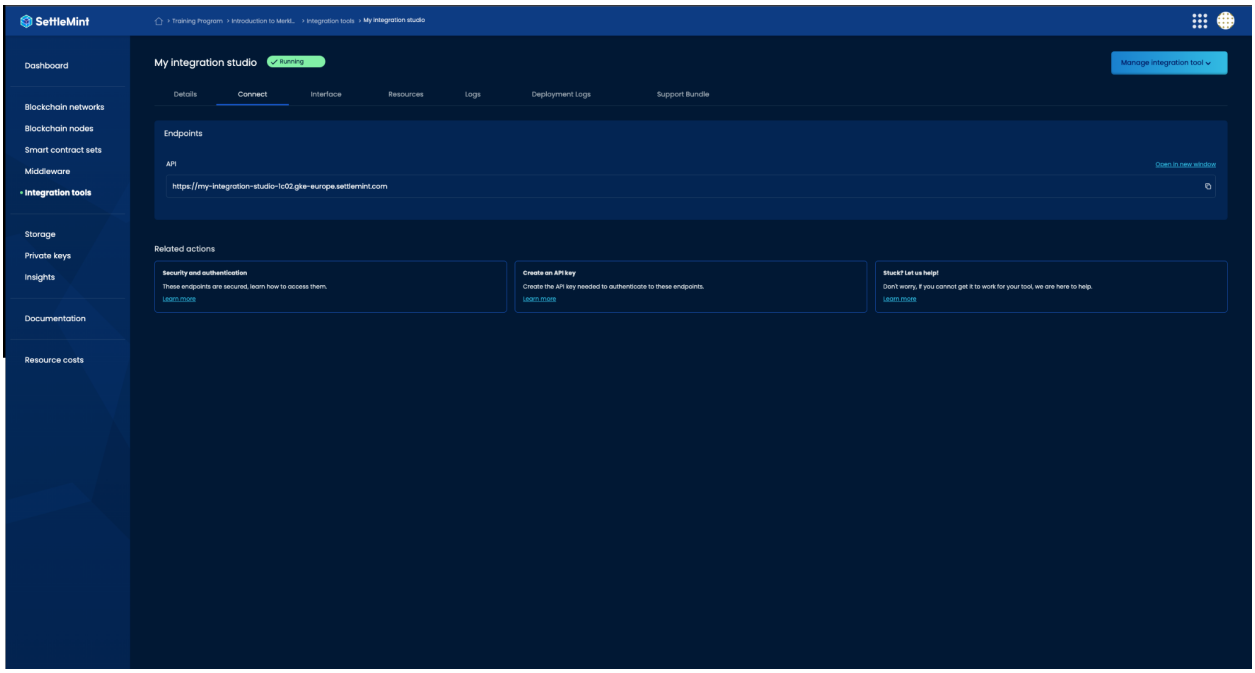
Json-rpc-url / Api key



ii. ABI: The ABI that comes with this flow is specific to the smart contract that you deployed, so you do not need to bring your own. However, in general, you will need to bring your own ABI, which you can obtain after the smart contract is compiled. (You will learn more about this in the Hardhat section.)

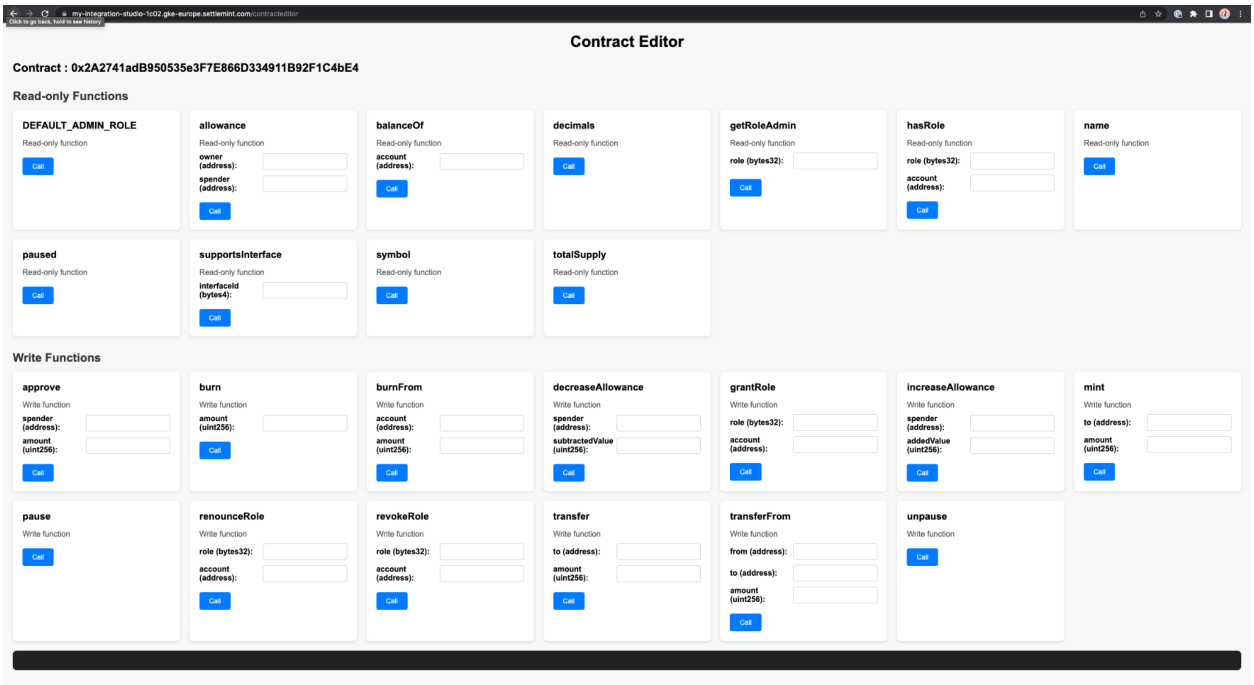


Step 9: Navigate back to your integration studio tab and press Connect.
Step 10: Copy the API link and paste it into your browser, followed by contractEditor.

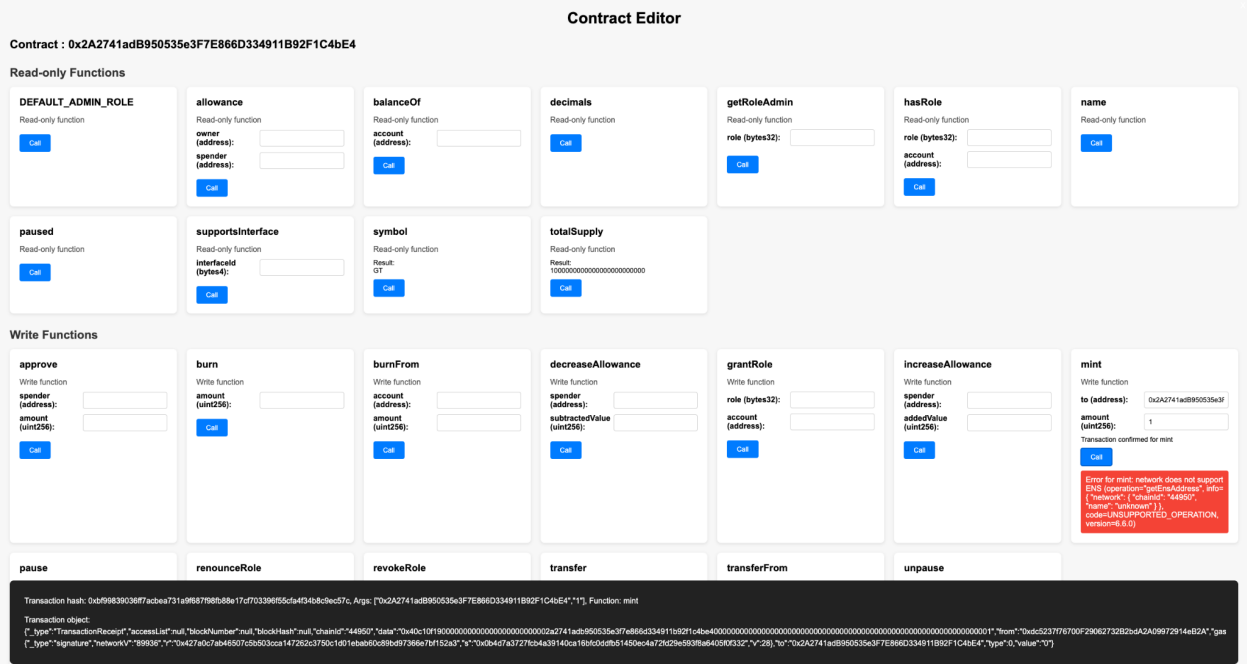


[?]My-intergration-studio-api-url / contracteditor
[?]

Step 11: You will now be able to see and interact with all the functions of the smart contract.



This interface will also show you errors and the transaction response.



Step 12: To complete the task, transfer a token to one of your colleagues and paste the transaction object in the chat

Slide 67:

Follow along the slides (if you can access remix ide)

1. Deploying contract Locally
2. Test net
3. Main net.

Slide 76: Hello World Smart Contract

Step 1: Open your IDE, navigate to the contracts folder and create a new file named “HelloWorld.sol”. Paste the following contract in it:

```
pragma solidity ^0.8.17;

contract HelloWorld {

    string public greetingMessage;

    constructor() {
        greetingMessage = "Hello world!";
    }

}
```

This contract is a simple Solidity smart contract. It declares a public string greetingMessage and assigns it the value "Hello World" upon initialization of the contract.

Step 2: In your IDE, navigate to the “deploy” folder, create a new file named “01_deploy_hello_world.ts”, and paste the following code:

```
import { DeployFunction } from 'hardhat-deploy/types';

const deploy: DeployFunction = async ({ deployments, getNamedAccounts }) => {
  const { deploy } = deployments;
  const { deployer } = await getNamedAccounts();

  await deploy('HelloWorld', {
    from: deployer,
    args: [],
    log: true,
  });
};
```

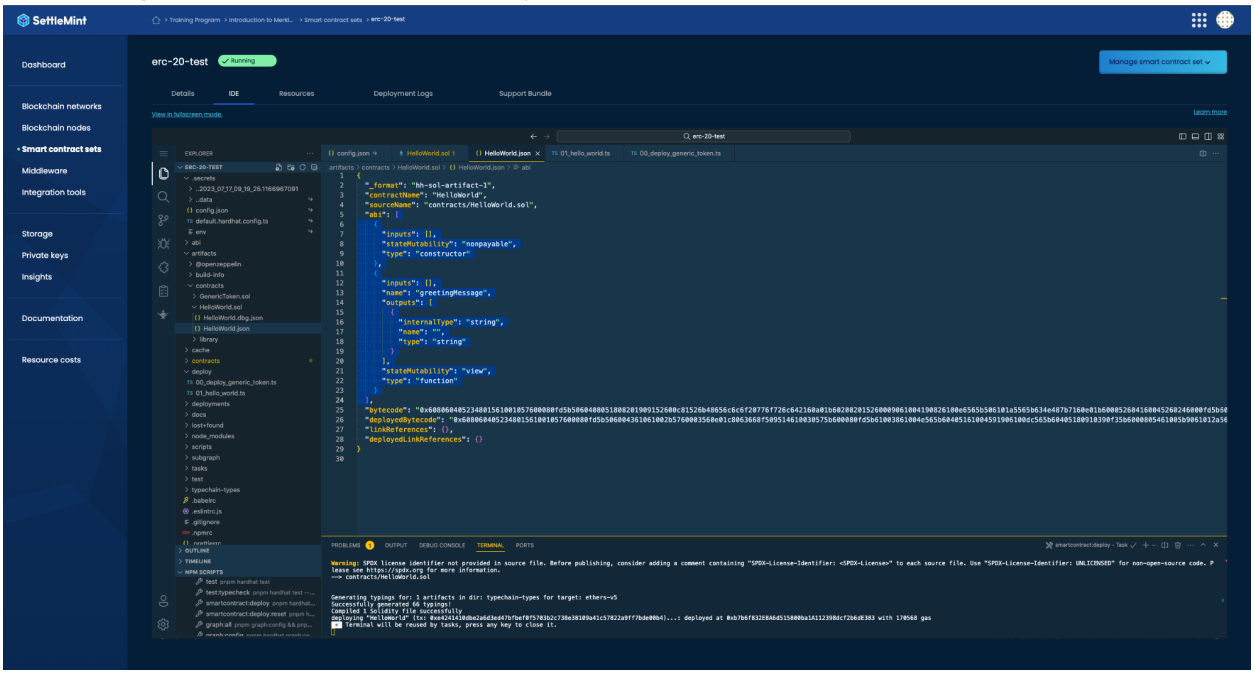
```
deploy.tags = ['HelloWorld'];
export default deploy;
```

Step 3: Run the command “pnpm hardhat compile”.

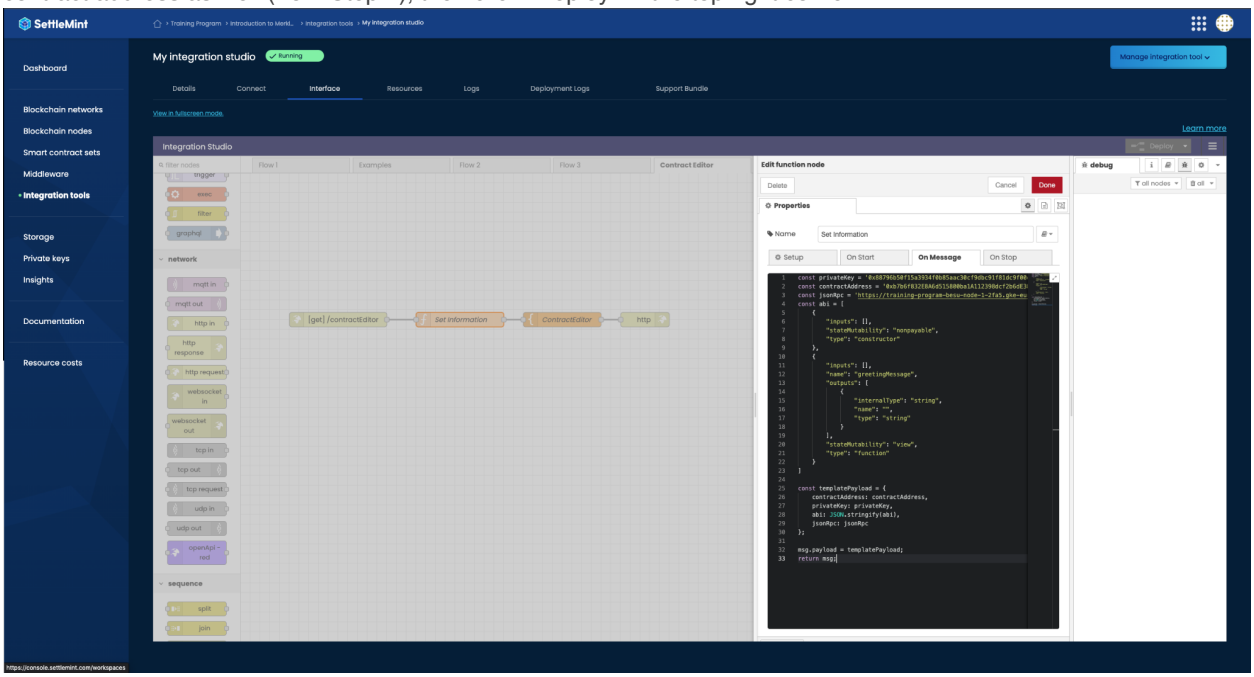
Step 4: Execute the command `smartcontract:deploy:reset` and copy the smart contract address for later use.

Step 5: Go to the integration studio, open your contract editor flow > 'Set Information' node > update the contractAddress variable with the address of the new HelloWorld contract we just deployed

Step 6: Navigate to artifact, contract, HelloWorld.json and copy the ABI from the smart contract.

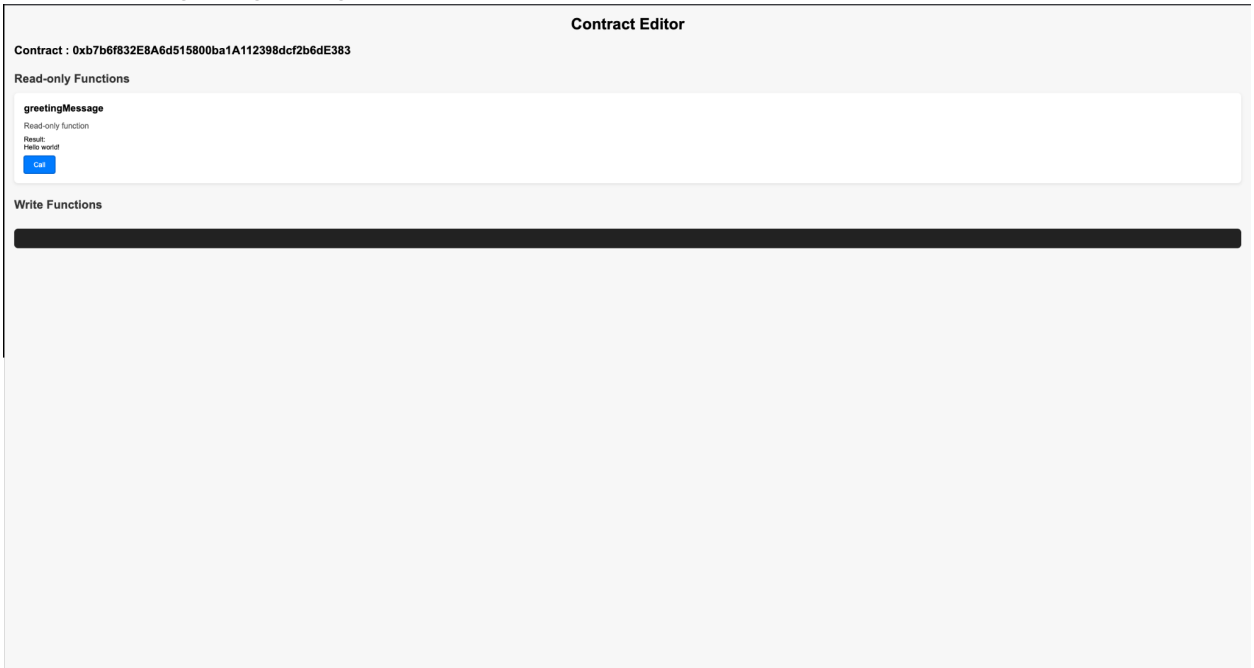


Step 7: Paste the ABI in Integration Studio's "Contract editor" flow, in the "Set Information" node. Import the smart contract address as well (from Step 4), then click "Deploy" in the top right corner.



Step 8: Navigate to the contract editor page on your Integration Studio API URL, e.g., my-integration-studio-api-url/contracteditor.com.

Step 9: Call the greetingMessage variable.



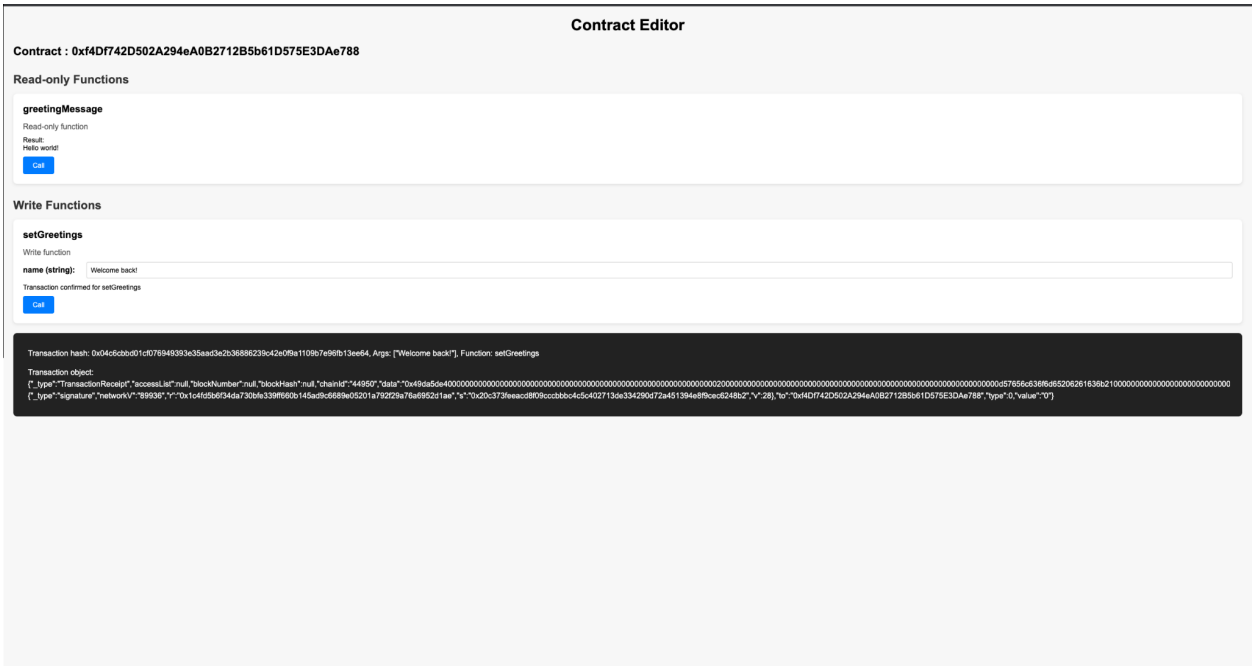
Step 10: Update the smart contract to allow a new value for greetingMessage by creating a new setGreetings function:

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

contract HelloWorld {
  string public greetingMessage;

  constructor() {
    greetingMessage = "Hello world!";
  }

  function setGreetings(string memory name) public returns (string memory) {
    greetingMessage = string(abi.encodePacked("Hello, ", name, "!"));
    return greetingMessage;
  }
}
```

This new function setGreetings accepts a string name as a parameter, concatenates it with the string "Hello, " and a "!" at the end, and assigns this new string to greetingMessage. The function then returns this new greeting.

Step 11: Compile and deploy the updated contract using your IDE. “smartcontract:deploy:reset

Step 12: Insert the new contract's address and ABI into the respective fields in the ContractEditor flow's "Set information" node.



Step 13: Navigate to your-integration-studio-api/contractEditor and input a new greeting message in the set greeting function. Call your getGreetings variable and you should see the new value you assigned.
And that's it!

Don't forget to send your transaction hash once completed.

Slide 90: Verifying contracts on a public chain / block explorer - Follow along

- if you can access remix ide- Else watch!

1. Deploying contract on a public network
2. Verify the contract on a public Block Explorer

Slide 91: blockscout - Verifying contracts on a private chain / block explorer

https://docs.blockscout.com/for-users/verifying-a-smart-contract

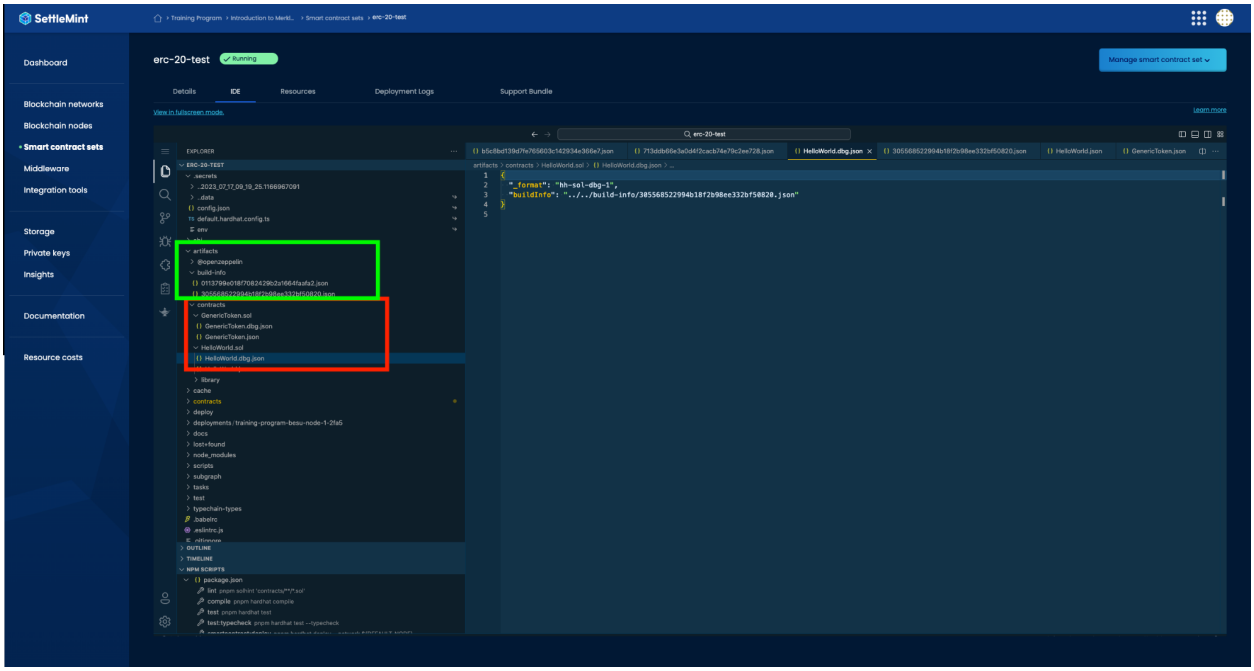
Step 1: Access the Blockscout Verification Page
Find one of your contracts on blockscout

Step 2: Specify the Contract Name
Enter the specific name for your contract, enter it in the format: MyContract.

Step 3: Decide on Nightly Builds
Choose whether to include nightly builds in your verification process. You can select either "Yes" or "No" depending on your compiler.

Step 4: Select the Compiler
Choose the compiler version that was used to compile your smart contract. This should be a dropdown menu with various version options.

Step 5: Attach the Standard Input JSON
Copy the contents of your Standard Input JSON file. You can locate this file by first opening contracts/ ContractName.sol/ContractName.dbg.json, where ContractName is the actual name of your contract.
The correct Standard Input JSON file name can be found within the ContractName.dbg.json file. Once you have the correct file name, you can find the Standard Input JSON file in the artifacts/build-info directory.



Step 6: Decide on Constructor Arguments
Choose whether to fetch constructor arguments automatically. Select "No" if you want to provide ABI-encoded Constructor Arguments or if the contract does not have any.
Step 7: Provide ABI-encoded Constructor Arguments
If you chose "No" in the previous step, fill in the ABI-encoded Constructor Arguments field. If the contract does not have any constructor arguments, you can leave this field blank.
Step 8: Verify and Publish

After filling out the form, click the "Verify & Publish" button and wait for the response. This will initiate the verification process.

Slide 120: Tictactoe

In the upcoming steps, we're going to navigate through the process of deploying a blockchain-based Tic-Tac-Toe game. Once it's up and running, challenge a colleague to a round.

Step 1: Contract Deployment - Now that we've become proficient in this area, let's use the steps from our previous session to deploy the following contract: TicTacToe.sol. Go to the contracts folder > Create a new file called TicTacToe.sol > Paste the content from https://github.com/SaeeDawod/training-program/blob/master/course2-part1/tikTacToe/TikTacToe.sol
.

```
contract TicTacToe {
    uint8 constant ARRAY_LENGTH = 3;
    address public player1;
    address public player2;
    bool public player1Turn = true;
    uint[3][3] public board;
    bool public gameOver = false;

    // Declare events
    event TurnEnded(address player, uint8 x, uint8 y);
    event GameEnded(address winner);

    constructor(address _player2) {
        player1 = msg.sender;
        player2 = _player2;
    }

    function makeMove(uint8 _x, uint8 _y) public {
        require(!gameOver, "Game Over!");
        require(msg.sender == player1 && player1Turn || msg.sender == player2 && !player1Turn, "Not your turn!");
        require(board[_x][_y] == 0, "Cell is not empty!");
        require(_x < ARRAY_LENGTH && _y < ARRAY_LENGTH, "Invalid cell!");

        if (player1Turn) {
```

```
            board[_x][_y] = 1;
        } else {
            board[_x][_y] = 2;
        }

        // Emit turn ended event
        emit TurnEnded(msg.sender, _x, _y);

        if (checkWinner(_x, _y)) {
            gameOver = true;
            // Emit game ended event
            emit GameEnded(msg.sender);
        } else {
            player1Turn = !player1Turn; // switch turn
        }
    }

    function checkWinner(uint8 _x, uint8 _y) internal view returns (bool) {
        uint8 player = player1Turn ? 1 : 2;
        // horizontal
        if (board[_x][0] == player && board[_x][1] == player && board[_x][2] == player) {
            return true;
        }
        // vertical
        if (board[0][_y] == player && board[1][_y] == player && board[2][_y] == player) {
            return true;
        }
        // diagonal
        if (_x == _y && board[0][0] == player && board[1][1] == player && board[2][2] == player) {
            return true;
        }
        // anti-diagonal
        if (_x + _y == 2 && board[0][2] == player && board[1][1] == player && board[2][0] == player) {
            return true;
        }
        return false;
    }

    function getBoard() public view returns (uint[3][3] memory) {
        return board;
    }
}
```

https://github.com/SaeeDawod/training-program/blob/master/course2-part1/tikTacToe/TikTacToe.sol

Step 2: Creating the deploy script to deploy the contract the TicTacToe contract on chain:

Go to the ‘deploy’ folder in the IDE > Create a file named ‘02_deploy_tic_tac_toe.ts’ > Paste the

contents from this link:
https://gist.github.com/snigdha920/666c627bb5298dc71820de8697150d02 to the file

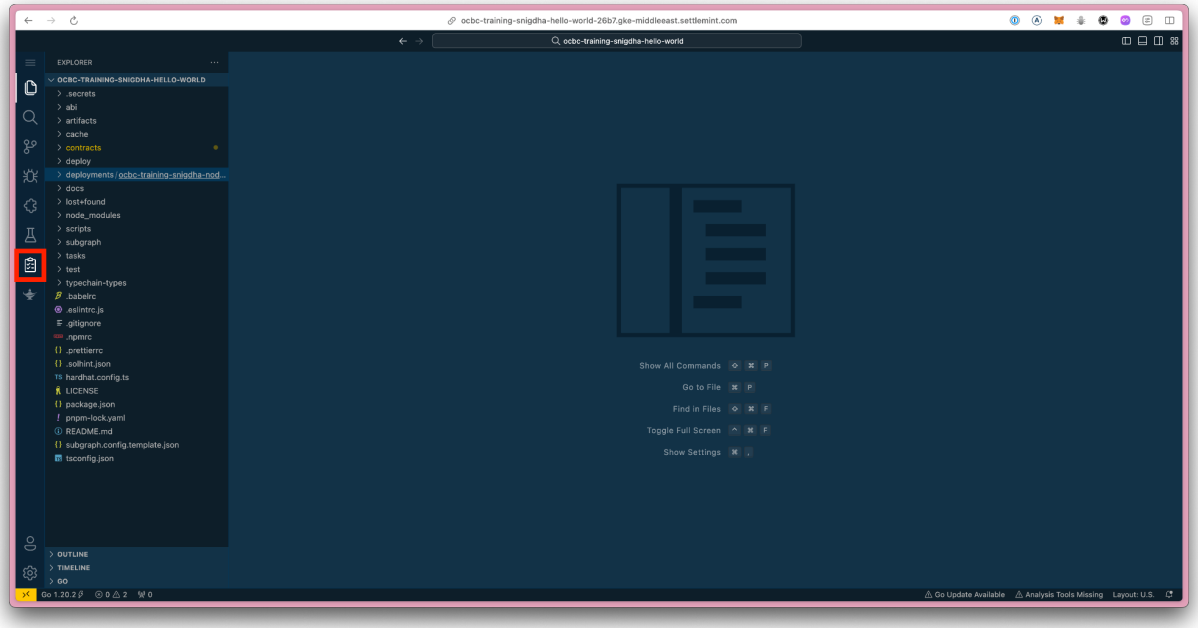
Step 3: Set the correct arguments for the deploy script: To deploy this contract we need to give an address of the second player. To get this address, ask for the address of the private key of your partner.

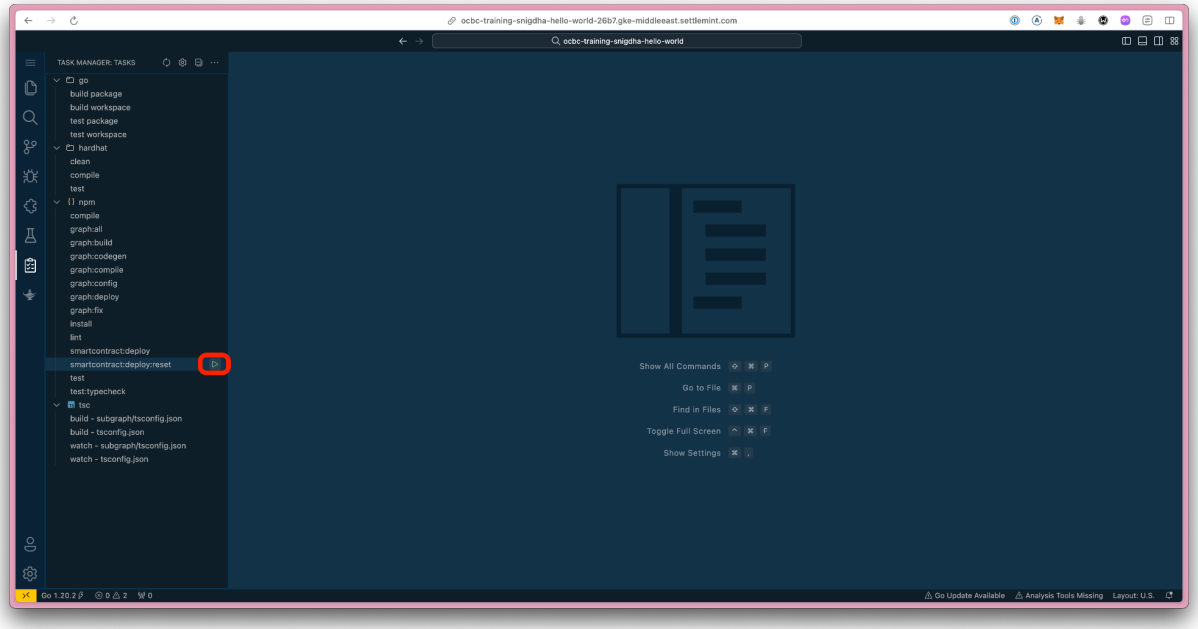
Once you have that, go to the deploy script we created in Step 3, and paste the address in the place of ‘<address-of-second-private-key>’ on line 9. It will look like:

```
await deploy('TicTacToe', {
  from: deployer,
  args: ['0x2220Cb92701f272327daaCcCF150cD4d1461Bc3B'],
  log: true,
});
```

Ensure you have the key address enclosed in ‘’!

Step 4: Deploy the TicTacToe contract on chain: Once you have the deploy script ready, go to the tasks bar by clicking on the following icon: and choosing ‘smartcontract:deploy:reset’:





Your TicTacToe contract will be deployed!

Step 2: Integration Studio - Open your Integration Studio and and import the following flow


```
[
  {
    "id": "1b72e8d575aef462",
    "type": "tab",
```

```
        "label": "Flow 1",
        "disabled": false,
        "info": "",
        "env": []
    },
    {
        "id": "f400539731192b8a",
        "type": "template",
        "z": "1b72e8d575aef462",
        "name": "Game",
        "field": "payload",
        "fieldType": "msg",
        "format": "handlebars",
        "syntax": "mustache",
        "template": "<!DOCTYPE html>\n<html>\n\n<head>\n    <title>Tic Tac Toe</title>\n    <style>\n        body {\n            display: flex;\n            flex-direction: column;\n            align-items: center;\n            height: 100vh;\n            font-family: Arial, sans-serif;\n        }\n\n        #feedback {\n            margin-bottom: 20px;\n            font-size: 1.5em;\n            font-weight: bold;\n        }\n\n        #board {\n            display: grid;\n            grid-template-columns: repeat(3, 1fr);\n            gap: 10px;\n        }\n\n        .cell {\n            width: 60px;\n            height: 60px;\n            border: 1px solid #000;\n            display: flex;\n            justify-content: center;\n            align-items: center;\n            font-size: 2em;\n        }\n\n        #player-turn {\n            margin-bottom: 20px;\n            font-size: 1.5em;\n        }\n    </style>\n</head>\n\n<body>\n    <div id=\"feedback\">Tic Tac Toe</div>\n    <div id=\"player-turn\"> </div>\n    <div id=\"board\"></div>\n    <script src=\"https://cdn.ethers.io/lib/ethers-5.2.umd.min.js\" type=\"application/javascript\"></script>\n    <script>\n        (async () => {\n            // Replace with your contract address\n            const contractAddress = \"0x686497408BDB2Ed17D730bE91714C0168b957e9d\";\n\n            // Replace with your contract ABI\n            const contractABI = [\n                {\n                    \"inputs\": [\n                        {\n                            \"internalType\": \"address\",\n                            \"name\": \"_player2\",\n                            \"type\": \"address\"\n                        }\n                    ],\n                    \"stateMutability\": \"nonpayable\",\n                    \"type\": \"constructor\"\n                },\n                {\n                    \"anonymous\": false,\n                    \"inputs\": [\n                        {\n                            \"indexed\": false,\n                            \"internalType\": \"address\",\n                            \"name\": \"winner\",\n                            \"type\": \"address\"\n                        }\n                    ],\n                    \"name\": \"GameEnded\",\n                    \"type\": \"event\"\n                },\n                {\n                    \"inputs\": [\n                        {\n                            \"internalType\": \"uint8\",\n                            \"name\": \"_x\",\n                            \"type\": \"uint8\"\n                        },\n                        {\n                            \"internalType\": \"uint8\",\n                            \"name\": \"_y\",\n                            \"type\": \"uint8\"\n                        }\n                    ],\n                    \"name\": \"makeMove\",\n                    \"outputs\": [],\n                    \"stateMutability\": \"nonpayable\",\n                    \"type\": \"function\"\n                },\n                {\n                    \"anonymous\": false,\n                    \"inputs\": [\n                        {\n                            \"indexed\": false,\n                            \"internalType\": \"address\",\n                            \"name\": \"player\",\n                            \"type\": \"address\"\n                        },\n                        {\n                            \"indexed\": false,\n                            \"internalType\": \"uint8\",\n                            \"name\": \"x\",\n                            \"type\": \"uint8\"\n                        },\n                        {\n                            \"indexed\": false,\n                            \"internalType\": \"uint8\",\n                            \"name\": \"y\",\n                            \"type\": \"uint8\"\n                        }\n                    ],\n                    \"name\": \"TurnEnded\",\n                    \"type\": \"event\"\n                },\n                {\n                    \"inputs\": [\n                        {\n                            \"internalType\": \"uint256\",\n                            \"name\": \"\",\n                            \"type\": \"uint256\"\n                        },\n                        {\n                            \"internalType\": \"uint256\",\n                            \"name\": \"\",\n                            \"type\": \"uint256\"\n                        }\n                    ],\n                    \"name\": \"board\",\n                    \"outputs\": [\n                        {\n                            \"internalType\": \"uint256\",\n                            \"name\": \"\",\n                            \"type\": \"uint256\"\n                        }\n                    ],\n                    \"stateMutability\": \"view\",\n                    \"type\": \"function\"\n                },\n                {\n                    \"inputs\": [],\n                    \"name\": \"gameOver\",\n                    \"outputs\": [\n                        {\n                            \"internalType\": \"bool\",\n                            \"name\": \"\",\n                            \"type\": \"bool\"\n                        }\n                    ],\n                    \"stateMutability\": \"view\",\n                    \"type\": \"function\"\n                },\n                {\n                    \"inputs\": [],\n                    \"name\": \"getBoard\",\n
```

\"outputs\": [\n {\n \"internalType\": \"uint256[3][3]\",\n \"name\": \"\",\n \"type\": \"uint256[3][3]\"\n }\n],\n \"stateMutability\": \"view\",\n \"type\": \"function\"\n },\n {\n \"inputs\": [],\n \"name\": \"player1\",\n \"outputs\": [\n {\n \"internalType\": \"address\",\n \"name\": \"\",\n \"type\": \"address\"\n }\n],\n \"stateMutability\": \"view\",\n \"type\": \"function\"\n },\n {\n \"inputs\": [],\n \"name\": \"player1Turn\",\n \"outputs\": [\n {\n \"internalType\": \"bool\",\n \"name\": \"\",\n \"type\": \"bool\"\n }\n],\n \"stateMutability\": \"view\",\n \"type\": \"function\"\n },\n {\n \"inputs\": [],\n \"name\": \"player2\",\n \"outputs\": [\n {\n \"internalType\": \"address\",\n \"name\": \"\",\n \"type\": \"address\"\n }\n],\n \"stateMutability\": \"view\",\n \"type\": \"function\"\n }\n];\n\n // Create a new provider and set it as the default\n // Create a new provider and set it as the default\n const provider = new ethers.providers.JsonRpcProvider('https://rpc.ankr.com/polygon_mumbai');\n\n console.log('provider', provider);\n\n // Hardcoded private key\n // const privateKey = \"9b30a6fd1ce87cb530d2a51bb65c1233613e127c8efb5681d381621e34c68ae6\";\n const privateKey = \"be1d4269f7fa07e4f98c74a167ba5abefc899bc9242b31d12913e8f9948f273c\";\n\n const wallet = new ethers.Wallet(privateKey, provider);\n\n // Create a new contract instance\n const contract = new ethers.Contract(contractAddress, contractABI, wallet);\n\n console.log(contract);\n\n // Get players from contract\n const player1 = await contract.player1();\n const player2 = await contract.player2();\n\n console.log(player1, player2);\n\n const feedbackElement = document.getElementById(\"feedback\");\n const playerTurnElement = document.getElementById(\"player-turn\");\n // Initialize the board\n async function initializeBoard() {\n const gameBoard = await contract.getBoard();\n console.log(gameBoard);\n for (let x = 0; x < 3; x++) {\n for (let y = 0; y < 3; y++) {\n const cell = document.createElement(\"div\");\n cell.id = `cell-${x}-${y}`;\n cell.classList.add(\"cell\");\n\n const cellValue = gameBoard[x][y].toNumber(); // Convert BigNumber to number\n if (cellValue === 1) {\n cell.textContent = \"X\";\n } else if (cellValue === 2) {\n cell.textContent = \"O\";\n }\n\n document.getElementById(\"board\").appendChild(cell);\n\n // Add event listener to each cell\n cell.addEventListener(\"click\", async () => {\n // Check if it's the player's turn\n const isPlayer1Turn = await contract.player1Turn();\n if ((isPlayer1Turn && player1 !== wallet.address) || (!isPlayer1Turn && player2 !== wallet.address)) {\n feedbackElement.textContent = \"It's not your turn!\";\n return;\n }\n\n feedbackElement.textContent = \"Processing move...\";\n try {\n const [x, y] = cell.id.split(\"-\").slice(1).map(Number);\n const tx = await contract.makeMove(x, y, { gasLimit: 10000000 });\n await tx.wait();\n feedbackElement.textContent = \"Move successful!\";\n } catch (error) {\n console.error(error);\n feedbackElement.textContent = \"Move failed!\";\n }\n });\n }\n }\n\n await initializeBoard();\n\n // Listen for TurnEnded events\n contract.on(\"TurnEnded\", (player, x, y) => {\n // Update the UI with the new move\n console.log('player turn', player);\n const cell = document.getElementById(`cell-${x}-${y}`);\n updatePlayerTurn()\n cell.textContent = player === player1 ? \"X\" : \"O\";\n });\n\n // Listen for GameEnded events\n contract.on(\"GameEnded\", (winner) => {\n // Announce the winner\n alert(`Game Over! The winner is ${winner === player1 ? \"Player 1\" : \"Player 2\"}`);\n });\n\n // Update the player turn element\n async function updatePlayerTurn() {\n const isPlayer1Turn = await contract.player1Turn();\n const currentPlayer = isPlayer1Turn ? \"Player 1\" : \"Player 2\";\n console.log(isPlayer1Turn);\n playerTurnElement.innerText = `Current turn: ${currentPlayer}`;\n }\n\n // Update the player turn element initially\n await updatePlayerTurn();\n\n // Listen for TurnEnded events to update player turn\n contract.on(\"TurnEnded\", () => {\n updatePlayerTurn();\n });\n })();\n </script>\n</body>\n\n</html>",
"output": "str",
"x": 490,

```
        "y": 520,
        "wires": [
            [
                "68d535bcbdaadb3c"
            ]
        ]
    },
    {
        "id": "d91bb0c6e4dcc391",
        "type": "http in",
        "z": "1b72e8d575aef462",
        "name": "",
        "url": "/test",
        "method": "get",
        "upload": false,
        "swaggerDoc": "",
        "x": 310,
        "y": 520,
        "wires": [
            [
                "f400539731192b8a"
            ]
        ]
    },
    {
        "id": "68d535bcbdaadb3c",
        "type": "http response",
        "z": "1b72e8d575aef462",
        "name": "",
        "statusCode": "200",
        "headers": {},
        "x": 690,
        "y": 520,
        "wires": []
    }
]
```


A. Input your unique contract address.
B. Enter the private key of your private key.
C. Provide your blockchain RPC-URL endpoint / API key.
D. Share the link <integration-studio-api-url>/test with your partner! Who will emerge victorious?

Upon completion of a round against another player, please answer the following questions in the group chat:

- What are the names of the contract events and how are we monitoring them?
- How does the contract manage turns?
- Can you explain how we established a connection to the smart contract, how we invoked its functions, and how we read its variables?"

Please note the UI might need to be refreshed between turns

Slide 121: Simplified Bank Contract

Training Task: Simple Bank System

A -

Learning Goals:

- Understand how to define and use Solidity variables, functions, mappings, and modifiers.
- Get familiar with basic contract design in Solidity.
- Learn to manage Ethereum transactions with Solidity.

Task Description:

Create a simple Bank contract with the following features:

1. Account creation: A user can create a bank account with their Ethereum address.
2. Deposit: A user can deposit Ether into their account.
3. Withdraw: A user can withdraw Ether from their account, but cannot withdraw more than their balance.
4. Balance check: A user can check the balance of their account.

Task Steps:

1. Define an Account struct that contains an id, owner, and balance.
2. Create a mapping that associates an Ethereum address with an Account.
3. Create a modifier onlyAccountOwner that allows only the owner of an account to execute certain functions.
4. Implement a function createAccount that allows a user to create an account.
5. Implement a function deposit that allows a user to deposit Ether into their account.
6. Implement a function withdraw that allows a user to withdraw Ether from their account.
7. Implement a function getBalance that allows a user to check the balance of their

The starter contract:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

contract SimpleBankSystem {
    struct Account {
        uint256 id;
        address owner;
        uint256 balance;
    }
```

```
    mapping(address => Account) public accounts;

    // Modifiers
    modifier onlyAccountOwner(address owner) {
        // TODO
        _;
    }

    // TODO: Implement the createAccount function

    // TODO: Implement the deposit function

    // TODO: Implement the withdraw function

    // TODO: Implement the getBalance function
}
```

Hints:

1. createAccount should initialize an Account and add it to the accounts mapping.
2. deposit should update the balance of the sender's Account and should be payable to accept Ether.
3. withdraw should decrease the balance of the sender's
4.

B -

Training Task: Bank System with Loan Management

Learning Goals:

- Further understand how to use Solidity structs and mappings.
- Learn to manage complex functions and operations within a contract.
- Learn how to handle more complex conditions and transactions.

Task Description:

After implementing the features of the bank system in Step A, extend the contract with the following loan management features:

1. Loan Creation: A user can create a loan associated with their account.
2. Loan Repayment: A user can repay the loan.
3. Outstanding Loan Balance Check: A user can check their outstanding loan balance.

Task Steps:

1. Define a Loan struct that contains an id, accountId, amount, and outstandingBalance.
2. Create a mapping that associates a loan id with a Loan.

3. Implement a function createLoan that allows a user to create a loan associated with their account. For simplicity, the initial loan amount is the outstanding balance.
4. Implement a function repayLoan that allows a user to repay their loan. The repaid amount should be subtracted from their account balance and the loan's outstanding balance.
5. Implement a function getOutstandingBalance that allows a user to check the outstanding balance of their loan

Hints:

1. createLoan should initialize a Loan and add it to the loans mapping.
2. repayLoan should decrease the balance of the sender's Account and the loan's outstanding balance by the repayment amount. Remember to ensure the account balance is sufficient for the repayment amount.
3. getOutstandingBalance should return the outstanding balance of the specified loan.

Course 2 - Part 2

Slide 24: Hardhat

In the previous section, we leveraged Hardhat to deploy contracts, albeit without delving into the intricate details. In this section, we will thoroughly explore Hardhat, learning about debugging, deploying, and testing contracts, among other things.

Step 1: Contract Deployment - Debugging the Initial Issue

1.1 Create a new file named 'HardHatDemo.sol' in the smart contract folder and input the following contract:

```
contract HardHatDemo {
  uint256 public counter;
  constructor(uint256 initialCounterValue) {
    counter = initialCounterValue;
  }
  function add() public {
    counter++;
  }
  function decrease() public {
    counter--;
  }
```

```
}
```

Hint: what does the error say?

Step 2: Deploying with Hardhat Deploy Syntax

2.1 If you're working locally, ensure the hardhat-deploy plugin is installed by running npm install hardhat-deploy.

2.2 Subsequently, create a new deployment script in the 'deploy' folder with the following content:

```
// deploy/01_deploy_hardhat_demo.ts
const { deploy } = require("hardhat-deploy");

module.exports = async ({ getNamedAccounts, deployments }) => {
  // Get the deployer account from the named accounts
  const { deployer } = await getNamedAccounts();

  // Destructure the deploy function from the deployments object
  const { deploy } = deployments;

  // Deploy the HardHatDemo contract
  await deploy("contract name", {
    from: deployer, // Specify the deployer's address
    args: [], // Specify the initial counter value here
    log: true, // Log deployment details to the console
  });
};

module.exports.tags = ["deployment tag"];
```

2.3 Explanation of the script:

A. The script imports the deploy function from the hardhat-deploy package. The module.exports line defines the script as the default export of the module. The script is an async function that accepts an object with two properties: getNamedAccounts and deployments. These properties are provided by the Hardhat deployment process.
B. const { deployer } = await getNamedAccounts(); extracts the deployer property from the object returned by the getNamedAccounts function. This represents the address of the deployer account.
C. const { deploy } = deployments; extracts the deploy function from the deployments object. This function is used to deploy contracts.

D. await deploy("HardHatDemo", { ... }); deploys the "HardHatDemo" contract using the deploy function.
E. from: deployer specifies the address of the deployer account.

F. args: [10] specifies an array of arguments to be passed to the contract's constructor. In this case, it provides the initial counter value of 10.
G. log: true enables logging of deployment details to the console.
H. module.exports.tags = ["HardHatDemo"]; sets the tags property of the module's exports to ["HardHatDemo"]. This associates the "HardHatDemo" tag with the deployment script.
I. The script deploys the "HardHatDemo" contract with the specified initial counter value using the deploy function. The script is executed during the Hardhat deployment process.

2.4 To use this script:

A. Place the script in the 'deploy' folder of your Hardhat project.
B. Modify the 'args' field to specify the desired initial counter value.
C. Run the command:

npx hardhat deploy --network <network-name>,

where <network-name> is the desired network specified in your Hardhat configuration file (hardhat.config.js).

D. Run the command

npx hardhat deploy --network {network name} --tags {tag name}

Step 3: Deploying from the Scripts Folder

3.1 Create a new script in the 'Scripts' folder. Let's call it 'deployHardHatScript.ts'.

```
const { ethers } = require("hardhat");
async function main() {
  const [deployer] = await ethers.getSigners();
  const HardHatDemo = await ethers.getContractFactory("contract name");
  const contract = await HardHatDemo.connect(deployer).deploy(arg);
  console.log(contract);
  await contract.decrease();
  console.log("New count:", (await contract.counter()).toString());
}
main()
  .then(() => process.exit(0))
  .catch((error) => {
    console.error(error);
    process.exit(1);
  });
```

To run the deployment, execute the command

npx hardhat run scripts/01_deploy_hardhat_demo.js --network <network-name>,

```
```

where <network-name> is the desired network specified in your Hardhat configuration file (hardhat.config.js).

Step 4: Tasks to Increment and Decrement the Counter

4.1 Create two tasks, 'increment' and 'decrement', in the 'scripts' folder.

For the 'increment' task (scripts/increment-counter.js), use the following syntax and adjust it accordingly:

```
task("say-hello", "Prints a greeting to the console")
  .addOptionalParam("name", "The name of the person to greet")
  .setAction(async (taskArgs, hre) => {
    if (taskArgs.name) {
      console.log(`Hello, ${taskArgs.name}!`);
    } else {
      console.log("Hello, world!");
    }
  });
```

Paste the following require statements into the hardhat config file:

```
require('./tasks/decrement-counter');
require('./tasks/increment-counter');
```

To run the tasks, execute the commands:

```
npx hardhat run increment-counter.ts --network <network-name>

npx hardhat run decrement-counter.ts --network <network-name>
```

Replace <network-name> with the desired network specified in your Hardhat configuration file (hardhat.config.js).

Step 5: Writing and Running a Simple Test

5.1 Create a test file in the 'test' folder, such as 'hardhatdemo-counter-test.ts', and paste in it the script from /training-program/counter-test.ts.

```
import { ethers } from "hardhat";
import { Contract, Signer, ContractFactory } from "ethers";
import { expect } from "chai";
describe("HardHatDemo Contract", function() {
  let HardHatDemoContractFactory: ContractFactory;
  let hardHatDemoContractInstance: Contract;
  let owner: Signer;
  beforeEach(async function() {
    [owner] = await ethers.getSigners();
    HardHatDemoContractFactory = await ethers.getContractFactory("HardHatDemo");
    hardHatDemoContractInstance = await HardHatDemoContractFactory.deploy(0);
    await hardHatDemoContractInstance.deployed();
  });
  it("Should initialize counter correctly", async function() {
    expect((await hardHatDemoContractInstance.counter()).toNumber()).to.equal(0);
  });
  it("Should increment the counter", async function() {
    await hardHatDemoContractInstance.connect(owner).add();
    expect((await hardHatDemoContractInstance.counter()).toNumber()).to.equal(1);
  });
  it("Should decrement the counter", async function() {
    await hardHatDemoContractInstance.connect(owner).add();
    await hardHatDemoContractInstance.connect(owner).decrease();
    expect((await hardHatDemoContractInstance.counter()).toNumber()).to.equal(0);
  });
});
```

To run the test, execute the command:

```
npx hardhat test.
```

Step 6: Deploying to a Testnet

Step 7: Verifying the Contract

Note: If not installed, install the Hardhat Etherscan plugin (it’s installed on the settlement platform).

Import hardhat-etherscan:

```
import "@nomiclabs/hardhat-etherscan";
```

Get API keys from the block explorer and place the keys inside the Config object in the hardhat.ts file:https://docs.polygonscan.com/getting-started/viewing-api-usage-statistics

```
const config: HardhatUserConfig = {
  // other configurations...
  etherscan: {
    apiKey: "Your Etherscan API key",
  },
};
```

Run the command in your terminal:

```
npx hardhat verify --network NETWORK_NAME DEPLOYED_CONTRACT_ADDRESS "Constructor argument 1"
```

Congratulations, you have verified your contract!

Step 8: Changing Network

You can easily switch the network for which you are deploying your contract using the same hardhat.config file. All you have to do is to add the network inside the hardhat config object, and deploy the contract to that network using your preferred deployment method.

```
const config: HardhatUserConfig = {
  // other configurations...
  networks: {
    ether-network-name: “RPC url”,
  },
};
```

Then deploy contracts to it

```
npx hardhat run increment-counter.ts --network <ether-network-name>
```

This concludes our comprehensive guide on using Hardhat for contract deployment, debugging, and testing. By following these steps, you should now have a solid understanding of how to use Hardhat in your blockchain development workflow. Remember, practice is key when it comes to mastering these concepts, so don't hesitate to experiment and try out different things.

slide 25 - Debugging Car rental contract:

Introduction

You are given a Solidity contract for a car rental service. The contract has several bugs and it's your job to find and fix them. The contract allows for the following:

- A contract owner (the car rental service) can add cars to the inventory.
- Users can view available cars, rent a car by paying a deposit, and specifying a rental period.
- Users can return the car and the contract calculates any penalties based on the delay in returning the car.
- The contract owner can withdraw the payments collected.

However, the contract is not working as expected due to several bugs. Your task is to identify these bugs and fix them.

Contract Code

Here is the buggy contract code:

https://github.com/SaeeDawod/training-program/blob/master/course2-part2/debugging/car-loan-bugged.sol

```
contract CarRental {
    struct Car {
        string name;
        uint256 deposit;
        bool rented;
    }

    struct Rental {
        uint256 carId;
        string rentalPeriod; // BUG 1
        uint256 startTimestamp;
        bool active;
    }

    mapping(uint256 => Car) public cars;
    mapping(address => Rental) public rentals;
    uint256 public carCounter = 0;
    address public owner;
    bool public stopped = false;

    modifier onlyOwner {
        require(msg.sender == owner, "Only the contract owner can call this function.");
        _;
    }

    constructor() {
        owner = msg.sender; // The contract deployer becomes the owner.
```

```
    }

    function addCar(string memory _name, uint256 _deposit) public onlyOwner {
        cars[carCounter] = Car(_name, _deposit, false);
        carCounter++;
    }

    function rentCar(uint256 _carId, string memory _rentalPeriod) public payable { // BUG 2
        require(cars[_carId].rented == false, "Car is already rented.");
        require(msg.value == cars[_carId].deposit, "Must pay the deposit amount.");
        rentals[msg.sender] = Rental(_carId, _rentalPeriod, block.timestamp, true); // BUG 3
        cars[_carId].rented = true;
    }

    function approveRental(address _renter) public { // BUG 4
        Rental storage rental = rentals[_renter];
        require(rental.active, "No active rental found.");
        rental.active = false;
        cars[rental.carId].rented = false;
    }

    function returnCar() public payable {
        Rental storage rental = rentals[msg.value]; // BUG 5
        require(rental.active, "No active rental found.");
        uint256 penalty = rental.rentalPeriod * 1 ether; // BUG 6
        require(msg.value == penalty, "Must pay the penalty.");
        rental.active = false;
        cars[rental.carId].rented = false;
    }
}
```

Instructions

1. Read and understand the contract code.
2. Identify the bugs in the contract. There are subtle issues with types of variables, visibility of functions, and more.
3. Fix the bugs and ensure the contract works as expected.

Hints

1. Check the types of all variables. Are they appropriate for their intended use? //BUG 1
2. Look at the visibility modifiers for the functions. Are they correct? //BUG 4
3. Pay attention to the calculation of the penalty in the returnCar function. //BUG 6
4. Look carefully at the rentCar function. //BUG 2, BUG 3
5. Don't forget to check the modifiers used in the functions.

Slide 34: Capture the flag

Let's run some Ethernanut CTF together!

https://ethernaut.openzeppelin.com/

Slide 59: Multisig Wallet

Task: Creating a Simple Multisig Wallet in Solidity

Objective: Write a simple multisig wallet contract in Solidity.

Instructions:

1. Define the contract and set the number of required signatures and the owners of the wallet.
2. Create a struct for transactions that includes the destination address, the value of the transaction, and a boolean to indicate whether the transaction has been executed.
3. Implement a function to submit a transaction. This function should add the transaction to a pending transactions array and emit an event.
4. Implement a function for owners to confirm transactions. This function should check that the sender is an owner and that they have not already confirmed the transaction.
5. Implement a function to execute a transaction once the required number of confirmations has been reached. This function should send the specified value to the destination address and mark the transaction as executed.
6. Add additional owners
7. Test your contract with different participants and scenarios to ensure it works as expected.

Slide 61: Proxy

Task: Creating and Upgrading a Simple UUPS Contract using Hardhat

Objective: Write a simple storage contract in Solidity, deploy it using Hardhat, and then upgrade it using a UUPS proxy contract.
Instructions:

Import the plugin in your Hardhat config: Add the following line to your hardhat.config.js file:

```
require('@openzeppelin/hardhat-upgrades');
```

Write your contracts: Define a Box contract and a BoxV2 contract in separate Solidity files.

The Box contract should have a state variable to store a single value and a function to retrieve that value. Here's an example:

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

contract Box {
    uint private _value;

    function setValue(uint newValue) public {
        _value = newValue;
    }

    function getValue() public view returns (uint) {
        return _value;
    }
```

```
}
```

The BoxV2 contract should be an upgraded version of Box with additional functionality. For example, you could add a function to double the stored value:

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

contract BoxV2 {
    uint private _value;

    function setValue(uint newValue) public {
        _value = newValue;
    }

    function getValue() public view returns (uint) {
        return _value;
    }

    function doubleValue() public {
        _value = _value * 2;
    }
}
```

Deploy the initial contract: Write a Hardhat script to deploy your Box contract using the deployProxy function from the upgrades plugin. Here's an example:

```
const { ethers, upgrades } = require("hardhat");

async function main() {
    const Box = await ethers.getContractFactory("Box");
    const box = await upgrades.deployProxy(Box, [42]);
    await box.deployed();
    console.log("Box deployed to:", box.address);
}

main()
    .then(() => process.exit(0))
    .catch(error => {
```

```
    console.error(error);
    process.exit(1);
  });
```

1. **Interact with the contract**: Write a Hardhat script to interact with your Box contract. Call the setValue function to change the value stored in the contract, and then call the getValue function to verify the updated value.
2. **Upgrade the contract**: Write a Hardhat script to upgrade your Box contract to BoxV2 using the upgradeProxy function from the upgrades plugin. Here's an example:

```
const { ethers, upgrades } = require("hardhat");

async function main() {
  const BoxV2 = await ethers.getContractFactory("BoxV2");
  const upgradedBox = await upgrades.upgradeProxy(existingProxyAddress, BoxV2);
  console.log("Box upgraded to:", upgradedBox.address);
}

main()
  .then(() => process.exit(0))
  .catch(error => {
    console.error(error);
    process.exit(1);
  });
```

Course 2 - Part 3

For the following tasks feel free to use any tool at your disposal in the Settlemint platform

Tutorial: Introduction to Merkle Trees, Zero-Knowledge Proofs, and zkRollups

Introduction

In this tutorial, we will delve into the concepts of Merkle Trees, Zero-Knowledge Proofs (ZKPs), and zkRollups. We will adopt a hands-on approach by developing a basic application that incorporates these principles.

Our practical exercise involves creating an application that utilizes Merkle Trees to whitelist users, enabling them to mint a "point" without disclosing user information to the public. Only users who know their "secret" can generate a hash that can be abstracted from the root hash of the Merkle Tree.

Zero-Knowledge Proofs
Zero-Knowledge Proofs are a cryptographic technique where one party (the prover) can prove to another party (the verifier) that they know a value x, without conveying any information apart from the fact that they know the value x.

zkRollups
zkRollups are a Layer-2 scaling solution that employs Zero-Knowledge Proofs to bundle multiple off-chain transactions into a single on-chain proof that can be verified by anyone. They provide a solution to Ethereum's scalability issues by enabling more transactions to be processed off-chain while still maintaining on-chain security.

Merkle Trees
A Merkle Tree is a binary tree of hashes. It begins with the "leaves" at the bottom, which are hashes of data blocks. Each pair of leaves is then hashed together to form the next level up in the tree. This process is repeated until we reach the tree's root, also known as the Merkle Root. This structure allows us to create a "proof" that a certain piece of data is in the tree, without needing to know all the other data in the tree. This proof is simply the minimum set of hashes required to compute the Merkle Root.
The "top hash" is also referred to as "the root hash".
If you're still confused, take a moment to watch this explanatory video: https://www.youtube.com/watch?v=fB41w3JcR7U

Prerequisites:

- Basic knowledge of the SettleMint platform
- Deployed network and node
- Basic understanding of Smart Contracts and how to deploy them
- Familiarity with ethers.js

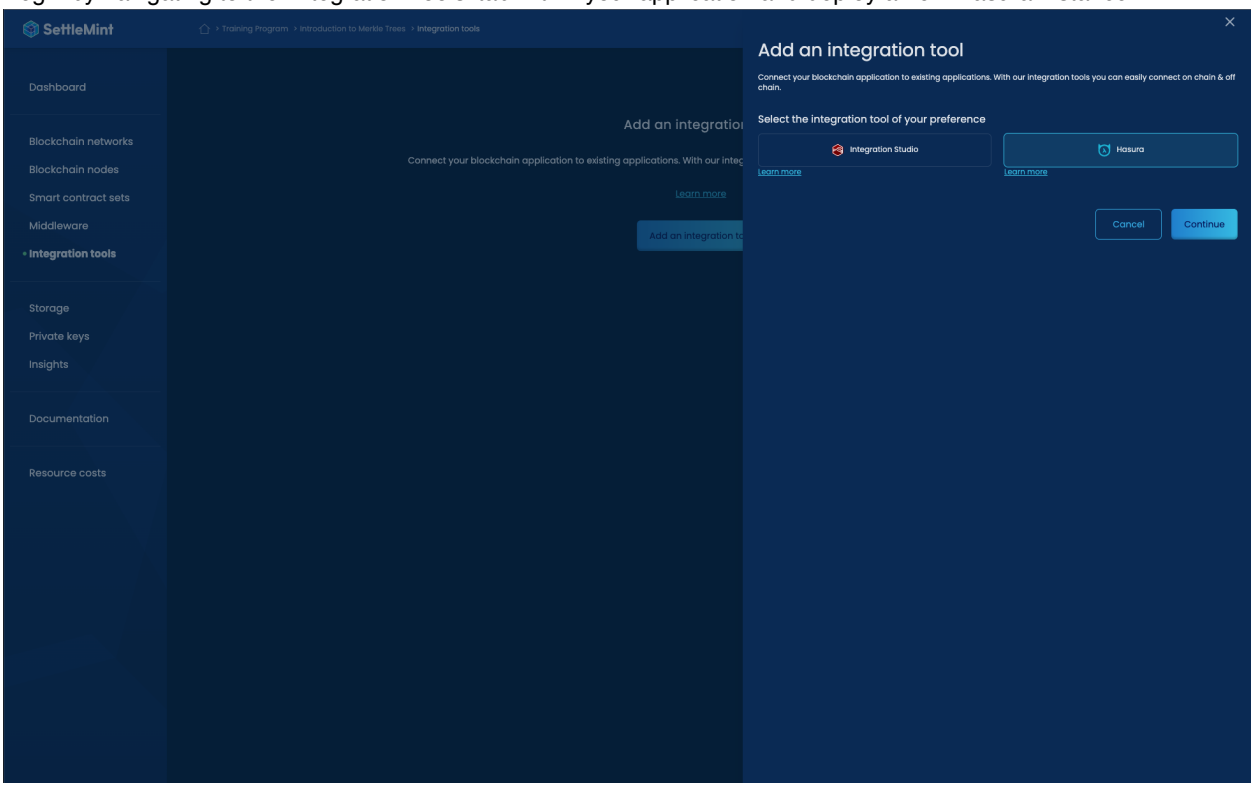
Tools:
1 - Integration Studio
2 - Hasura

Steps:

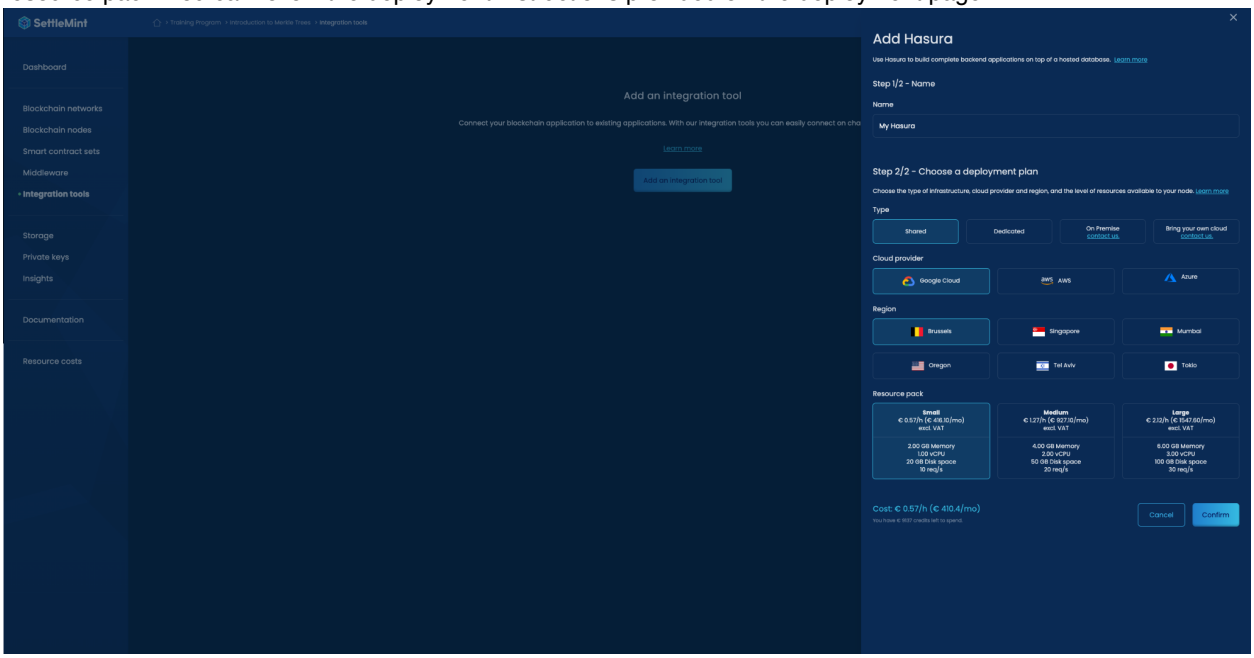
- Create a users table in Hasura.
- Create a deterministic way of hashing the data in Integration Studio.
- Create a smart contract that stores the RootHash and checks the leaves against it.
- Develop helper functions in Integration Studio that produce the proof and leaves to be checked against the RootHash in the smart contract.

Step 1: Create users table in Hasura

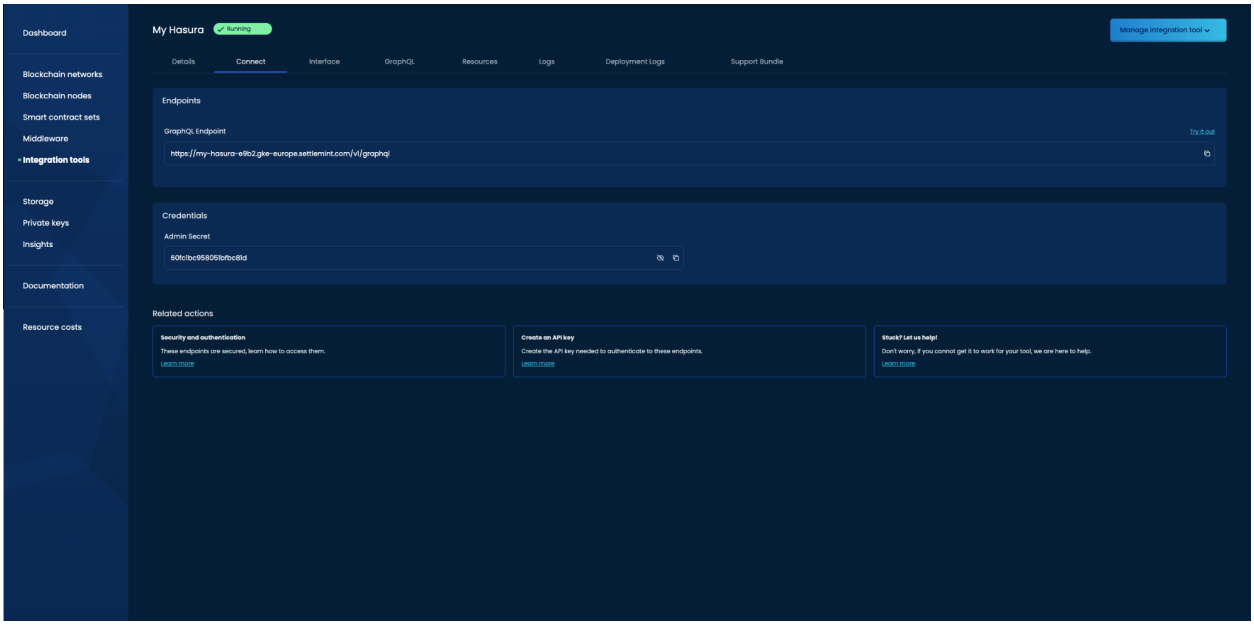
a. Begin by navigating to the 'Integration Tools' tab within your application and deploy a new Hasura instance.



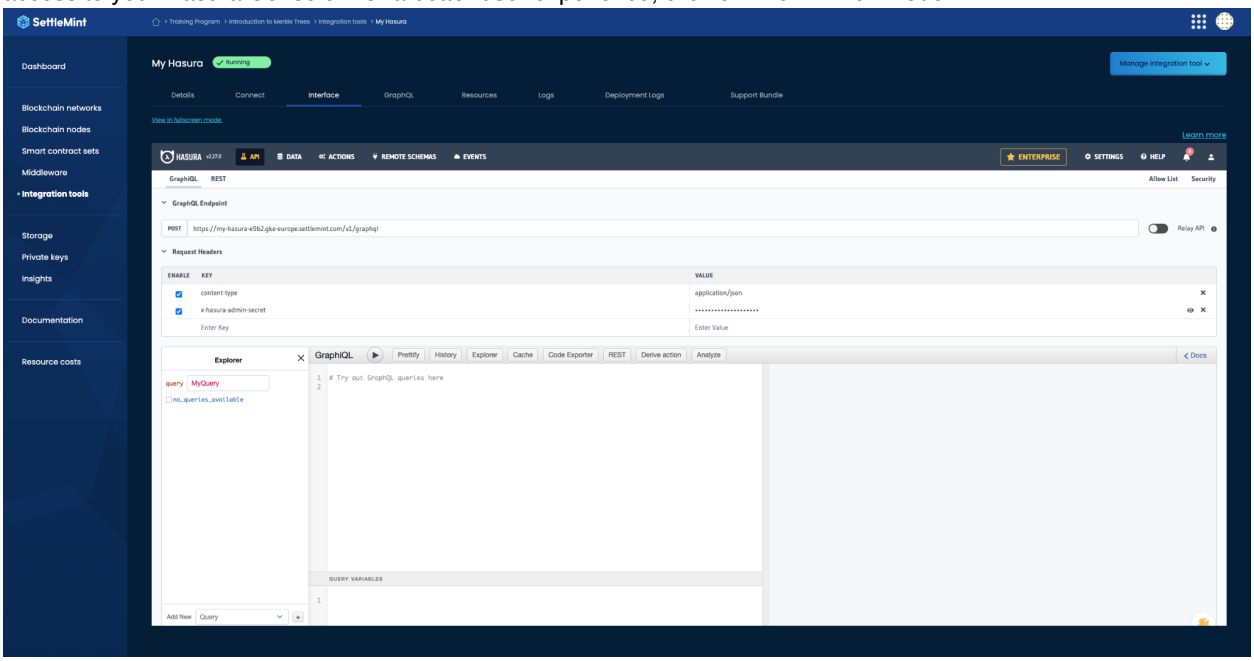
b. For the purpose of this demonstration, we will deploy the instance using the shared plan with a small resource pack. You can follow the deployment instructions provided on the deployment page.



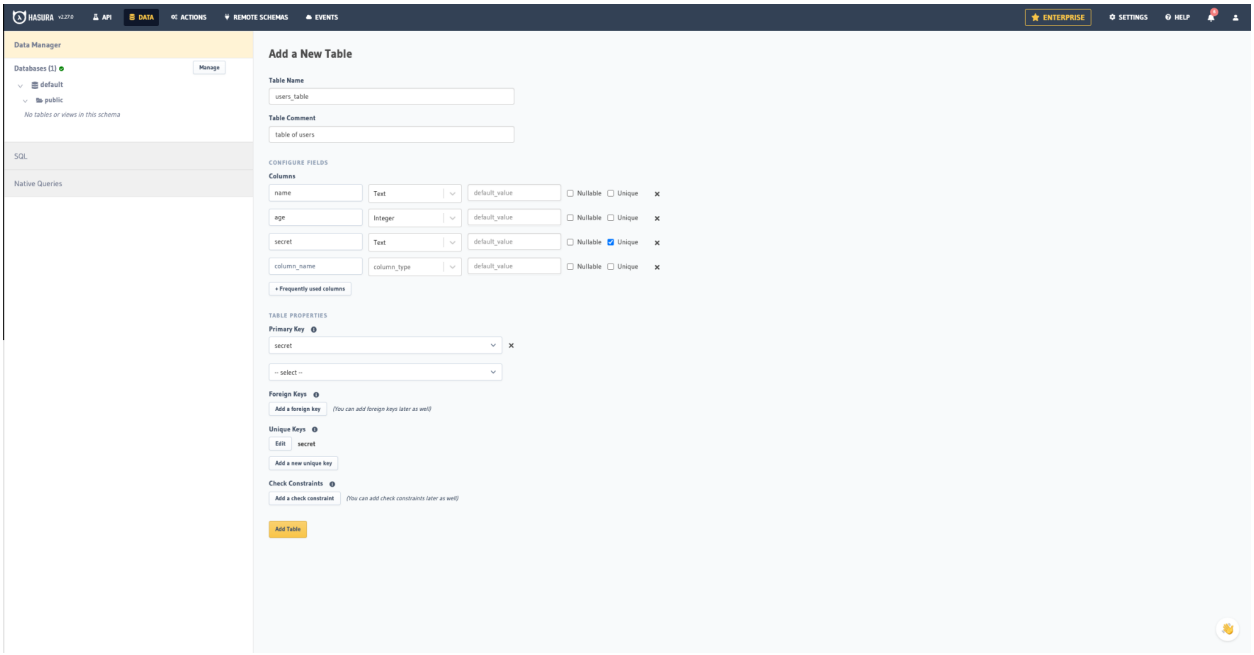
c. Once Hasura is deployed, click on the Hasura instance and navigate to the 'Connection' tab. Here, copy the Hasura admin secret.



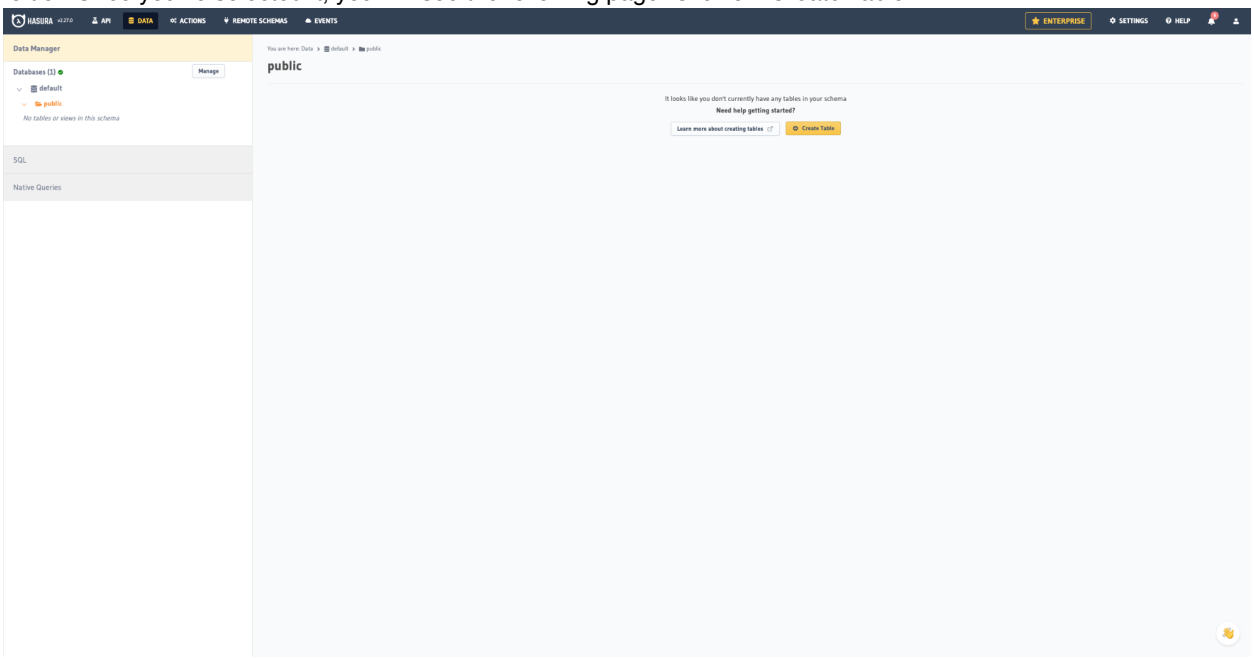
d. Next, navigate to the 'Interface' tab and paste the admin secret. Upon entering the secret, you will gain access to your Hasura Console. For a better user experience, click on 'View in Full Mode'.



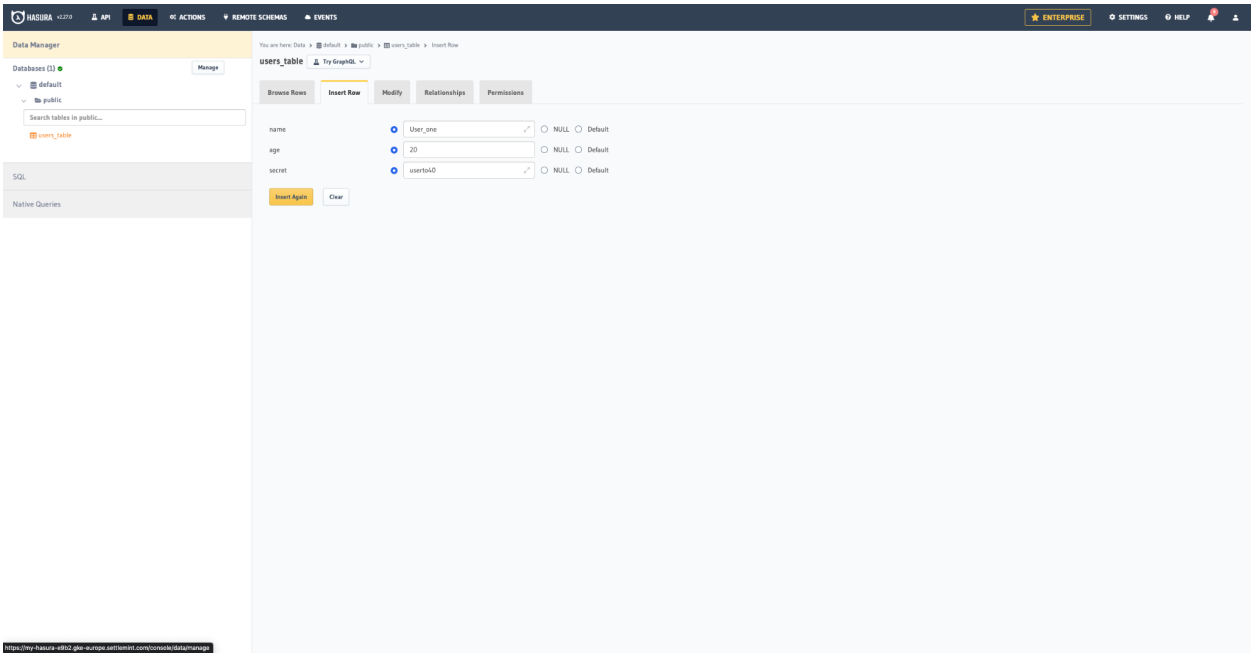
e. Hasura allows you to connect to almost any existing SQL database. However, for simplicity, we will create a new table using the Hasura Console.



f. Navigate to the 'Data' tab on the top navigation bar. In the left menu under 'Data Manager', select the 'public' folder. Once you've selected it, you will see the following page. Click on 'Create Table'.



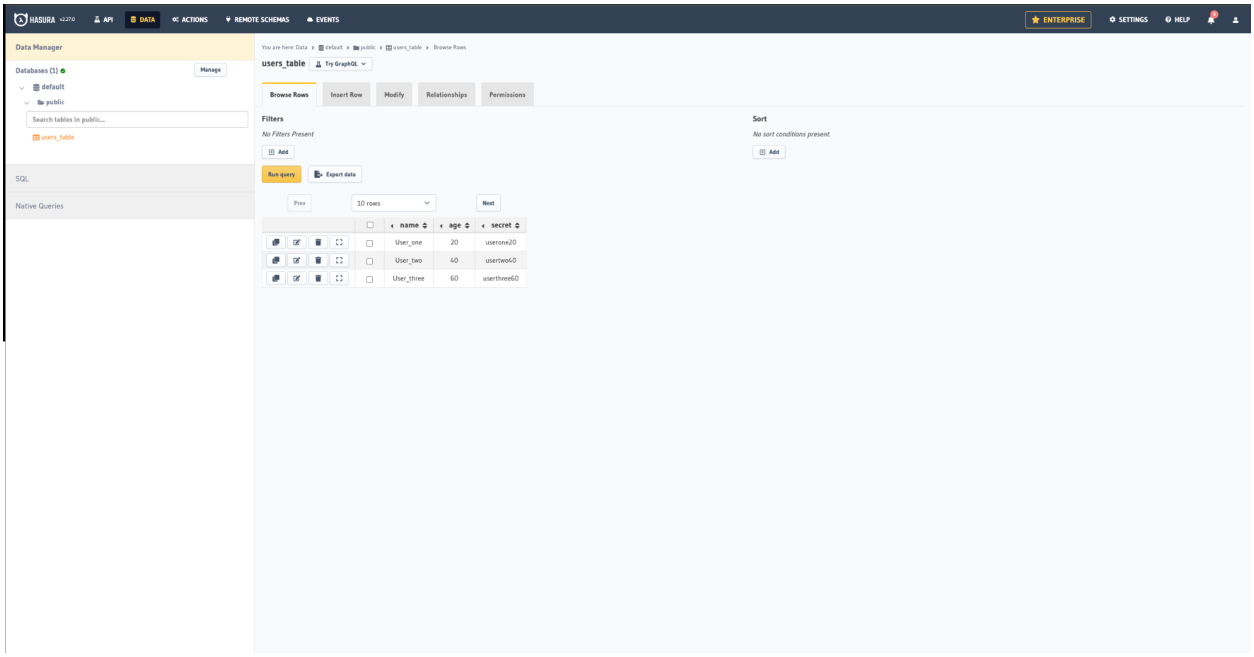
g. Configure the table as shown below and click 'Add Table' when you're done.



h. Take a moment to navigate through the different pages. In the 'Modify' tab, you can edit the table. The 'Browse Rows' tab allows you to view the tables. The 'Permission' tab helps you manage access control.

i. When you're ready, visit the 'Insert Row' tab and add three users. Ensure that the secret is something easy for you to remember as we will use it later on.

j. If you visit the 'Browse Rows' tab, you will see the three new users that we added.



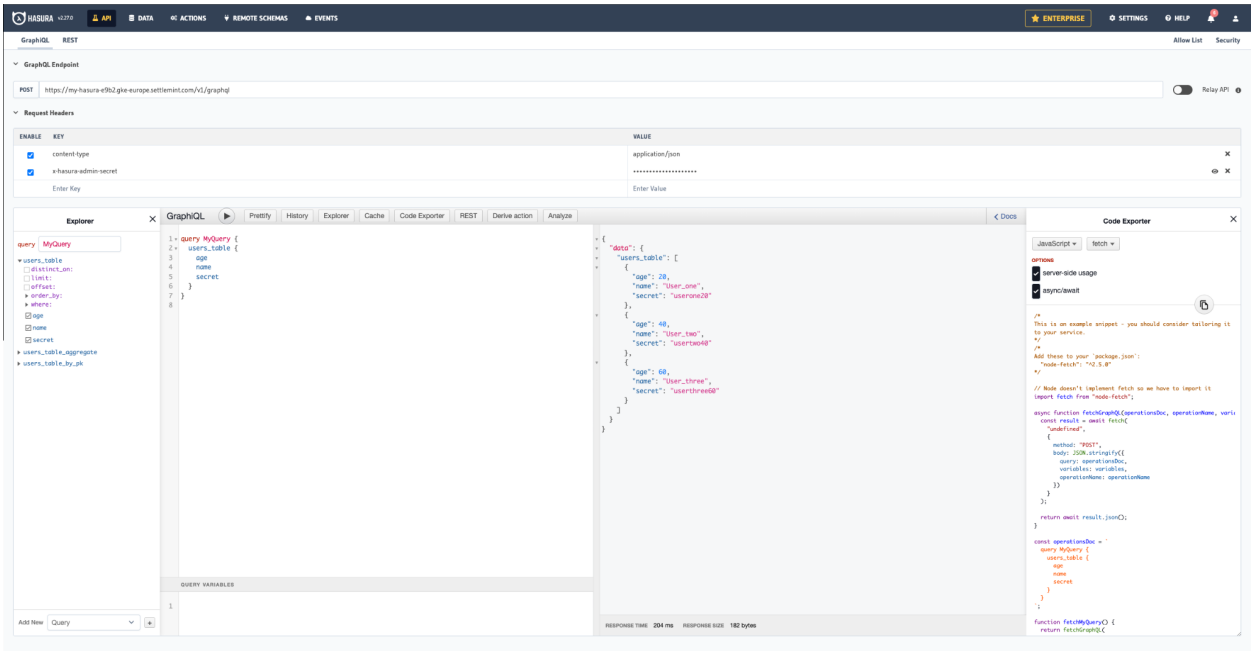
k. Congratulations, your database is now ready!

Step 1.5: Exporting the Queries from Hasura

In this step, we will export the queries from Hasura for use in Integration Studio in Step 2. We aim to create a function that fetches the data, aggregates it, and then uses the MerkleJS library to generate a Root hash.

To import the data from Hasura, we will need to create an API key. You can follow this guide to do so: [API Keys Guide](https://console.settlemint.com/documentation/docs/using-platform/api-keys/)

Next, get the code from the Hasura code generator. You can do this by building the query in the explorer tab on the left, and then clicking the “Code Exporter” button.



You will need to add the Headers field to the fetchGraphQL function in order to authenticate the requests.

```
headers: {
  "Content-Type": "application/json",
  "x-hasura-admin-secret": "Hasura admin key",
  "x-auth-token":"Settlemint API key"
},
```

Your fetchGraphQL function should look like this:

```
async function fetchGraphQL(operationsDoc, operationName, variables) {
  const result = await fetch(
    "YOUR HASURA URL",
    {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
        "x-hasura-admin-secret": "Your Hasura admin key",
        "x-auth-token":"Your Settlemint API key"
      },
      body: JSON.stringify({
        query: operationsDoc,
        variables: variables,
        operationName: operationName
      })
    }
  );
```

```
  return await result.json();
}
```

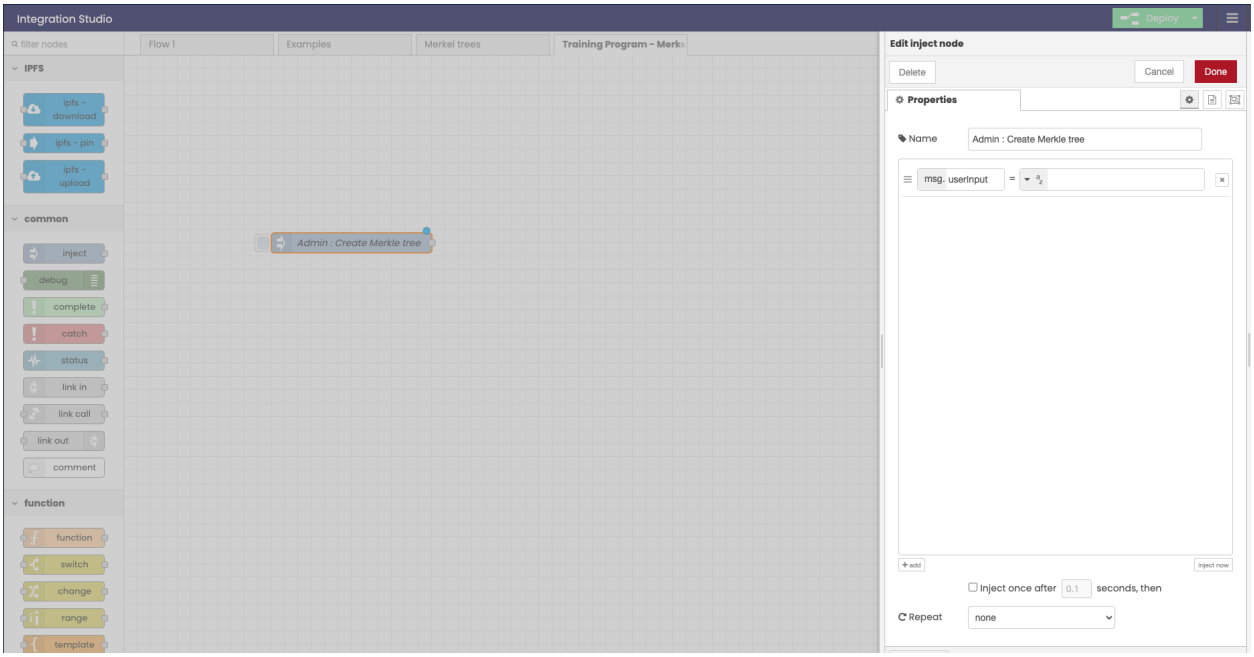
Remember to replace "YOUR HASURA URL", "Your Hasura admin key", and "Your Settlemint API key" with your actual information.

Part 2: Integration Studio- Admin Flow

a. Deploy an instance of Integration Studio in the same manner you deployed Hasura.

b. Create a function node that, when triggered (using inject), returns the root hash.

To do this, navigate to your new Integration Studio flow and drag an "inject" node to the dashboard. Double click on it and rename it to "Admin: Create Merkle Tree". This will be used to trigger the subsequent function node.



2. Add a function node that fetches data from Hasura and creates Merkle trees.

2.1 Explanation:

Next, we will add a function node to our Integration Studio flow. This function node will fetch data from our Hasura database using the script we created. It will extract each user's age, name, and secret, combine them into a single string, and hash that string.

The hashed strings will then be used to create a root hash to be hosted on the blockchain. This allows us to verify if our user's "secret" or any other information about them is correct without exposing their information to the blockchain.

Let's dive deeper into how this is done:

2.1. Before we prepare the script that will run the JavaScript we copied in Step 1.5 "export queries from Hasura", it's important to understand the process. After that, we will provide you with the code snippets!

We will use the code from the Hasura code exporter to fetch the information from the database. We will get an object that looks like this:

```
{
  users_table: [
    { age: 20, name: 'User_one', secret: 'userone20' },
    { age: 40, name: 'User_two', secret: 'usertwo40' },
    { age: 60, name: 'User_three', secret: 'userthree60' }
  ]
}
```

Our goal now is to create a hash for each user using their information. We will do this by extracting the values of each user, combining them into a single string, and then hashing it with keccak256.

After fetching the database, we will parse it with the following function. This function simply takes the values of the object and creates a single keccak hash from them:

```
function organizeAndHashUsers(users) {
    const hashes = [];
    for (const user of users) {
        const { age, name, secret } = user;
        const dataToHash = age.toString() + name + secret;
        const hash = keccak256(dataToHash).toString('hex');
        hashes.push(hash);
    }
    return hashes
}
```

So, the object:

```
{ age: 20, name: 'User_one', secret: 'userone20' }
```

Will become:

20User_oneuserone20

Then we will run this string in the keccak256 function (imported from the keccak256 - ethers library):

keccak256(20User_oneuserone20)

This will result in something like this:

41bb1e44bc70ec73abff22e9968938cf52f18cfc15cd3cc28fa49873ed7d5a79

To match the solidity keccak256, we will add the prefix "0x". So, the end result will be a hash looking like:

0x41bb1e44bc70ec73abff22e9968938cf52f18cfc15cd3cc28fa49873ed7d5a79

For each user in the database, we will have a keccak256 hash representing them in an array.

```
[
  '41bb1e44bc70ec73abff22e9968938cf52f18cfc15cd3cc28fa49873ed7d5a79',
  'dce01d42480af923fab205a2e4291d4803291389410a084504b786e4fdcf8b61',
  '4007e85b723e582d22cc6061558c51eaf16719a19ddb95a749f33375f6ee0080'
]
```

From this array, we will derive a root hash using the Merkle tree algorithm. Our end result will be a 32-byte hash that we will deploy on the blockchain and use to check if our user is allowed to run the function "getPoint".

Merkle Tree JS Example:

```
const addresses = msg.addresses
let leaves = addresses.map(keccak256)
let merkleTree = new merkletreejs.MerkleTree(leaves, keccak256, { sortPairs: true })
let rootHash = merkleTree.getRoot().toString('hex')

global.set("merkleTree", merkleTree);
global.set("rootHash", '0x' + rootHash);

msg.payload = '0x' + rootHash

return msg
```

The root hash would look something like this:

0x4fcbd8c9f9b5f45d45f2a9fb150a4e696158e621593e45cb4a7877d094dc6a08

Now that we have explained the functions and concept, we are ready for the next step.

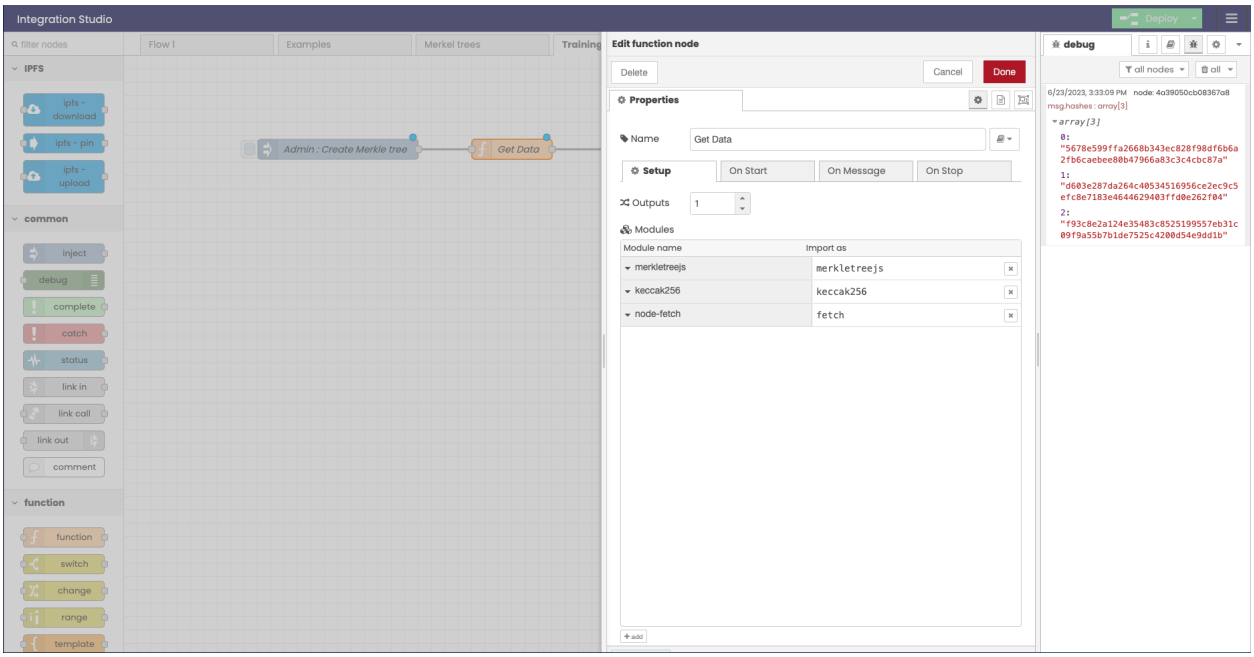
2.2 Create a "GetData" Function Node

1. Drag a new function node onto the workspace.

2. Double-click on it and rename it to "GetData".

3. After naming the function node, you need to import the following libraries. You can do this by double-clicking on the function node:

   a. Keccak256
   b. Node-fetch



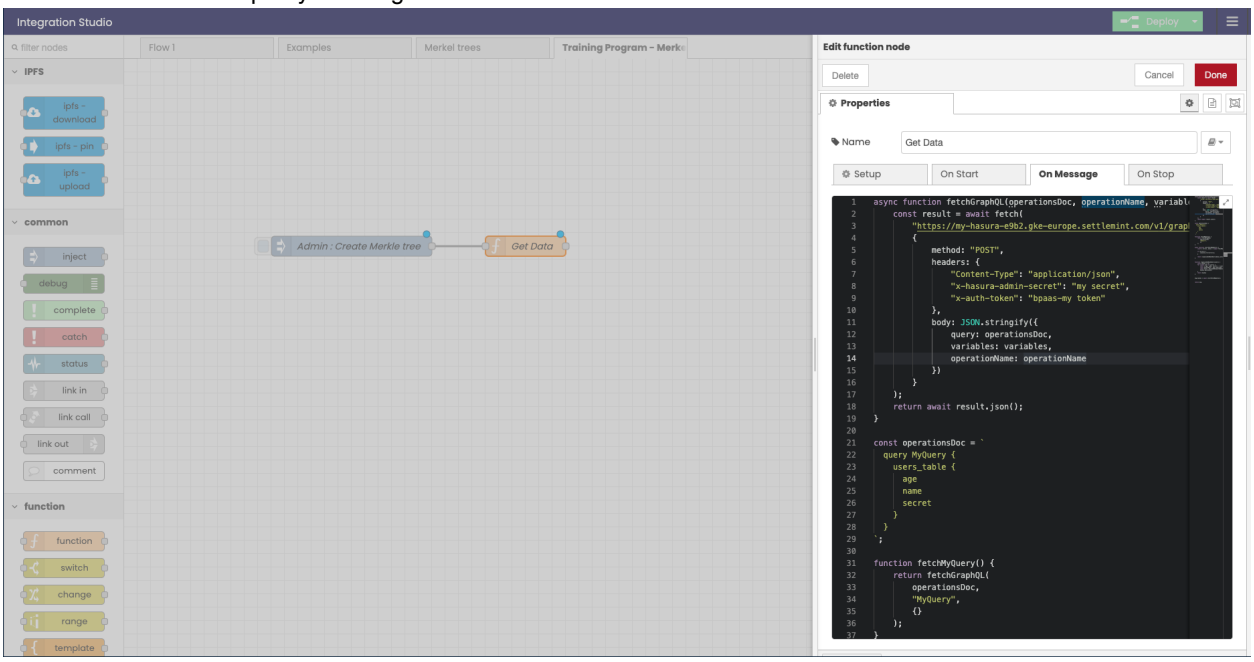
4. Import the script from Hasura's code exporter and integrate the `organizeAndHashUsers(user)` function, which we developed earlier.

Here's how the complete code snippet should look. Remember to replace the placeholders with your actual information.

```
async function fetchGraphQL(operationsDoc, operationName, variables) {
  const result = await fetch(
    "https://my-hasura-e9b2.gke-europe.settlemint.com/v1/graphql",
    {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
```

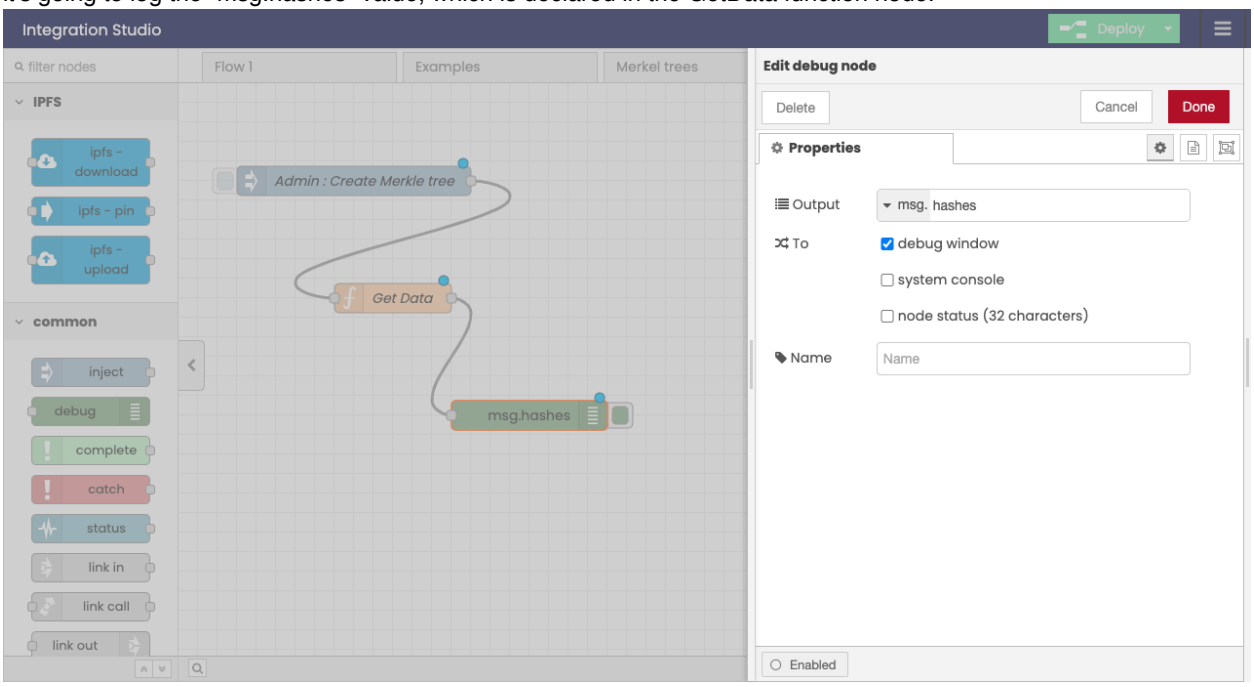
```
            "x-hasura-admin-secret": "my secret",
            "x-auth-token": "bpaas-my token"
          },
          body: JSON.stringify({
            query: operationsDoc,
            variables: variables,
            operationName: operationName
          })
        }
    );
    return await result.json();
}
const operationsDoc = `
  query MyQuery {
    users_table {
      age
      name
      secret
    }
  }
`;
function fetchMyQuery() {
    return fetchGraphQL(
        operationsDoc,
        "MyQuery",
        {}
    );
}
async function startFetchMyQuery() {
    const { errors, data } = await fetchMyQuery();
    if (errors) {
        console.error(errors);
    }
    return organizeAndHashUsers(data.users_table);
}
function organizeAndHashUsers(users) {
    const hashes = [];
    for (const user of users) {
        const { age, name, secret } = user;
        const dataToHash = age.toString() + name + secret;
        const hash = keccak256(dataToHash).toString('hex');
        hashes.push(hash);
    }
    return hashes;
}
msg.hashes = await startFetchMyQuery();
return msg;
```

This should be the setup of your Integration Studio.



2.3 ADDING A DEBUG NODE:

In Integration Studio, you can debug by adding a debug node, which will log a value from the msg object. In this case, it's going to log the `msg.hashes` value, which is declared in the GetData function node.

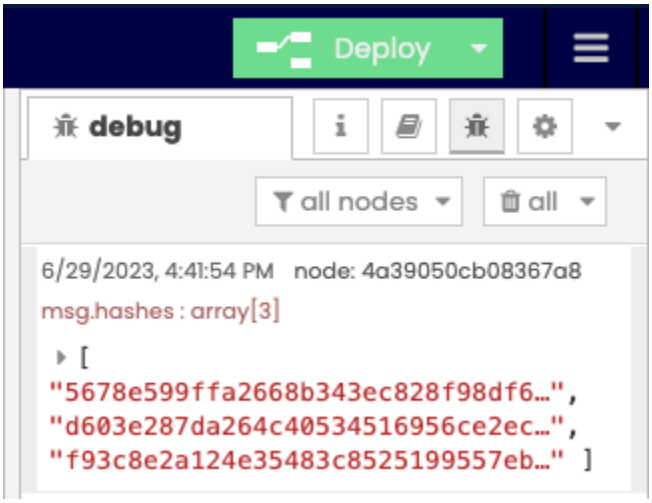


//Get Data function node

```
74
75    msg.hashes = await startFetchMyQuery();
76
```

Now, if you trigger this command, you should see in your debug console the hashes that were generated from your Hasura DB. Here's an example of the output:

[
  '41bb1e44bc70ec73abff22e9968938cf52f18cfc15cd3cc28fa49873ed7d5a79',
  'dce01d42480af923fab205a2e4291d4803291389410a084504b786e4fdcf8b61',
  '4007e85b723e582d22cc6061558c51eaf16719a19ddb95a749f33375f6ee0080'
]



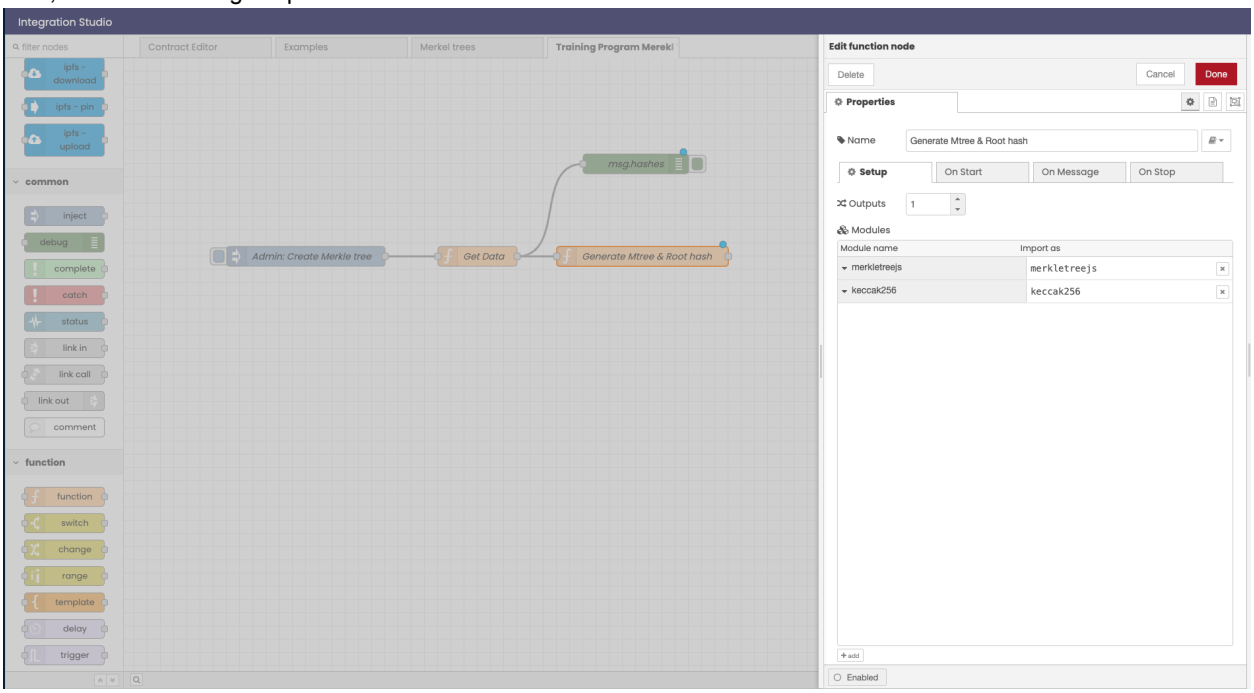
2.4 GENERATING THE MERKLE TREE AND EXTRACTING THE ROOT HASH:

You don't have to be a cryptography master to create Merkle trees. You can use the libraries that run the algorithm for you.

To do this, create a new function node and call it "Generate Merkle tree". We'll create a tree and extract the root hash from it for on-chain storage.

Don't forget to import the merkle.js module and keccak256 to the new function node.

Now, add the following script to this function node:



Now lets add to the this function node the following script,

```
const leaves = msg.hashes;
let merkleTree = new merkletreejs.MerkleTree(leaves, keccak256, { sortPairs: true });
let rootHash = merkleTree.getRoot().toString('hex');
global.set("merkleTree", merkleTree);
global.set("rootHash", '0x' + rootHash);
msg.roothash = '0x' + rootHash;
msg.mtree = merkleTree;
return msg;
```

This script takes the hashes generated in the previous function node (GetData), inputs them into the Merkle tree algorithm, and generates the Merkle tree for us.

Then we use another built-in function in the merkle.js library to generate the root hash.

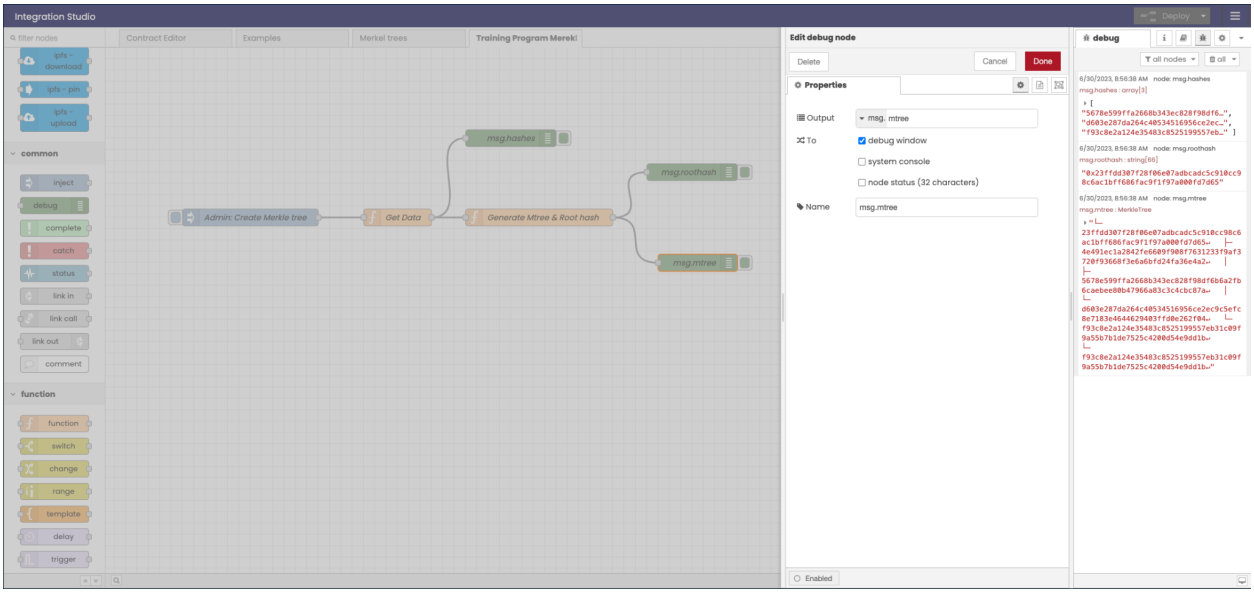
Finally, we initialize the root hash and the Merkle tree as global variables, and also add them to the msg object.

We'll also add a couple of debugger nodes: one will log the entire Merkle tree (msg.mtree = merkleTree)

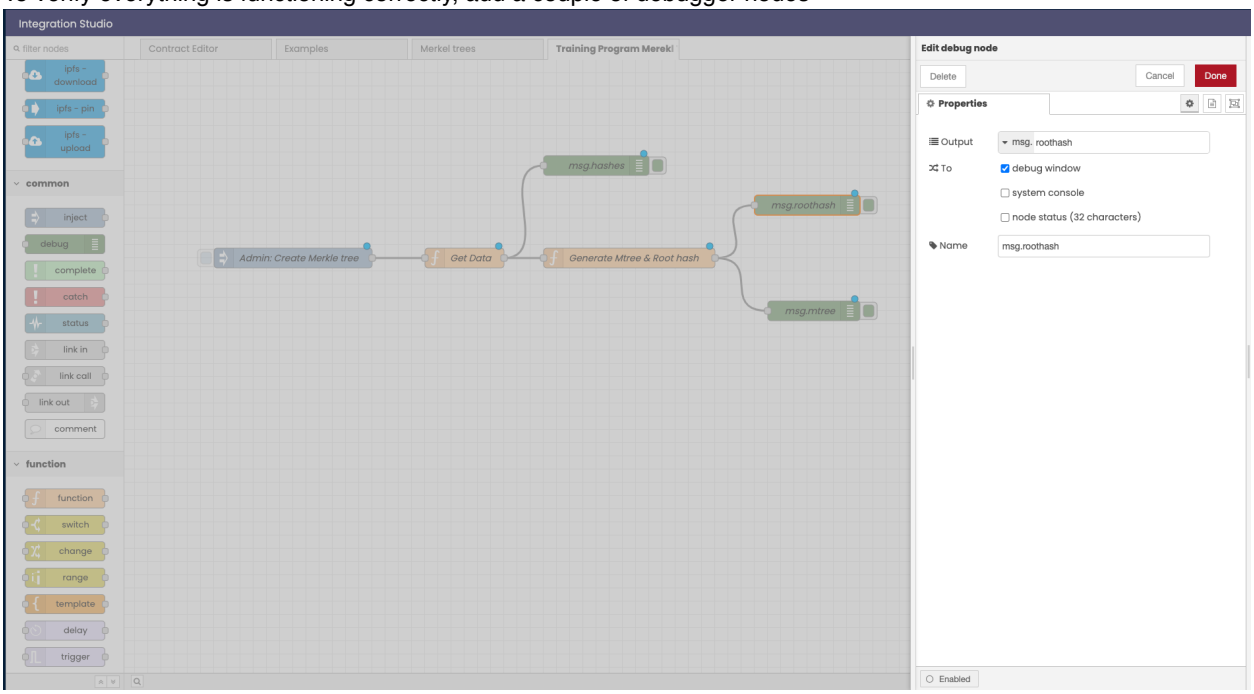
```
msg.mtree
```

and the second will log the root hash (msg.roothash = '0x' + rootHash).

msg.rootHash

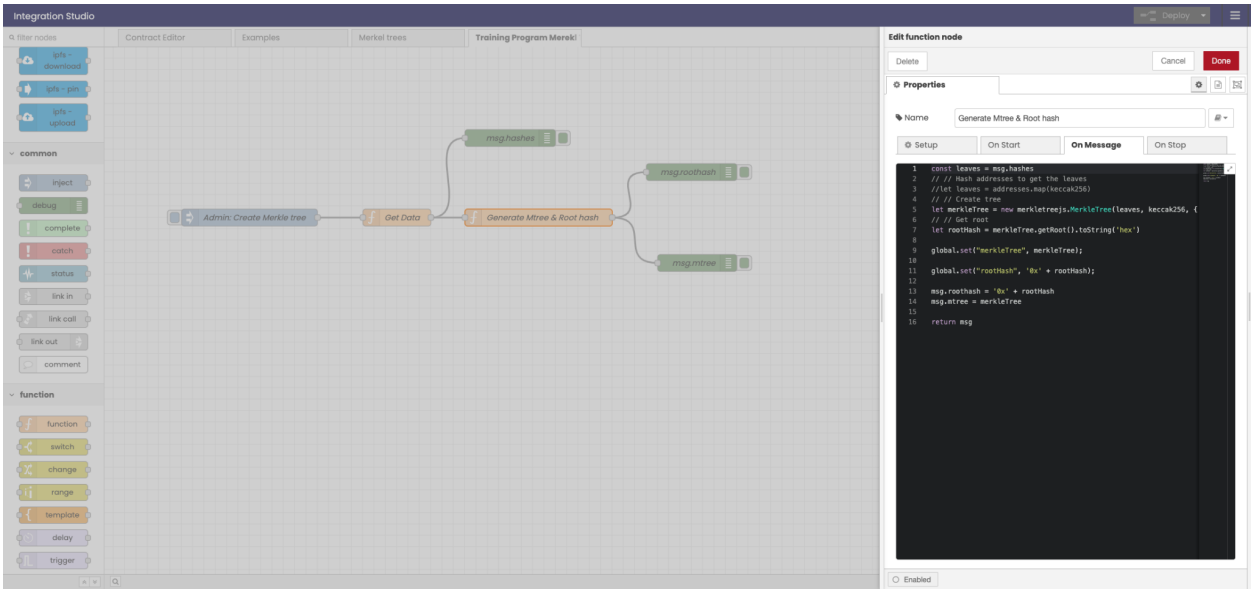


To verify everything is functioning correctly, add a couple of debugger nodes



.

Here's how your flow should look so far:



3. USER FLOW IN INTEGRATION STUDIO

In the previous section, we handled the admin flow, creating a Merkle tree using our database. We will later use the root hash of this tree.

Now, let's create a flow that simulates a user who is interacting with the contract.

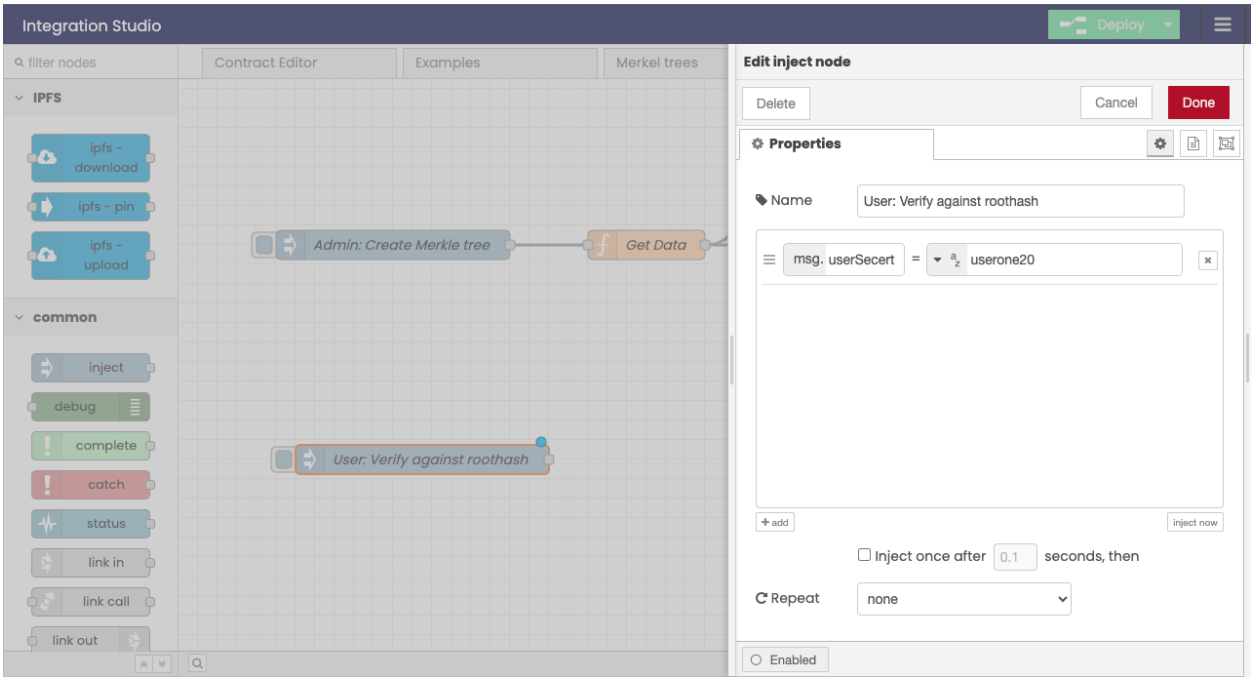
The user provides his "secret". Using this secret, we fetch the rest of his information from Hasura, and then create a hash using the same method we used in Step 2.

Then, we can check against the root hash to determine if our user's hash belongs in that tree.

3.1 CREATE A NEW INJECT NODE:

Start by creating a new "Inject Node". Double-click on it and rename it to "User: Verify against Root Hash" or any other name you prefer. Next, press "Add" to insert a new item into the msg object as a user secret (msg.userSecret), as illustrated in the image below.

The value that is injected will serve as the secret. Shortly, we will use this secret to fetch the rest of the user's data.

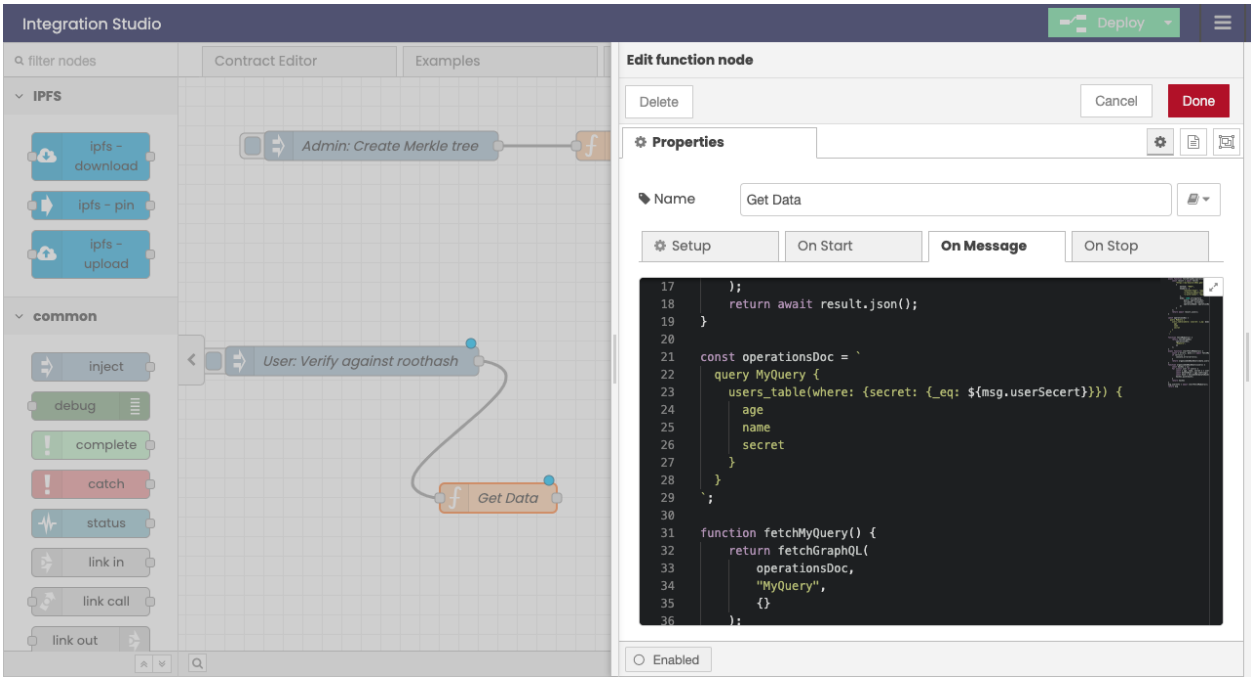


3.2 ADDING GET DATA FUNCTION NODE:

Next, we are going to enhance the users flow by adding a new "Get Data" function node. This function node will operate much like the "Get Data" function node in the "Admin: Create Merkle" flow, but with one significant change: we will dynamically change the "operationDoc" variable (the query) to incorporate the msg.userSecret value that we input in the "User: Verify against Root Hash" inject node, which we created in the previous section (3.1).

```
20
21  const operationsDoc = `
22    query MyQuery {
23      users_table(where: {secret: {_eq: ${msg.userSecert}}}) {
24        age
25        name
26        secret
27      }
28    }
29  `;
30
```

The full script should look something like this. Please ensure that you input your own values. The easiest way to achieve this would be to just copy the previous Get Data node and only change the operation Doc variable.

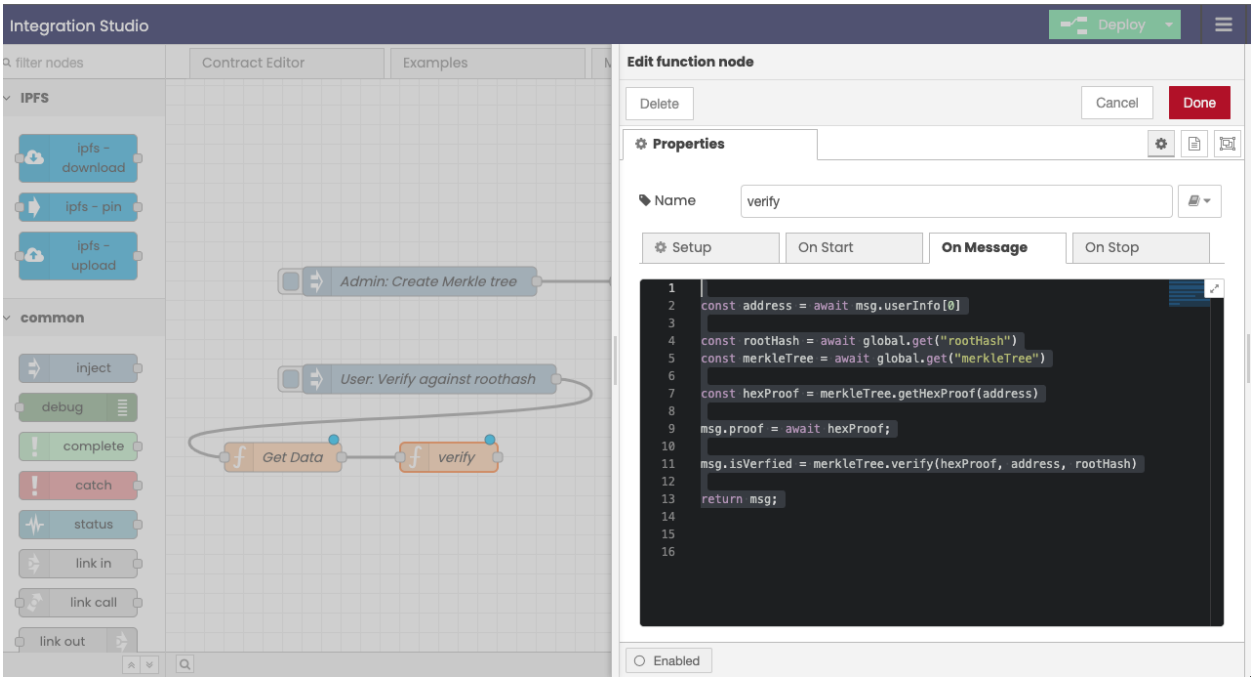


3.3 VERIFYING:

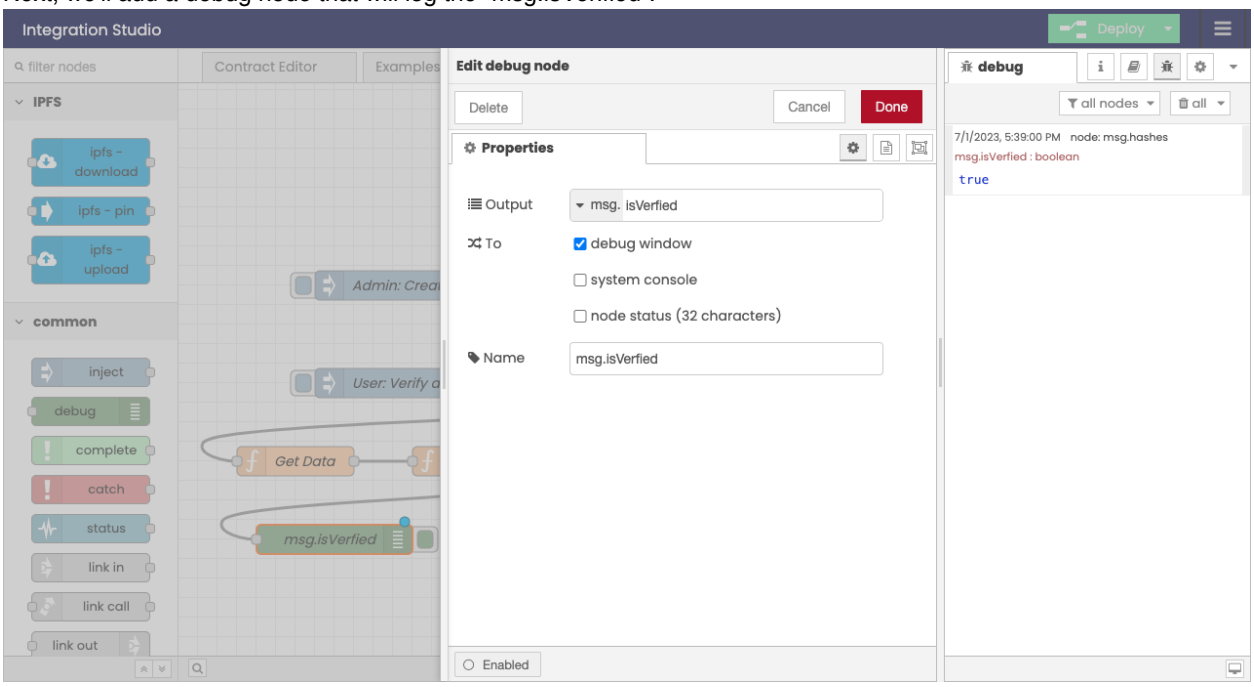
Next, we'll create a function node that checks the user's hash against the root hash of the tree. This can be accomplished by adding a function node named "verify" and inserting the following code into it:

```
const leaf = await msg.userInfo[0]
const rootHash = await global.get("rootHash")
const merkleTree = await global.get("merkleTree")
const hexProof = merkleTree.getHexProof(leaf)
msg.proof = await hexProof;
msg.isVerified = merkleTree.verify(hexProof, leaf, rootHash)
return msg;
```

This code retrieves the root hash and Merkle tree that we created in the "admin flow" (step 2). It then returns the verification status in the `msg.isVerified`.



Next, we'll add a debug node that will log the `msg.isVerified`.



Of course, this does not yet verify anything against the blockchain. Naturally, the next step will be to deploy the smart contract!

4. THE CONTRACT!

4.1 - Writing the Contract

In your Settlemint IDE, create a file in the contract folder. You can name it "MerkleExample", for instance, and copy the following contract into it:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0 <0.9.0;

import "@openzeppelin/contracts/utils/cryptography/MerkleProof.sol";

contract MerkleExample {

    bytes32 public rootHash;

    constructor(bytes32 _rootHash) {
        rootHash = _rootHash;
    }

    function checkValidity(bytes32[] calldata _merkleProof, bytes32 leaf) public view returns (bool) {
        return MerkleProof.verify(_merkleProof, rootHash, leaf);
    }
}
```

The primary function of this smart contract is to store the rootHash. This allows our users to verify, against the blockchain, whether their hash can be derived from the rootHash.

In the smart contract, we will import OpenZeppelin's MerkleProof library, just as we imported the Merkle JS library in Integration Studio.

```
import "@openzeppelin/contracts/utils/cryptography/MerkleProof.sol";
```

We will store the root hash in a `bytes32` state variable:

```
public rootHash =  0x0000000000000000000000000000000000000000000000000000000000000000;
```

Just like the Verify function node in the previous section, the contract calls the Merkle tree library to perform the verification. Please note that it accepts the same parameters, but in a different order:

```
MerkleProof.verify(_merkleProof, rootHash, leaf)
```

4.2 Deployment Script

To create a new deployment script, go to the deploy folder and use the following syntax (00_name.ts). Replace the argument with the rootHash that your admin flow creates, and then deploy the contract.

```
import { DeployFunction } from 'hardhat-deploy/types';

const func: DeployFunction = async function (hre) {
  const { deployments, getNamedAccounts } = hre;
  const { deploy } = deployments;
  const { deployer } = await getNamedAccounts();

  await deploy('MerkleTest', {
    from: deployer,
    args: ["REPLACE WITH ROOTHASH!"], // replace with your rootHash
    log: true,
  });
};

//export default func;
func.tags = ['MerkleTest'];
```

To deploy it, run:

```
npx hardhat deploy --network <network-name> --tags MerkleTest
```

5. Let's Test It!

5.1 Set up the smart contract interface by copying over the ABI and contract address into the contract editor flow.

5.2 Run the “get proofs” command, and check the information in the contract interface.

5.3 If the function returns true, it means that you've made it! Congrats! Now, you know how to whitelist any information you'd like on the blockchain without exposing it, and you're ready to tackle the next challenge.

6. Task - Upgrade:

Add an "addPoints" function, which checks if the user is verified, and if they are, it adds a point to that user each time they call that function.

Hint: Map the points to addresses.

Cross Chain bridge - atomic swaps

Task Description:

Implement Cross-Chain Atomic Swaps using keccak256 Hash
Objective: Your task is to implement a system for performing cross-chain atomic swaps. This should be implemented using two Ethereum smart contracts, one for each chain involved in the swap.

Steps:

- Secret Generation: Generate a secret and its corresponding hash. The secret should be a pseudo-random number generated within a smart contract. Use the keccak256 function to generate a hash of the secret.
- Contract Deployment on Chain A:
    - Write a contract in Solidity that represents the initiation of the atomic swap.
    - The contract should hold the assets to be swapped. For simplicity, you can use the native currency of the chain (Ether for Ethereum) for this task.
    - The contract should contain the hash of the secret in its constructor, and a function that allows the participant on Chain B to claim the funds if they can provide the secret that matches the stored hash.
    - This contract should also specify the participant on Chain B, and only this participant should be able to claim the funds.
    - Deploy this contract on Chain A.
- Contract Deployment on Chain B:
    - Write a similar contract for Chain B.
    - This contract should also hold assets, contain the hash of the secret, and specify the participant from Chain A.
    - Deploy this contract on Chain B.
- Secret Revelation and Swap Confirmation on Chain B:

    - The participant from Chain A should call a function on the contract on Chain B to confirm the swap. This function should require the secret as a parameter, and should release the funds to the participant from Chain A if the provided secret matches the stored hash.
    - If the secret is correct, the contract should emit an event with the secret to confirm the swap.
- Swap Confirmation on Chain A:
    - The participant from Chain B, now having the secret from the transaction on Chain B, should call a function on the contract on Chain A to confirm the swap.
    - Again, if the secret is correct, the contract should release the funds to the participant from Chain B and emit an event with the secret to confirm the swap.

Deliverables:

- Source code for the contracts deployed on both Chain A and Chain B.
- Detailed comments in the code explaining the functionality.
- Instructions on how to deploy and interact with these contracts using a tool like Remix or Truffle.
- Tests demonstrating the atomic swap process

Remember, the goal of this task is to ensure that the swap is atomic - either both asset transfers happen, or neither do. Good luck with your implementation!

Simplified ENS - Domain name space

Task: Building a Simplified ENS (Ethereum Name Service) System

Overview:

In this task, you will build a simplified version of the Ethereum Name Service (ENS) system. ENS allows users to register and resolve human-readable names (domains) to Ethereum addresses. This task will give you hands-on experience in building a basic domain registration and resolution system using smart contracts and JavaScript.

Requirements:

- Basic understanding of Ethereum, smart contracts, and JavaScript.
- Development environment with Solidity compiler, Hardhat, and Node.js installed.
- Basic knowledge of Hardhat and Ethereum development tooling.

Steps:

- Project Setup:
    - Settlemint IDE
    - Integration studio
    - blockchain node deployed
- Contract Development:
    - Create a new Solidity contract file, e.g., ENS.sol, and define the contract with the required functions and data structures.
    - Implement the logic for domain registration and resolution:
        - Design the data structure to store domain-to-address mappings.
        - Write a function to register a domain by associating it with an Ethereum address.
        - Write a function to resolve a domain to its associated Ethereum address.
    - Compile and deploy the contract using Hardhat.
- ts Interaction:
    - Create a new Typescript file, e.g., ens.ts, to deploy the contract
    - Set up the necessary connections to an Ethereum network provider using ethers.js.
    - Write JavaScript functions to perform the following actions (can be done in integration studio)
        - Register a domain by calling the contract's register function.
        - Resolve a domain by calling the contract's resolve function.
        - Send a transaction to the resolved address using the resolved address as the recipient.
    - Test the functionality of your JavaScript code using Hardhat's testing framework or a separate test script.

Additional Tips:

- Break down the tasks into smaller steps, and explain each step thoroughly to provide beginners with a clear understanding of what they need to do and why.
- Encourage participants to ask questions and seek help if they encounter any difficulties.
- Provide additional resources and references for further learning about ENS, smart contracts, and Ethereum development.

Note: This is a simplified task to help beginners understand the basic concepts of an ENS system. In a real-world scenario, building a complete and secure ENS system would involve more complex considerations and features.

Feel free to modify the task based on your specific requirements and the level of expertise of the participants

Bonus task 1 : Building a Decentralized Voting Application

Overview:
In this task, you will build a decentralized voting application using Ethereum smart contracts and JavaScript. The application allows users to create polls, vote on different options, and view the results in a transparent and tamper-proof manner.

Requirements:

- Solid understanding of Ethereum, smart contracts, and JavaScript.
- Development environment with Solidity compiler, Hardhat, and Node.js installed.
- Basic knowledge of Hardhat and Ethereum development tooling.

Steps:

- Project Setup:
    - Set up a new project in your settlemint application
    - make sure you have a blockchain node deployed
- Contract Development:
    - Create a new Solidity contract file, e.g., Voting.sol, and define the contract with the required functions and data structures.
    - Implement the logic for creating polls, voting on options, and tracking the results:
        - Design the data structures to store poll information, options, and votes.
        - Write functions to create a poll, vote on an option, and retrieve the poll results.
    - Compile and deploy the contract using Hardhat.
- typescript/ javascript Interaction:
    - Create a new file/ integration studio flow, to interact with the deployed contract.
    - Set up the necessary connections to an Ethereum network provider using ethers.js.
    - Write JavaScript functions to perform the following actions:
        - Create a new poll by calling the contract's createPoll function.
        - Vote on an option in a poll by calling the contract's vote function.
        - Retrieve the results of a poll by calling the contract's getResults function.
    - Test the functionality of your JavaScript code using Hardhat's testing framework or a separate test script.

Additional Tips:

- Consider adding additional features to the voting application, such as time-limited polls, multiple-choice questions, or weighted voting.
- Encourage participants to explore additional security considerations, such as preventing multiple votes from the same address.
- Provide participants with sample scenarios or use cases to guide them in creating meaningful polls for their application.

Bonus task 2 : Building a Decentralized Crowdfunding Application

Task 2: Developing a Decentralized Crowdfunding Platform

Overview:
In this task, you will develop a decentralized crowdfunding platform using Ethereum smart contracts and JavaScript. The platform enables project creators to create crowdfunding campaigns, receive contributions from backers, and release funds upon successful completion of the campaign.
Requirements:

- Solid understanding of Ethereum, smart contracts, and JavaScript.
- Development environment with Solidity compiler, Hardhat, and Node.js installed.
- Basic knowledge of Hardhat and Ethereum development tooling.

Steps:

- Project Setup:
    - Set up a new project in your settlemint application
    - make sure you have a blockchain node deployed
- Contract Development:
    - Create a new Solidity contract file, e.g., Crowdfunding.sol, and define the contract with the required functions and data structures.
    - Implement the logic for creating campaigns, contributing to campaigns, and releasing funds:
        - Design the data structures to store campaign details, contributions, and funding goals.
        - Write functions to create a campaign, contribute to a campaign, and release funds upon campaign completion.
    - Compile and deploy the contract using Hardhat.
- JavaScript /typescript Interaction:
    - Create a new file/ integration studio flow, to interact with the deployed contract.
    - Set up the necessary connections to an Ethereum network provider using ethers.js.
    - Write JavaScript functions to perform the following actions:
        - Create a new crowdfunding campaign by calling the contract's createCampaign function.
        - Contribute funds to a campaign by calling the contract's contribute function.
        - Release funds from a completed campaign by calling the contract's releaseFunds function.
    - Test the functionality of your JavaScript code using Hardhat's testing framework or a separate test script.

Additional Tips:

- Consider adding additional features to the crowdfunding platform, such as stretch goals, campaign updates, or refund mechanisms.
- Encourage participants to think about campaign verification and dispute resolution mechanisms.
- Discuss the advantages of decentralized crowdfunding, such as reduced fees, global accessibility, and increased transparency.

Feel free to modify the tasks according to your preferences and the skill level of the participants. Add any specific requirements or constraints as needed.