**Deploy Script**

Let's begin by creating a new file in the /script directory called DeployRaffle.sol and importing the Raffle contract.

pragma solidity ^0.8.19;

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

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

🗒️ **NOTE**  
There are two ways to import files in Solidity: using a direct path or a relative path. In this example, we are using a relative path, where the Raffle.sol file is inside the src directory but one level up (..) from the current file's location.

**The deployContract Function**

Next, let's define a function called deployContract to handle the **deployment process**. This function will be similar to the one we used in the FundMe contract.

contract DeployRaffle is Script {

function run() external {

deployContract();

}

function deployContract() internal returns (Raffle, HelperConfig) {

// Implementation will go here

}

}

To deploy our contract, we need various parameters required by the Raffle contract, such as entranceFee, interval, vrfCoordinator, gasLane, subscriptionId, and callbackGasLimit. The values for these parameters will vary *depending on the blockchain network we deploy to*. Therefore, we should create a HelperConfig file to specify these values based on the target deployment network.

**The HelperConfig.s.sol Contract**

To retrieve the correct network configuration, we can create a new file in the same directory called HelperConfig.s.sol and define a **Network Configuration Structure**:

contract HelperConfig is Script {

struct NetworkConfig {

uint256 entranceFee;

uint256 interval;

address vrfCoordinator;

bytes32 gasLane;

uint32 callbackGasLimit;

uint256 subscriptionId;

}

}

We'll then define two functions that return the *network-specific configuration*. We'll set up these functions for Sepolia and a local network.

function getSepoliaEthConfig() public pure returns (NetworkConfig memory) {

return NetworkConfig({

entranceFee: 0.01 ether, // 1e16

interval: 30, // 30 seconds

vrfCoordinator: 0x9DdfaCa8183c41ad55329BdeeD9F6A8d53168B1B,

gasLane: 0x787d74caea10b2b357790d5b5247c2f63d1d91572a9846f780606e4d953677ae,

callbackGasLimit: 500000, // 500,000 gas

subscriptionId: 0

});

}

function getLocalConfig() public pure returns (NetworkConfig memory) {

return NetworkConfig({

entranceFee: 0.01 ether,

interval: 30, // 30 second

vrfCoordinator: address(0),

gasLane: "",

callbackGasLimit: 500000,

subscriptionId: 0

});

}

We will then create an abstract contract CodeConstants where we define some network IDs. The HelperConfig contract will be able to use them later through inheritance

abstract contract CodeConstants {

uint256 public constant ETH\_SEPOLIA\_CHAIN\_ID = 11155111;

uint256 public constant LOCAL\_CHAIN\_ID = 31337;

}

These values can be used inside the HelperConfig constructor:

👀❗**IMPORTANT**  
We are choosing the use of **constants** over magic numbers

constructor() {

networkConfigs[ETH\_SEPOLIA\_CHAIN\_ID] = getSepoliaEthConfig();

}

We also have to build a function to fetch the appropriate configuration based on the actual chain ID. This can be done first by verifying that a VRF coordinator exists. In case it does not and we are not on a local chain, we'll revert.

function getConfigByChainId(uint256 chainId) public view returns (NetworkConfig memory) {

if (networkConfigs[chainId].vrfCoordinator != address(0)) {

return networkConfigs[chainId];

} else if (chainId == LOCAL\_CHAIN\_ID) {

return getOrCreateAnvilEthConfig();

} else {

revert HelperConfig\_\_InvalidChainId()

}

}

In case we are on a local chain but the VRF coordinator has already been set, we should use the existing configuration already created.

function getOrCreateAnvilEthConfig() public returns (NetworkConfig memory) {

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

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

return localNetworkConfig;

}

This approach ensures that we have a robust configuration mechanism that adapts to the actual deployment environment.