**Deploy a mock priceFeed**

**Testing locally**

In the previous lesson, we refactored our contracts to avoid being forced to use Sepolia every single time when we ran tests. The problem is we didn't quite fix this aspect. We made our contracts more flexible by changing everything for us to input the `priceFeed` address only once. We can do better!

It is very important to be able to run our all tests locally. We will do this using a \*\*mock contract\*\*.

Before we dive into the code, let's emphasize why this practice is so beneficial. By creating a local testing environment, you reduce your chances of breaking anything in the refactoring process, as you can test all changes before they go live. No more hardcoding of addresses and no more failures when you try to run a test without a forked chain. As a powerful yet simple tool, a mock contract allows you to simulate the behavior of a real contract without the need to interact with a live blockchain.

Thus, on our local Anvil blockchain we will deploy a contract that mimics the behavior of the Sepolia `priceFeed`.

**Where the magic happens**

Please create a new file in your `script` folder called `HelperConfig.s.sol`. Here we'll write the logic necessary for our script to deploy mocks when it detects we are performing tests on our local anvil chain. Also, here we will keep track of all the contract addresses we will use across all the different chains we will interact with.

The start:

// SPDX-License-Identifier: MIT

pragma solidity 0.8.19;

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

contract HelperConfig {

// If we are on a local Anvil, we deploy the mocks

// Else, grab the existing address from the live network

}

Copy the following functions inside the contract:

struct NetworkConfig {

address priceFeed; // ETH/USD price feed address

}

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

NetworkConfig memory sepoliaConfig = NetworkConfig({

priceFeed: 0x694AA1769357215DE4FAC081bf1f309aDC325306

});

return sepoliaConfig;

}

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

}

We decided to structure the information we need depending on the chain we are testing on. We use a struct to hold this information for every chain. You might think that we could have gotten away with a simple address variable but that changes if we need to store multiple addresses or even more blockchain-specific information.

For now, we created a getSepoliaEthConfig that returns the NetworkConfig struct, which contains the priceFeed address.

What do we need to do to integrate this inside the deployment script?

First of all, we need to be aware of the chain we are using. We can do this in the constructor of the HelperConfig contract.

Update the HelperConfig as follows:

NetworkConfig public activeNetworkConfig;

struct NetworkConfig {

address priceFeed; // ETH/USD price feed address

}

constructor(){

if (block.chainid == 11155111) {

activeNetworkConfig = getSepoliaEthConfig();

} else {

activeNetworkConfig = getAnvilEthConfig();

}

}

As you can see, we've defined a new state variable, called activeNetworkConfig which will be the struct that gets queried for blockchain-specific information. We will check the block.chainId at the constructor level, and depending on that value we select the appropriate config.

The block.chainId in Ethereum refers to the unique identifier of the blockchain network in which the current block is being processed. This value is determined by the Ethereum network itself and is typically used by smart contracts to ensure that they are interacting with the intended network. Go on [chainlist.org](https://chainlist.org/) to find out the ChainID's of different blockchains.

Let's update the DeployFundMe.s.sol to use our newly created HelperConfig.

import {HelperConfig} from "./HelperConfig.s.sol";

Add the following before the vm.startBroadcast line inside the run function:

// The next line runs before the vm.startBroadcast() is called

// This will not be deployed because the `real` signed txs are happening

// between the start and stop Broadcast lines.

HelperConfig helperConfig = new HelperConfig();

address ethUsdPriceFeed = helperConfig.activeNetworkConfig();

Run the forge test --fork-url $SEPOLIA\_RPC\_URL command to check everything is fine. All tests should pass.

Great, let's keep going.

Now that we've configured it for one chain, Sepolia, we can do the same with any other chain that has a priceFeed address available on [Chainlink Price Feed Contract Addresses](https://docs.chain.link/data-feeds/price-feeds/addresses?network=ethereum&page=1#overview). Simply copy the getSepoliaEthConfig function, rename it and provide the address inside it. Then add a new block.chainId check in the constructor that checks the current block.chainId against the chainId you find on [chainlist.org](https://chainlist.org/). You would also need a new RPC\_URL for the new blockchain you chose, which can be easily obtained from Alchemy.

This type of flexibility elevates your development game to the next level. Being able to easily test your project on different chains just by changing your RPC\_URL is an ability that differentiates you from a lot of solidity devs who fumble around with hardcoded addresses.

In the next lessons, we will learn how to use Anvil in our current setup. Stay tuned.

**Update to Pricefeed Version**

*Last updated on January 6, 2025*

**Note!**

Chainlink has updated the version of their pricefeed on mainnet. Tests forking mainnet, as shown in the video, may fail. Adjust your test as follows to account for this change:

function testPriceFeedVersionIsAccurate() public {

if (block.chainid == 11155111) {

uint256 version = fundMe.getVersion();

assertEq(version, 4);

} else if (block.chainid == 1) {

uint256 version = fundMe.getVersion();

assertEq(version, 6);

}

}