**Encoding SVGs to be stored onchain**

**SVG NFT Encoding**

Before we even begin, I want to say you *can* pass the SVG itself to a constructor and encode it on-chain (and I'll show you how this works a bit later), but if we encode the SVG with base64 *first* and pass this to our constructor, it'll save us a step.

Here's the encoded SVGs again for those without base64 installed.

HappySVG:

data:image/svg+xml;base64,

SadSVG:

data:image/svg+xml;base64,

❗ **NOTE** Those who have decided to use their own custom SVG images, remember you can acquire the encoding with the command base64 -i <filename> while in the img directory!

Now, if we're going to be passing *already encoded* imageURIs to our constructor, it's probably a good idea to adjust the naming of our storage variables for clarity. Let's do this before moving on.

contract MoodNft is ERC721 {

uint256 private s\_tokenCounter;

string private s\_sadSvgImageUri;

string private s\_happySvgImageUri;

constructor(string memory sadSvgImageUri, string memory happySvgImageUri) ERC721("Mood NFT", "MN"){

s\_sadSvgImageUri = sadSvgImageUri;

s\_happySvgImageUri = happySvgImageUri;

}

}

❗ **IMPORTANT** **tokenURI != imageURI**

It's important to remember that imageURI is one property of a token's tokenURI. A tokenURI is usually a JSON object!

At this point you may be asking, if the tokenURI is a JSON object, how do we store this on-chain? The answer: We can encode it in much the same way!

OpenZeppelin actually offers a [**Utilities**](https://docs.openzeppelin.com/contracts/4.x/utilities) package which includes a Base64 function which we can leverage to encode our tokenURI on-chain.

We've already got OpenZeppelin contracts installed, so we can just import Base64 into our NFT contract.

import { Base64 } from "@openzeppelin/contracts/utils/Base64.sol";

Let's start off our tokenURI function by defining a variable, string memory tokenMetadata. We can set this equal to our JSON object in string format like so:

string memory tokenMetadata = abi.encodePacked(

'{"name: "',

name(),

'", description: "An NFT that reflects your mood!", "attributes": [{"trait\_type": "Mood", "value": 100}], "image": ',

imageURI,

'"}';

)

In the above, we're using abi.encodePacked to concatenate our disparate strings into one object. This allows us to easily parameterize things a little bit.

In order to determine our imageURI we'll need to derive this from the mood which has been set to our NFT token. As such, we're going to need a way to track the mood of each token. This sounds like a mapping to me. We can even spice it up a little bit and map our choice to an enum, which would allow someone to set more moods, if they wanted to expand on things in the future.

contract MoodNft is ERC721 {

uint256 private s\_tokenCounter;

string private s\_sadSvgImageUri;

string private s\_happySvgImageUri;

enum Mood {

HAPPY,

SAD

}

mapping(uint256 => Mood) private s\_tokenIdToMood;

}

When an NFT is minted, they'll need a default mood, let's default them to happy.

function mintNft() public {

\_safeMint(msg.sender, s\_tokenCounter);

s\_tokenIdToMood[s\_tokenCounter] = Mood.HAPPY;

s\_tokenCounter++;

}

Now, back in our tokenURI function, we can define a conditional statement which will derive what our imageURI should be.

function tokenURI(

uint256 tokenId

) public view override returns (string memory) {

string memory imageURI;

if (s\_tokenIdToMood[tokenId] == Mood.HAPPY) {

imageURI = s\_happySvgImageUri;

} else {

imageURI = s\_sadSvgImageUri;

}

string memory tokenMetadata = string.concat(

'{"name: "',

name(),

'", description: "An NFT that reflects your mood!", "attributes": [{"trait\_type": "Mood", "value": 100}], "image": ',

imageURI,

'"}'

);

}

Alright, this looks good, but we're not done yet. We'll add a way to flip our NFTs mood soon. For now, we just have our metadata as a string in our contract, we need to convert this to the hashed syntax that our browser understands.

This is where things might get a little wild.

Currently we have a string, in order to acquire the Base64 hash of this data, we need to first convert this string to bytes, we can do this with some typecasting.

bytes(

abi.encodePacked(

'{"name: "',

name(),

'", description: "An NFT that reflects your mood!", "attributes": [{"trait\_type": "Mood", "value": 100}], "image": ',

imageURI,

'"}'

)

);

Now we can apply our Base64 encoding to acquire our hash.

Base64.encode(

bytes(

abi.encodePacked(

'{"name: "',

name(),

'", description: "An NFT that reflects your mood!", "attributes": [{"trait\_type": "Mood", "value": 100}], "image": ',

imageURI,

'"}'

)

)

);

At this point, our tokenURI data is formatting like our imageUris were. If you recall, we needed to prepend our data type prefix(data:image/svg+xml;base64,) to our Base64 hash in order for a browser to understand. A similar methodology applies to our tokenURI JSON but with a different prefix. Let's define a method to return this string for us. Fortunately the ERC721 standard has a \_baseURI function that we can override.

function \_baseURI() internal pure override returns (string memory) {

return "data:application/json;base64,";

}

Now, in our tokenURI function again, we can concatenate the result of this \_baseURI function with the Base64 encoding of our JSON object... and finally we can type cast all of this as a string to be returned by our tokenURI function.

return string(

abi.encodePacked(

\_baseURI(),

Base64.encode(

bytes(

abi.encodePacked(

'{"name: "',

name(),

'", description: "An NFT that reflects your mood!", "attributes": [{"trait\_type": "Mood", "value": 100}], "image": ',

imageURI,

'"}'

)

)

)

)

);

Admittedly, this is a lot at once. Before we add any more functionality, let's consider writing a test to make sure things are working as intended. To summarize what's happening:

1. We created a string out of our JSON object, concatenated with abi.encodePacked.
2. typecast this string as a bytes object
3. encoded the bytes object with Base64
4. concatenated the encoding with our \_baseURI prefix
5. typecast the final value as a string to be returned as our tokenURI

**Testing tokenURI**

Given the complexity of our tokenURI function, let's take a moment to write a quick test and assure it's returning what we'd expect it to. Create the file test/MoodNftTest.t.sol and set up our usual boilerplate.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.18;

import {Test} from "forge-std/Test.sol";

import {MoodNft} from "../src/MoodNft.sol";

contract MoodNftTest is Test {

MoodNft moodNft;

function setUp() public {

}

}

We'll need to declare our Happy and Sad SVG URIs as constants in our test, we can use these in the deployment of our MoodNft contract within the setUp function of our Test.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.18;

import {Test} from "forge-std/Test.sol";

import {MoodNft} from "../src/MoodNft.sol";

contract MoodNftTest is Test {

MoodNft moodNft;

string public constant HAPPY\_SVG\_URI = "data:image/svg+xml;base64,";

string public constant SAD\_SVG\_URI = "data:image/svg+xml;base64,";

function setUp() public {

moodNft = new MoodNft(SAD\_SVG\_URI, HAPPY\_SVG\_URI);

}

}

Finally we can write a test function. All that's required is to mint one of our MoodNft tokens, and then we can console out the tokenURI of that tokenId(0)! We'll need to create a user to do this.

❗ **PROTIP** Don't forget to import console!

import {Test, console} from "forge-std/Test.sol";`

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.18;

import {Test, console} from "forge-std/Test.sol";

import {MoodNft} from "../src/MoodNft.sol";

contract MoodNftTest is Test {

...

address USER = makeAddr("USER");

function setUp() public {

moodNft = new MoodNft(SAD\_SVG\_URI, HAPPY\_SVG\_URI);

}

function testViewTokenURI() public {

vm.prank(USER);

moodNft.mintNft();

console.log(moodNft.tokenURI(0));

}

}

Now let's run it (make sure you're back in your root directory)!

forge test --mt testViewTokenURI -vvvv

Logs:

data:application/json;base64,

This looks pretty good! If we paste this into our browser we should see...

... That looks like a JSON to me! Now, let's copy that imageURI into our browser...

Close enough!

**Wrap Up**

Amazing! We've written most of our MoodNFT contract and we've gotta ahead of the game with our tests, verifying that our tokenURI function is infact returning a correctly formatting tokenURI which has been derived from our NFT's current mood setting.

In the next lesson we'll set up the functionality necessary to flip our NFT's mood!