**Fund subscription**

**Funding a subscription programmatically**

In the previous lessons, we learned how to create a subscription using both the Chainlink UI and programmatically. Let's see how we can fund the subscription programmatically.

This is what the subscription creation snippet from DeployRaffle looks like:

if (subscriptionId == 0) {

CreateSubscription createSubscription = new CreateSubscription();

(subscriptionId, vrfCoordinator) = createSubscription.createSubscription(vrfCoordinator);

}

Below the subscriptionId line, we need to continue with the funding logic.

For that let's open the Interactions.s.sol and below the existing contract create another contract called FundSubscription:

contract FundSubscription is Script, CodeConstants {

uint256 public constant FUND\_AMOUNT = 3 ether;

function fundSubscriptionUsingConfig() public {

}

function run() external {

fundSubscriptionUsingConfig();

}

}

I know this step looks very similar to what we did in the subscription creation lesson. That is completely fine and desirable!

One thing we need and we currently don't have configured is the LINK token. If you remember in the previous lesson, we funded our subscription with LINK, and we need to do the same thing here.

What do we need:

1. Sepolia testnet has already a LINK contract deployed, we need to have that address easily accessible inside our HelperConfiguration. To always make sure you get the correct LINK contract access the following [link](https://docs.chain.link/resources/link-token-contracts?parent=vrf).
2. Anvil doesn't come with a LINK contract deployed. We need to deploy a mock LINK token contract and use it to fund our subscription.

Let's start modifying our HelperConfig.s.sol:

struct NetworkConfig {

uint256 entranceFee;

uint256 interval;

address vrfCoordinator;

bytes32 gasLane;

uint256 subscriptionId;

uint32 callbackGasLimit;

address linkToken;

}

[...]

function getSepoliaEthConfig()

public

view

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

link: 0x779877A7B0D9E8603169DdbD7836e478b4624789

});

}

We've added the LINK address in the NetworkConfig struct and hardcoded it in the getSepoliaEthConfig function. This modification also requires some adjustments in the Interactions.s.sol:

Now the fun part! Patrick conveniently provided us with a mock LINK token contract. You can access it [here](https://github.com/Cyfrin/foundry-smart-contract-lottery-cu/blob/efac6e2a5c2df6d1936d117f40f93575d25cf694/test/mocks/LinkToken.sol).

Inside the test folder, create a new folder called mocks. Inside that, create a new file called LinkToken.sol. Copy Patrick's contract in the new file. Looking through it, we can see that it imports ERC20 from a library called Solmate which self-describes itself as a Modern, opinionated, and gas optimized building blocks for smart contract development. We need to install it with the following command:

forge install transmissions11/solmate --no-commit

Add the following line inside your remappings.txt:

@solmate/=lib/solmate/src

Back in our HelperConfig.s.sol we need to import the LinkToken:

import {LinkToken} from "test/mocks/LinkToken.sol";

And now, with this new import, we can deploy the token in case we use Anvil like so:

function getOrCreateAnvilEthConfig() internal returns (NetworkConfig memory) {

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

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

return localNetworkConfig;

}

// Deploy mocks, etc

vm.startBroadcast();

VRFCoordinatorV2\_5Mock vrfCoordinatorMock = new VRFCoordinatorV2\_5Mock(

MOCK\_BASE\_FEE,

MOCK\_GAS\_PRICE\_LINK,

MOCK\_WEI\_PER\_UNIT\_LINK

);

LinkToken linkToken = new LinkToken();

vm.stopBroadcast();

localNetworkConfig = NetworkConfig({

entranceFee: 0.01 ether,

interval: 30, // 30 seconds

vrfCoordinator: address(vrfCoordinatorMock),

// Doesn't matter for the gasLane value

gasLane: 0x787d74caea10b2b357790d5b5247c2f63d1d91572a9846f780606e4d953677ae,

subscriptionId: 0,

callbackGasLimit: 500\_000,

linkToken: address(linkToken)

});

return localNetworkConfig;

}

Amazing work!

Now we need to look through our code and make sure we have enough fields everywhere we use the NetworkConfig struct, which increased from 6 fields to 7 fields because we've added the link address.

Most people don't remember all the places, and that's alright. Run forge build.

It should look something like this:

[⠒] Solc 0.8.24 finished in 1.34s

Error:

Compiler run failed:

Error (7364): Different number of components on the left hand side (6) than on the right hand side (7).

--> script/DeployRaffle.s.sol:12:9:

|

12 | (

| ^ (Relevant source part starts here and spans across multiple lines).

Error (7407): Type tuple(uint256,uint256,address,bytes32,uint64,uint32,address) is not implicitly convertible to expected type tuple(uint256,uint256,address,bytes32,uint64,uint32).

--> test/unit/RaffleTest.t.sol:42:13:

|

42 | ) = helperConfig.localNetworkConfig();

| ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Even if this looks scary, it tells you where you need to perform the changes.

Control + Click the paths (script/DeployRaffle.s.sol:12:9:) to go to the broken code and fix it by making sure it takes the newly added address linkToken parameter.

Inside the Raffle.t.sol make sure to define the address linkToken in the state variables section. Then add it in here as well:

HelperConfig.NetworkConfig memory config = helperConfig.getConfig();

entranceFee = config.entranceFee;

interval = config.interval;

vrfCoordinator = config.vrfCoordinator;

gasLane = config.gasLane;

subscriptionId = config.subscriptionId;

callbackGasLimit = config.callbackGasLimit;

linkToken = LinkToken(config.linkToken);

And then, import our mock Link token contract:

import {LinkToken} from "test/mocks/LinkToken.sol";

Take care of both the places where we call HelperConfig() to set our config inside Interactions.s.sol:

function fundSubscriptionUsingConfig() public {

HelperConfig helperConfig = new HelperConfig();

address vrfCoordinator = helperConfig.getConfig().vrfCoordinator;

uint256 subscriptionId = helperConfig.getConfig().subscriptionId;

address linkToken = helperConfig.getConfig().linkToken;

fundSubscription(vrfCoordinator, subscriptionId, linkToken);

}

function fundSubscription(address vrfCoordinator, uint256 subscriptionId, address linkToken) public {

console.log("Funding subscription: ", subscriptionId);

console.log("Using vrfCoordinator: ", vrfCoordinator);

console.log("On chainId: ", block.chainid);

if(block.chainid == ETH\_ANVIL\_CHAIN\_ID) {

vm.startBroadcast();

VRFCoordinatorV2\_5Mock(vrfCoordinator).fundSubscription(subscriptionId, FUND\_AMOUNT);

vm.stopBroadcast();

} else {

console.log(LinkToken(linkToken).balanceOf(msg.sender));

console.log(msg.sender);

console.log(LinkToken(linkToken).balanceOf(address(this)));

console.log(address(this));

vm.startBroadcast();

LinkToken(linkToken).transferAndCall(vrfCoordinator, FUND\_AMOUNT, abi.encode(subscriptionId));

vm.stopBroadcast();

}

}

Try another forge build. This time it compiled on my side, but if it didn't compile on your side just keep control clicking through the errors and fixing them. If you get stuck please come on Cyfrin Discord in the Updraft section and ask for help.

Great! Now our script uses the right LINK address when we work on Sepolia, and deploys a new LinkToken when we work on Anvil.

Let's come back to Interactions.s.sol and finish our FundSubscription contract:

contract FundSubscription is Script, CodeConstants {

uint256 public constant FUND\_AMOUNT = 3 ether;

function fundSubscriptionUsingConfig() public {

HelperConfig helperConfig = new HelperConfig();

address vrfCoordinator = helperConfig.getConfig().vrfCoordinator;

uint256 subscriptionId = helperConfig.getConfig().subscriptionId;

address linkToken = helperConfig.getConfig().linkToken;

if (subscriptionId == 0) {

CreateSubscription createSub = new CreateSubscription();

(uint256 updatedSubId, address updatedVRFv2) = createSub.run();

subscriptionId = updatedSubId;

vrfCoordinator = updatedVRFv2;

console.log("New SubId Created! ", subscriptionId, "VRF Address: ", vrfCoordinator);

}

fundSubscription(vrfCoordinator, subscriptionId, linkToken);

}

function fundSubscription(address vrfCoordinator, uint256 subscriptionId, address linkToken) public {

console.log("Funding subscription: ", subscriptionId);

console.log("Using vrfCoordinator: ", vrfCoordinator);

console.log("On chainId: ", block.chainid);

if(block.chainid == ETH\_ANVIL\_CHAIN\_ID) {

vm.startBroadcast();

VRFCoordinatorV2\_5Mock(vrfCoordinator).fundSubscription(subscriptionId, FUND\_AMOUNT);

vm.stopBroadcast();

} else {

console.log(LinkToken(linkToken).balanceOf(msg.sender));

console.log(msg.sender);

console.log(LinkToken(linkToken).balanceOf(address(this)));

console.log(address(this));

vm.startBroadcast();

LinkToken(linkToken).transferAndCall(vrfCoordinator, FUND\_AMOUNT, abi.encode(subscriptionId));

vm.stopBroadcast();

}

}

function run() public {

fundSubscriptionUsingConfig();

}

}

This seems like a lot, but it isn't, let's go through it step by step:

* Like any other Script our's has a run function that gets executed
* Inside we call the fundSubscriptionUsingConfig function
* Inside the fundSubscriptionUsingConfig function we get the activeNetworkConfig that provides the chain-appropriate vrfCoordinator, subscriptionId and link token address
* At the end of fundSubscriptionUsingConfig we call the fundSubscription, a function that we are going to define
* We define fundSubscription as a public function that takes the 3 parameters as input
* We console log some details, this will help us debug down the road
* Then using an if statement we check if we are using Anvil, if that's the case we'll use the fundSubscription method found inside the VRFCoordinatorV2\_5Mock
* If we are not using Anvil, it means we are using Sepolia. The way we fund the Sepolia vrfCoordinator is by using the LINK's transferAndCall function.

**Note:** The transferAndCall function is part of the ERC-677 standard, which extends the ERC-20 token standard by adding the ability to execute a function call in the recipient contract immediately after transferring tokens. This feature is particularly useful in scenarios where you want to atomically transfer tokens and trigger logic in the receiving contract within a single transaction, enhancing efficiency and reducing the risk of reentrancy attacks. In the context of Chainlink, the LINK token implements the transferAndCall function. When a smart contract wants to request data from a Chainlink oracle, it uses this function to send LINK tokens to the oracle's contract address while simultaneously encoding the request details in the \_data parameter. The oracle's contract then decodes this data to understand what service is being requested.

Don't worry! You'll get enough opportunities to understand these on the way to becoming the greatest Solidity dev/auditor!

For now, let's run a forge build. Everything compiles, great!

Take a break and continue watching Patrick running the newly created script to fund the subscription he created via the UI in the past lesson.