Exercises: Create Simple Casino with Solidity, Truffle, MetaMask, Provable and IPFS

Context

Provable Things (https://provable.xyz/) is the leading **oracle** service for smart contracts and blockchain applications, serving thousands of requests every day on **Ethereum** and **Bitcoin** (Rootstock). In the **blockchain** space, an **oracle** is a party which provides **real world data**. The need for such a figure arises from the fact that blockchain applications such as Bitcoin scripts and smart contracts cannot access and directly fetch the data they require: price feeds for assets and financial applications; weather-related information for peer-to-peer insurance; random number generation for gambling, etc.

Prerequisites

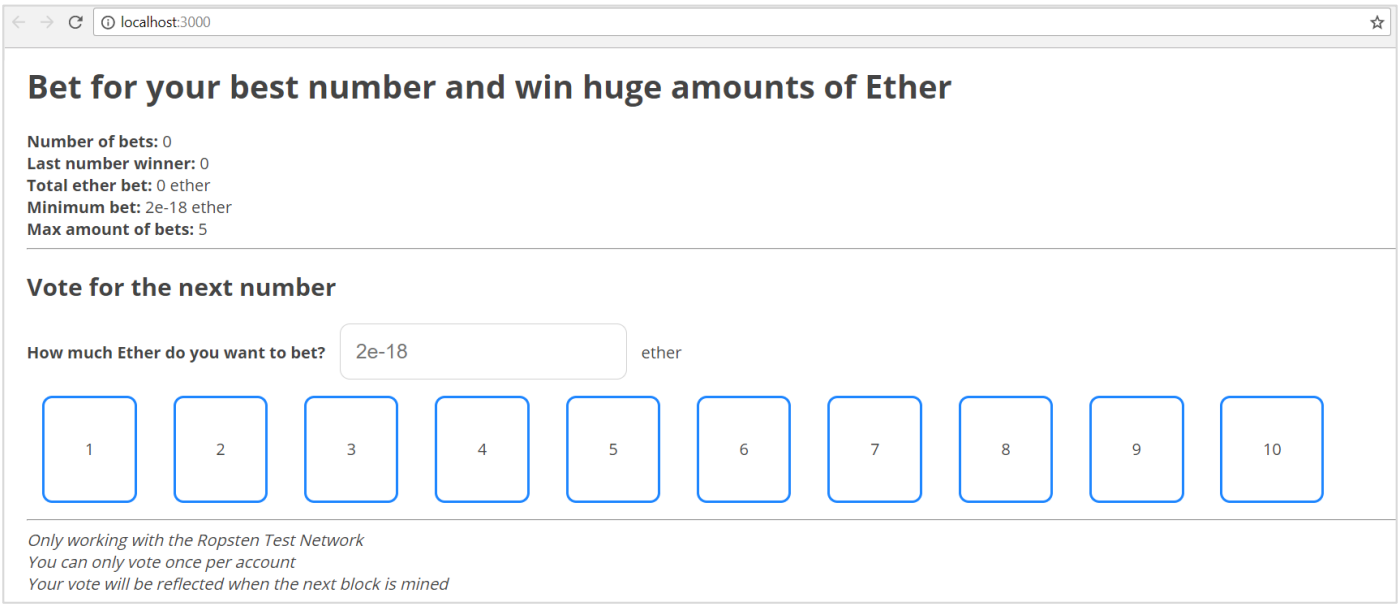
You must have the following software installed, along with corresponding versions:

- **NodeJS** v13.5.0
  - *Check:* ***node -v***
- **NPM** (includes NPX) v6.13.4
  - Check: ***npm -v*** or ***npx -v***
- Able to access https://remix.ethereum.org/
- **Truffle** v5.1.17

Note: If the screenshots in this document seem small/blurry. Zooming in will help improve clarity.

Goal

We are going to create a **decentralized casino application** where users are able to bet money for a number between **1 and 10** and if they are correct, they win a portion of all the ether money staked after 100 total bets.



Setup the Project

1. Clone/Download the project and code templates:

```
git clone https://github.com/kingsland-innovation-center/decentralized-casino.git
```

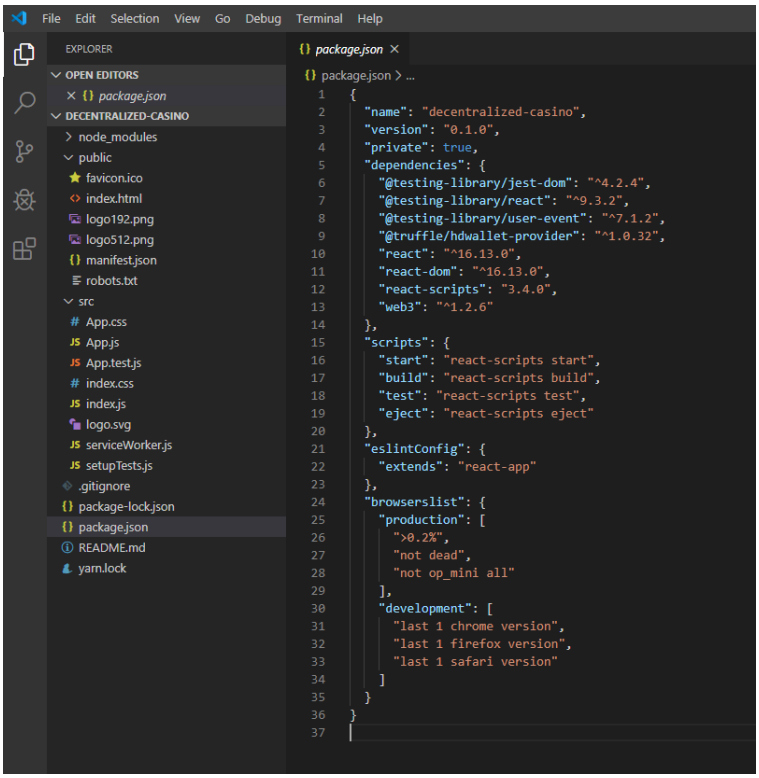
Or you may go here and manually download the project:

https://github.com/kingsland-innovation-center/decentralized-casino

Go to your project folder.
From now on, this will be your workspace. Make sure that the files created and commands executed are being done in this directory:

```
cd decentralized-casino
```

2. Install the dependencies.
Dependency files can be found in the project's **package.json** file.



To install these project dependencies, run:

```
npm install
```

Install **Truffle** and its dependencies globally (don't' forget to run your command shell as **administrator**):

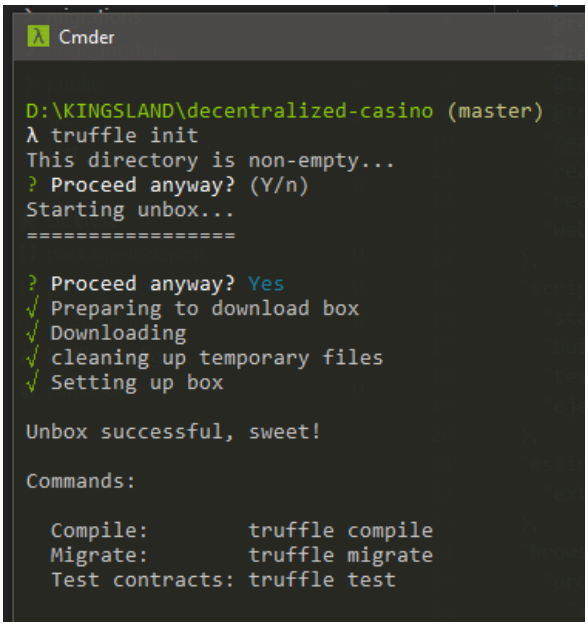
```
If you are using windows, you may want to install this dependency first if there are
errors installing truffle. This will take a while, go grab a snack, perhaps coffee:

npm install --global --production windows-build-tools
```

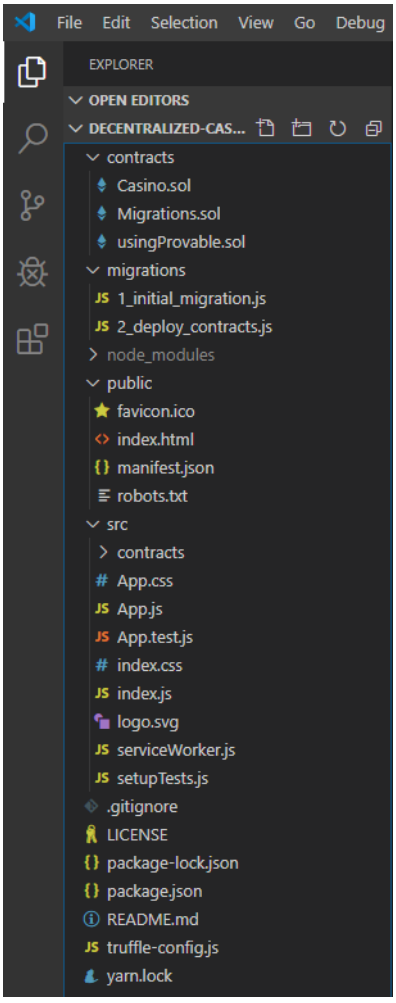
```
npm install --global truffle@5.1.17
```

3. If you are building your own project from scratch, use this command to initialize a **truffle** project with recommended directory structures. **Since you have a preset project, there is no need to do this step as this has already been done for you**:

```
truffle init
```

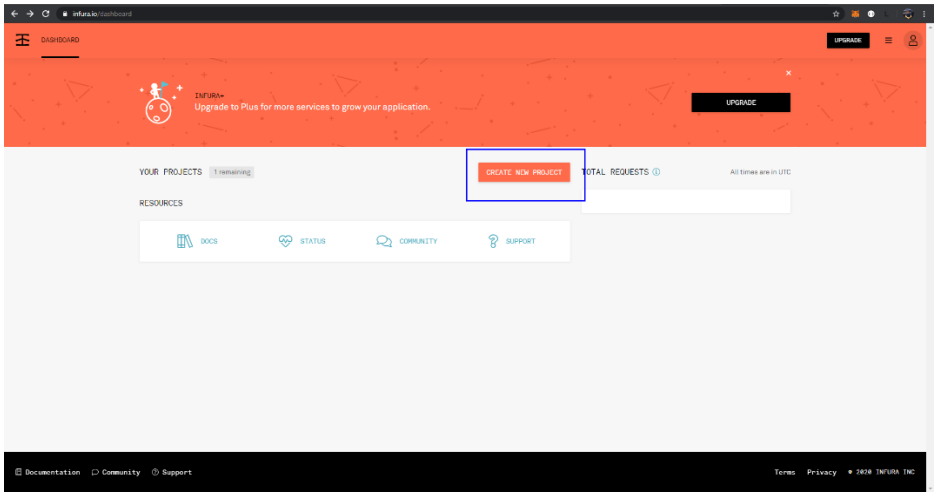


4. You will now have this structure in your project directory:

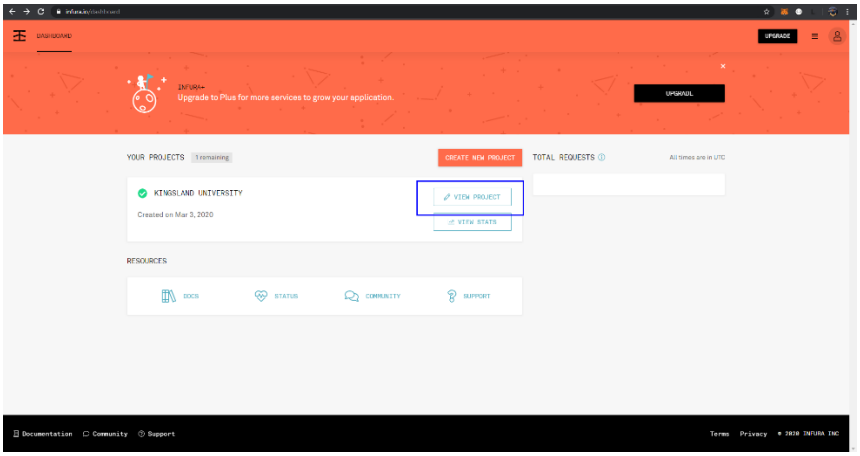


5. Get your Project ID on Infura.
   If you are new, register an account: https://infura.io/register
   If you are an existing user, login: https://infura.io/login

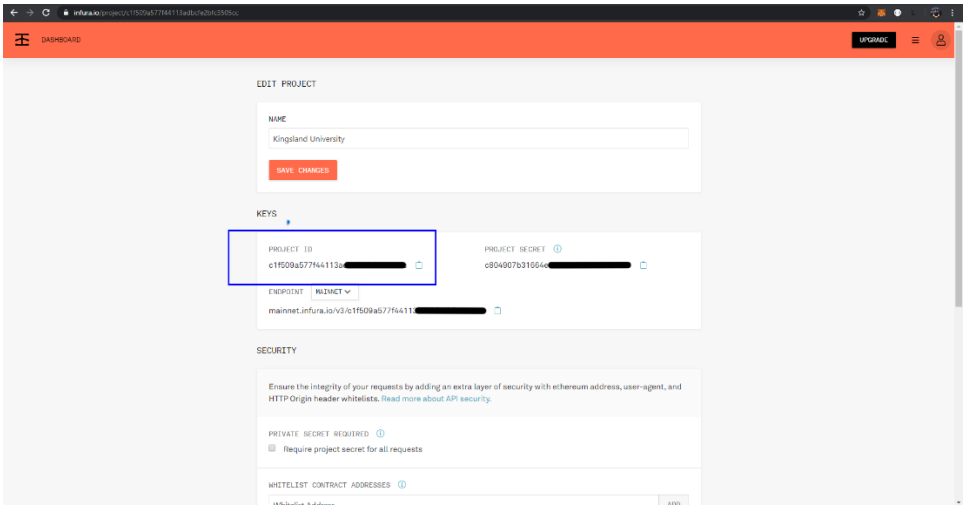
   After logging in, you may use an existing key or create one.



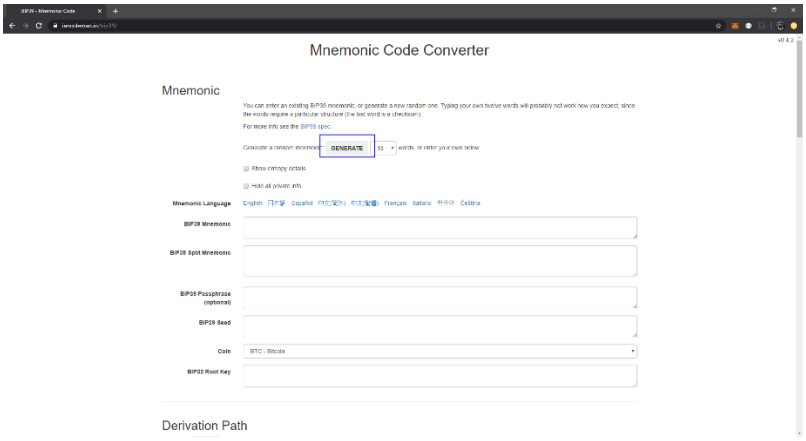
   View your project:



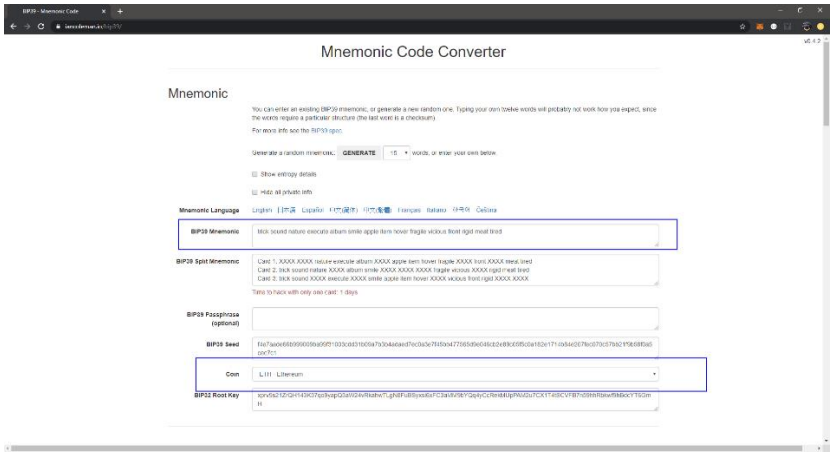
   Take note of your Infura Project ID, you will use this later:



Generate a mnemonic. If you already have your own, you may skip this step.
Let us use a tool to do this quickly: https://iancoleman.io/bip39/
Once you're on the page, click **GENERATE**:

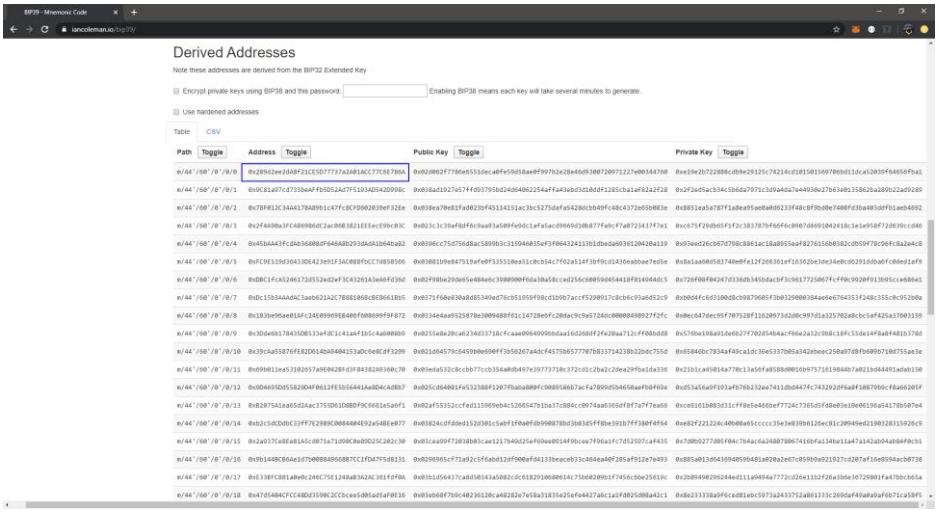


Then, change the **Coin** selection to **ETH – Ethereum** and **copy the mnemonic for use later.**



Scroll down a little further and you'll see a list of addresses along with their corresponding private/public keys. These are the keys that are associated with this mnemonic and will stay the same whenever you use the mnemonic with any BIP39-compliant wallet.

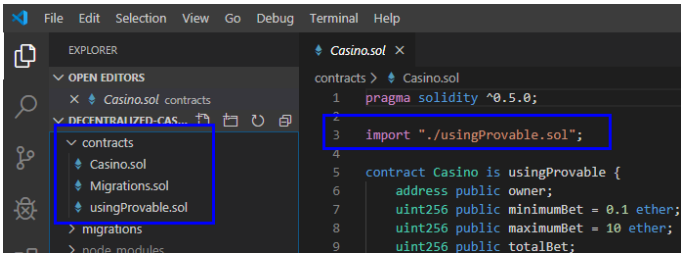
Select the first address and **send** some Ethers to it (using Metamask or your own favorite wallet application) as this account will be used later to deploy the contract:



Problem 1. Create Smart Contracts

1. We will use the **ProvableAPI.** Copy the file linked below to the **`contracts/`** folder with **`usingProvable.sol`** as the file name if you're starting from scratch. Otherwise, you may already have it in your project.

`https://github.com/provable-things/ethereum-api/blob/master/provableAPI_0.5.sol`



2. Create the file **contracts/Casino.sol**, this is the main Solidity contract that we will be writing.

   The Casino contract will have:

   - **`owner`** – **Address** of the owner.
   - **`minimumBet`** (default **0.1 ether**) – The minimum bet a user has to make.
   - **maximumBet** (default **10 ether**)– The maximum bet that can be made for each game.
   - **numberOfBets** – Number of bets that the users have made.
   - **maxNumberOfBets** (default **100**)– The maximum number of bets at a given time to avoid excessive gas consumption.
   - **winningNumber** – The lucky number that decides the winner.
   - **An array of all the players.**
     - `address[] public players;`
   - **A structure for storing the players and their bets.**
     - `mapping(address => uint256) public playerBets;`
   - **A structure for storing each number and which players have bet on that particular number.**
     - `mapping(uint256 => address payable[]) public bets;`
   - **Events** that log some actions of the contract.

```
contracts > Casino.sol
1  pragma solidity ^0.5.0;
2
3  import "./usingProvable.sol";
4
5  contract Casino is usingProvable {
6      address public owner;
7      uint256 public minimumBet = 0.1 ether;
8      uint256 public maximumBet = 10 ether;
9      uint256 public numberOfBets;
10     uint256 public maxNumberOfBets = 100;
11     uint256 public winningNumber;
12
13     address[] public players;
14     mapping(address => uint256) public playerBets;
15     mapping(uint256 => address payable[]) public bets;
16
17     event generatedRandomNumber(string randomNumber);
18     event LogNewProvableQuery(string description);
19
```

3. Now create the constructor that is used to configure:
    - The minimum bet that each user has to make in order to participate in the game.
    - The maximum number of bets that are required for each game.

    You **may** also set the type of authenticity proof of **provable** which are simply cryptographic guarantees proving the authenticity of the data (read more here):

```
19
20      constructor(uint256 _minimumBet, uint256 _maxNumberOfBets) public {
21          owner = msg.sender;
22
23          if (_minimumBet > 0) {
24              minimumBet = _minimumBet;
25          }
26
27          if (_maxNumberOfBets > 0) {
28              maxNumberOfBets = _maxNumberOfBets;
29          }
30
31          provable_setProof(proofType_Ledger);
32      }
33
```

4. Implement the **bet** function:

```
33
34      function bet(uint256 numberToBet) public payable {
35          require(numberOfBets < maxNumberOfBets, "Bet table is full.");
36          require(
37              numberToBet >= 1 && numberToBet <= 10,
38              "Choose a bet number between 1 and 10."
39          );
40          require(
41              msg.value >= minimumBet,
42              "Bet is lesser than specified minimum bet."
43          );
44          require(
45              playerBets[msg.sender] == 0,
46              "You are not allowed to change your bet."
47          );
48
49          playerBets[msg.sender] = numberToBet;
50          bets[numberToBet].push(msg.sender);
51          players.push(msg.sender);
52
53          numberOfBets += 1;
54
55          if (numberOfBets >= maxNumberOfBets) {
56              generateWinningNumber();
57          }
58      }
59
```

5. We should generate winner number by using provable function **provable_newRandomDSQuery** which takes **delay**, **numberRandomBytes** and **callbackGas:**

```
59
60      function generateWinningNumber() public payable {
61          uint256 delay = 0;
62          uint256 numberRandomBytes = 7;
63          uint256 callbackGasLimit = 400000;
64
65          provable_newRandomDSQuery(delay, numberRandomBytes, callbackGasLimit);
66          emit LogNewProvableQuery(
67              "Provable query was sent, standing by for the answer..."
68          );
69      }
70
```

6. We should create a **callback** function which gets called by Provable when a random number is generated.
   - **`_queryID`** – A unique ID that identifies a specific query done to Provable and it is returned to the contract as a parameter of the callback transaction.
   - **`_result`** – A string that contains the generated random number from Provable.
   - **`_proof`** – A signature which proves that the response indeed came from Provable.

   These can then be verified by calling **`provable_randomDS_proofVerify__returnCode()`** function.

   When all goes well, we process the random number to get it within our "bounds" which is from 1 to 10. Then, we distribute the prizes.

```
70
71      function __callback(
72          bytes32 _queryID,
73          string memory _result,
74          bytes memory _proof
75      ) public {
76          require(msg.sender == provable_cbAddress(), "Bad callback");
77          require(
78              provable_randomDS_proofVerify__returnCode(
79                  _queryID,
80                  _result,
81                  _proof
82              ) ==
83                  0,
84              "Bad proof"
85          );
86          emit generatedRandomNumber(_result);
87
88          bytes32 encodedRandom = keccak256(abi.encodePacked(_result));
89          winningNumber = (uint256(encodedRandom) % 10) + 1;
90          distributePrizes();
91      }
92
```

7. Implement the function to send the corresponding Ether to each winner then reset the bets by deleting all the players for the next game and resetting the **total bet** and **number of bets**:

   Make sure to handle the case when there is no winner: **`bets[winningNumber].length != 0`**

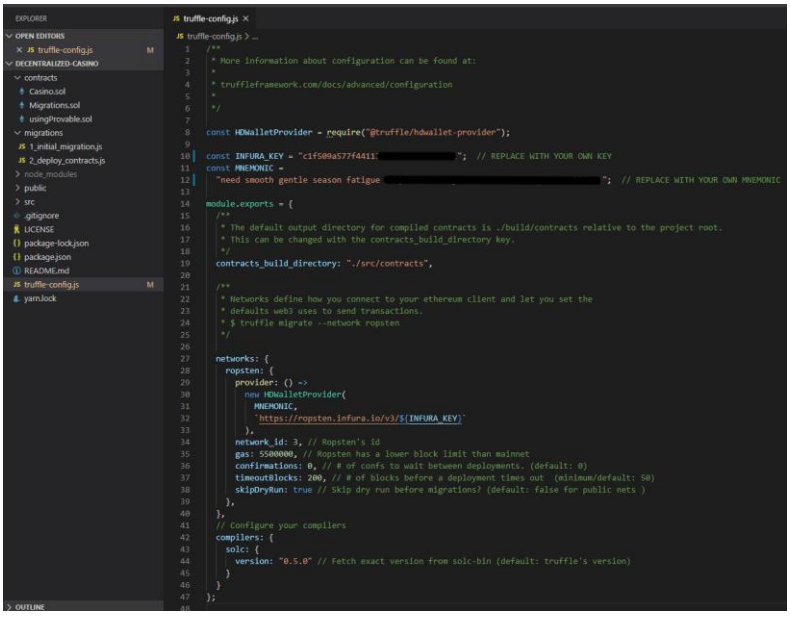
```
92
93      function distributePrizes() public {
94          if (bets[winningNumber].length != 0) {
95              uint256 prize = address(this).balance / bets[winningNumber].length;
96
97              for (
98                  uint256 index = 0;
99                  index < bets[winningNumber].length;
100                 index++
101             ) {
102                 address payable winner = bets[winningNumber][index];
103                 winner.transfer(prize);
104             }
105         }
106
107         resetBets();
108     }
109
110     function resetBets() private {
111         for (uint256 index = 1; index <= 10; index++) {
112             bets[index].length = 0;
113         }
114
115         for (uint256 index = 0; index < players.length; index++) {
116             playerBets[players[index]] = 0;
117         }
118         numberOfBets = 0;
119     }
120
```

8. Finally, implement a **`getContractBalance()`** view function so that the frontend can see the total contract balance.

```
120
121     function getContractBalance() public view returns(uint256 balance) {
122         return address(this).balance;
123     }
124 }
125
```

9. We are now ready to deploy the contracts to the Ropsten Test Network.
   Go to the **`migrations/`** folder and create the file **`2_deploy_contracts.js`** and write the code below:
   - First, we require the **`Casino.sol`** contract.
   - Then, in the **`.deploy()`** method we specify the minimum bet, in this case it's **0.1 ether** converted to wei with that function
   - Constructor arguments
     - **0.1** is the minimum bet. Use the *web3.utils* library to convert the unit into *wei*.
     - **2** is the maximum number of bets (for testing purposes).
   - Finally, the gas limit that we are willing to use to deploy the contract. Let's do **5,000,000**.

10. Open **truffle-config.js** from the root folder and customize your Truffle configuration
    - **Use your own Infura Project ID key from the Infura settings in an earlier step**
    - **Use your own mnemonic you had generated in an earlier step**



Problem 2. Deploy the application online with IPFS

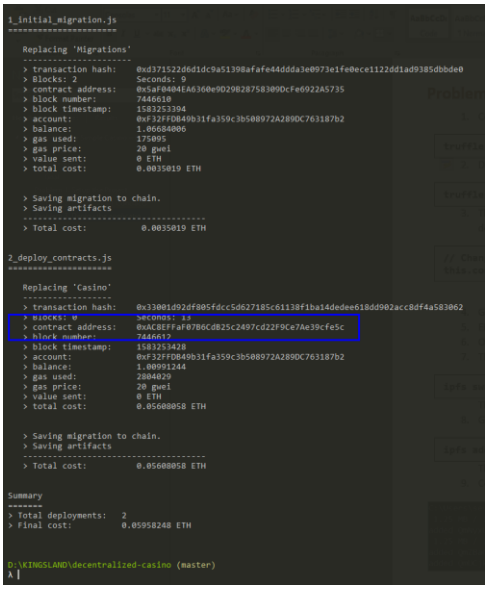
1. Compile your contract:

```
truffle compile
```

2. Deploy the contract on the Ropsten test network.

```
truffle migrate --network ropsten
```

Take note of the deployed Casino contract address:

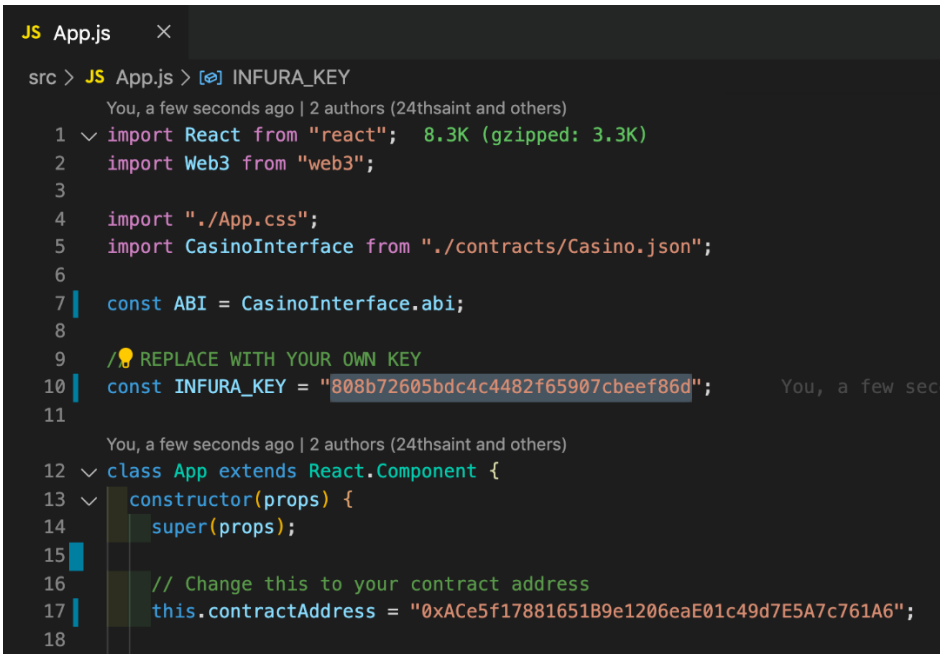


3. Then, in **src/app.js**, change the address of the contract instance to the address of the contract you deployed on Ropsten:

```
// Change this to your contract address
this.contractAddress = "0xACe5f17881651B9e1206eaE01c49d7E5A7c761A6"
```

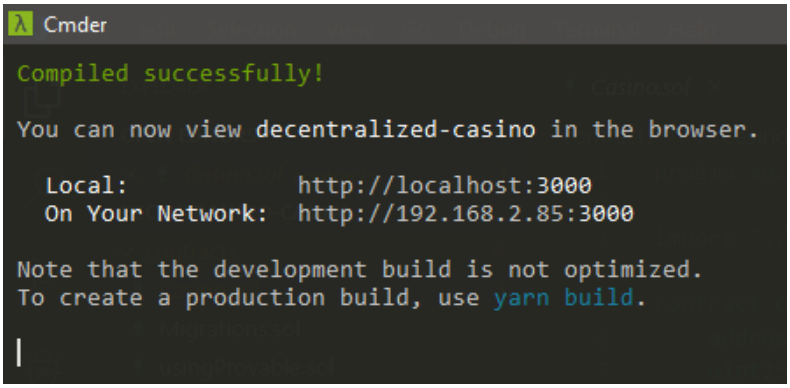
4. Also, in **src/app.js**, change the INFURA_KEY to the Project ID Key from the Infura settings in an earlier step:

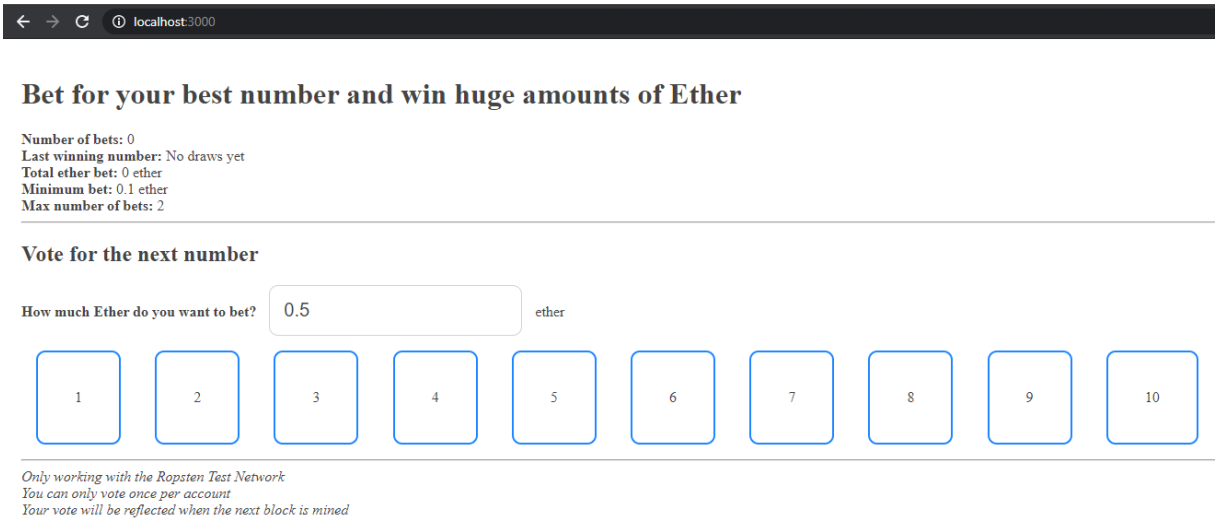
```
// REPLACE WITH YOUR OWN KEY
const INFURA_KEY = "808b72605bdc4c4482f65907cbeef86d";
```



5. Sanity Checking.
   Make sure that your dApp will run correctly. This command will create a local server for your files and automatically launch your browser at **http://localhost:3000**

```
npm start
```





At this point, you can interact with your dApp.

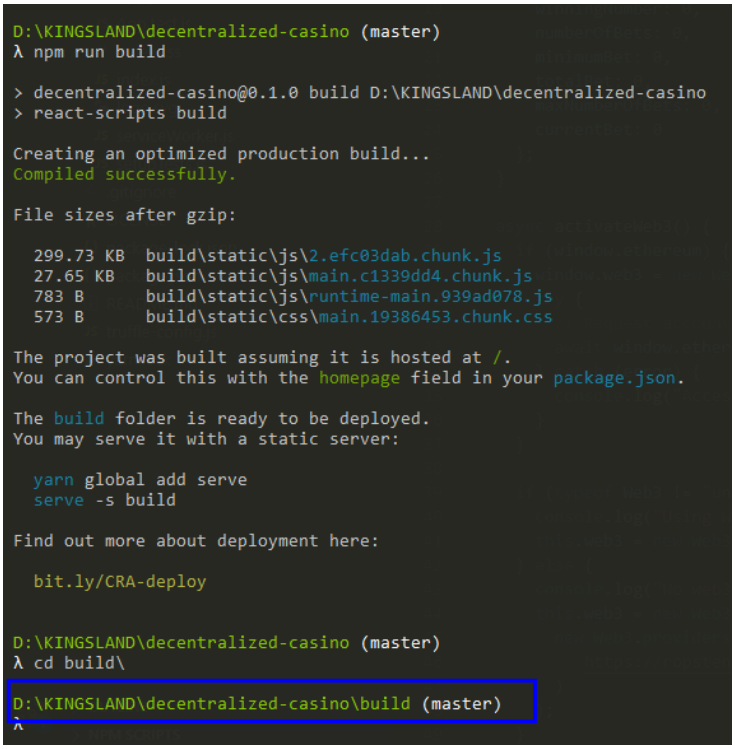
Try to place some bets, change your account in Metamask, and place another bet.
Keep doing until you reach the maximum number of bets and your smart contract generates a random number from Provable.

6. You are now ready to deploy a **decentralized application on IPFS**!

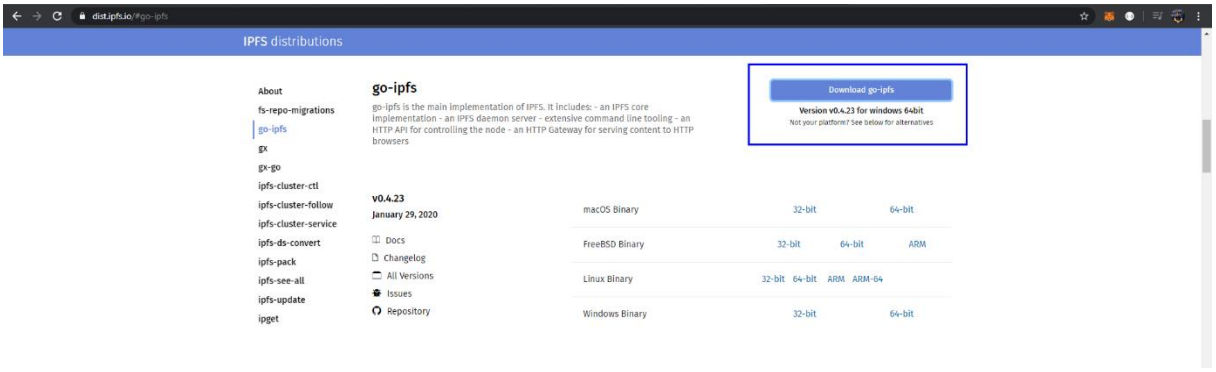
   Compile your ReactJS project.

```
npm run build
```

   This will create a directory named **`build/`**
   This contains all your source files in **`src/`** bundled together with optimizations.
   Take note of the location of this directory.



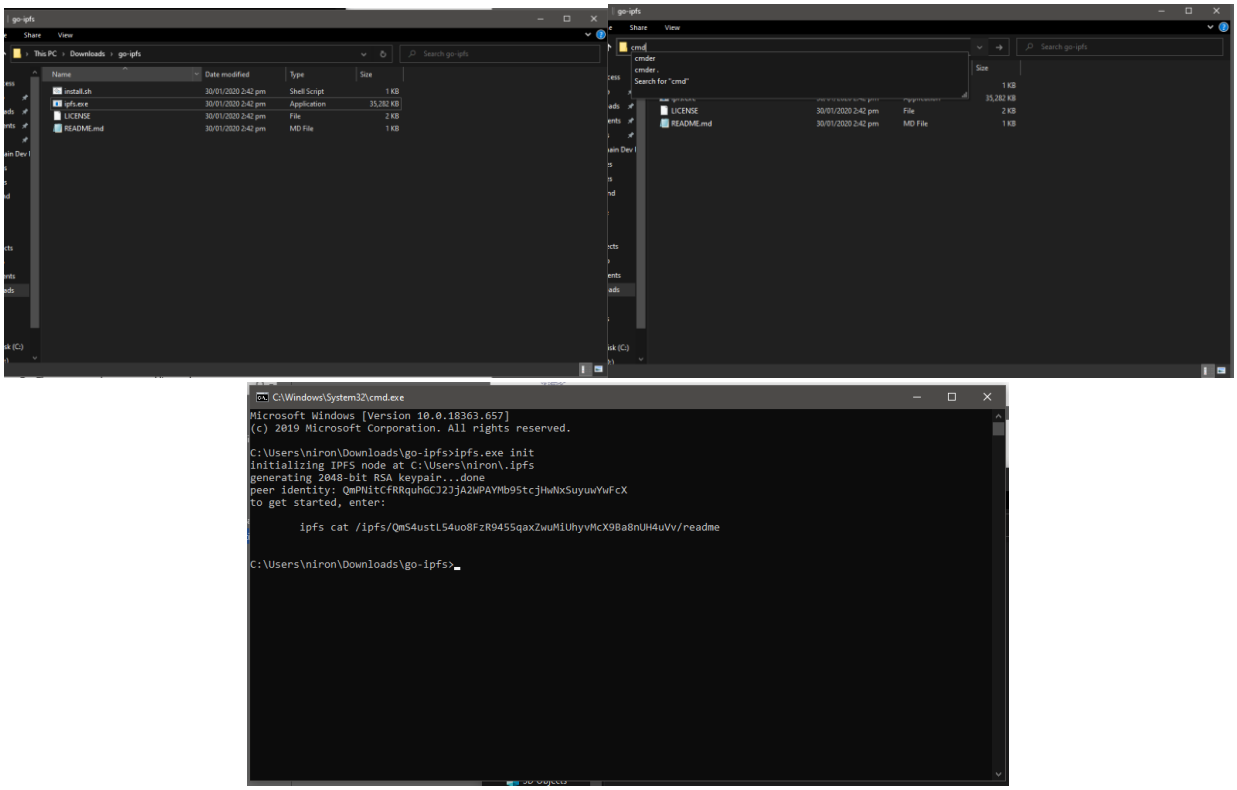
7. Go to https://dist.ipfs.io/#go-ipfs and download **go-ipfs** then extract it.



8. Go *to the IPFS folder* and run in Command Prompt:

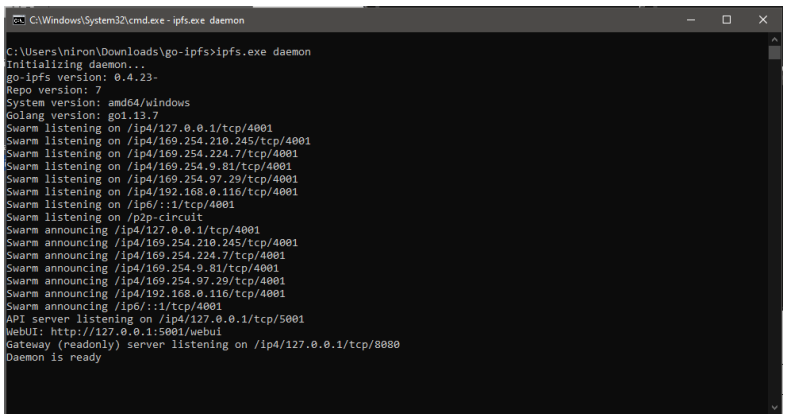
ipfs.exe init
If you are using Mac OS, run the install script first, then initialize: **./install.sh** **ipfs init**

(Advanced users: You may add IPFS as a *path variable* to easily access the command anywhere.)



9. Open command line and type **ipfs daemon.** This will make your machine node and IPFS node.

Ipfs.exe daemon
If you are using Mac OS, type: **ipfs daemon**

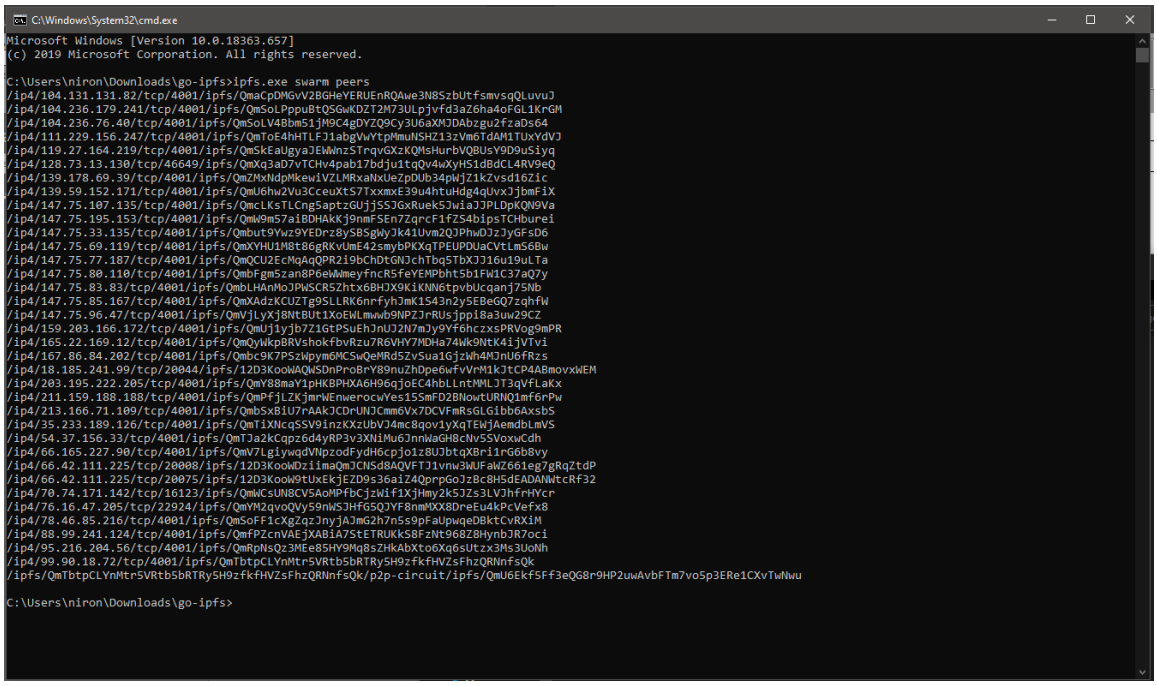


Keep this terminal running, do not terminate the process.

10. Open another *separate* command line and type:

```
ipfs swarm peers
```

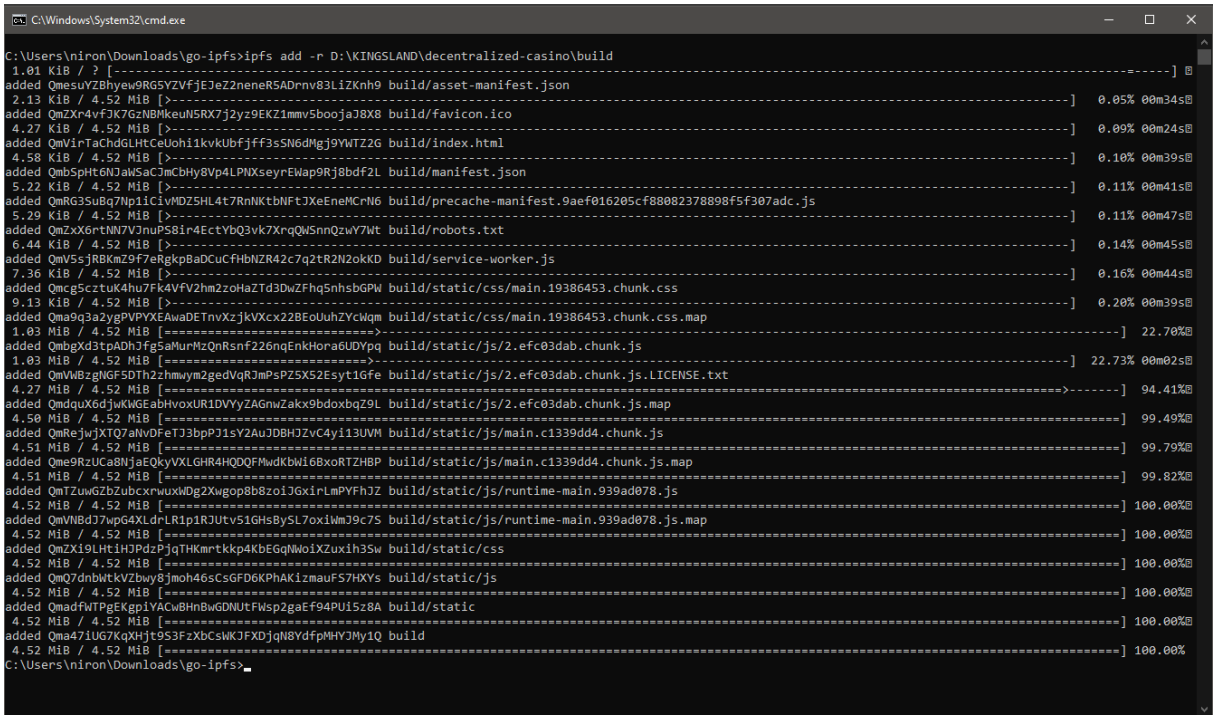
This will get you the peers that your machine has established a connection with. These peers are ready to share your published content.



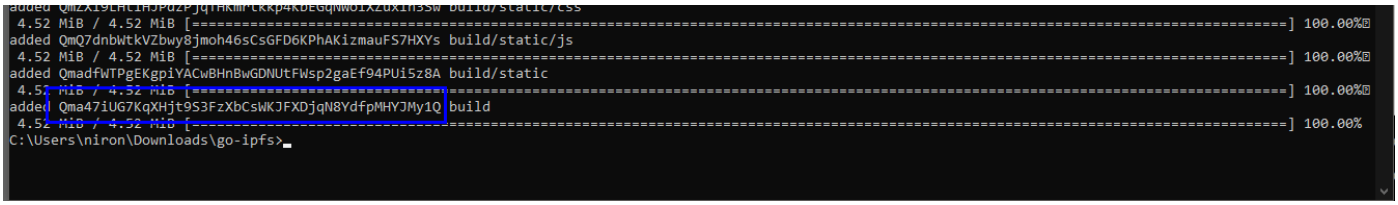
11. Get the path of your **`build/`** folder (*review step 6*) and run the command:

```
ipfs add -r <build_folder_location>
```

This will add your folder to the IPFS network.

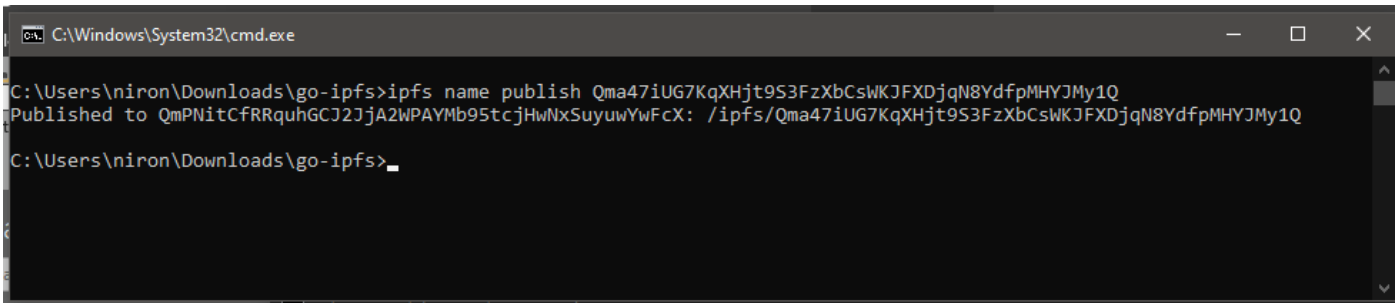


12. Copy the last hash. For example, **"Qma47iUG7KqXHjt9S3FzXbCsWKJFXDjqN8YdfpMHYJMy1Q":**



13. Run the following command to finally publish your files in the IPFS network:

```
ipfs name publish <hash_of_build_folder>
```

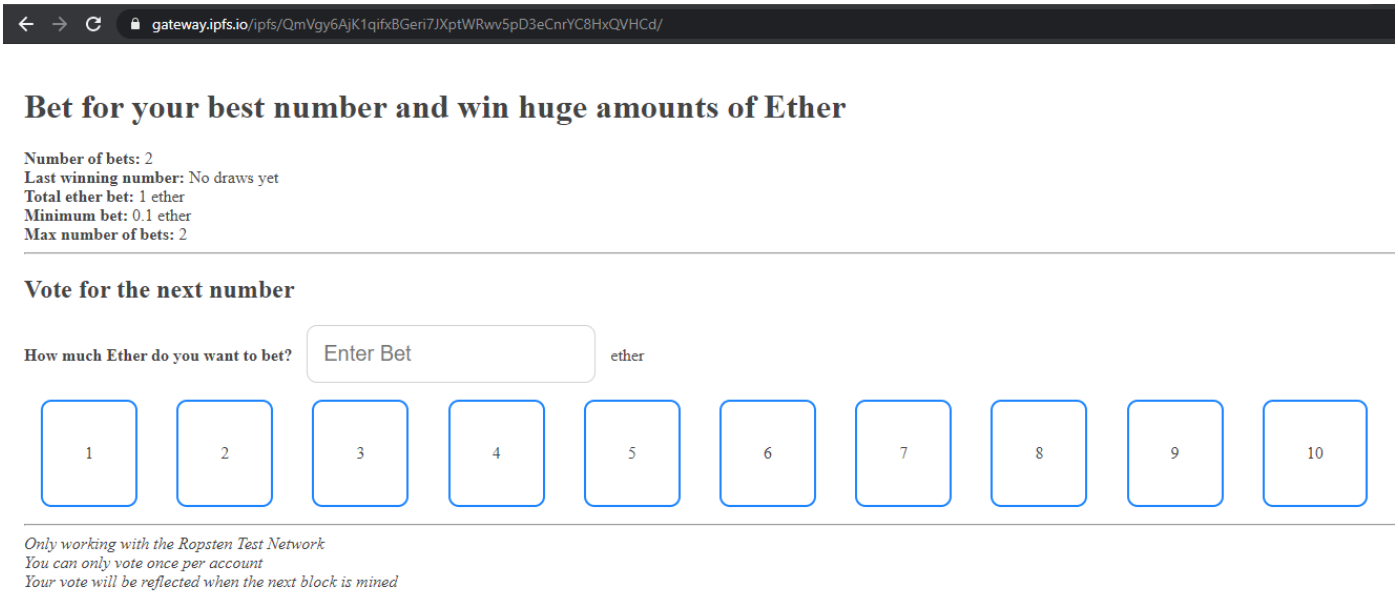


14. Open the following link. For example:
    https://gateway.ipfs.io/ipfs/QmbiA7itaE3uomsJq6XPDqbwbiwYzMCzrYL6uz7Xa4AwY8/

15. If you make changes to your files remember to execute:

```
npm run build
ipfs add -r <build_folder_location>
ipfs name publish <hash_of_build_folder>
```

16. Congratulations, you just deployed your decentralized application in the IPFS network!



What to Submit?

Create a **zip file** (e.g. **`your-name-decentralized-casino-metamask-provable-exercise.zip`**) containing the following:

1. A **links.txt** file containing your:
    a. IPFS dApp link.
    b. Etherscan contract link.
2. Screenshots of the terminal: **truffle compilation, truffle migration**.

Submit your **zip** file as **homework** at the course platform.