**Create the deployment script**

**SVG NFT Deploy Script**

In this lesson, we'll jump right into creating the script to deploy our MoodNFT collection. We'll look at how this can be used to upgrade our tests, making them more dynamic and we'll discuss the value of integration testing.

To begin, we'll need to create the file script/DeployMoodNft.s.sol and fill it with our script boilerplate.

// SPDX-License-Identifier:MIT

pragma solidity ^0.8.18;

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

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

contract DeployMoodNft is Script {

function run() external returns (MoodNft) {}

}

Looks great! Now we should consider how we're mention to deploy MoodNft.sol. We know that the constructor arguments for this contract take a sadSvgImageUri and a happySvgImageUri, so much like we did in MoodNftTest.t.sol, we *could* hardcode these values. A better approach however may be to write our deploy script to read this data itself from our workspace. Our script can even do all the encoding for us.

Let's start with creating this encoding function.

function svgToImageURI(string memory svg) public pure returns (string memory){

string memory baseURL = "data:image/svg+xml;base64,";

}

Set up like this, we can now use the Base64 offering from OpenZeppelin to encode the data passed to this function, and then concatenate it with our baseURI. We'll need to import Base64.

// SPDX-License-Identifier:MIT

pragma solidity ^0.8.18;

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

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

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

contract DeployMoodNft is Script {

function run() external returns (MoodNft) {}

function svgToImageURI(string memory svg) public pure returns (string memory){

string memory baseURL = "data:image/svg+xml;base64,";

string memory svgBase64Encoded = Base64.encode(bytes(svg));

return string(abi.encodePacked(baseURL, svgBase64Encoded));

}

}

The above function is taking the svg string parameter, encoding it with the OpenZeppelin Base64.encode function, and then prepends the encoded value with our baseURL. Great job!

❗ **PROTIP** You can replace abi.encodePacked with the more up-to-date string.concat!

Before moving on, we should write a quick test to verify this is encoding things we way we expect.

**Testing Our Encoding**

Let's test the function we just wrote. To keep things clean, create a new file test/DeployMoodNftTest.t.sol. The setup for this file is going to be the same as always.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.18;

import {DeployMoodNft} from "../script/DeployMoodNft.s.sol";

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

contract DeployMoodNftTest is Test {

DeployMoodNft public deployer;

function setUp() public {

deployer = new DeployMoodNft();

}

Easy enough, we're definitely getting good at this by now.

Next we'll need a test function to verify that our SVG is being converted to a URI appropriately. We should have an example to compare the results of our test to. I've included an example URI below, feel free to encode your own SVG if you'd like!

**Sample SVG:**



In our test now, we can assign an expectedUri variable to this string. We'll need to also define the svg which we'll pass to the function.

function testConvertSvgToUri() public view {

string memory expectedUri = "";

string memory svg = '<svg xmlns="http://www.w3.org/2000/svg" width="500" height="500"><text x="200" y="250" fill="black">Hi! You decoded this! </text></svg>';

string memory actualUri = deployer.svgToImageURI(svg);

}

Great! Now we'll need to assert that our expectedUri is equal to our actualUri. Remember, we can't compare strings directly since they're effectively bytes arrays. We need to hash these for easy comparison.

assert(

keccak256(abi.encodePacked(expectedUri)) ==

keccak256(abi.encodePacked(actualUri))

);

All that's left is to run our test!

forge test --mt testConvertSvgToUri

Nailed it! Our solidity scripted encoding is working just like our command line.

DeployMoodNft.sol isn't currently defining what our svg parameters are, we hardcoded these into our test. Let's make the deploy script a little more dynamic by leverage the [**Foundry Cheatcode readFile**](https://book.getfoundry.sh/cheatcodes/fs?highlight=readFile#signature).

Before we can allow Foundry to read our files into our deploy script, we'll need to set some permissions in foundry.toml. Add this to your foundry.toml:

fs\_permissions = [{access = "read", path = "./img/"}]

❗ **NOTE** This line provides the Foundry framework read permissions, specifically in the img directory. This is much safer than setting FFI = true!

With this in place, we can now use the readFile cheatcode to access these SVG files in our deploy script.

// SPDX-License-Identifier:MIT

pragma solidity ^0.8.18;

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

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

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

contract DeployMoodNft is Script {

function run() external returns (MoodNft) {

string memory sadSvg = vm.readFile("./img/sadSvg.svg");

string memory happySvg = vm.readFile("./img/happySvg.svg");

}

function svgToImageURI(string memory svg) public pure returns (string memory){

string memory baseURL = "data:image/svg+xml;base64,";

string memory svgBase64Encoded = Base64.encode(bytes(svg));

return string(abi.encodePacked(baseURL, svgBase64Encoded));

}

}

Now we can deploy our MoodNft.sol contract in our run function, passing it the data read in from these files.

function run () external returns (MoodNft) {

string memory sadSvg = vm.readFile("./img/sadSvg.svg");

string memory happySvg = vm.readFile("./img/happySvg.svg");

vm.startBroadcast();

MoodNft moodNft = new MoodNft(svgToImageURI(sadSvg), svgToImageURI(happySvg));

vm.stopBroadcast();

return moodNft;

}

Because we're now using a deployment script, our testing framework is changing a little bit. The test we just wrote is more correctly classified as an integration test than a unit test. Let's keep things distinct by adjusting our test folder a bit first.

Create the directories test/integration and test/unit. Within test/integration create a copy of our MoodNftTest.t.sol and name it something like MoodNftIntegrationsTest.t.sol, and move our BasicNft.t.sol file here as well (it uses a deployer too!).

We'll adjust MoodNftIntegrationsTest.t.sol to use our deployer next.

❗ **NOTE** Moving your test files about may have broken some of your imports! You can add ../ to the beginning of each import to "back it out" of a directory. Things should work again!

**MoodNftIntegrationsTest.t.sol**

The changes to be made in this file are fairly small, but impactful. Instead of deploying with:

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

We can use our newly written deployer. It'll need to be imported.

<details> <summary>MoodNftIntegrationsTest.t.sol</summary>

//SPDX-License-Identifier: MIT

pragma solidity ^0.8.18;

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

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

import {DeployMoodNFT} from "../../script/DeployMoodNFT.s.sol";

contract MoodNFTTest is Test {

MoodNFT moodNFT;

address USER = makeAddr("USER");

DeployMoodNFT deployer;

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 {

deployer = new DeployMoodNFT();

moodNFT = deployer.run();

}

function testViewTokenURIIntegration() public {

vm.prank(USER);

moodNFT.mintNft();

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

}

}

</details>

With these adjustments, our tests should function identically to before.

**Testing Flipping the URI**

One thing we definitely haven't tested yet, and we should do quickly, is our flipMood function. Lets assure this properly swaps between happy and sad when called.

function testFlipMoodIntegration() public {

vm.prank(USER);

moodNFT.mintNft();

vm.prank(USER);

moodNFT.flipMood(0);

assert(keccak256(abi.encodePacked(moodNft.tokenURI(0))) == keccak256(abi.encodePacked(SAD\_SVG\_URI)));

}

This test has our USER mint an NFT (which defaults as happy), and then flips the mood to sad with the flipMood function. We then assert that the token's URI matches what's expected.

Let's run it!

forge test --mt testFlipMoodIntegration

Uh oh. That ain't right.

**Wrap Up**

Wow, this was a big lesson. We've written a deploy script and refactored some of our tests into more secure integration style tests.

For some reason testFlipMoodIntegration is erroring on us though...

In the next lesson we'll get some practice debugging, I suppose!

See you there!