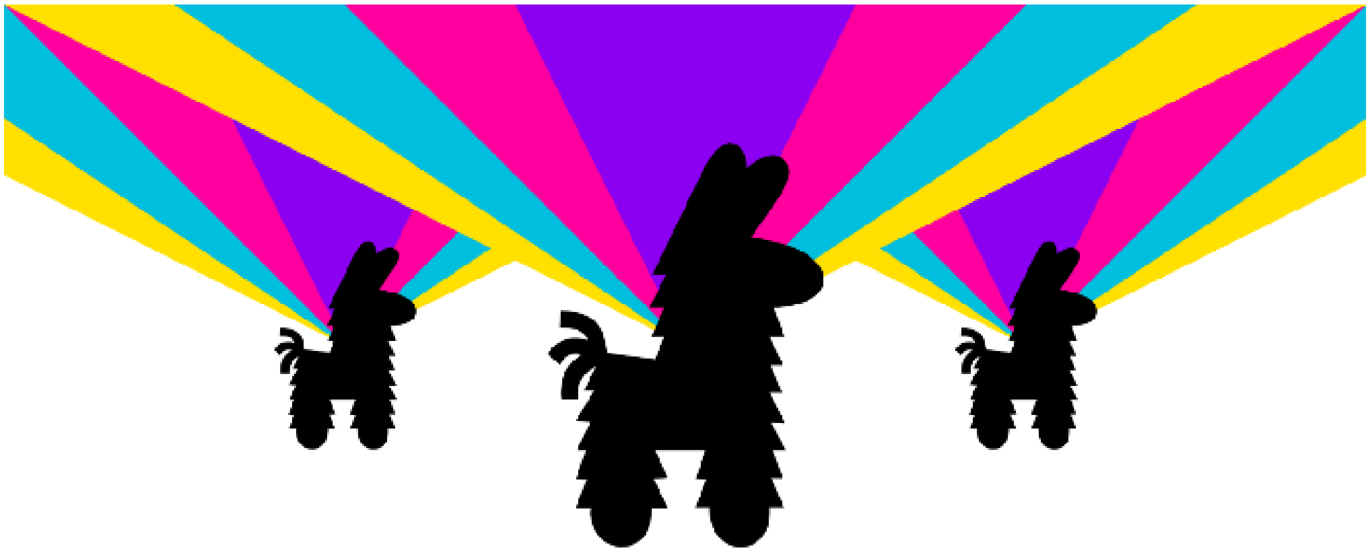
How to Display Your NFT Collectibles Like NBA Top Shot With Flow and IPFS

Using Pinata: Part 2

Justin Hunter Follow
Mar 4 · 8 min read



This is part two of a three-part series focused on the Flow blockchain, NFTs, and IPFS. You can read part one here:

How to Create NFTs Like NBA Top Shot With Flow and IPFS

Using Pinata

medium.com

In this tutorial, we're going to build a simple React application that interfaces with Flow smart contracts to authenticate and to fetch NFTs owned by a user. We will then resolve the NFT's metadata to get the IPFS location of the NFT's underlying digital asset (in this case a video). Quick reminder about the NFT we're working on — we're building NFTs that, like NBA Top Shot, back video content, but unlike Top Shot, the videos are of people smashing piñatas.

Setup

For this tutorial, you'll need to have your Flow emulator running. If you don't remember how to start that, take a look at the previous post or check out the Flow CLI docs. It's important to note that the Flow emulator is an in-memory representation of the Flow blockchain. So if between the first post and this one, you shut down the emulator, you'll need to go ahead and do the following:

- Start the emulator
- Deploy the project
- Mint your NFT

Each of those steps is covered in detail in part one of this series.

In addition, you're going to need to have NodeJS installed on your machine. You can install it here.

And as usual, you'll need a text editor.

Setting Up React And Dependencies

I'm going to be creating my React app within the parent `pinata-party` directory I started in part one of these tutorials. However, if you want, you can create your React app in a completely new directory.

To create our app, run:

```
npx create-react-app pinata-party-frontend
```

When everything is done installing, you'll have a new directory called `pinata-party-frontend`. Switch into that directory, and we'll need to install some dependencies.

First, per the documentation on Flow here, you'll need to install the Flow JS SDK. In fact, for the first part of the frontend setup, we're just going to follow the Flow documentation.

Let's continue. Run:

```
npm i @onflow/fcl @onflow/types
```

We're going to store some values as global variables for our application. We will make use of environment variables. In react, this means creating a `.env` file and setting key value pairs where the keys are prefixed with `REACT_APP`. If you follow the Flow documentation, you'll be set up to be connected with their testnet. For the sake of this tutorial, we're connecting to the Flow emulator. So, we'll need to make some changes. In your `.env` file add the following:

```
REACT_APP_ACCESS_NODE=http://localhost:8080

REACT_APP_WALLET_DISCOVERY=https://fcl-
discovery.onflow.org/testnet/authn

REACT_APP_CONTRACT_PROFILE=0xf8d6e0586b0a20c7
```

Replace the `REACT_APP_ACCESS_NODE` value with the local emulator url as indicated above. Replace the `REACT_APP_CONTRACT_PROFILE` value with the address you received when you deployed your project.

Now that we have that, we need to create a config file that we will use to interact with the Flow JS SDK. Create a file in the `src` directory called `config.js`. Add the following:

```
import {config} from "@onflow/fcl"
```

```
config()
.put("accessNode.api", process.env.REACT_APP_ACCESS_NODE)
.put("challenge.handshake", process.env.REACT_APP_WALLET_DISCOVERY)
.put("0xProfile", process.env.REACT_APP_CONTRACT_PROFILE)
```

This configuration file just helps the JS SDK work with the Flow blockchain (or emulator in this case). To make this file available throughout the app, open up the `index.js` file and add this line:

```
import "./config"
```

Now, let’s wire up some authentication. You don’t need to force people to authenticate into the site if you don’t want to, but it will be important to have authentication for part three of the tutorial which enables the transferring of NFT assets.

We need to create an authentication component. In your `src` directory, create a file called `AuthCluster.js` . Inside that file, add the following:

```
import React, {useState, useEffect} from 'react'
import * as fcl from "@onflow/fcl"

const AuthCluster = () => {
  const [user, setUser] = useState({loggedIn: null})
  useEffect(() => fcl.currentUser().subscribe(setUser), [])
  if (user.loggedIn) {
    return (
      <div>
        <span>{user?.addr ?? "No Address"}</span>
        <button className="btn-primary" onClick={fcl.unauthenticate}>Log Out</button>
      </div>
    )
  } else {
    return (
      <div>
        <button className="btn-primary" onClick={fcl.logIn}>Log In</button>
        <button className="btn-secondary" onClick={fcl.signUp}>Sign Up</button>
      </div>
    )
  }
}

export default AuthCluster
```

AuthCluster.js hosted with ❤ by GitHub   view raw

This is simple using a login and sign up button to leverage the Flow JS SDK's ability to connect to a wallet discovery provider. You'll be able to sign up for an account or sign in with an existing account. It's pretty nice!

Now, we need to get this component into our app. Let's keep it simple for now. Replace your `App.js` file with the following:

```
import './App.css';
import AuthCluster from './AuthCluster';

function App() {
  return (
    <div className="App">
      <AuthCluster />
    </div>
  );
}

export default App;
```

If you start the app now (npm start), you should see a page with a login and a sign up button. In fact, both of those buttons are functional. Try it out.

Ok, now that our React app is largely set up and our dependencies are installed, let's get to building out the ability to fetch NFTs for an account and display them.

Fetching NFTs From Flow

In order to display our NFT that minted in the first post, we need to be able to communicate with the Flow blockchain. In the case of this tutorial, we need to be able to communicate with the Flow emulator. We already told our app that the emulator was running on port 8080 when we set up our `.env` file. But now, how do we use JavaScript to talk to Flow?

Fortunately, Flow has this functionality built into their JS SDK. If you remember, we previously wrote a script to look up an NFT based on its token id and return the token's metadata. It looked like this:

```
import PinataPartyContract from 0xf8d6e0586b0a20c7

pub fun main() : {String : String} {
    let nftOwner = getAccount(0xf8d6e0586b0a20c7)
    // log("NFT Owner")
    let capability = nftOwner.getCapability<&{PinataPartyContract.NFTReceiver}>(/public/

    let receiverRef = capability.borrow()
        ?? panic("Could not borrow the receiver reference")

    return receiverRef.getMetadata(id: 1)
}
```

CheckTokenMetadata.cdc hosted with ❤ by GitHub view raw

Now, we just need to be able to convert that into a JavaScript call. Let's create a new component that will allow us to both fetch data and eventually display the NFT data. In your `src` directory, create a file called `TokenData.js` . In that file, add the following:

```
import React, { useState } from "react";
import * as fcl from "@onflow/fcl";

const TokenData = () => {
  const [nftInfo, setNftInfo] = useState(null)
  const fetchTokenData = async () => {
    const encoded = await fcl
      .send([
        fcl.script`
        import PinataPartyContract from 0xf8d6e0586b0a20c7
        pub fun main() : {String : String} {
          let nftOwner = getAccount(0xf8d6e0586b0a20c7)
          let capability = nftOwner.getCapability<&{PinataPartyContract.NFTReceiver}>(/p

          let receiverRef = capability.borrow()
              ?? panic("Could not borrow the receiver reference")

          return receiverRef.getMetadata(id: 1)
        }
      `
      ])

    const decoded = await fcl.decode(encoded)
    setNftInfo(decoded)
  };
  return (
    <div className="token-data">
      <div className="center">
        <button className="btn-primary" onClick={fetchTokenData}>Fetch Token Data</butto
      </div>
      {
        nftInfo &&
        <div>
          {
            Object.keys(nftInfo).map(k => {
              return (
                <p>{k}: {nftInfo[k]}</p>
              )
            })
          }
          <button onClick={() => setNftInfo(null)} className="btn-secondary">Clear Toke
        </div>
      }
    </div>
  );
};

export default TokenData;
```

In this file, we are creating a component that has a button to fetch token data. We are also creating a button to clear token data. When the fetch button is clicked, it calls a function we created called `fetchTokenData`. That function uses the Flow JS SDK to execute the exact same script we executed in part one of this tutorial from the command line, but in React. We take the results of the execution and we set the results to a state variable called `nftInfo`. If that variable exists, we render the key value pairs from the NFT metadata on screen and a button that lets us clear the data.

I've also added a bit of CSS to make things look less plain. If you want to add the same CSS, it's here, and all you have to do is replace `App.css` with this:

```css
.App {
  display: flex;
  flex-direction: column;
  min-height: 500px;
  justify-content: center;
  align-items: center;
}

button {
  padding: 10;
  height: 30px;
  min-width: 100px;
  cursor: pointer;
}

.btn-primary {
  border: none;
  background: rgb(255, 224, 0);
  color: #282828;
}

.btn-secondary {
  border: none;
  background: rgb(0, 190, 221);
  color: #282828;
}

.center {
  text-align: center;
}

.token-data {
  margin-top: 100px;
}
```

App.css hosted with ❤ by GitHub view raw

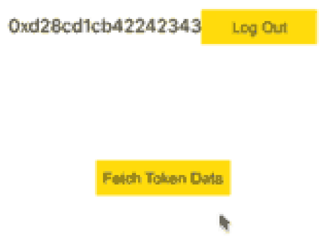
Now, just add your new component to `App.js` below the `AuthCluster` component like this:

```
import './App.css';
import AuthCluster from './AuthCluster';
import TokenData from './TokenData';

function App() {
  return (
    <div className="App">
      <AuthCluster />
      <TokenData />
    </div>
  );
}

export default App;
```

Check out your app now and try to fetch your token data. It should look like this:



This is pretty cool! We are looking up an NFT owned by the account we specified, then we are fetching the metadata from that token. We are displaying that metadata on screen, but we know that one value in that metadata resolves to a video file. Let's see if we can get that rendered on screen shall we?

Getting Media From IPFS

You already signed up for a Pinata account and you added your video file to IPFS through the Pinata upload interface. That means you have been able to fetch content from IPFS already. In the Pin Explorer, when you click on a hash, you are taken to the Pinata IPFS gateway where your IPFS content is resolved and displayed to you. For the sake of this tutorial, we're going to keep things a little more general while at the same time highlighting the beauty of IPFS.

That is to say, you don't have to fetch your content from the Pinata gateway. Instead, we're going to fetch it from the public Protocol Labs gateway.

Let's dive in.

Back in your `TokenData.js` file, let's add a way to display the video file we ultimately retrieve from IPFS. Update your file to look like this:

```
import React, { useState } from "react";
import * as fcl from "@onflow/fcl";

const TokenData = () => {
  const [nftInfo, setNftInfo] = useState(null)
  const fetchTokenData = async () => {
    const encoded = await fcl
      .send([
        fcl.script`
        import PinataPartyContract from 0xf8d6e0586b0a20c7
        pub fun main() : {String : String} {
          let nftOwner = getAccount(0xf8d6e0586b0a20c7)
          let capability = nftOwner.getCapability<&{PinataPartyContract.NFTReceiver}>(/p

          let receiverRef = capability.borrow()
              ?? panic("Could not borrow the receiver reference")

          return receiverRef.getMetadata(id: 1)
        }
      `
      ])

    const decoded = await fcl.decode(encoded)
    setNftInfo(decoded)
  };
  return (
    <div className="token-data">
      <div className="center">
        <button className="btn-primary" onClick={fetchTokenData}>Fetch Token Data</butto
      </div>
      {
        nftInfo &&
        <div>
          {
            Object.keys(nftInfo).map(k => {
              return (
                <p>{k}: {nftInfo[k]}</p>
              )
            })
          }
          <div className="center video">
            <video id="nft-video" canplaythrough controls width="85%">
              <source src={`https://ipfs.io/ipfs/${nftInfo["uri"].split("://")[1]}`}
                    type="video/mp4" />
            </video>
            <div>
              <button onClick={() => setNftInfo(null)} className="btn-secondary">Clear T
            </div>
          </div>
        </div>
      }
    </div>
  );
};

export default TokenData;
```

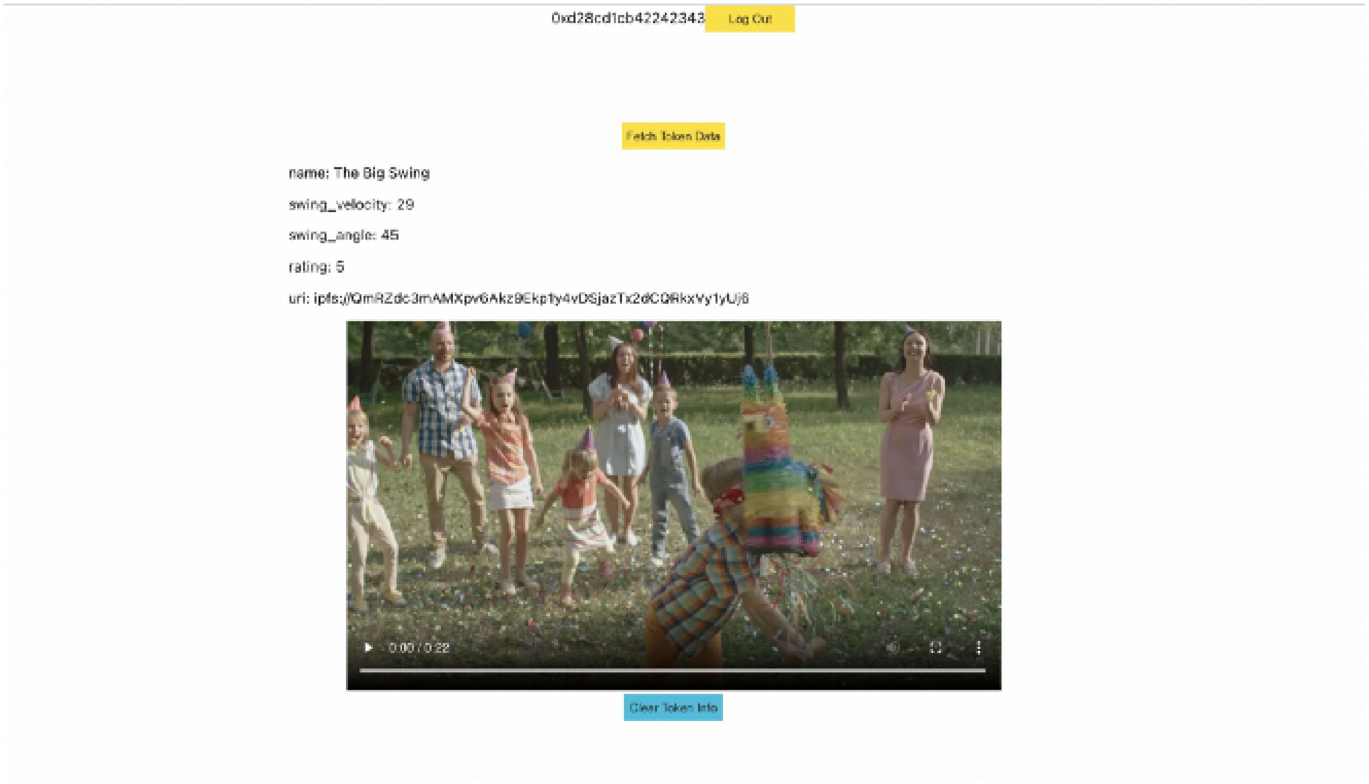
We've added a video element with the source pointing to the file on IPFS. You'll notice that we had to split the `uri` value to get just the IPFS hash so that we could fetch the content from the IPFS gateway. Let's talk about that URI for a moment.

The uri we created with our NFT look like `ipfs://Qm...` . We created it that way because the IPFS desktop client will allow you to click on and open links that look like that by default. Also, the Brave browser supports pasting in links that look like that. We think this link format will be supported more and more as we go.

However, in this case, we need the hash so that we can fetch the content from the IPFS public gateway. When we're done, the link would look like this:

https://ipfs.io/ipfs/QmRZdc3mAMXpv6Akz9Ekp1y4vDSjazTx2dCQRkxVy1yUj6

Now, if you try fetching the token data in our app, you should get something like this:



```
name: The Big Swing
swing_velocity: 29
swing_angle: 45
rating: 5
uri: ipfs://QmRZdc3mAMXpv6Akz9Ekp1y4vDSjazTx2dCQRkxVy1yUj6
```

Our NFT is no longer a faceless token on a blockchain. It's a real live digital asset! Your video will probably be different, but hopefully the result you see in your app is essentially the same experience.

Wrapping Up

This is a very simple app, and there is a lot you can do to make it prettier, make it more interactive, and even add more Flow elements to it. The Flow JS SDK is powerful, so I recommend reading through the docs.

We successfully added authentication to our app using Flow, we created an interface to fetch information about an NFT, and we created a way to display not just the raw metadata but the underlying digital asset. All of this is secured by both the Flow blockchain and IPFS. We know that the NFT is owned by the person who says they own it, and we know the content displayed is valid because the hash is encoded into the NFT.

In the final installment of this series, we'll focus on creating a mini-marketplace that will allow us transfer NFTs.

Happy Pinning!

Go to part 3:

How To Create an NFT Marketplace on Flow With IPFS
Using Pinata: Part 3
medium.com

Additional reading:

The Secret NFT
How to Submarine an NFT with IPFS
medium.com

Introducing Scoped API Keys
Create keys to pin on IPFS with Pinata
medium.com

Nft Flow Blockchain Ipfs Nba Top Shot Digital Collectibles

Download on the App Store

GET IT ON Google Play