

# ERC721 + Vue.js CryptoKitties-like Dapp in under 10 minutes



Sam Benemerito

Follow

Feb 19 · 9 min read



You might have heard of CryptoKitties, an Ethereum-based platform where you can collect, buy, sell, and even breed digital cats, and about how people have been spending a crazy amount of money in the game.

Making a CryptoKitties-like application is actually quite easy! And in this tutorial, we'll be doing just that.


## What We Are About To Build

From here onwards, we will be building a simple CryptoKitties-like dapp, which we will call CryptoVipers (vipers, because why not?). In our simple application, we will be able to buy vipers, breed vipers, and view the ones we own. Which will be something like this:

## CryptoVipers

Collect and breed digital vipers.

Buy a random Viper



Buy

Each Viper costs 0.02 Ether

or

Breed two of the Vipers you own to make a new one!

Matron ID:

Sire ID:

Breed Vipers

Breeding Vipers cost 0.05 Ether

Owned Vipers

## Function List

1. Viper Creation—we should be able to own new vipers by **buying** one or **breeding** two of them
2. View Viper Details—we should be able to retrieve our vipers' details and **display** it.

## Tools

1. Smart Contract  
[Solidity](#), [Remix](#), [Metamask](#)
2. Frontend  
[Web3.js](#), [Vue.js](#), [Vue-cli](#), [Bootstrap-vue](#)

Aside from that, we will also be using [Git](#) later to clone our boilerplate :)

## Prerequisites

Before proceeding with this tutorial, you should first read the following articles:

1. [Getting Started with MetaMask](#)
2. [Compile and Deploy Using Remix IDE](#)
3. [Introduction to Smart Contracts and Solidity](#)

## Understanding ERC721

**ERC721** is a standard that describes how non-fungible (or **unique tokens**) on the Ethereum blockchain should be made. With ERC721, each item or token is unique, meaning it is not equal to any other token. You could think of each one as a rare, **one-of-a-kind collectible**.

### ERC721 Interface

According to the specifications, ERC721 defines a minimum interface a smart contract must implement to allow unique tokens to be managed, owned, and traded:

- **balanceOf( \_owner )**—Returns the number of tokens in a specific \_owner's wallet.
- **ownerOf( \_tokenId )**—Returns the wallet address of the specific token's owner.
- **totalSupply()**—Returns the total amount of tokens created
- **transfer( \_to, \_tokenId )**—Transfers token with \_tokenId from sender's wallet to a specific wallet.
- **takeOwnership( \_tokenId )**—Claims the ownership of a given token ID
- **approve( \_to, \_tokenId )**—Approves another address to claim for the ownership of the given token ID

Also, it defines two events: **Transfer**, and **Approval**. A *Transfer* event is emitted when a token is transferred from one wallet to another. On the other hand, an *Approval* event is emitted when an address (user) approves another address to claim the ownership of a certain token that he owns.

### OpenZeppelin ERC721 Implementation

OpenZeppelin provides reusable smart contracts with implementations of standards like ERC20 and ERC721. In this case, we will be importing their ERC721 implementation so we don't have to write it ourselves from scratch.

# Making the Project

## Workflow

1. Creating the Smart Contract
2. Building the Web App

## Creating the Smart Contract

Since our clone will also be based on Ethereum, our token will also be made using Solidity, one of the programming languages used for creating smart contracts.

In Remix, create a new file named **ViperToken.sol** and add the following code:

```

1  // We will be using Solidity version 0.5.3
2  pragma solidity 0.5.3;
3  // Importing OpenZeppelin's ERC-721 Implementation
4  import "https://github.com/OpenZeppelin/openzeppelin-
5  // Importing OpenZeppelin's SafeMath Implementation
6  import "https://github.com/OpenZeppelin/openzeppelin-
7
8
9  contract ViperToken is ERC721Full {
10     using SafeMath for uint256;
11     // This struct will be used to represent one viper
12     struct Viper {
13         uint8 genes;
14         uint256 matronId;
15         uint256 sireId;
16     }
17
18     // List of existing vipers
19     Viper[] public vipers;
20
21     // Event that will be emitted whenever a new viper
22     event Birth(
23         address owner,
24         uint256 viperId,
25         uint256 matronId,
26         uint256 sireId,
27         uint8 genes
28     );
29
30     // Initializing an ERC-721 Token named 'Vipers' w
31     constructor() ERC721Full("Vipers", "VPR") public
32     {
33
34     // Fallback function
35     function() external payable {
36     }
37
38     /** @dev Function to determine a viper's character
39     * @param matron ID of viper's matron (one parent)
40     * @param sire ID of viper's sire (other parent)
41     * @return The viper's genes in the form of uint

```

```
42     */
43     function generateViperGenes(
44         uint256 matron,
45         uint256 sire
46     )
47         internal
48         pure
49         returns (uint8)
50     {
51         return uint8(matron.add(sire)) % 6 + 1;
52     }
53
54     /** @dev Function to create a new viper
55      * @param matron ID of new viper's matron (one p
56      * @param sire ID of new viper's sire (other par
57      * @param viperOwner Address of new viper's owne
58      * @return The new viper's ID
59      */
60     function createViper(
61         uint256 matron,
62         uint256 sire,
63         address viperOwner
64     )
65         internal
66         returns (uint)
67     {
68         require(viperOwner != address(0));
69         uint8 newGenes = generateViperGenes(matron, s
70         Viper memory newViper = Viper({
71             genes: newGenes,
72             matronId: matron,
73             sireId: sire
74         });
75         uint256 newViperId = vipers.push(newViper).su
76         super._mint(viperOwner, newViperId);
77         emit Birth(
78             viperOwner,
79             newViperId,
80             newViper.matronId,
81             newViper.sireId,
82             newViper.genes
83         );
```

```
84         return newViperId;  
85     }  
86
```

We're basically importing OpenZeppelin's ERC721 implementation, and adding our custom functions. We're also importing SafeMath to avoid Integer Overflow and Underflow. Now, to explain each function:

First, the function **generateViperGenes** is responsible for determining the new viper's genes, especially when it is being created through breeding (getting both parents' genes). This is a very simple implementation though, and in this case, we use a simple number from 1–6 to determine how the viper will look like. You could do something better if you feel like it :)

Then, we have the function **createViper** which basically creates a new viper and gives it to the address passed in the parameter, "viperOwner".

The function **buyViper** is a *payable* function (accepts payments) and calls the **createViper** function we previously defined. The word `payable` is a modifier that is used to indicate that this function can receive ether when you execute it. Now, you might have noticed this part:

```
require(msg.value == 0.02 ether);
```

The **require** statement checks a condition and raises an error when false. In Solidity, `msg.value` holds the amount of Ether being sent by the one who called the function. In this case, we want the user to pay 0.02 ether for buying a random viper. You may change this amount, or even remove it if you wish to.

We then have the **breedVipers** function which takes 0.05 ether as payment, and creates a new viper based off the two parents' genes.

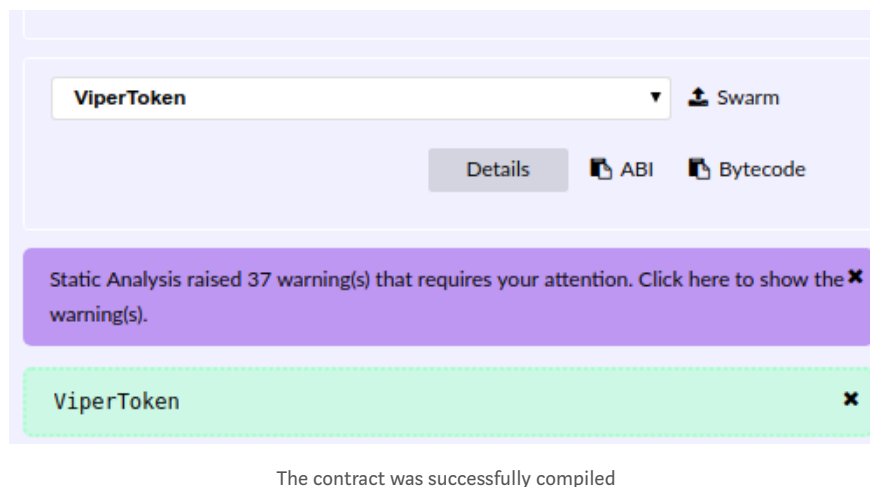
Next, we have the function **getViperDetails** which just returns details such as genes, and parent IDs of a given viper.

Finally, we have the **ownedVipers** function which returns a list of viper IDs that the function caller owns. The word `msg.sender` is the address of the user that executed that function.

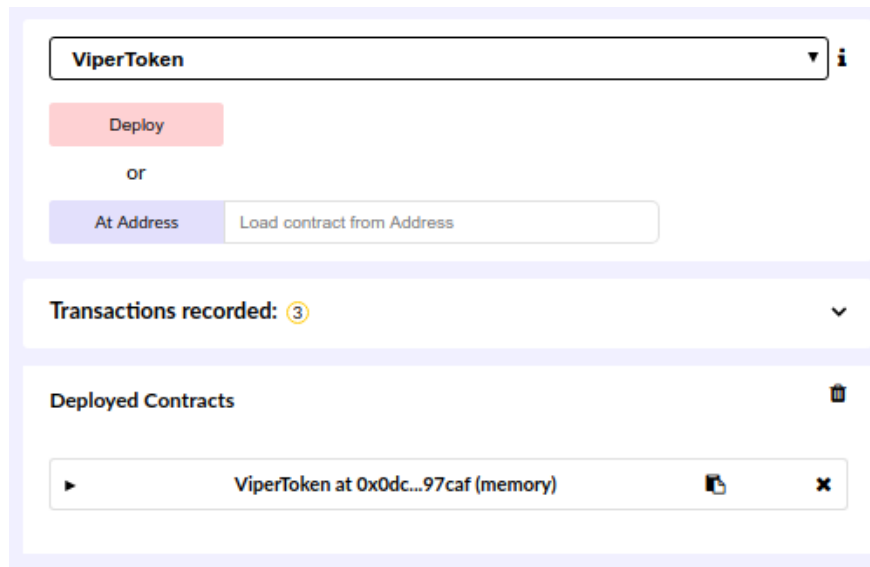
Now, compile the **ViperToken** contract (make sure you select compiler version at the right side of Remix, and choose **0.5.3+commit.10d17f24** because we are using Solidity version 0.5.3) and deploy it to the **Ropsten Test Network**. Make sure you are compiling and deploying the **ViperToken** contract.

## — ✓ Let's check it out. —

To check that our contract was deployed, you should see the following:







The contract was successfully deployed and is listed in 'Deployed Contracts'

## Building the Web App

Our smart contract now works, but there's no fun in just looking at numbers so we'll be making a simple web application.

### Setting Up

To get up to speed, let's clone a boilerplate project (found [here](#)) by doing the following in a Terminal (or Command Prompt/Powershell for Windows):

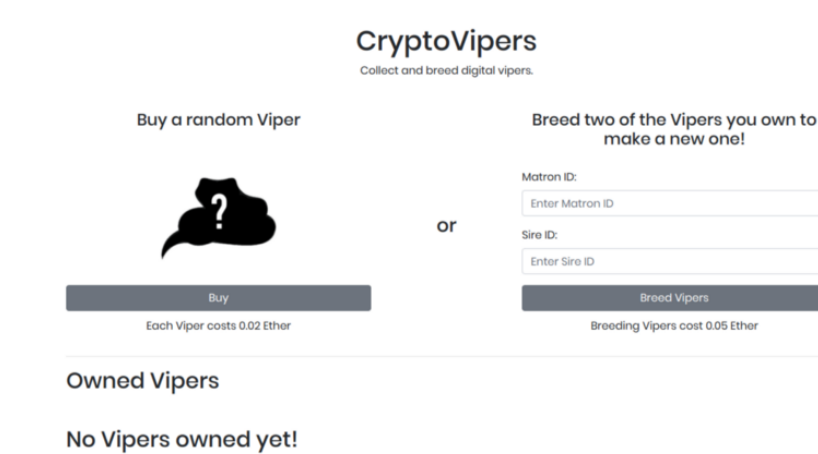
```
# Cloning the boilerplate from GitHub
git clone -b boilerplate --single-branch
https://github.com/openberry-ac/cryptovipers.git

# Navigating to the directory and installing packages
cd cryptovipers
npm install

# Installing Web3
npm install -s web3@1.0.0-beta.37

# To run the app
npm run dev
```

Voila! In a few minutes, you should see the app running through a browser on <http://localhost:8080> looking like this (though it is not functional yet):



CryptoVipers' index page

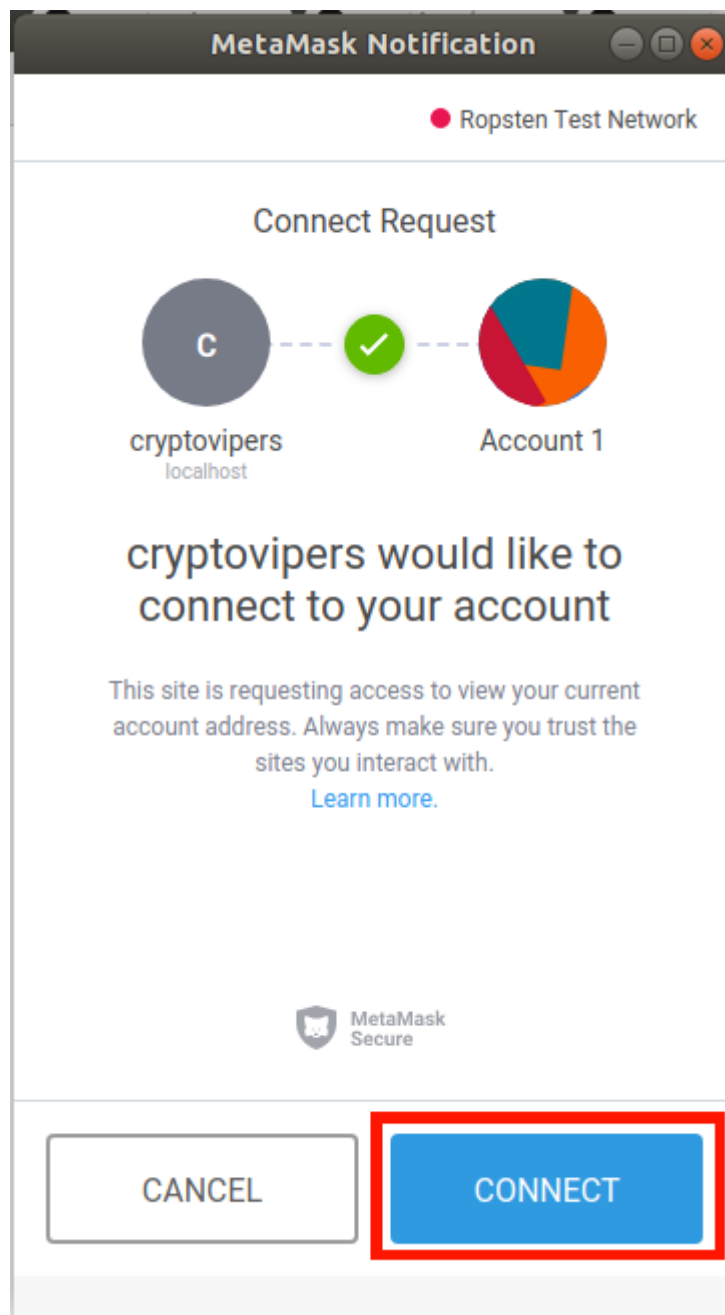
## Connecting to Our Smart Contract Instance

To enable our web app to interact with our smart contract, we will be using [web3.js](#). We already have the package installed, you can see how it is called in the file named **web3.js** inside the “contracts” folder, where we should put something like this:

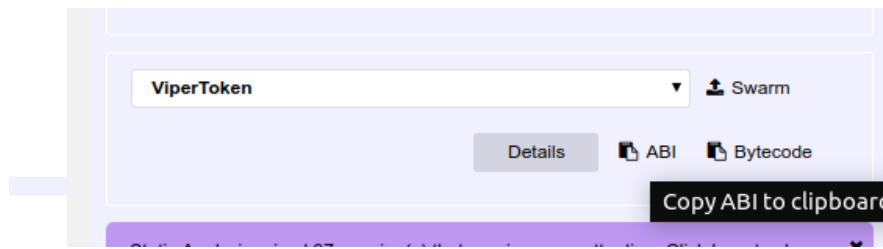
```
1  import Web3 from 'web3';
2
3  const getWeb3 = () => new Promise((resolve) => {
4    window.addEventListener('load', () => {
5      let currentWeb3;
6
7      if (window.ethereum) {
8        currentWeb3 = new Web3(window.ethereum);
9        try {
10          // Request account access if needed
11          window.ethereum.enable();
12          // Accounts now exposed
13          resolve(currentWeb3);
14        } catch (error) {
15          // User denied account access...
16          alert('Please allow access for the app to work');
17        }
18      } else if (window.web3) {
19        window.web3 = new Web3(window.web3.currentProvider);
```

It basically loads the web3 instance the MetaMask extension initializes, which we will be needing later to interact with our smart contract.

You might encounter a MetaMask pop-up window that asks for access permission. This is because we have `ethereum.enable()` where the app requests for account (or wallet) access. You should just click the 'Connect' button right here:



Now, we need our smart contract's ABI to connect it to our web app. To get the ABI, go back to Remix, go to the **Compile** tab, and click **ABI** beside the Details button as shown in the picture:

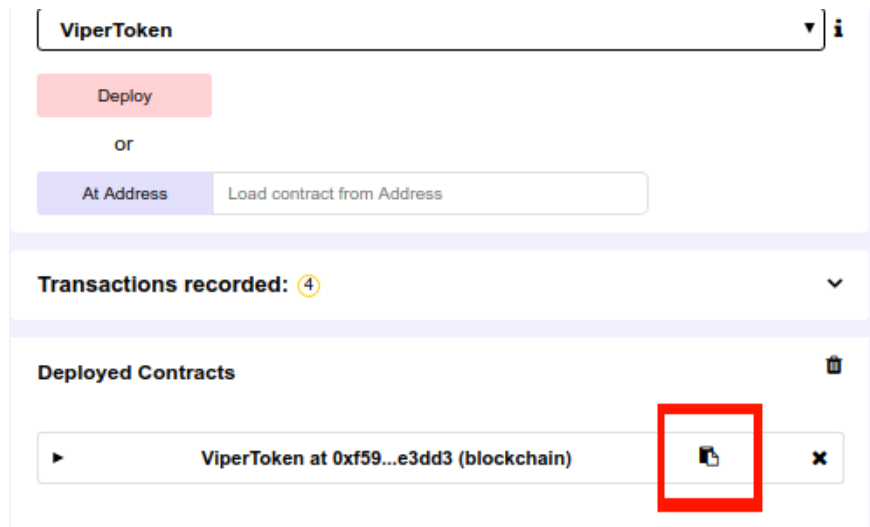


Copy the ABI using the ABI Button

After getting it, open the file named **abi.js** in the **contracts** folder, then paste it as the variable **contractAbi**'s value, like this:

```
1  const contractAbi = // PASTE ABI CODE HERE
2
3  export default contractAbi;
```

There should be an example in the file, which you can always refer to. Then, we will have to specify the smart contract's instance address too, which you can get by going to [Remix's](#) Deploy tab, and clicking the copy icon on your deployed contract, as shown in this picture:



Copy the Instance address using the Copy Button

Open **App.vue** found inside the **src** folder, and paste your contract address on line 86 as the variable **contractAddress**' value (there should also be comments in the file, which you can always refer to):

```
const contractAddress = ''; // Right here!  
// Ex: const contractAddress =  
'0xf59c4c3c79071d3e11034a9344789bd3';
```

## Defining the Methods

You might notice that the user interface is there, but the buttons aren't functional. That's because we have not defined our functions yet, which we will be doing now. Go back to **App.vue**, and go to line 116 where you can see **methods**, but everything just contains a *console.log()*.

Our first function is for buying a viper. Let's modify the *buyViper()* function to look like this:

```
1  buyViper() {  
2    this.isLoading = true;  
3    this.contractInstance.methods.buyViper().send({  
4      from: this.account,  
5      value: web3.toWei(0.02, 'ether'),  
6    }).then((receipt) => {  
7      this.addViperFromReceipt(receipt);  
8      this.isLoading = false;  
9    }).catch((err) => {
```

For buying a single viper, we have a fee of **0.02 ether**, so we need to pay it by sending our account details and 0.02 ether. We then call the *buyViper()* function from our smart contract, which returns the details of the new viper. We will save these details inside the **vipers** array.

Next, we modify the function for viper breeding, *breedVipers()*:

```

1  breedVipers() {
2    this.isLoading = true;
3    this.contractInstance.methods.breedVipers(this.matron,
4      from: this.account,
5      value: web3.toWei(0.05, 'ether'),
6    }).then((receipt) => {
7      this.addViperFromReceipt(receipt);
8      this.isLoading = false;
9    }).catch((err) => {

```

We need to pay **0.05 ether** to access this function. The *breedVipers()* function in our smart contract requires two integer parameters, *matron* and *sire*, so we pass the two integers to the function. This function also returns the details of the new viper and we need to save it to the `vipers` array.

Finally, we define the method for retrieving the details of vipers we own, called *getVipers()*:

```

1  getVipers() {
2    this.isLoading = true;
3    this.contractInstance.methods.ownedVipers().call({
4      from: this.account,
5    }).then((receipt) => {
6      for (let i = 0; i < receipt.length; i += 1) {
7        this.contractInstance.methods.getViperDetails(receipt[i],
8          from: this.account,
9        ).then((viper) => {
10         this.vipers.push({
11           id: viper[0],
12           genes: viper[1],
13           matron: viper[2],
14           sire: viper[3],
15           url: vipersMap[viper[1]],
16         });
17       }).catch((err) => {

```

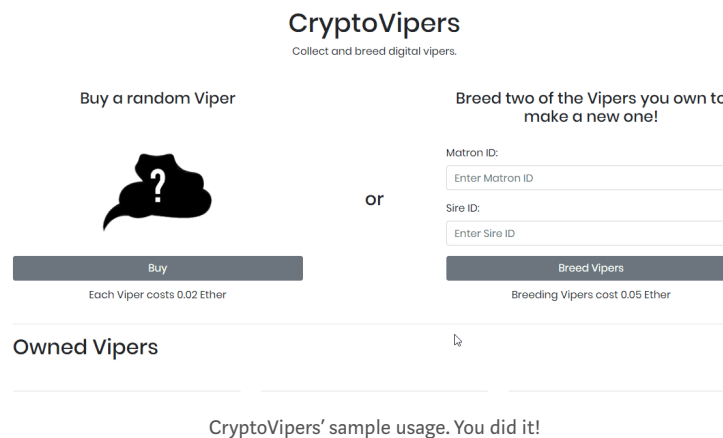
There are two functions to get the details, *ownedVipers()* and *getViperDetails()*. The first function is to get an array of our vipers then

the second function is to get the details of each of the vipers. After getting the details of each viper, we save it to the `vipers` array.

And, we're done!

## — ✓ Let's check it out. —

Refresh your browser to see the changes. This time, the whole web app is complete, and everything is functional! You should then be able to use it like this:



You can see the final result on this [GitHub repository](#) (master branch) which you can always refer to :)

## Conclusion

We just finished making our simple Cryptokitties clone! Awesome!

We learned how to create our own ERC721 implementation, and define our own custom functions. We also learned how to set up our own project using Vue.js, and created a simple application.

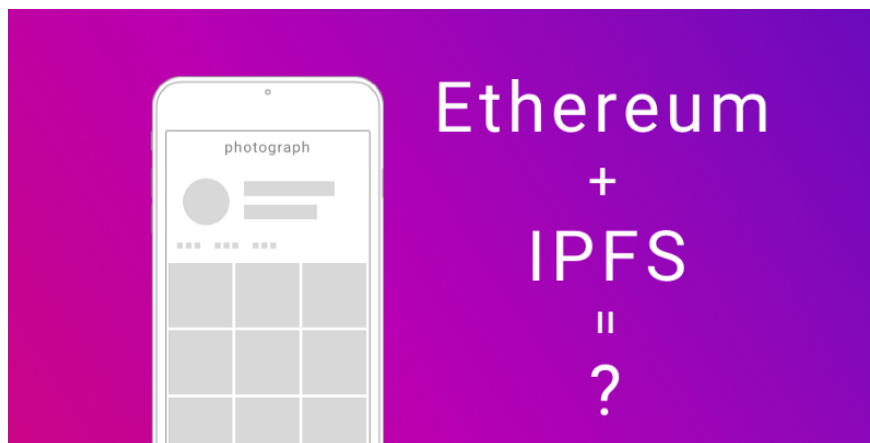




Chuck Norris approves

So what's next?

You might want to write the token code (ERC721 implementation) from scratch while referencing to [OpenZeppelin's implementation](#) or the [standard specification](#). Or perhaps, you'd like to expand from what we have done and add more features like trading with other users, which is a good idea too 👍



On a side note, you might want to check out openberry's previous tutorial, [creating an Instagram-like DApp with IPFS + Vue.js](#).

'til the next tutorial!

openberry is a tutorial marketplace, designed to allow anyone to learn blockchain programming.

openberry | blockchain tutorial marketplace

openberry is a tutorial marketplace, designed to enable anyone to learn blockchain programming.

[www.openberry.ac](http://www.openberry.ac)



openberry (@openberry\_ac) | Twitter

The latest Tweets from openberry (@openberry\_ac). openberry is a tutorial...  
[twitter.com](https://twitter.com/openberry_ac)



