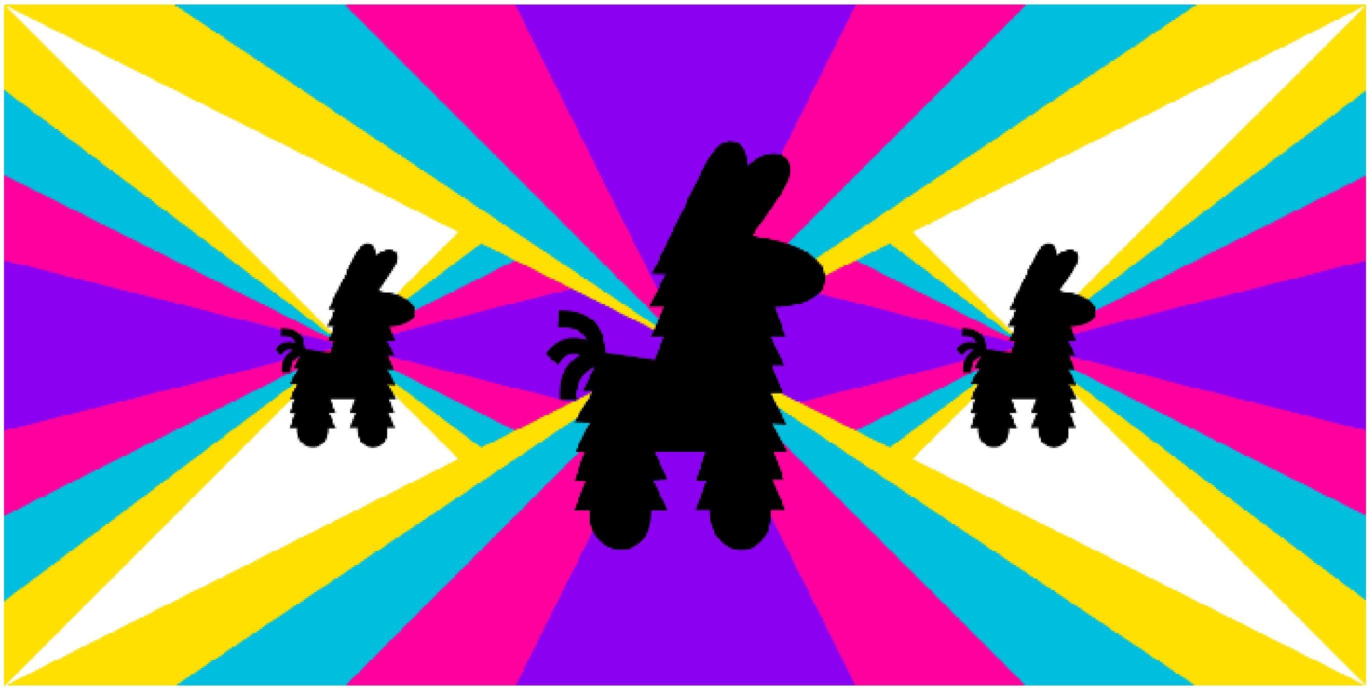
How to Create NFTs Like NBA Top Shot With Flow and IPFS

Using Pinata: Part 1

Justin Hunter Follow
Feb 27 · 13 min read



As the non-fungible tokens (NFTs) market reaches a fever pitch, it's interesting to look back on the relatively early days of NFTs and remember the challenges that were exposed by CryptoKitties. The platform, built by the team at Dapper Labs, was the first real example of potential mass adoption and the first real strain on the Ethereum blockchain.

Since then, NFTs have taken off with platforms like Rarible , OpenSea, Foundation, and Sorare springing up. These platforms are seeing millions of dollars flow through monthly. Most of this has happened, despite the early struggles, on the Ethereum blockchain. However, the team at Dapper Labs, after their experience with CryptoKitties, set out to build a new blockchain that would be general-purpose but would also be a great fit for the NFT use case. In doing so, their goal was to solve many of the problems they saw with NFTs on Ethereum while providing a better experience to both developers and collectors in the space. Their new blockchain, Flow, has proven itself capable of landing some big names. The NBA, UFC, and even Dr. Seuss are on Flow.

We recently wrote about creating NFTs with built-in asset support on IPFS, and we've talked about the responsibility problem in the NFT space and how we think IPFS can help. It's now time to talk about how to create NFTs on Flow backed by IPFS. One of the major early applications of the Flow blockchain is NBA Top Shot. We're going to build a very basic replica of the NFT minting process and then back the NFT metadata and assets on IPFS.

Since we love piñatas, instead of videos of NBA highlights, our NFTs are going to be focused on trade-able videos of piñatas being smashed at parties.

This is a three-part tutorial:

1. Creating the contract and minting a token
2. Creating an app to view NFTs created through this contract
3. Creating a marketplace to transfer NFTs to others while also transferring the NFT's underlying assets on IPFS

Let's get started with the first tutorial.

Setting Up

We need to get the Flow CLI installed. There are some good install instructions within Flow's documentation, but I'll copy them here:

macOS

```
brew install flow-cli
```

Linux

```
sh -ci “$(curl -fsSL https://storage.googleapis.com/flow-cli/install.sh)"
```

Windows

```
iex “& { $(irm ‘https://storage.googleapis.com/flow-cli/install.ps1') }”
```

We are going to be storing asset files on IPFS. To make that easier, we can use Pinata. You can sign up for a free account here and grab an API Key here. We’ll use the API in the second post of this tutorial, but we’ll use the Pinata website in this post.

We also need to have NodeJS installed and a text editor that can help with syntax highlighting of Flow smart contract code (which is written in a language called Cadence). You can install Node here. Visual Studio Code has an extension that supports Cadence.

Let’s create a directory to house our project.

```
mkdir pinata-party
```

Change into that directory and initialize a new flow project:

```
cd pinata-party
flow project init
```

Now, open the project in your favorite code editor (again, if you use Visual Studio Code, grab the Cadence extension), and let’s get to work.

You’ll see a `flow.json` file that we will be making use of soon. First, create a folder called `cadence`. Within that folder, add another folder called `contracts`. And finally, create a file within the `contracts` folder called `PinataPartyContract.cdc`.

Before we move forward, it’s important to point out that everything we do in regards to the Flow blockchain from this point forward will be done on the emulator. However, deploying a project to testnet or mainnet is as simple as updating configuration settings in your `flow.json` file. Let’s set that file up for the emulator environment now and then we can start writing our contract.

Update the contracts object in `flow.json` to look like this:

```
"contracts": {
     "PinataPartyContract":
"./cadence/contracts/PinataPartyContract.cdc"
}
```

Then, update the `deployments` object in that file to look like this:

```
"deployments": {
     "emulator": {
          "emulator-account": ["PinataPartyContract"]
     }
}
```

This is telling the Flow CLI to use the emulator to deploy our contract, it's also referencing the account (on the emulator) and the contract we will write soon. Actually…

Let's start writing that bad boy now.

The Contract

Flow has a great tutorial on creating NFT contracts. It's a good reference point, but as Flow points out themselves, they have not yet solved the NFT metadata problem. They would like to store metadata on chain. That's a great idea, and they will surely come up with a logical approach to it. However, we want to mint some tokens with metadata now, AND we want media files associated with the NFT. Metadata is only one component. We need to also point to the media that the token ultimately represents.

If you're familiar with NFTs on the Ethereum blockchain, you may know that many of the assets those tokens back are stored on traditional data stores and cloud hosting providers. This is OK except when it's not. We've written in the past about the genius of content-addressable content and the downsides to storing blockchain-adjacent data on traditional cloud platforms. It all boils down to two main points:

- The assets should be verifiable

- It should be easy to transfer the maintenance responsibilities

IPFS takes care of both of these points. Pinata then layers in an easy way to pin that content long-term on IPFS. This is exactly what we want for the media that backs our NFTs, right? We want to make sure that we can prove ownership (the NFT), provide data about the NFT (the NFT), and ensure that we have control over the underlying asset (IPFS) —media or otherwise—and not some replica.

With all this in mind, let's write a contract that mints NFTs, associates metadata to the NFT, and ensures that metadata points to the underlying asset stored on IPFS.

Open up the `PinataPartyContract.cdc` and let's get to work.

The first step is defining our contract. We're going to add a whole lot more to this, but we start by defining `PinataPartyContract` and within that, we create a `resource`. Resources are items stored in user accounts and accessible through access control measures. In this case, the `NFT` resource ultimately because the thing that is used to represent NFTs owned. NFTs have to be uniquely identifiable. The `id` property allows us to identify our tokens.

Next, we need to create a resource interface that we will use to define what capabilities are made available to others (i.e. people who are not the contract owner):

Put this right below the NFT resource code. This `NFTReceiver` resource interface is saying that whoever we define as having access to the resource will be able to call the following methods:

- `deposit`
- `getIDs`
- `idExists`
- `getMetadata`

Next, we need to define our token collection interface. Think of this as the wallet that houses all a user's NFT.

There's a lot going on in this resource, but it should make sense soon. First, we have a variable called `ownedNFTs`. This one is pretty straightforward. It keeps track of all the NFTs a user owns from this contract.

Next, we have a variable called `metadataObjs`. This one is a little unique because we are extending the Flow NFT contract functionality to store a mapping of metadata for each NFT. This variable maps a token id to its associated metadata, which means we need the token id before we can set it.

We then initialize our variables. This is required for variables defined in a resource within Flow.

Finally, we have all of the available functions for our NFT collection resource. Note that not all of these functions are available to the world. If you remember, we defined the functions that would be accessible to anyone earlier in our `NFTReceiver` resource interface.

I do want to point out the `deposit` function. Just as we extended the default Flow NFT contract to include the `metadataObjs` mapping, we are extending the default `deposit` function to take an additional parameter of `metadata`. Why are we doing this here? We need to make sure that only the minter of the token can add that metadata to the token. To keep this private, we keep the initial addition of the metadata confined to the minting execution.

We're almost done with our contract code. So, right below the `Collection` resource, add this:

First, we have a function that creates an empty NFT collection when called. This is how a user who is first interacting with our contract will have a storage location created that maps to the `Collection` resource we defined.

After that, we create one more resource. This is an important one, because without it, we can't mint tokens. The `NFTMinter` resource includes an `idCount` which is incremented to ensure we never have duplicate ids for our NFTs. It also has a function that actually creates our NFT.

Right below the `NFTMinter` resource, add the main contract initializer:

This initializer function is only called when the contract is deployed. It does three things:

1. Creates an empty Collection for the deployer of the collection so that the owner of the contract can mint and own NFTs from that contract.
2. The `Collection` resource is published in a public location with reference to the `NFTReceiver` interface we created at the beginning. This is how we tell the contract that the functions defined on the `NFTReceiver` can be called by anyone.
3. The `NFTMinter` resource is saved in account storage for the creator of the contract. This means only the creator of the contract can mint tokens.

The full contract can be found here.

Now that we have a contract ready to go, let's deploy it, right? Well, we should probably test it on the Flow Playground. Go there and click on the first account in the left sidebar. Replace all the code in the example contract with our contract code, then click Deploy. If all goes well, you should see a log like this in the log window at the bottom of the screen:

```
16:48:55 Deployment Deployed Contract To: 0x01
```

Now we're ready to deploy our contract to the locally running emulator. From the command line, run this:

```
flow project start-emulator
```

Now, with our emulator running and our `flow.json` file configured properly, we can deploy our contract. Simply run this command:

```
flow project deploy
```

If all went well, you should see an output like this:

```
Deploying 1 contracts for accounts: emulator-account

PinataPartyContract -> 0xf8d6e0586b0a20c7
```

We now have a contract live on the Flow emulator, but we want to mint a token. Let's close out this blog post by taking care of that.

Minting The NFT

In the second post in this tutorial, we'll work on making the minting process more user friendly with an app and a user interface. For the sake of getting something minted and showing how metadata will work with our NFTs on Flow, we're going to use Cadence scripts and the command line.

Let's create a new directory within the root of our `pinata-party` project, and let's call it `transactions`. Once you have that folder created, create a new file within in that is called `MintPinataParty.cdc`.

For us to write our transaction, we need to have a file to reference in the metadata we provide to the NFT. For that, we will upload a file to IPFS via Pinata. For the sake of this tutorial, and because our NFTs are focused on trade-able videos of piñatas being smashed at parties, we will be uploading a video of a kid hitting a piñata at a birthday party. You can upload any video file you'd like. You can really upload any asset file you'd like and associate that with your NFT, but the second post in this tutorial series will be expecting video content. Once you have your video file ready to go, upload it here.

When you've uploaded the file, you will be provided an IPFS hash (often referred to as a content identifier or CID). Copy this hash because we will use it in the minting process.

Now, inside your `MintPinataParty.cdc` file, add the following:

This is a pretty simple transaction, thanks in large part to the work Flow has put in to make things easy, but let's walk through it. First, you'll notice the import statement at the top. If you remember, when we deployed the contract, we received an account for it.

That's what we need to reference. So replace `0xf8d6e0586b0a20c7` with the account address from your deployment.

Next we define the transaction. Everything that happens in here is related to the transaction we plan to execute.

The first thing we do in our transaction is define two reference variables, `receiverRef` and `minterRef`. In this case we are both the receiver of the NFT and the minter of the NFT. Those two variables are referencing resources we created in our Contract. If the person executing the transaction doesn't have access to the resource, the transaction will fail.

Next, we have a `prepare` function. This function takes the account information of the person trying to execute the transaction and does some validations. We try to "borrow" the capabilities available on the two resources we defined `NFTMinter` and `NFTReceiver`. This is where things will fail if the person executing the transaction does not have access to those resources.

Finally, we have our `execute` function. This function is where we build up the metadata for our NFT, mint the NFT, then associate the metadata prior to depositing the NFT in our account. If you notice, I created a metadata variable. Within that variable, I added some info about our token. Since our token is representing an event where a piñata is smashed at a party, and because we are trying to replicate much of what you see in NBA Top Shot, I have defined some statistics in the metadata. The velocity at which the kid swings the stick to hit the piñata, the angle of the swing, and a rating. I'm just having fun with these stats. You would enter whatever information makes sense for your token in a similar way, though.

You'll note, I am also defining a `uri` property in the metadata. This is going to point to the IPFS hash that hosts our asset file associated with the NFT. In this case, it is the actual video of the piñata being hit. You can replace the hash with the hash you received after you uploaded your file earlier.

We are prefixing the hash with `ipfs://` for a couple of reasons. This is the correct reference for a file on IPFS and can be used with IPFS's desktop client and browser extension. We can also paste it directly into the Brave browser now that they offer native support for IPFS content.

We call the `mintNFT` function which creates the token. We then have to call the `deposit` function to put it into our account. This is also where we pass in the metadata. If you remember, we defined in our `deposit` function a variable association that adds the metadata to the associated token id.

Finally, we simply log out the fact that the token has been minted and deposited.

We're now almost ready to send our transaction and mint the NFT. But first, we need to prepare our account. From the command line within your project's root folder, let's create a new private key for signing.

Run the following command:

```
flow keys generate
```

This will give you a public and a private key. **ALWAYS PROTECT YOUR PRIVATE KEY**

We will need the private key to sign the transaction, so we can paste that into our `flow.json` file. We also need to specify the signing algorithm. Here's what your `accounts` object in the `flow.json` file should look like now:

```
"accounts": {
  "emulator-account": {
     "address": "YOUR ACCOUNT ADDRESS",
     "privateKey": "YOUR PRIVATE KEY",
     "chain": "flow-emulator",
     "sigAlgorithm": "ECDSA_P256",
     "hashAlgorithm": "SHA3_256"
  }
},
```

If you intend to store any of this project on github or any remote git repository, make sure you do not include the private key. You may want to `.gitignore` your entire `flow.json`. Even though we are only using a local emulator, it's good practice to protect your keys.

Now that we've updated this, we can send our transaction. Doing so is as simple as running this command:

```
flow transactions send --code ./transactions/MintPinataParty.cdc --
signer emulator-account
```

We are referencing the transaction file we wrote and our signer account from the `flow.json`. If everything went well, you should see an output like this:

```
Getting information for account with address 0xf8d6e0586b0a20c7 ...

Submitting transaction with ID
4a79102747a450f65b6aab06a77161af196c3f7151b2400b3b3d09ade3b69823 ...

Successfully submitted transaction with ID
4a79102747a450f65b6aab06a77161af196c3f7151b2400b3b3d09ade3b69823
```

Now, the last thing we need to do is verify the token is in our account and fetch the metadata. Do do this, we're going to write a very simple script and call it from the command line.

From the root of your project, create a new folder called `scripts`. Inside of that, create a file called `CheckTokenMetadata.cdc`. In that file, add the following:

This script can be thought of in a similar way to read-only methods on Ethereum smart contracts are utilized. They are free and simply return data from the contract.

In our script, we are importing our contract from the deployed address. We are then defining a `main` function (which is the required function name for a script to run). Inside this function, we're defining three variables:

- nftOwner: This is simply the account that owns the NFT. We minted the NFT from the account that also deployed the contract, so in our example those two addresses are the same. That may not always be true depending on the design of your contracts in the future.
- capability: We need "borrow" the available capabilities (or functions) from the deployed contract. Remember, these capabilities are access-controlled, so if a capability is not available to the address trying to borrow it, the script will fail. We are borrowing capabilities from the `NFTReceiver` resource.
- receiverRef: This variable simply takes our capability and tells the script to borrow from the deployed contract.

Now, we can call functions (that are available to us). In this case, we want to make sure the address in question actually has received the NFT we minted, and then we want to see the metadata associated with the token.

Let's run our script and see what we get. Run the following on the command line:

```
flow scripts execute ./scripts/CheckTokenMetadata.cdc
```

You should see an output something like this for the metadata output:

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

Congratulations! You successfully created a Flow smart contract, minted a token, and associated metadata to the token, and stored the token's underlying digital asset on IPFS. Not too bad for part one of a tutorial.

Next up, we have a tutorial on building a front-end React application that will allow you to display your NFT by fetching the metadata and resolving that metadata.

Happy Pinning!

Go to part two:

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

Using Pinata: Part 2

medium.com

Additional reading:

Who Is Responsible for NFT Data?

Is Da Vinci Responsible for Maintaining the Mona Lisa?

medium.com

The File Requirements for NFTs

Purpose-Built, Secure, and Ad-Free

medium.com

Nft Flow Blockchain Ipfs Nba Top Shot Digital Collectibles