**Test and deploy the lottery smart contract pt.2**

**Next steps**

Great! We've written some amazing code, but you know our job here is not done! We need to test it. Let's be smart about testing, what do we need to be able to properly test the contract and what kind of tests shall we do?

**Plan:**

1. Write deploy scripts
2. Write tests
   1. Local chain
   2. Forked Testnet
   3. Forked Mainnet
3. Maybe deploy and run on Sepolia?

**Deployment scripts**

Please create a new file called DeployRaffle.s.sol inside the script folder.

And now you know the drill, go write as much of it as you can! After you get stuck or after you finish come back and check it against the version below:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.19;

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

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

contract DeployRaffle is Script {

function run() external returns (Raffle) {

}

}

We've started with the traditional SPDX declaration, then specified the pragma solidity version. We imported the Script from Foundry and the Raffle contract because we want to do a Raffle deployment script, declared the contract's name and made it inherit Script and created the run function which will return our Raffle contract deployment. Great!

Let's work smart, looking again over the plan we see that we'll have to deploy the Raffle contract on at least 3 different chains. Let's stop here with the deployment script and work on the HelperConfig.

Create a new file called HelperConfig.s.sol in the script folder.

Inside let's create the HelperConfig contract:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.19;

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

contract HelperConfig is Script {

struct NetworkConfig {

uint256 entranceFee;

uint256 interval;

address vrfCoordinator;

bytes32 gasLane;

uint64 subscriptionId;

uint32 callbackGasLimit;

}

}

We start with the SPDX and pragma solidity declarations. Then, we import Script from Foundry, name the contract and make it inherit Script. Cool! Now what do we need to deploy the Raffle contract? That information can be easily found in the Raffle contract's constructor:

constructor(uint256 entranceFee, uint256 interval, address vrfCoordinator, bytes32 gasLane, uint64 subscriptionId, uint32 callbackGasLimit)

We created a new struct called NetworkConfig and we matched its contents with the Raffle's constructor input.

Great! Now let's design a function that returns the proper config for Sepolia:

function getSepoliaEthConfig()

public

pure

returns (NetworkConfig memory)

{

return NetworkConfig({

entranceFee: 0.01 ether,

interval: 30, // 30 seconds

vrfCoordinator: 0x9DdfaCa8183c41ad55329BdeeD9F6A8d53168B1B,

gasLane: 0x787d74caea10b2b357790d5b5247c2f63d1d91572a9846f780606e4d953677ae,

subscriptionId: 0, // If left as 0, our scripts will create one!

callbackGasLimit: 500000, // 500,000 gas

});

}

The function above returns a NetworkConfig struct with data taken from [here](https://docs.chain.link/vrf/v2-5/supported-networks#sepolia-testnet). The interval, entranceFee and callbackGasLimit were selected by Patrick.

Ok, we need a couple more things. We need a constructor that checks what blockchain we are on and attributes a state variable, let's call it activeNetworkConfig, the proper config for the chain used.

NetworkConfig public activeNetworkConfig;

constructor() {

if (block.chainid == 11155111) {

activeNetworkConfig = getSepoliaEthConfig();

} else {

activeNetworkConfig = getOrCreateAnvilEthConfig();

}

}

Good, we only missing the getOrCreateAnvilEthConfig function.

For now, let's create only a part of it:

function getOrCreateAnvilEthConfig()

public

returns (NetworkConfig memory anvilNetworkConfig)

{

// Check to see if we set an active network config

if (activeNetworkConfig.vrfCoordinator != address(0)) {

return activeNetworkConfig;

}

}

We check if the activeNetworkConfig is populated, and if is we return it. If not we need to deploy some mocks. But more on that in the next lesson.