Creating an Instagram-like DApp with IPFS + Vue.js

Gwen Danielle Merida Follow
Feb 12 · 9 min read

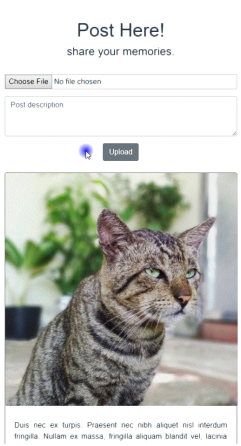
Instagram is a social platform where you can share your photos and videos to all the people you are connected to.

Now, imagine creating your own social platform like Instagram. An Instagram-like DApp that is running on top of Blockchain technology.

What We Are Trying To Build

In this tutorial, we will be building an Instagram-like platform integrated with both technologies, InterPlanetary File System **IPFS** and **Ethereum** blockchain. These two have been tightly coupled for some time now because of their similar nature, it decentralizes the data. And with that, when making a decentralized app, they synergize pretty well. You will understand this later on, as you read along the tutorial.

Also, we will be using Vue.js for the application's front-end. We won't be discussing much about this tool since I don't want you to get the idea that it should just be Vue.js, there are many available front-end languages that you can use for creating this app.



Function List

1. Upload data (POST image in IPFS)
2. Retrieve data (GET image from IPFS)

Tools

1. Smart Contract
   Solidity, Remix, Metamask
2. Frontend
   Web3.js, ipfs-http-client
   Vue.js, Vue-cli, Boostrap-vue

Content should be read in advance

1. How to deploy in Remix and use Metamask
2. Basic Solidity

Github

If you just want to get the code you can get it here:
https://github.com/openberry-ac/instagram

Why IPFS?

In basic terms, InterPlanetary File System is a peer-to-peer storing and sharing file system. Using IPFS, you can upload files such as text,

images and even videos.

In centralized storages, you can only get data from their servers, and the speed of that transfer depends on the distance between you and that server. But unlike a peer-to-peer storage such as IPFS, you can get the data from anyone who has the exact data you need, so the transfer is a lot faster.

IPFS uses content-addressable hashes to verify that the data you're getting is right, since all data has their own unique hash value.

```
# An example of a file's unique hash ID:
Qmf7bwqcd4BD7ohtkCBQvHu1BGMc8wMSP2nrLxsTBLDP4t
```

So when you upload a file in IPFS, you will get the data's unique hash ID as return, then you can retrieve it by using the same hash through a gateway , it's that simple.

```
# Gateway: https://ipfs.io/ipfs/
# Retrieve: Gateway + Unique Hash ID
https://ipfs.io/ipfs/ +
Qmf7bwqcd4BD7ohtkCBQvHu1BGMc8wMSP2nrLxsTBLDP4t
```

Try this one out!
https://ipfs.io/ipfs/Qmf7bwqcd4BD7ohtkCBQvHu1BGMc8wMSP2nrLxsTBLDP4t

Build the Project!

Workflow

- Writing our Smart Contract
- Setting Web3.js, Contract Instance and IPFS

- Getting Users Account
- Posting data in IPFS
- Fetching data from IPFS

Writing our Smart Contract

In creating this contract, we will be using Solidity.

> *Solidity is an object-oriented, high-level language for implementing smart contracts.*
> https://solidity.readthedocs.io/en/v0.5.3/

Here are the function that will be in our contract:

- **sendHash(_imgHash, _textHash)**—sends the image and text hashes and stores it.
- **getCounter()**—gets the total number of posts that are already stored.
- **getHash(_index)**—gets the image and text hashes using an index number.

We will name our contract, **InstagramPosting.sol**, and use the current latest and stable version of Solidity, version 0.5.3. Also, we imported **SafeMath** library for safe math operations.

```
pragma solidity 0.5.3
import "https://github.com/OpenZeppelin/openzeppelin-s

contract InstagramPosting {
    using SafeMath for uint256;

    // This struct is for the properties of a post.
    struct Post{
        address owner;
        string imgHash;
        string textHash;
    }

    // A mapping list for posts from Post struct.
    mapping(uint256 => Post) posts;
```

In this snippet of code, we created a struct, named **Post**. This holds the data: **owner**, **imgHash** and **textHash**, that will be used to store the post's data.

- **owner(address)**—the address of the post's owner.
- **imgHash(string)**—the hashes of images stored in IPFS.
- **textHash(string)**—the hashes of image's captions stored in IPFS.

Then, we declared a public mapping for Post. We created a mapping named **posts**, to list and store the data, with uint256 as its key reference. Lastly, we created a uint256, named **postCtr**, to traverse inside the posts mapping.

```
    // Event which will notify new posts.
    event NewPost();

    /**
     * @dev Function to store image & text hashes.
     * @param _img hash from IPFS.
     * @param _text hash from IPFS.
     */
    function sendHash(
        string memory _img,
        string memory _text
    )
        public
    {
        postCtr = postCtr.add(1);
```

Next, is the **sendHash** function, this is where the image and text hashes retrieved from an IPFS upload are sent (As seen on parameters **_img** and **_text**) and stored. Before storing, first increment the **postCtr** variable by 1 using **SafeMath's add()** function to avoid overflow, and set it as the new index reference of the post. After that, the `msg.sender` which refers to the address of the sender is stored as the owner of the post, followed by storing the two hashes in **imgHash** and **textHash**.

At the last part, we see the emit **NewPost(),** this refers to the event above the function. It is an event watcher that notifies the web application if the sendHash() function transaction is finished.

Send function is done, now let's create a function that retrieves it.

```
    /**
     * @dev Function to get image & text hashes.
     * @param _index number from total posts iteration
     * @return Stored image & text hashes.
     */
    function getHash(uint256 _index)
        public
        view
        returns (
            string memory img,
            string memory text,
            address owner
        )
    {
        owner = posts[_index].owner;
        img = posts[_index].imgHash;
```

As you would notice, the **getCounter()** plays a role in getting a hash, since it returns the number of total posts based on the **postCtr** variable.

For the **getHash()** function, by simply choosing an index number (As seen on **_index** parameter) within the range of the returned value from **getCounter()**, we can traverse inside and get our desired post. The return data are, **img**, **text**, and **owner**, and each are populated by the posts mapping with the chosen index.

InstagramPosting.sol

```
pragma solidity 0.5.3;
import "https://github.com/OpenZeppelin/openzeppelin-s


contract InstagramPosting{
    using SafeMath for uint256;

    // This struct is for the properties of a post.
    struct Post{
        address owner;
        string imgHash;
        string textHash;
    }

    // A mapping list for posts from Post struct.
    mapping(uint256 => Post) posts;

    // A counter for the posts mapping list.
    uint256 postCtr;

    // Event which will notify new posts.
    event NewPost();

    /**
     * @dev Function to store image & text hashes.
     * @param _img hash from IPFS.
     * @param _text hash from IPFS.
     */
    function sendHash(
        string memory _img,
        string memory _text
    )
        public
    {
        postCtr = postCtr.add(1);
        Post storage posting = posts[postCtr];
        posting.owner = msg.sender;
        posting.imgHash = _img;
        posting.textHash = _text;

        emit NewPost();
    }
```

```
42        }
43
44        /**
45         * @dev Function to get image & text hashes
```

Setting Web3.js, Contract Instance and IPFS

For this tutorial, a template project with frontend is provided which you can use while following this tutorial. you can clone the project:

```
# git clone the project template
git clone -b boilerplate --single-branch
https://github.com/openberry-ac/instagram.git
```

```
#go to folder
cd instagram
```

```
# install packages needed in the web application
npm install
```

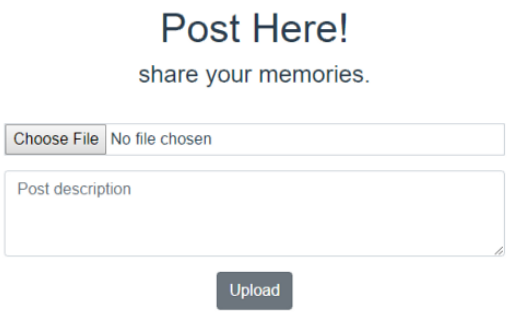
Using the template provided, on the root folder, let's install the packages (web3 and ipfs-http-client):

```
# install web3
npm install -s web3@1.0.0-beta.37
```

```
# install ipfs-http-client
npm install -s ipfs-http-client
```

```
# run web application
npm run dev
```

When you run the app, the page should looks like this.



Now let's go in, **web3.js,** inside the **contract** folder**.** Here is where we will setup and initialize our web3, by importing the web3 package, named **Web3** and then declare a constant variable named, **web3** and instantiate **Web3** on it.

The code should look like this:

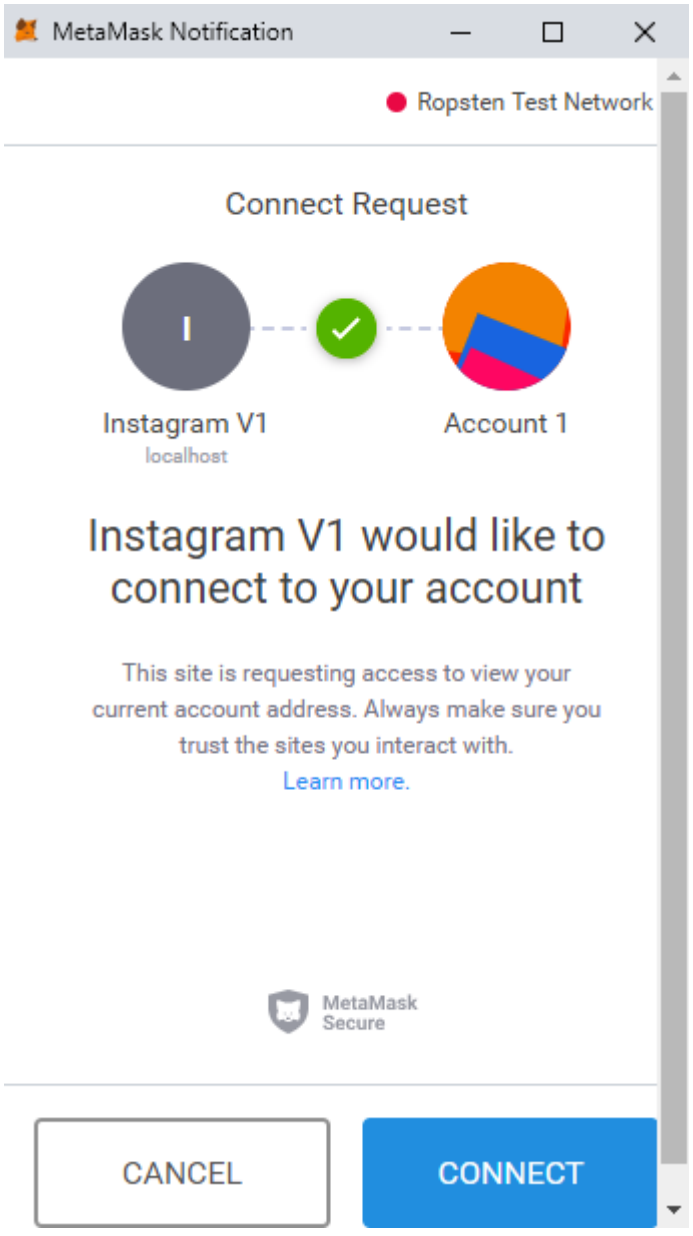
web3.js

```
//imports the Web3 API
import Web3 from 'web3';

/**
 * creates & exports new instance for
 * Web3 using provided service by Metamask.
 */
let currentWeb3;

if (window.ethereum) {
    let instance = new Web3(window.ethereum);
    try {
        // Request account access if needed
        window.ethereum.enable();
        // Acccounts now exposed
        currentWeb3 = instance;
    } catch (error) {
        // User denied account access...
        alert('Please allow access for the app to work
```

We have successfully created web3.js. Now go back to your browser and refresh the page, you must be redirected on connection request in metamask.

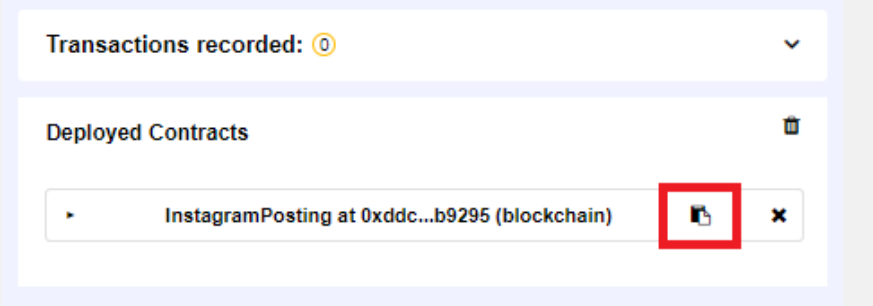


Click **CONNECT,** and your contract is now connected to the Ethereum Network.

Then in **contractInstance.js,** which is also in **contract** folder, we will create an instance of our contract using the **ABI** and declare the **contract address**.

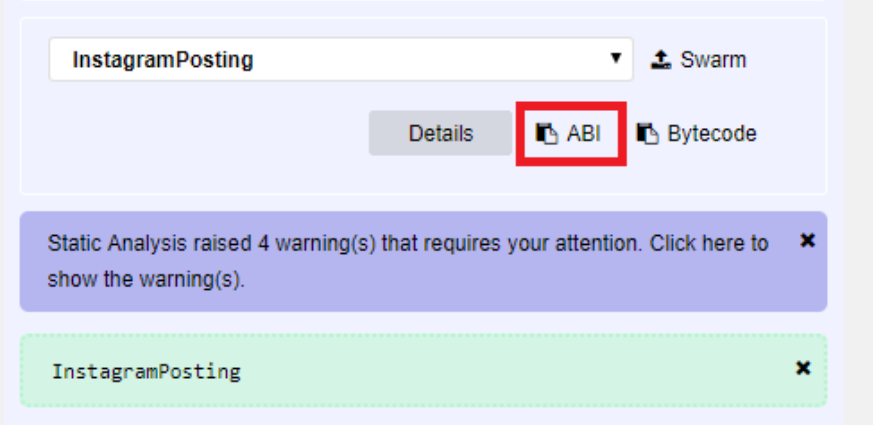
To connect the smart contract on Ethereum, ABI and its contract address are needed.

Then back on Remix, under the `Run` tab, in the list of deployed contract click the copy button beside the desired contract, to get the **contract address**.



Then, on **contractInstance.js**, declare it also on a constant variable named, **address**

Back to Remix again, move to `compile` tab and copy **ABI** by clicking the button.



Now paste it in our **contractInstance.js**, by declaring it on a constant variable named, **abi.**

contractInstance.js should look like this, with your copied address and abi.

```
import web3 from './web3';
const address = "Paste the copied contract address in t
const abi = /* Paste the copied ABI here! */
export default new web3.eth.Contract(abi, address);
```

Now you can connect to the contract.

Now to connect to IPFS, go to ipfs.js which is also in **contract** folder, then import ipfs-http-client and name it **IPFS**. Then instantiate **IPFS** in a constant variable, **ipfs**, to connect to infura's gateway. We will be using infura's gateway, to post and get data in IPFS.

ipfs.js

```
//imports the IPFS API
import IPFS from 'ipfs-http-client';

/**
 * creates & exports new instance for
 * IPFS using infura as host, for use.
 */
```

Now that we are done in instantiating IPFS, We have to declare contract in main.js, so we could call the functions in our smart contract.

main.js

We need to declare the contract instance that was already imported above, so add **contract** in **data** to declare the contract instance. With it, we can call methods from our deployed smart contract.

```
data: {
    // ..
    contract
  },
```

Finally, we are done setting up web3, contract and IPFS. It's now time to learn how to post and get data in IPFS.

Posting data in IPFS

First, we will teach you how to post the data in IPFS. But, before we can do that, first we have to get and set the user's wallet address. In **main.js**, inside **methods** we will create an async function called, **updateAccount()**, to get the current account being used in metamask.

```
1     /**
2       * gets current account on web3 and
3       * store it on currentAccount variable.
4       */
5      async updateAccount() {
6        const accounts = await web3.eth.getAccounts();
7        const account = accounts[0];
```

Now let's learn how we can upload post in the Ethereum Blockchain and IPFS. First, let's open the **App.vue** which can be seen inside **src** folder.

We will only be working inside the script tag, now under **methods**, is the empty async function **onSubmit(),** which is being called in the **handleOk()** function, which checks if the input is not empty. **onSubmit()** function, sends file to IPFS and the hashes retrieved are sent to our contract.

```
/**
 * submits buffered image & text to IPFS
 * and retrieves the hashes, then store
 * it in the Contract via sendHash().
 */
onSubmit() {
  alert('Uploading on IPFS...');
  this.$root.loading = true;
  let imgHash;
  ipfs.add(this.buffer)
    .then((hashedImg) => {
      imgHash = hashedImg[0].hash;
      return this.convertToBuffer(this.caption);
    }).then(bufferDesc => ipfs.add(bufferDesc)
      .then(hashedText => hashedText[0].hash)).the
      this.$root.contract.methods
        .sendHash(imgHash, textHash)
        .send({ from: this.$root.currentAccount },
          (error, transactionHash) => {
            if (typeof transactionHash !== 'undefi
              alert('Storing on Ethereum ...');
```

We also need a function that will handle the choosing of file and converts it to buffer and a function that will return the converted file to buffer. This can be done in **captureFile().**

```
/* used to catch chosen image &
 * convert it to ArrayBuffer.
 */
captureFile(file) {
  const reader = new FileReader();
  if (typeof file !== 'undefined') {
    reader.readAsArrayBuffer(file.target.files[0])
    reader.onloadend = async () => {
      this.buffer = await this.convertToBuffer(rea
```

and to make the code cleaner we have created **convertToBuffer()** function for the conversion of file to buffer.

```
    /**
     * converts ArrayBuffer to
     * Buffer for IPFS upload.
     */
    async convertToBuffer(reader) {
      return Buffer.from(reader);
```

Let's check it out.

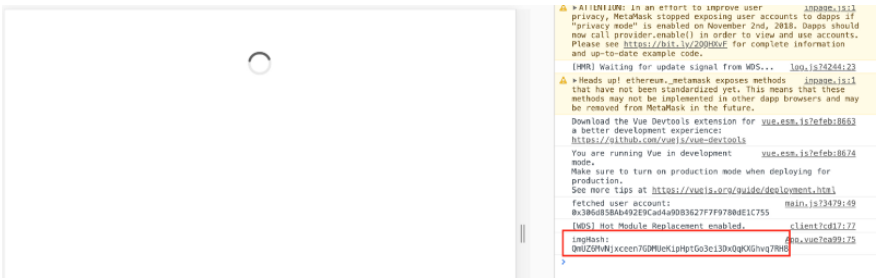
Ok let's have a checkpoint on what we have done so far. This part of the tutorial is about uploading image in the IPFS. So let's try to upload and check if it is really stored in the IPFS.

But before uploading, let's add a console log to check the hashed of the image to be uploaded in the IPFS. On **onSubmit()** function, let's add the logger after we get the image hash from ipfs and before we return the value. This part of the code, should look like this:

```
// ..
.then((hashedImg) => {
   imgHash = hashedImg[0].hash;
   console.log("imgHash: " + imgHash);
   return this.convertToBuffer(this.caption);
//
```

Note: you can delete the console.log() after this checkpoint.

Now upload image in our page and check "imgHash" in the browser's console.



As I mentioned above, use this url to check if the image was uploaded in IPFS:

https://ipfs.io/ipfs/ + imgHash

Fetching data from IPFS

We don't want to always check our uploaded image in https://ipfs.io/ipfs/, We want to show it in our page. To do that, in **main.js** file, in async function **getPosts(),** we need to get the images in IPFS and here is how we do it:

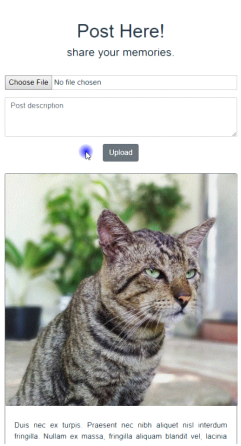
```
    /**
     * using the Smart Contract instance:
     * getCounter() - gets the length of total posts
     * getHash() - gets the image & text hashes using
     *
     * index is from the iteration of the retrieved to
     * post count. every loop gets the hashes and fetc
     * text & image using the IPFS gateway URL.
     */
    async getPosts() {
      this.loading = false;
      const posts = [];
      const counter = await contract.methods.getCounte
        from: this.currentAccount,
      });

      if (counter !== null) {
        const hashes = [];
        const captions = [];
        for (let i = counter; i >= 1; i -= 1) {
          hashes.push(contract.methods.getHash(i).call
            from: this.currentAccount,
          }));
        }

        const postHashes = await Promise.all(hashes);

        for (let i = 0; i < postHashes.length; i += 1)
          captions.push(fetch(`https://gateway.ipfs.io
            .then(res => res.text()));
        }
```

Alright! we are done with everything. This should be the output:



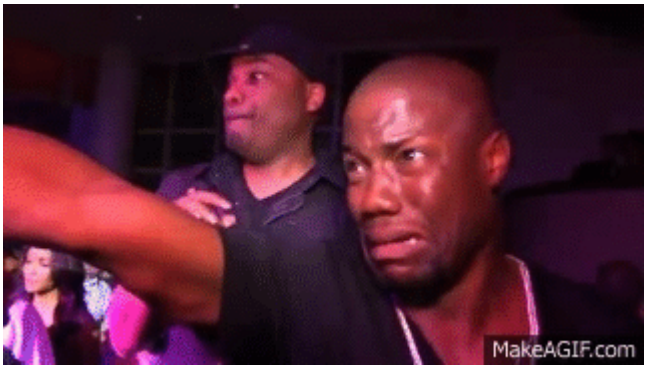
If you want the full code of this tutorial, you can check it here: https://github.com/openberry-ac/instagram

Summary

You just made your very own Instagram web application integrated with IPFS and Ethereum! You did great for getting this far, you can totally show this off to your developer friends!

From this tutorial you learned how to send and retrieve files from IPFS, and how to establish connection with it. Also you have learned how to interact with your smart contract by connecting to Web3. Man, you have come a long way, and you owned it!

Me being proud right now:



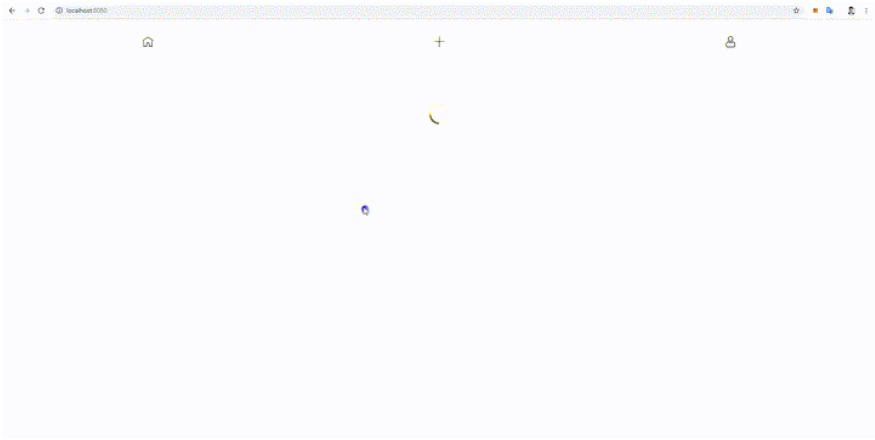
For your next step you might want to explore more about IPFS and Solidity. There's so much more that you can learn and can do from this technologies, so go ahead and click the links I gave just now!

You can also add more features to this project you created and deploy your very own live Instagram-like decentralized application, your call!

Well then, see you on the next tutorial!

If you get interested in the following topics:

- state management
- page routing



let us know by giving a clap.

---

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