**Test the NFTs smart contract**

**Basic NFT Tests**

Once the setup is complete, it's time to jump into tests. Writing an array of tests serves to validate the functionality of our contract, we'll start with something simple and verify that our NFT name is set correctly.

Start with the usual boilerplate for our test contract. Create the file test/BasicNftTest.t.sol. Our test contract will need to import BasicNft, and our deploy script as well as import and inherit Foundry's Test.sol.

//SPDX-License-Identifier: MIT

pragma solidity ^0.8.18;

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

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

import {DeployBasicNft} from "../script/DeployBasicNFT.s.sol";

contract BasicNFTTest is Test {

DeployBasicNft public deployer;

BasicNFT public basicNft;

function setUp() public {

deployer = new DeployBasicNft();

basicNft = deployer.run();

}

}

To confirm that the Name of your NFT is correct, declare a function testNameIsCorrect and specify it as public view.

function testNameIsCorrect() public view {

string memory expected = "Doggie";

string memory actual = basicNft.name();

assert(expected == actual);

}

Now, you may believe that we can simply do something like the above. We know the ERC721 standard allows us to call the name function to verify what what set, so that should be it, right?

We actually run into an issue here.

**Comparing Strings**

If you recall from previous lessons, strings are actually a special data type. Under the hood, strings exist as an array of bytes, arrays can't be compared to arrays in this way, this is limited to primitive data types. Primitive data types include things like int, bool, bytes32, address etc.

So, how do we compare these strings? Since it's an array, we could loop through the elements of the array and compare each of them. This is entirely doable, but it's computationally expensive and going take a long time if the strings were very large!

A more elegant approach would be to encode each of our string objects into a hash and compare the hashes.

This is a point where I may use Foundry's tool, chisel, to sanity check myself as I go.

Chisel

Use chisel to create a couple simple strings

string memory cat = "cat";

string memory dog = "dog";

Now if you type cat, you should get a kinda crazy output that's representing the hex of that string.

We'll leverage abi.encodePacked to convert this to bytes, then finally we can use keccak256 to hash the value into bytes32, which we can can use in our value comparison.

❗ **NOTE** I know we haven't covered encoding or abi.encodePacked in great detail yet, but don't worry - we will.

If we apply this encoding and hashing methodology to our BasicNft test, we should come out with something that looks like this:

function testNameIsCorrect() public view {

string memory expectedName = "Doggie";

string memory actualName = basicNft.name();

assert(keccak256(abi.encodePacked(expectedName)) == keccak256(abi.encodePacked(actualName)));

}

In the above, we're encoding and hashing both of our strings before comparing them in our assertion. Now, if we run our test with forge test --mt testNameIsCorrect...

Great work! Let's write a couple more tests together.

**Testing Mint and Balance**

The next test we write will assure a user can mint the NFT and then change the user's balance. We'll need to create a user to prank in our test. Additionally, we'll need to provide our mint function a tokenUri, I've provided one below for convenience. If you've one prepared from the previous lesson, feel free to use it!

contract BasicNftTest is Test {

...

address public USER = makeAddr("user");

string public constant PUG =

"ipfs://bafybeig37ioir76s7mg5oobetncojcm3c3hxasyd4rvid4jqhy4gkaheg4/?filename=0-PUG.json";

...

function testCanMintAndHaveABalance() public {

vm.prank(USER);

basicNft.mintNft(PUG);

assert(basicNft.balanceOf(USER) == 1);

assert(keccak256(abi.encodePacked(PUG)) == keccak256(abi.encodePacked(basicNft.tokenURI(0))));

}

}

With this, we again should just be able to run forge test and see how things resolve.

**Wrap Up**

Great work, again! Our tests are looking great. In the next lesson we'll look at how to set up an interactions script for the contract so that we can test things on a public test net with some integration testing.

See you soon!