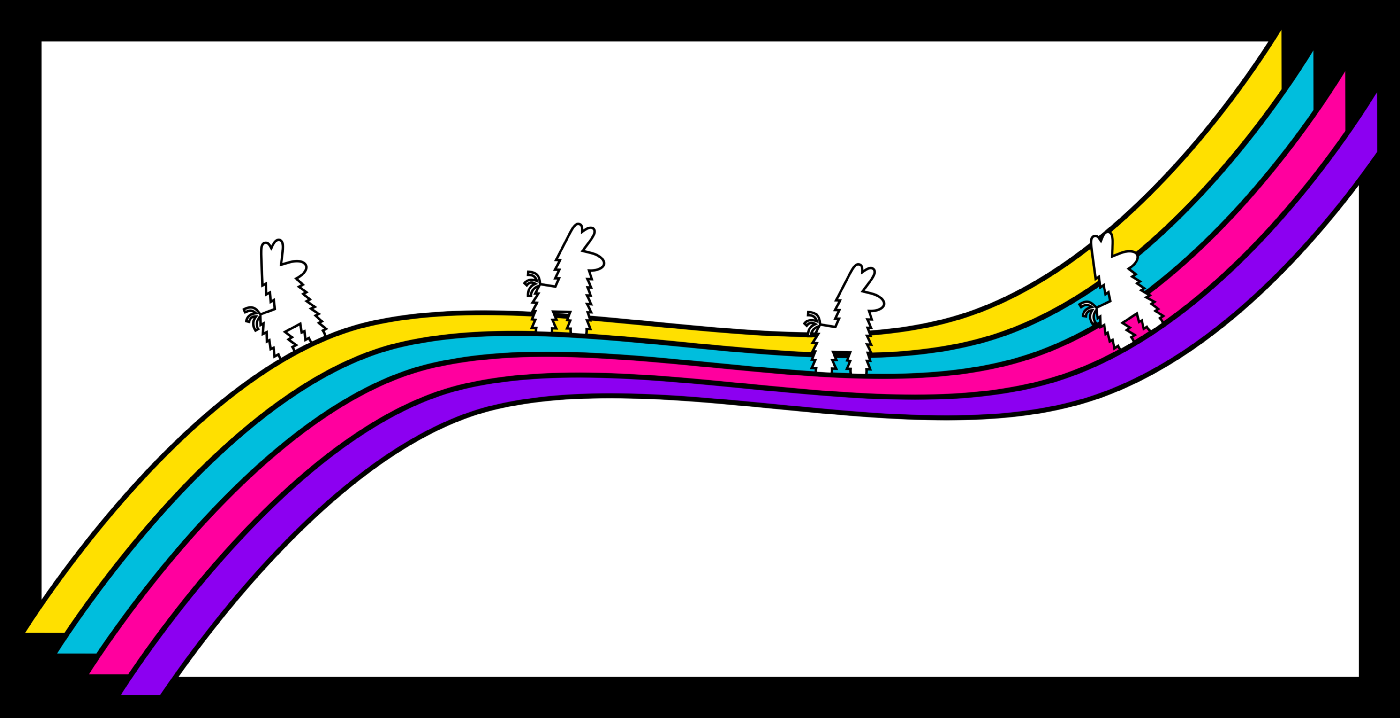
# **How to Build ERC-721 NFTs with IPFS**

## Using Open Zeppelin, Truffle, and Pinata



The [ERC-721 standard](https://eips.ethereum.org/EIPS/eip-721) has given rise to the non-fungible token (NFT) market on Ethereum. ERC-721 is a standard for creating an NFT—a complicated way of saying “unique items”. Any unique *thing* can be an NFT. A house, a baseball card, a piece of art, etc. But the power is not in the item just being unique and digitized. The power is in verifiability. That’s where the ERC-721 standard shines.

The problem with creating ERC-721 tokens comes from storing the underlying assets. Blockchains are not great for storing large pieces of data. In 2017, Jamila Omar of Interplanetary Database estimated the cost of storing 1GB of data on Ethereum would be more than $4 million.

*In general, the cost to store data on Ethereum works out to approximately 17,500 ETH/GB, or around $4,672,500 at today’s prices.*

* [Jamila Omar, 2017](https://medium.com/ipdb-blog/forever-isnt-free-the-cost-of-storage-on-a-blockchain-database-59003f63e01)

So if we know the cost of storing the assets tied to an NFT is too high to use a blockchain, what are the alternatives? We could store assets using traditional cloud storage. Amazon’s S3 and Microsoft’s Azure provide cheap storage solutions. However, traditional cloud storage as we know it has a major flaw.

*It’s not cryptographically verifiable.*

**Verifiability**

The whole point of an NFT is digital verification and control of what might be a physical or digital asset. If we cannot verify the underlying asset itself in a similar way to verifying the ownership of the token that represents the asset, we have lost track of the ultimate goal.

The solution to both problems is [IPFS](https://ipfs.io/). IPFS is a distributed storage network. It works in a similar way to cloud storage. You make a request for content and that content is returned. However, the big difference is the content is stored by utilizing a global network of storage providers. IPFS leverages a tool called content addressability. This means that instead of you making a request to that data center in Ohio for a piece of content, you would instead make a request FOR the content itself. It might be located in Ohio. It might be located closer. With content addressability, you no longer have to rely on a single location for the retrieval of content. This is far more efficient for global blockchains.

IPFS also takes care of the verifiability for us. Because all content is defined and stored based on the content itself, should a piece of content be tampered with or changed, we would have a mismatch when trying to verify the content and know it is wrong. Let’s make this a little more clear with a simple example:

Alice stores a picture of a cat on IPFS and that cat picture is represented by a content identifier. For simplicity, let’s say the identifier is “C”.

Bob requests that cat picture and then draws a mustache on that poor cat. When Bob uploads his picture, he will no longer have the same identifier. Because he has changed the underlying data (from cat to mustachioed cat), Bob’s identifier might be “M”.

If Bob tried to pass his photo off as Alice’s anyone who bothered to check would know he was lying. Alice’s identifier does not match Bob’s and therefore, the image Bob is trying to pass off as Alice’s is verifiably fake.

I think you can start to see how this comes in handy when trying to verify digital assets like NFTs. So, with that in mind, let’s see how we can create an NFT and store the associated asset on IPFS.

# **Getting Started**

For this tutorial, you’re going to need a few things:

* A good text editor
* IPFS installed
* Ganache — Ethereum’s local blockchain—installed
* Truffle installed
* NodeJS installed
* Pinata API Key

You can choose any text editor you’d like. I prefer [Visual Studio Code](https://code.visualstudio.com/), but it’s up to you.

Once you’ve figured out your editor, let’s make sure IPFS is installed. Follow the instructions [directly from IPFS here](https://docs.ipfs.io/how-to/command-line-quick-start/).

You’ll be able to grab [Truffle](https://www.trufflesuite.com/truffle) and [Ganache](https://www.trufflesuite.com/ganache) from the same place as they fall under the full

[Truffle](https://medium.com/u/9c9995b303d5?source=post_page-----e76a21d8f914-----------------------------------)

Suite.

If you don’t already have

[Node.js](https://medium.com/u/96cd9a1fb56?source=post_page-----e76a21d8f914-----------------------------------)

on your machine, you’ll need to download that as well. You can find the [most recent version here](https://nodejs.org/en/).

Finally, you want to make sure your precious asset for which you create an NFT is permanently stored on IPFS. This can be done by [running your own IPFS node](https://medium.com/pinata/how-to-deploy-an-ipfs-node-on-digital-ocean-c59b9e83098e) or by using what’s known as an IPFS [Pinning Service](https://medium.com/pinata/what-is-an-ipfs-pinning-service-f6ed4cd7e475). For simplicity, we are going to replicate it through the Pinata pinning service. [Sign up for an account here](https://pinata.cloud/).

# **Writing the Smart Contract**

**Let’s get a quick disclaimer out of the way here. I am not a professional smart contract developer. I know enough to be dangerous, and in the blockchain world, dangerous could equal losing money. So be careful, do your research, and certainly find best practices. This guide is intended as an educational starting point and there are probably better ways to do what I’m showing you here.**

ERC-721 tokens are governed by smart contracts. Fortunately, the folks over at [OpenZeppelin](https://openzeppelin.com/) make ensuring you write good contracts easy. We’ll use their ERC-721 contract to help us get started.

First, in your terminal, create a new project folder.

mkdir mySpecialAsset && cd mySpecialAsset

Next, we’ll initialize our project directory using NPM.

npm init -y

Now, we can utilize Truffle to initialize our smart contract project.

truffle init

Now, we need to get access to those sweet, sweet Open Zeppelin contracts. To do that, we’ll install Open Zeppelin’s solidity library like this:

npm install @openzeppelin/contracts

We will need to think of a good name for our token. I’m going to call mine UniqueAsset because we are ensuring NFTs must be associated with unique underlying assets. You can call your contract anything you’d like. We want to house the contract in a contracts folder, so we’ll create our file like this:

touch contracts/UniqueAsset.sol

We can now open that file and get to work. We need to specify the version (or versions of solidity that are compatible with our contract. We will also need to import the contract frameworks from Open Zeppelin. And we can start to implement the contract itself. Let’s take a look:

pragma solidity ^0.6.0;

import "@openzeppelin/contracts/token/ERC721/ERC721.sol";

import "@openzeppelin/contracts/utils/Counters.sol"

contract UniqueAsset is ERC721{

constructor() public ERC721("UniqueAsset", "UNA") {}

}

We need to specify the version of Solidity we’re using. In this case, I’m using 0.6.0. You can make sure that Truffle compiles your contract using the correct version of Solidity by editing the truffle-config.jsfile. I am importing two contracts from Open Zeppelin: ERC721 and Counters. Counters is just helping us increment the token ids after issuance. Finally, in the constructor for our contract, we are defining our token name and symbol. Again, you can set this to be whatever you’d like.

We’re going to need to add some logic into our contract. Let’s do that now.

First, let’s think about what we’re trying to do here. We want to issue NFTs for specific assets. We want those assets to be verifiable as much as we want ownership to be verifiable. So a couple of things we need to consider here:

1. We want to associate the NFT with an IPFS content identifier (hash).
2. We don’t ever want to mint (or create) an NFT that maps to the same IPFS hash as another NFT.

Let’s start by defining variables in our contract that we’ll use to help control the above two points. Above your constructor in the contract code, add the following:

using Counters for Counters.Counter;

Counters.Counter private \_tokenIds;

mapping(string => uint8) hashes;

constructor() public ERC721("UniqueAsset", "UNA") {}

...

We are using Counters to help us increment the identifiers for the tokens we mint. We are also creating a \_tokenIds variable to keep track of all of the tokens we’ve issued. And finally, for our top-level variables, we’re creating a mapping for the IPFS hashes associated with tokens. This will help prevent issuing tokens mapped to a hash previously associated with another token.

Your contract should now look like this:

pragma solidity ^0.6.0;

import "@openzeppelin/contracts/token/ERC721/ERC721.sol";

import "@openzeppelin/contracts/utils/Counters.sol";

contract UniqueAsset is ERC721 {

using Counters for Counters.Counter;

Counters.Counter private \_tokenIds;

mapping(string => uint8) hashes;

constructor() public ERC721("UniqueAsset", "UNA") {}

}

We need to add a method to our contract that will allow us to mint an NFT for a specific IPFS hash if not token has been minted yet for that hash. We’ll do this just below our constructor.

function awardItem(address recipient, string memory hash, string memory metadata)

public

returns (uint256)

{

require(hashes[hash] != 1);

hashes[hash] = 1;

\_tokenIds.increment();

uint256 newItemId = \_tokenIds.current();

\_mint(recipient, newItemId);

\_setTokenURI(newItemId, metadata);

return newItemId;

}

There’s a lot going on here, so let’s walk through this line by line. Our function takes two parameters: An address variable called recipient, a string variable called hash, and a string variable called metadata. The address variable is the person’s wallet address who will receive the NFT. The string variable for the hash is the IPFS hash associated with the content we are creating the NFT for. And the string variable for metadatashould refer to a link to the JSON metadata for the asset. The metadata might include the asset name, a link to an image referencing the asset, or anything else you want.

Then, after defining our function, we are making it public. This just means it can be called from outside the smart contract. We are also defining the return value of our function to be of type uint256.

Inside the function, we are using Solidity’s built-in require to automatically reject the contract call if the hash has been used to mint an NFT before. You’ll notice we are checking if our hashes mapping has a matching hash with the integer of 1. If so, that hash has been used.

If the hash has not been used, we add the hash passed through our function to our hashes mapping and set its value to 1).

Finally, we increment our \_tokenIds variable because we’re about to create a new NFT and we mint our token, returning the token identifier.

That was a lot, so let’s quickly summarize without leaning on the code. Our contract right now takes in a person’s Ethereum wallet address and an IPFS hash. It checks to make sure that hash doesn’t match a previously minted NFT. If all is good, a new NFT is created specific to that IPFS hash.

Ok, we’ve written our contract. Now what?

Let’s compile it and deploy it. Remember I asked you to install Ganache. We’re going to use that now. Start Ganache either through the ganache-cli or by using the desktop client (I prefer the desktop client).

If you take a look within your project directory, you’ll see a folder called migrations. We need to create a new migration file. These files are very specific and in this case, we need it to run after the existing migration file that was created by default for you. To ensure that happens, we will name our migration file with a 2 at the beginning. Go ahead and call your new migration 2-deploy-contract.js. Within that file, add the following:

var UniqueAsset = artifacts.require("UniqueAsset");

module.exports = function(deployer) {

deployer.deploy(UniqueAsset);

};

Replace the references to UniqueAsset with whatever you are calling your contract.

Once that’s done and saved, in your terminal, within the project directory, run:

truffle compile

Assuming you didn’t hit any errors, your contract has been compiled and can now be deployed. Simply run:

truffle migrate

If you get an error, you may need to manually set the port that Ganache is running on. The desktop client makes this easy to find. In your truffle-config.js file, you can find the networks section and set the development network port appropriately.

If all went well, you just deployed your NFT smart contract. Here seems like a good opportunity to remind you that **I am not an expert smart contract developer. There are surely best practices I missed that you should research yourself if you are creating smart contracts.**

You can test your contract by writing tests in Truffle against it, or you can head over to [Remix](https://remix.ethereum.org/) and paste your contract in there and run tests. If you do test it, you’ll notice that you can mint an NFT associated with an IPFS hash, but if you try to mint another NFT for that same hash, the call to your contract will fail.

Now that we’ve taken care of the smart contract, we need to get our underlying asset onto IPFS and make sure it’s available when it’s time to mint an NFT associated with it.

# **Adding Asset to IPFS**

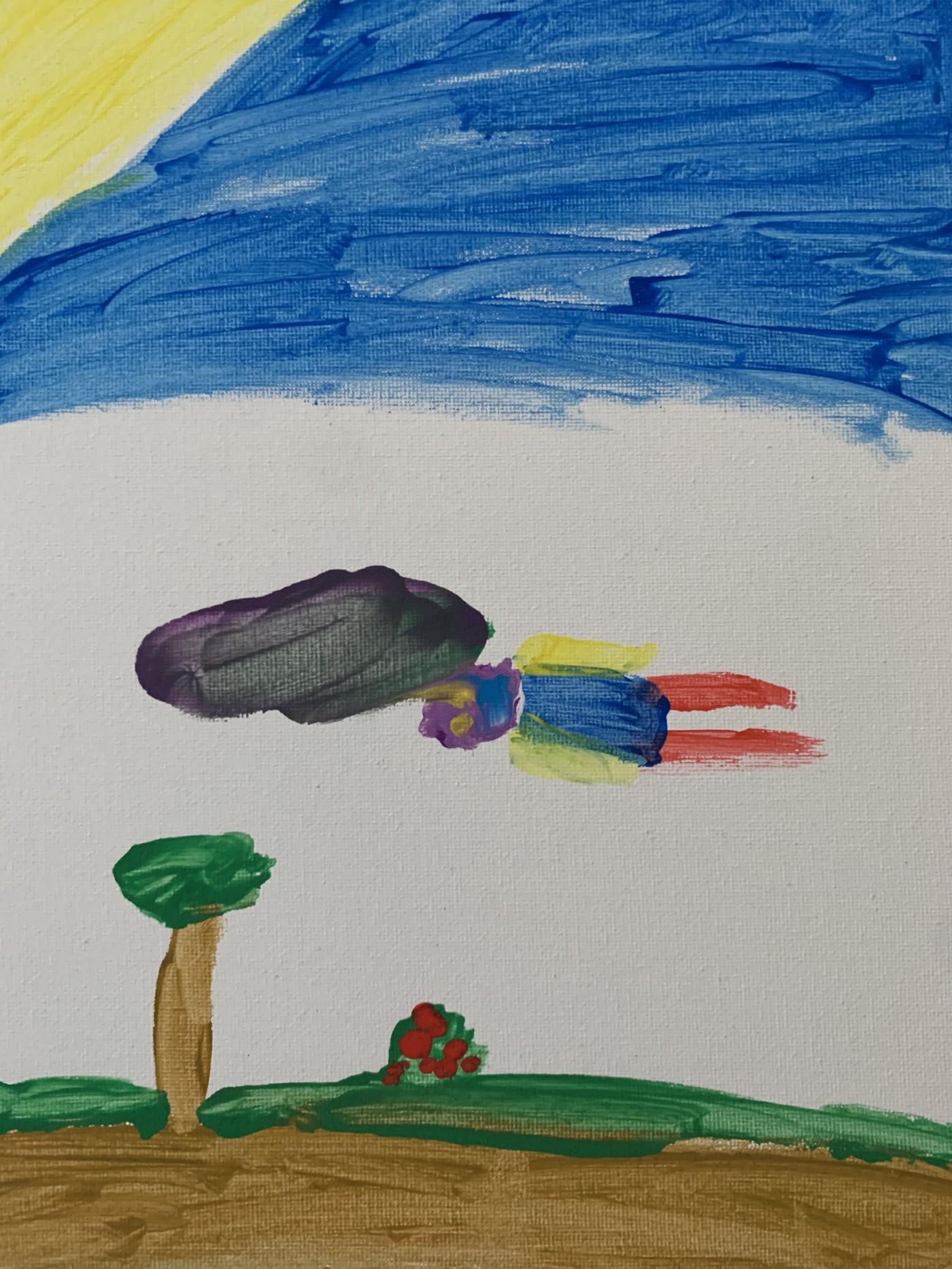
We’re going to use Pinata to add our asset to IPFS and ensure that it remains pinned. We will also add our JSON metadata to IPFS so that we can pass it through to our token contract. So, log into the account you previously created on Pinata. In the top-right, click the account dropdown and choose Account. There you will be able to see your API key. You can hover to see your API secret.

Copy both of those as we’ll be using them in our code to upload our asset file.

Click New API Key and make your selections. For me, I think I only want this key to be used once and I only want it to have access to the pinFileToIPFS endpoint since that’s how we’re going to be pushing our asset file to IPFS.

Once you have your API Key and Secret Key, you can write some code to add your asset file to IPFS. If you’re feeling tired after all that smart contract work, don’t worry because this is going to be super simple. In fact, if you want to skip the code entirely, Pinata [has a convenient upload feature](https://pinata.cloud/pinataupload) in the UI.

In your code editor, create a new file called uploadFile.js. This can be in the same directory where you created your smart contract. Before we write our code, it’d be good to have your asset file ready. Just make sure it’s saved somewhere on the computer you are using. For me, I’m going to be uploading a painting my son did:



Now that we have our asset saved and ready to upload, let’s write our code. We need to two dependencies to make this easier for us. In your terminal, at the root of your project, run the following:

npm i axios form-data

Now, in the uploadFile.js file, add the following:

const pinataApiKey = "YOURAPIKEY";

const pinataSecretApiKey = "YOURSECRETKEY";

const axios = require("axios");

const fs = require("fs");

const FormData = require("form-data");

const pinFileToIPFS = async () => {

const url = `https://api.pinata.cloud/pinning/pinFileToIPFS`;=

let data = new FormData();

data.append("file", fs.createReadStream("./pathtoyourfile.png"));

const res = await axios.post(url, data, {

maxContentLength: "Infinity",

headers: {

"Content-Type": `multipart/form-data; boundary=${data.\_boundary}`

pinata\_api\_key: pinataApiKey,

pinata\_secret\_api\_key: pinataSecretApiKey,

},

});

console.log(res.data);

};

pinFileToIPFS();

To run the script in this file, simply execute the following command from the terminal:

node uploadFile.js

Upon successful upload, you’ll get a result like this:

{

IpfsHash: 'QmfAvnM89JrqvdhLymbU5sXoAukEJygSLk9cJMBPTyrmxo',

PinSize: 2936977,

Timestamp: '2020-12-03T21:07:13.876Z'

}

That hash is the verifiable representation of your asset. It points to your asset on the IPFS network. If someone tampered with your asset and changed it, the hash would be different. That hash is what should be used when minting the NFTs through our smart contract. Any IPFS host offering a public gateway will be able to display the content for you now.

Pinata has a gateway, and you can [view the asset I just uploaded here](https://gateway.pinata.cloud/ipfs/QmfAvnM89JrqvdhLymbU5sXoAukEJygSLk9cJMBPTyrmxo/).

The last thing we need to do is create a JSON file that represents our assets and its metadata. This makes it easier for any service you might want to list your asset to show the appropriate metadata. Let’s create a simple JSON file like this:

{

"name":"My Kid's Art",

"hash": "QmfAvnM89JrqvdhLymbU5sXoAukEJygSLk9cJMBPTyrmxo",

"by": "Justin Huner"

}

You can add whatever metadata you’d like, but it’s important to include the hash. This is the reference to the actual asset. Now, upload this file the same way you uploaded the asset file using Pinata. When you get back the IPFS hash for the metadata, keep that handy. You’ll need that when creating the token.

Remember, the smart contract takes in a metadata string? That string is going to be the IPFS URL for the metadata. You’ll construct it like this:

ipfs://YOUR\_METADATA\_HASH

So to summarize, you’ll be passing three items into the smart contract function we previously created:

* Recipient Address
* Asset Hash
* Metadata URL

# **Bringing It All Together**

NFTs are an important improvement in the way we handle ownership of all kinds of goods. They are easily transferable and simplify the process of creating ownership and proving ownership. The missing piece, though, has been the verification of ownership of a specific item.

By saving assets to IPFS and associating the IPFS hash with the asset’s NFT, we can expand the verifiable ownership of the asset to verification of the asset’s validity itself.

[Pinata](https://pinata.cloud/) helps streamline this process by making the storage of assets on IPFS easy.

Happy pinning!