

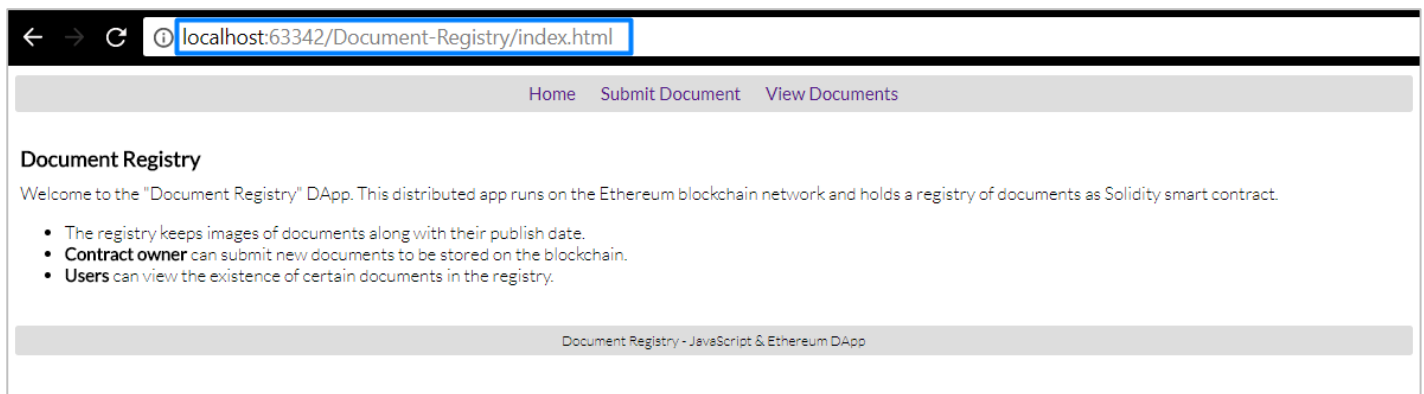
Exercises: Create Document Registry App

A document registry is a repository that contains links to documents added in the system. The Document Registry DApp is a simple registry for documents stored in a decentralized system which only the contract creator can add, but anyone can see them.

Goal

The goal of this exercise is:

- To deploy a simple Smart Contract using **Remix IDE** in **Ganache CLI** – command version of Ganache, a personal Ethereum blockchain perfect for our development purposes.
- To create a simple JavaScript app, which will interact with the contract using **Web3 API** and **MetaMask**.
- To utilize IPFS in storing binary data. For the example, document images will **not** be stored in the contract, but in IPFS. The contract should only store a representation of the images (eg. IPFS hash).



Prerequisites

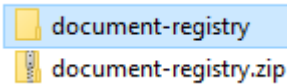
- Able to access Remix IDE <https://remix.ethereum.org/>
- Ganache CLI **v6.12.2** `npm install -g ganache-cli@6.12.2`
- MetaMask **v7.7.8**
- NodeJS **v13.5.0** <https://nodejs.org/en/download/releases/>
- NPM **v6.13.4**
- IPFS **v0.4.23** <https://dist.ipfs.io/go-ipfs/v0.4.23>

You might find that your version is higher, that is okay. Avoid using versions lower than what is specified as libraries used in this exercise might not be available old older versions or screenshots may not match the ones that you see on your screen.

You may or may not experience difficulties, but if you do, make sure to comply with these versions first before reporting to the instructors.

Problem 1. Setup the Project

1. Download the code template from exercise-resources:



2. Navigate to that folder in the terminal and install the dependencies:

```
npm install
```

Problem 2. Create the Smart Contract

1. Create a **DocumentRegistry.sol** file which will hold the smart contract implementation and define the version:

```
1 pragma solidity ^0.8.3;
2
3 contract DocumentRegistry {
4
5 }
6
```

2. Create a simple **structure** called **Document**, which stores the string hash of the document in IPFS and the date it has been added. The contract will have an array of documents and an address which will store the owner's address:

```
4 struct Document {
5     string hash;
6     uint256 dateAdded;
7 }
8
9 Document[] private documents;
10 address contractOwner;
11
```

3. A simple modifier **onlyOwner** which makes sure the caller of the method is the contract owner:

```
11
12 modifier onlyOwner() {
13     require(msg.sender == contractOwner, "Not a contract owner");
14     _;
15 }
16
```

4. Create the **constructor**, which will make the executor owner of the contract:

```
17 constructor() public {
18     contractOwner = msg.sender;
19 }
```

5. Create an add function, which adds a document in the arrayp from its hash and the current time (**block.timestamp**) in the structure. The function **returns** the date of the added document:

```
20
21 ~   function add(string memory hash) public onlyOwner returns (uint256 dateAdded) {
22       uint256 currentTime = block.timestamp;
23       Document memory doc = Document(hash, currentTime);
24       documents.push(doc);
25       return currentTime;
26   }
27
```

6. Finally, create **getDocumentsCount()** and **getDocument(uint256 index)** functions. This returns the count of the documents and a document by a given index:

```
27
28 ~   function getDocumentsCount() public view returns (uint256 documentCount) {
29       return documents.length;
30   }
31
32 ~   function getDocument(uint256 index) public view returns (string memory hash, uint256 dateAdded) {
33       Document memory document = documents[index];
34       return (document.hash, document.dateAdded);
35   }
36 }
37
```

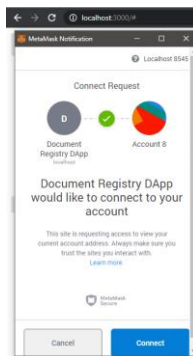
Problem 3. Create JavaScript Front-end

1. Create the jQuery function which will run as soon as the DOM becomes safe to manipulate. As of **Metamask v4.14.0**, a call to **ethereum.enable()** is necessary to gain access to a user's accounts in Metamask and enable RPC interactions. Read more here if you want to look at more details: <https://eips.ethereum.org/EIPS/eip-1102>.

Allow access using the injected Web3 in the browser with:

```
$(document).ready(function () {  
  try {  
    window.ethereum.enable();  
  } catch (e) {  
    alert("Access Denied.");  
  }  
})
```

This results in:



2. Next, **insert** the constants of the contract address and contract **ABI**. We will get back to this and replace it with the correct values after the contract is deployed. Also, initialize the IPFS library:

```
const documentRegistryContractAddress = "";  
const documentRegistryContractABI = [];  
  
const IPFS = window.IpfsApi("localhost", "5001");  
const Buffer = IPFS.Buffer;
```

3. Prepare the function for **uploading** the document. There are preset codes before this code separated with *User Interface Handlers Start/End* markers. This takes care of adding handlers to the user interface components:

```
// === User Interface Interactions End ===  
  
function uploadDocument() {  
  // Todo: Implementation  
}
```

4. Check if there is any file to **upload**, if not, show **error message**:

```
function uploadDocument() {
  if ($("#documentForUpload")[0].files.length === 0) {
    showError("Please select file to upload!");
    return;
  }
}
```

5. Create a **file reader** to read the given document as an array buffer:

```
function uploadDocument() {
  if ($("#documentForUpload")[0].files.length === 0) {
    showError("Please select file to upload!");
    return;
  }

  let fileReader = new FileReader();
  fileReader.onload = function () {
    // Todo: Handler
  };

  fileReader.readAsArrayBuffer($("#documentForUpload")[0].files[0]);
}
```

6. In the **onload** property check whether web3 is included.
Then, use **Buffer** to get a buffer from the file reader result.

Initialize the contract using the ABI and contract address.

```
fileReader.onload = function () {
  if (typeof web3 === "undefined") {
    showError(
      "Please install Metamask to access the Ethereum Web3 API from your browser"
    );
    return;
  }

  let fileBuffer = Buffer.from(fileReader.result);
  let contract = web3.eth
    .contract(documentRegistryContractABI)
    .at(documentRegistryContractAddress);
  console.log(contract);
}
```

7. Next, use IPFS to add the document.

After it is uploaded, call the contract by its **ABI** and **address** and add the document.

Putting it all together, you end up with this **uploadDocument()** implementation:

```
function uploadDocument() {
  if ($("#documentForUpload")[0].files.length === 0) {
    showError("Please select file to upload!");
    return;
  }

  let fileReader = new FileReader();
  fileReader.onload = function () {
    if (typeof web3 === "undefined") {
      showError(
        "Please install Metamask to access the Ethereum Web3 API from your browser"
      );
      return;
    }

    let fileBuffer = Buffer.from(fileReader.result);
    let contract = web3.eth
      .contract(documentRegistryContractABI)
      .at(documentRegistryContractAddress);
    console.log(contract);

    IPFS.files.add(fileBuffer, (err, result) => {
      if (err) {
        showError(err);
        return;
      }

      if (result) {
        let ipfsHash = result[0].hash;
        contract.add(ipfsHash, (error, txHash) => {
          if (error) {
            showError("Smart contract call failed: " + error);
            return;
          }

          if (txHash) {
            showInfo(
              `Document ${ipfsHash} successfully added to the registry! Transaction hash: ${txHash}`
            );
            return;
          }
        });
      }
    });
  };

  fileReader.readAsArrayBuffer($("#documentForUpload")[0].files[0]);
}
```

8. Now the function showing the documents. First, it will check if web3 is defined:

```
function viewGetDocuments() {  
  if (typeof web3 === "undefined") {  
    showError(  
      "Please install Metamask to access the Ethereum Web3 API from your web browser."  
    );  
    return;  
  }  
  
  // Todo: Implementation  
}
```

9. Finally, call the contract and get the count of the documents.

After the documents count is returned, implement a for-loop that iterates through the index and retrieves each document.

For each document, from the received image hash create a URL with “<https://localhost:8080/ipfs/{hash}>” and add it as a source of an **** tag.

Beautify the received date and place it in a **<p>** tag. Then, append them both in a **<div>** and finally, append it to **#viewGetDocuments**.

```

function viewGetDocuments() {
  if (typeof web3 === "undefined") {
    showError(
      "Please install Metamask to access the Ethereum Web3 API from your web browser."
    );
    return;
  }

  let contract = web3.eth
    .contract(documentRegistryContractABI)
    .at(documentRegistryContractAddress);

  contract.getDocumentsCount((err, res) => {
    if(err) {
      showError("Smart contract failed: " + err);
      return;
    }

    let documentsCount = res.toNumber();

    if (documentsCount > 0) {
      let html = $("<div>");
      for (let index = 0; index < documentsCount; index++) {
        contract.getDocument(index, (error, result) => {
          if (error) {
            showError("Smart contract failed: " + error);
            return;
          }
          let ipfsHash = result[0];
          let contractPublishDate = result[1];
          let div = $("<div>");
          let url = "http://localhost:8080/ipfs/" + ipfsHash;

          let displayDate = new Date(
            contractPublishDate * 1000
          ).toLocaleDateString();
          div.append(`${<p>Document published on: ${displayDate}</p>`);
          div.append(`${`);
          html.append(div);
        });
      }
      html.append("</div>");
      $("#viewGetDocuments").append(html);
    } else {
      $("#viewGetDocuments").append("<div>No documents in the docuement registry!</div>");
    }
  });
}
});

```


Problem 4. Deploy the Document Registry Contract

1. Install **ganache** by the command:

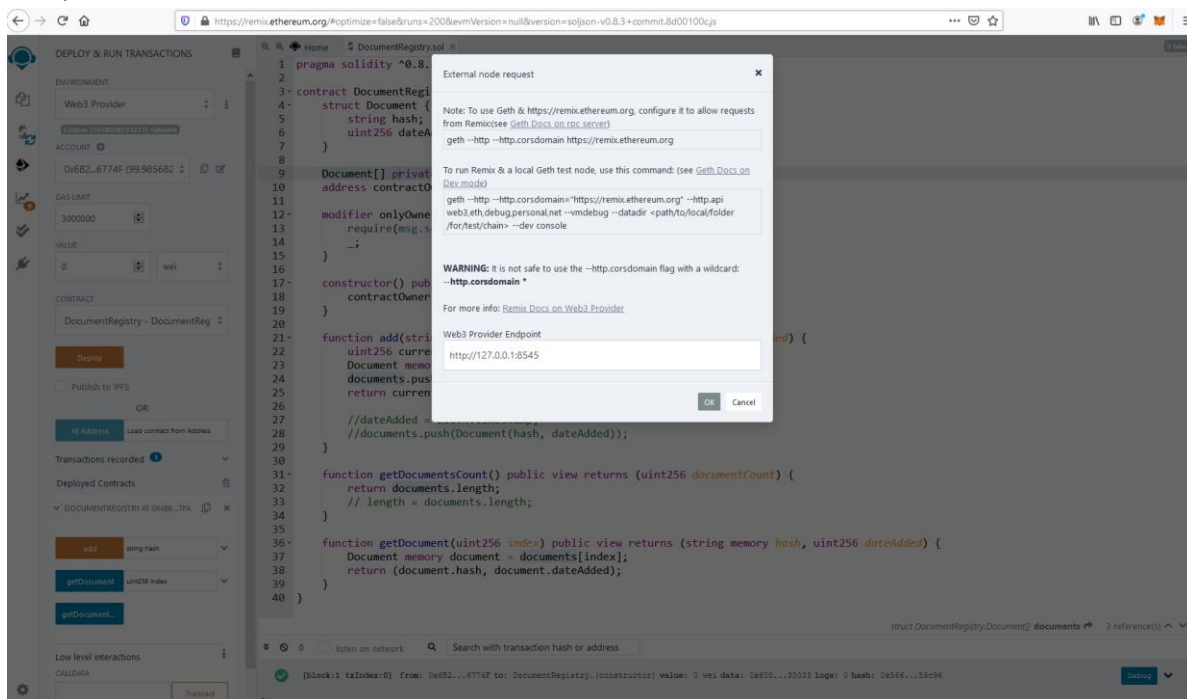
```
npm install -g ganache-cli@6.12.2
```

2. Start **ganache-cli** by issuing this command on the terminal:

```
ganache -cli
```

Do not close this terminal as you go along with the next activities.

3. Go to **remix.ethereum.org** and return to your **DocumentRegistry.sol** smart contract code.
4. Compile and choose the **Web3 Provider** to connect to **Ganache**.



5. If you get an error message, make sure you use **http** protocol (not https) when opening Remix.

JavaScript scripts. The following libraries are accessible:

```
version 1.0.0
```

```
.js
```

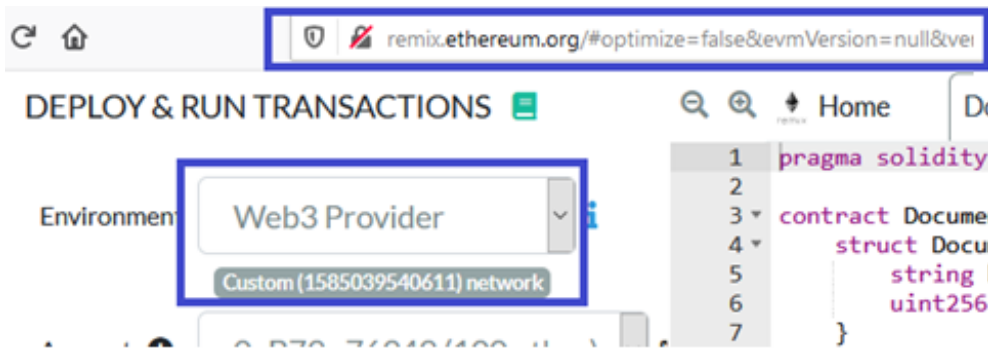
```
//
```

```
(run remix.help() for more info)
```

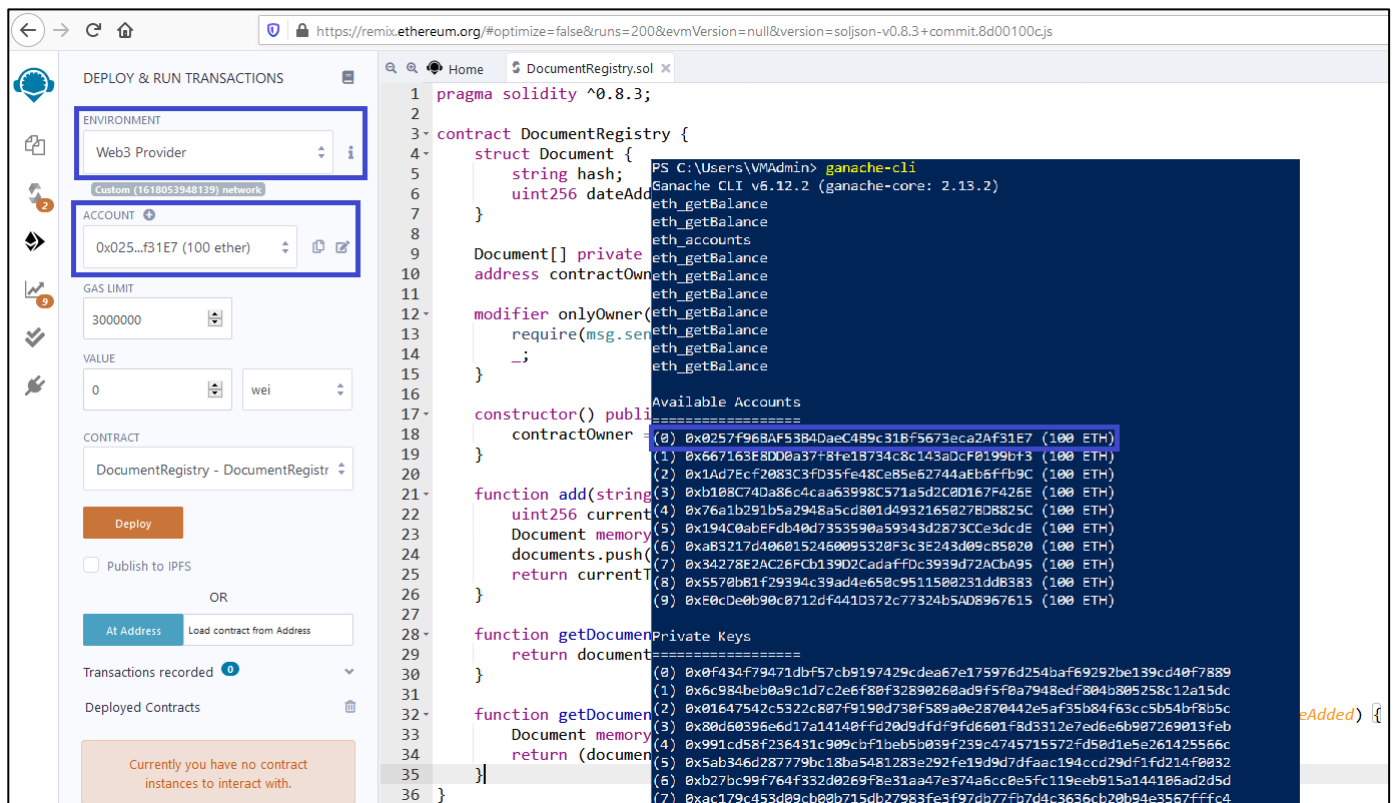
common command to interact with the Remix interface (see list of commands above). Note that these commands can also be included and run via script.

```
ts/.register(key,
```

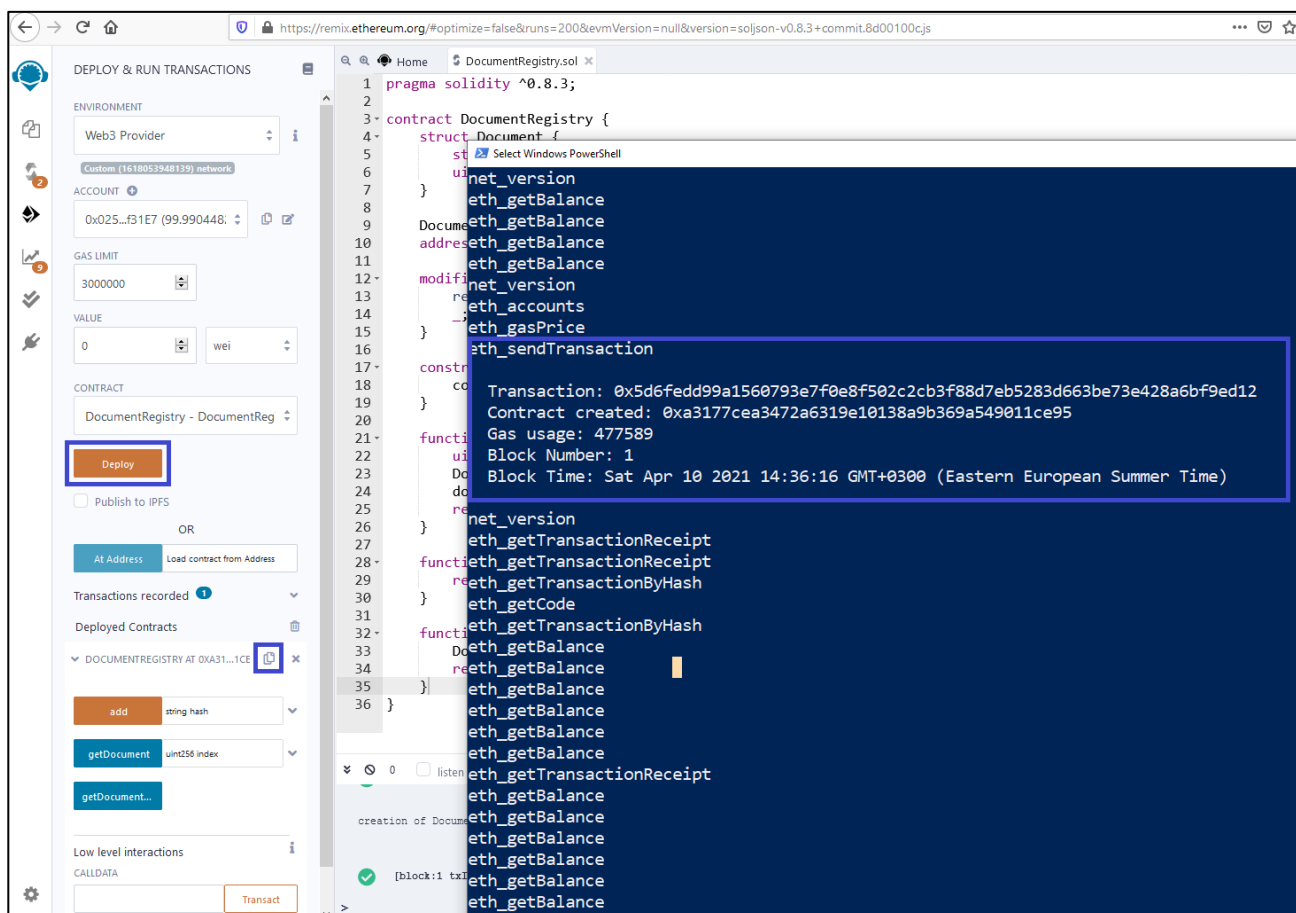
Not possible to connect to the Web3 provider. Make sure the provider is running and a connection is open (via IPC or RPC).



6. In Account you must see the accounts from your local Ethereum network.



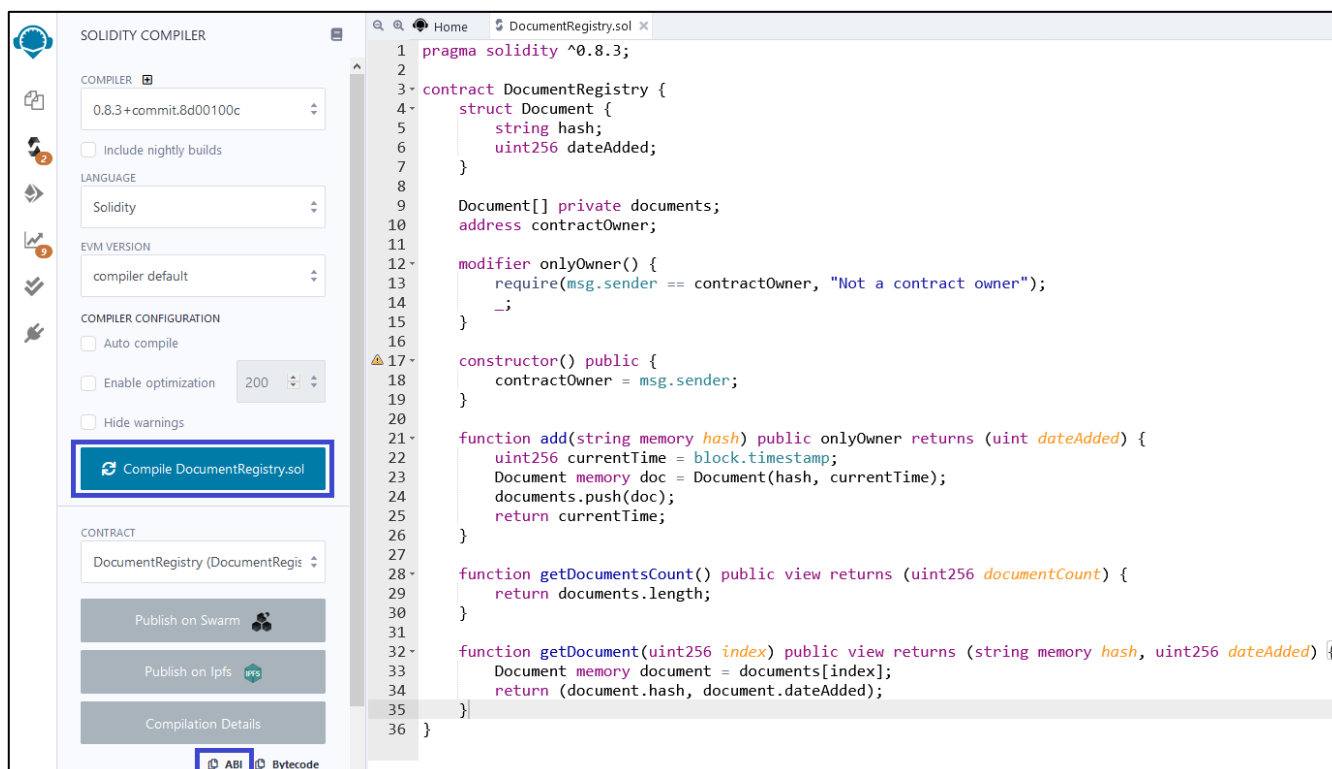
7. **Deploy** the contract. After the deployment, the contract will appear beneath. **Copy** the contract address:



The screenshot shows the Remix IDE interface. On the left, the 'DEPLOY & RUN TRANSACTIONS' sidebar is visible, with the 'Deploy' button highlighted. The main editor displays the Solidity code for 'DocumentRegistry.sol'. A console window on the right shows the deployment details:

```
Transaction: 0x5d6fdd99a1560793e7f0e8f502c2cb3f88d7eb5283d663be73e428a6bf9ed12
Contract created: 0xa3177cea3472a6319e10138a9b369a549011ce95
Gas usage: 477589
Block Number: 1
Block Time: Sat Apr 10 2021 14:36:16 GMT+0300 (Eastern European Summer Time)
```

8. To get the **ABI**, go back to **Solidity compiler**, copy the output:



The screenshot shows the Solidity Compiler panel in Remix IDE. The 'Compiler' section shows the version '0.8.3+commit.8d00100c'. The 'Language' is set to 'Solidity'. The 'EVM Version' is set to 'compiler default'. The 'Compiler Configuration' section has 'Auto compile' checked. The 'Compile DocumentRegistry.sol' button is highlighted. The main editor shows the Solidity code for 'DocumentRegistry.sol'.

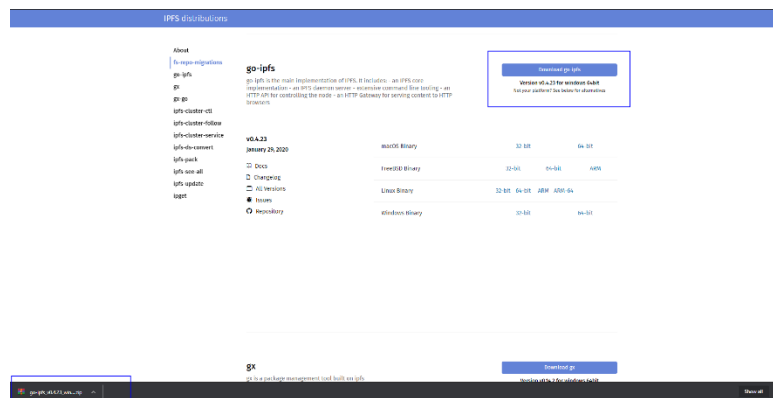
9. Go back to the project and paste the **Contract Address** and the **ABI** for the variables we previously wrote:

```
const documentRegistryContractAddress =
    "0xecb2c0da71e85e0a6a0dd91aa8808d12229e5511";

const documentRegistryContractABI =
[
    {
        "constant": false,
        "inputs": [
            {
                "internalType": "string",
                "name": "hash",
                "type": "string"
            }
        ],
        "name": "add",
        "outputs": [
            {
                "internalType": "uint256",
                "name": "dateAdded",
                "type": "uint256"
            }
        ],
        "payable": false,
        "stateMutability": "nonpayable",
        "type": "function"
    },
    {
        "inputs": [],
        "payable": false,
        "stateMutability": "nonpayable",
        "type": "constructor"
    },
]
```

Problem 5. IPFS Configuration

1. You should have installed IPFS and added it to your Path or simply extract it somewhere and launch a terminal on that particular folder.



2. In your IPFS configuration file, set the API HTTP CORS header so that there would be no CORS-related errors in the browser:
 - For Windows, the configuration file can be found in: **%homepath%\ipfs**
 - For Unix-like platforms, the configuration file is located in: **~/.ipfs**

```
{
  "API": {
    "HTTPHeaders": {
      "Access-Control-Allow-Origin": [
        "*"
      ]
    }
  },
  "Addresses": {
    "API": "/ip4/127.0.0.1/tcp/5001",
    "Announce": [],
    "Gateway": "/ip4/127.0.0.1/tcp/8080",
    "NoAnnounce": [],
    "Swarm": [
      "/ip4/0.0.0.0/tcp/4001",
      "/ip6:::/tcp/4001",
      "/ip4/0.0.0.0/udp/4001/quic",
      "/ip6:::/udp/4001/quic"
    ]
  }
}
```

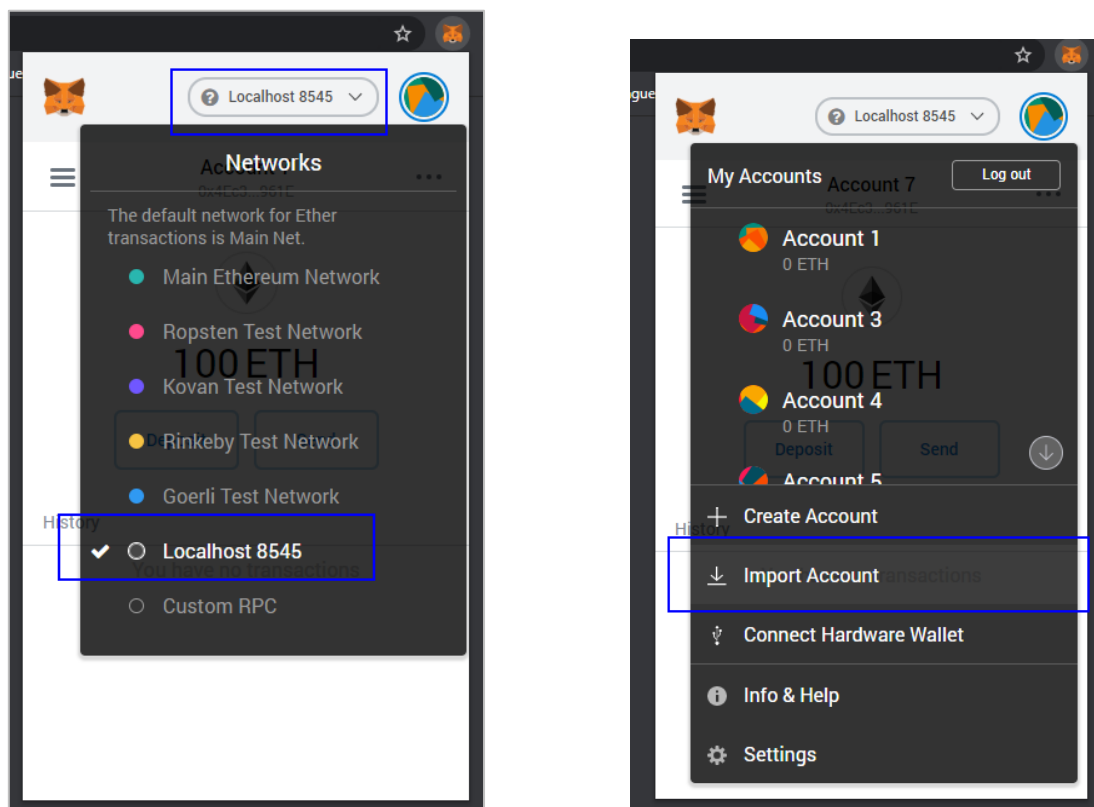
3. Start the IPFS daemon:

```
ipfs daemon
```

```
Command Prompt - ipfs daemon
C:\Users\VMAdmin>ipfs daemon
Initializing daemon...
go-ipfs version: 0.10.0
Repo version: 11
System version: amd64/windows
Golang version: go1.16.8
Swarm listening on /ip4/127.0.0.1/tcp/4001
Swarm listening on /ip4/127.0.0.1/udp/4001/quic
Swarm listening on /ip4/192.168.91.128/tcp/4001
Swarm listening on /ip4/192.168.91.128/udp/4001/quic
Swarm listening on /ip6:::1/tcp/4001
Swarm listening on /ip6:::1/udp/4001/quic
Swarm listening on /p2p-circuit
Swarm announcing /ip4/127.0.0.1/tcp/4001
Swarm announcing /ip4/127.0.0.1/udp/4001/quic
Swarm announcing /ip4/192.168.91.128/tcp/4001
Swarm announcing /ip4/192.168.91.128/udp/4001/quic
Swarm announcing /ip6:::1/tcp/4001
Swarm announcing /ip6:::1/udp/4001/quic
API server listening on /ip4/127.0.0.1/tcp/5001
WebUI: http://127.0.0.1:5001/webui
Gateway (readonly) server listening on /ip4/127.0.0.1/tcp/8080
Daemon is ready
```

Problem 6. Interacting with the Contract and IPFS

1. Log-in to **MetaMask** and switch to Local Network and prepare to import a private key:



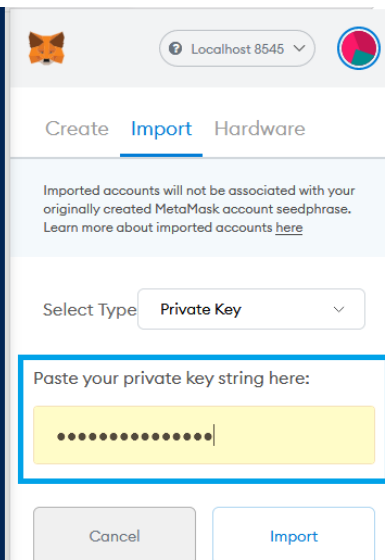
2. Import the private keys from **ganache-cli**:
 - Contract creator account – The account used to deploy the contract previously in Remix IDE.
 - One random account.

```

Available Accounts
=====
(0) 0x0257f96BAF53B4DaeC4B9c31Bf5673eca2Af31E7 (100 ETH)
(1) 0x667163E8DD0a37f8fe1B734c8c143aDcF0199bf3 (100 ETH)
(2) 0x1Ad7EcF2083C3fD35fe48CeB5e62744aEb6ffB9C (100 ETH)
(3) 0xb108C74Da86c4caa63998C571a5d2C0D167F426E (100 ETH)
(4) 0x76a1b291b5a2948a5cd801d4932165027BDB825C (100 ETH)
(5) 0x194C0abEFdb40d7353590a59343d2873CCe3dcdE (100 ETH)
(6) 0xaB3217d4060152460095320F3c3E243d09cB5020 (100 ETH)
(7) 0x34278E2AC26FCb139D2CadaffDc3939d72ACbA95 (100 ETH)
(8) 0x5570bB1f29394c39ad4e650c9511500231ddB383 (100 ETH)
(9) 0xE0cDe0b90c0712df441D372c77324b5AD8967615 (100 ETH)

Private Keys
=====
(0) 0x0f434f79471dbf57cb9197429cdea67e175976d254baf69292be139cd40f7889
(1) 0x6c984beb0a9c1d7c2e6f80f32890260ad9f5f0a7948edf804b805258c12a15dc
(2) 0x01647542c5322c807f9190d730f589a0e2870442e5af35b84f63cc5b54bf8b5c
(3) 0x80d60396e6d17a14140ffdd20d9dfd9fd6601f8d3312e7ed6e6b907269013feb
(4) 0x991cd58f236431c909cbf1beb5b039f239c4745715572fd50d1e5e261425566c
(5) 0x5ab346d287779bc18ba5481283e292fe19d9d7dfaacc194ccd29df1fd214f0032
(6) 0xb27bc99f764f332d0269f8e31aa47e374a6cc0e5fc119eeb915a144106ad2d5d
(7) 0xac179c453d09cb00b715db27983fe3f97db77fb7d4c3636cb20b94e3567fffc4
(8) 0x77928cea0c0fd72079b4df9dd063639f835b29750e701f8a560908320927ca33
(9) 0xd5cd8af21b4fc45d40d9c85f5cc7091b53c15a1c3280208fa80f58e731a712c1

```



Localhost 8545

Create **Import** Hardware

Imported accounts will not be associated with your originally created MetaMask account seedphrase. [Learn more about imported accounts here](#)

Select Type: **Private Key**

Paste your private key string here:

.....

Cancel Import

3. Go back to your **document-registry** workspace and launch the client:

```
npm start
```

```
D:\KINGSLAND\document-registry (document-registry@1.0.0)
λ npm start

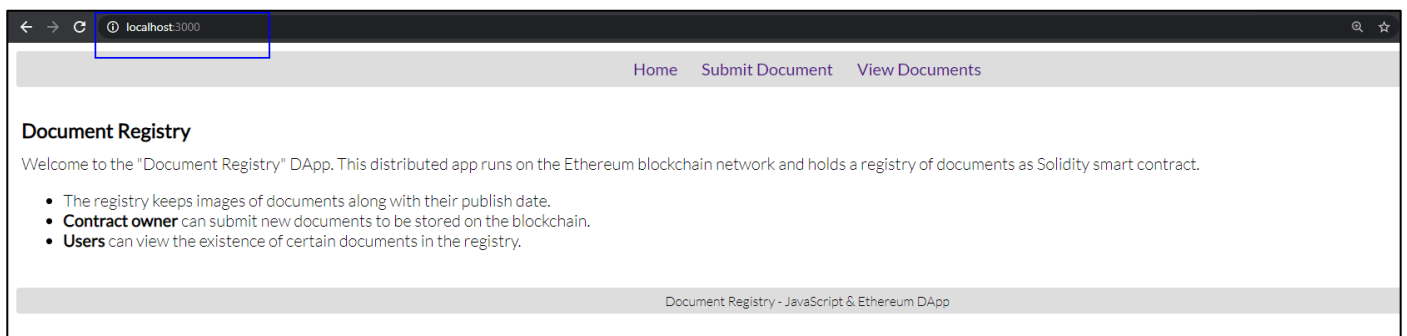
> document-registry@1.0.0 start D:\KINGSLAND\document-registry
> serve -S public -l 3000

Serving!

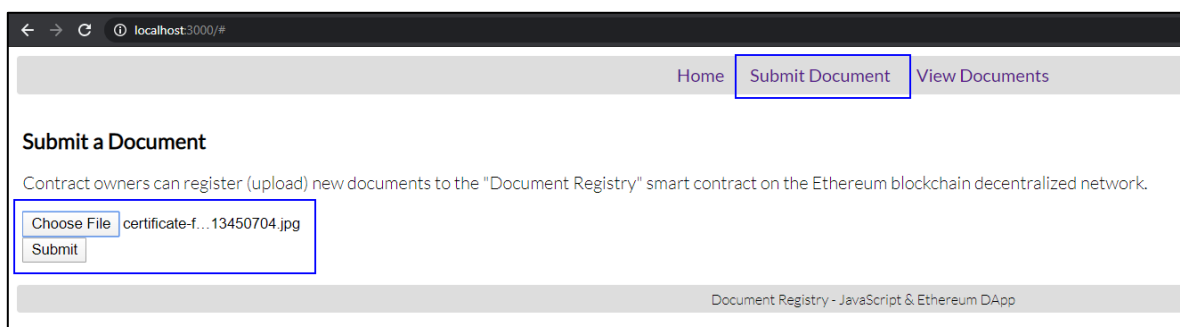
- Local:      http://localhost:3000
- On Your Network: http://192.168.2.89:3000

Copied local address to clipboard!
```

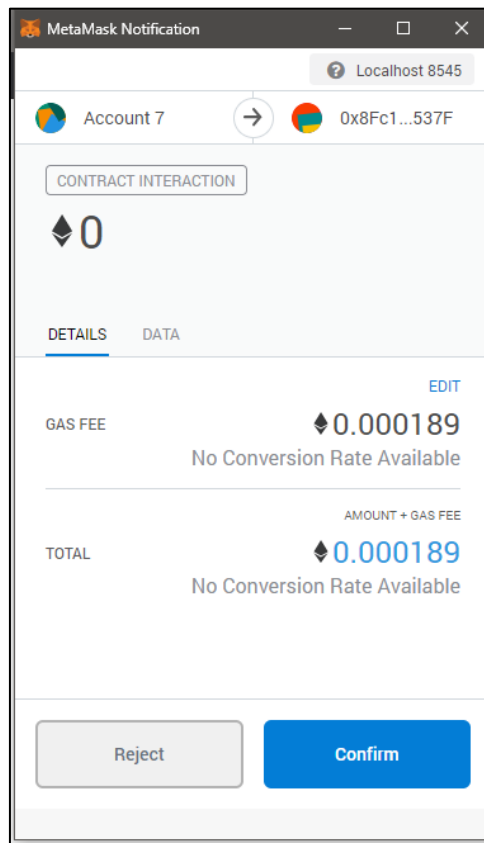
4. Go to your browser and visit <http://localhost:3000>:



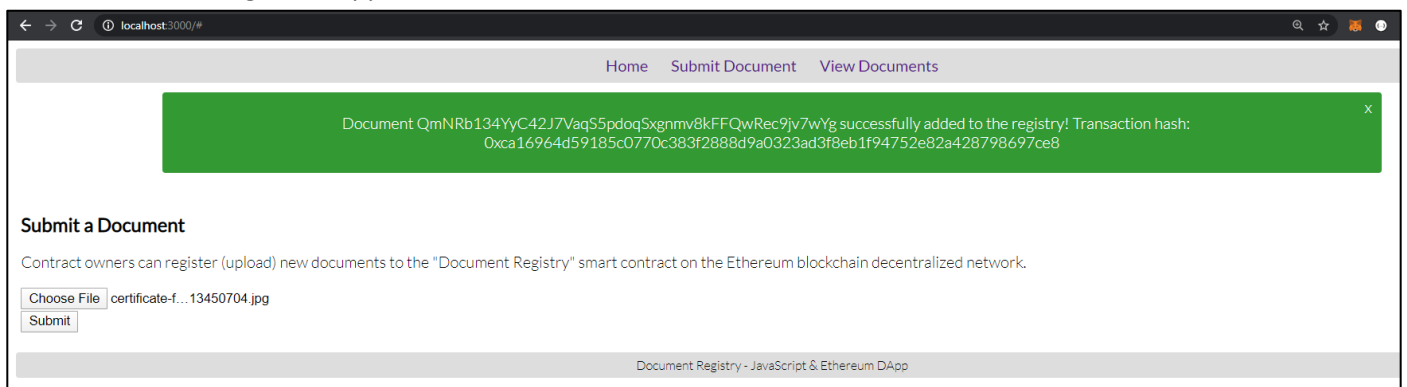
5. Upload any kind of a file in the **[Submit Document]** section. Click **[Submit]**.



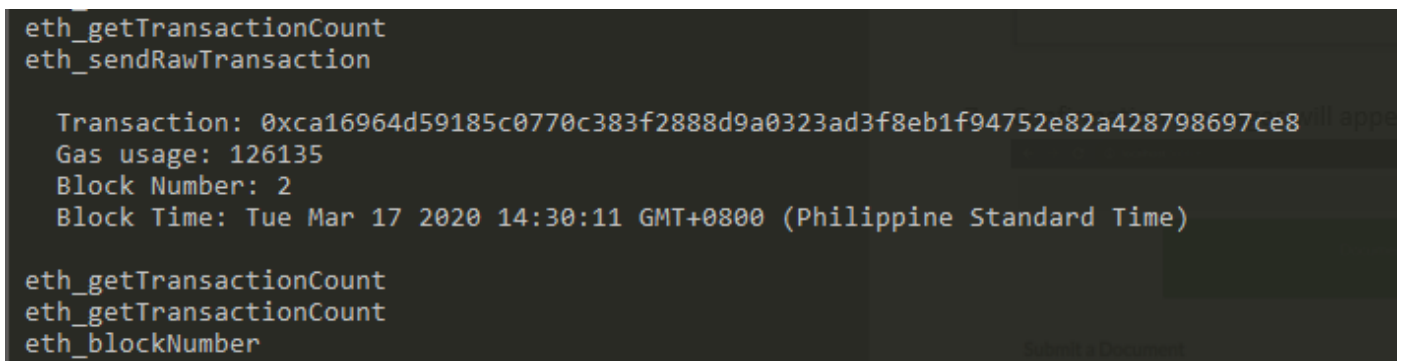
6. A MetaMask prompt will appear, click **[Confirm]**:



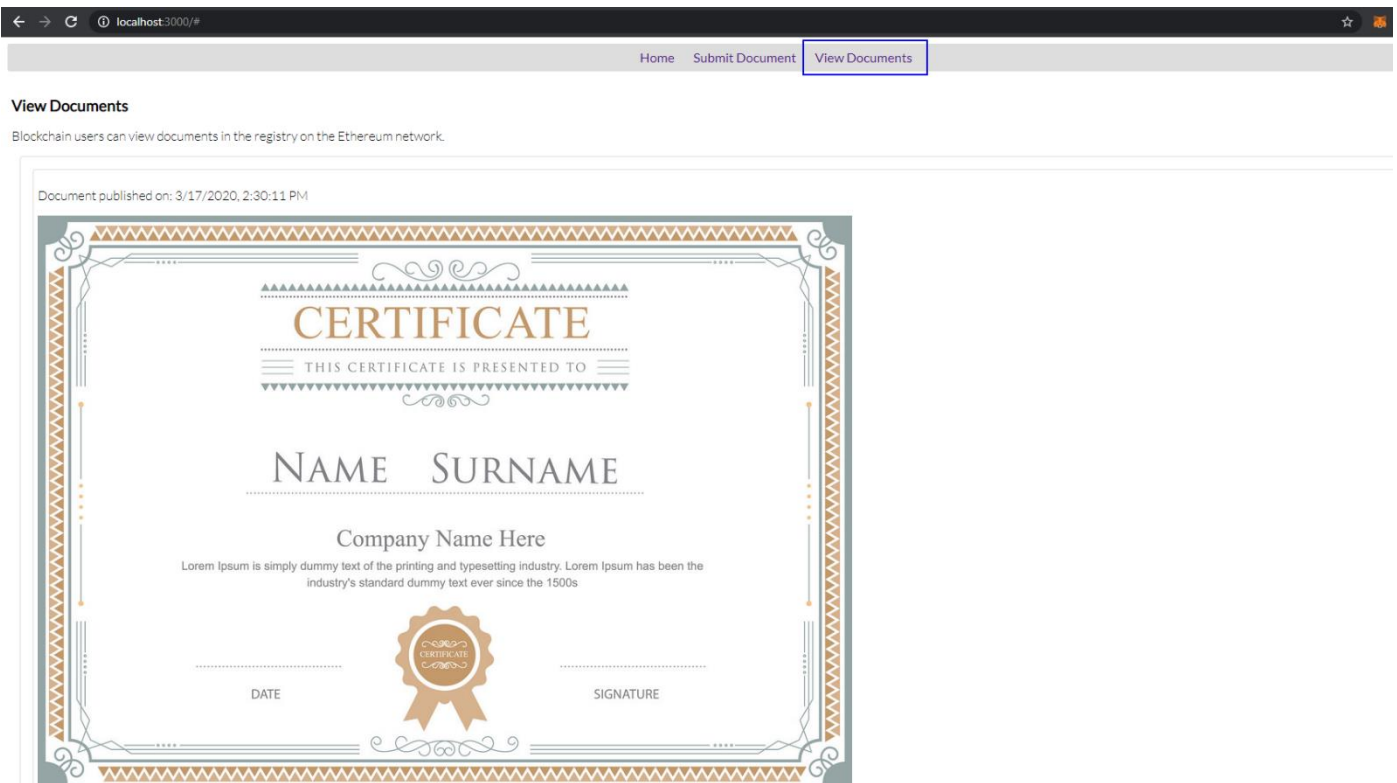
7. Confirmation messages will appear:



In Ganache:



8. Go to [View Documents]:



What to Submit?

Submit a **zip file** (e.g. **your-username-document-registry.zip**) containing the following:

- The source code
 - The smart contract.
 - The client code (remove *node_modules*).
- At least two screenshots
 - The successful submission of a document on the “Submit Document” section.
 - The “View Documents” section containing the submitted document.