**Subscribing to events**

**Creating a subscription**

Picking up from where we left in the previous lesson. Our test failed pointing to some error named InvalidConsumer(). Let's rerun the test with verbosity to see where is the problem:

forge test --mt testDontAllowPlayersToEnterWhileRaffleIsCalculating -vvvvv

At the end, we see this:

├─ [31556] Raffle::performUpkeep(0x)

│ ├─ [5271] VRFCoordinatorV2Mock::requestRandomWords(0x474e34a077df58807dbe9c96d3c009b23b3c6d0cce433e59bbf5b34f823bc56c, 0, 3, 500000 [5e5], 1)

│ │ └─ ← [Revert] InvalidConsumer()

│ └─ ← [Revert] InvalidConsumer()

└─ ← [Revert] InvalidConsumer()

It looks like when we call performUpkeep it internally calls requestRandomWords, and somewhere inside we hit an error.

Go to HelperConfig.s.sol and try to follow the path of the VRFCoordinatorV2Mock. Inside we can see why our function failed:

modifier onlyValidConsumer(uint64 \_subId, address \_consumer) {

if (!consumerIsAdded(\_subId, \_consumer)) {

revert InvalidConsumer();

}

\_;

}

This modifier checks if our consumer is added to the subscriptionId we've provided. We didn't do that and that's why it fails.

If you remember, we did this using the Chainlink UI in [Lesson 6](https://updraft.cyfrin.io/courses/foundry/smart-contract-lottery/solidity-random-number-chainlink-vrf). But we are developers, we need to do this programmatically.

We need to update the deployment script to make sure we can run the failing test.

Open DeployRaffle.s.sol.

The first order of business is to ensure we have a valid subscriptionId. If we have one, our test should pick it up, if we don't have one then we should create one.

Inside the script folder create a new file called Interactions.sol. This is where we'll take care of the subscription creation.

Let's start with the basics:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.19;

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

contract CreateSubscription is Script {

}

Every script needs a run function. Inside the run function we will call the createSubscriptionUsingConfig.

function createSubscriptionUsingConfig() public returns (uint64) {

}

function run() external returns (uint64) {

return createSubscriptionUsingConfig();

}

Let's pause and talk about what we are doing and what we need to make things happen. Thinking back about what we did in Lesson 6. We created a subscription, we added a consumer and we funded the subscription. Open the VRFCoordinatorV2Mock and let's look for functions that we need to do it programmatically:

function createSubscription() external override returns (uint64 \_subId) {

s\_currentSubId++;

s\_subscriptions[s\_currentSubId] = Subscription({owner: msg.sender, balance: 0});

emit SubscriptionCreated(s\_currentSubId, msg.sender);

return s\_currentSubId;

}

[...]

function addConsumer(uint64 \_subId, address \_consumer) external override onlySubOwner(\_subId) {

if (s\_consumers[\_subId].length == MAX\_CONSUMERS) {

revert TooManyConsumers();

}

if (consumerIsAdded(\_subId, \_consumer)) {

return;

}

s\_consumers[\_subId].push(\_consumer);

emit ConsumerAdded(\_subId, \_consumer);

}

[...]

function fundSubscription(uint64 \_subId, uint96 \_amount) public {

if (s\_subscriptions[\_subId].owner == address(0)) {

revert InvalidSubscription();

}

uint96 oldBalance = s\_subscriptions[\_subId].balance;

s\_subscriptions[\_subId].balance += \_amount;

emit SubscriptionFunded(\_subId, oldBalance, oldBalance + \_amount);

}

Great! Now we need to call all of them, but before that, we first need to pull the VRFv2 address, available in the HelperConfig.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.19;

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

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

contract CreateSubscription is Script {

function createSubscriptionUsingConfig() public returns (uint64) {

HelperConfig helperConfig = new HelperConfig();

(

,

,

address vrfCoordinator,

,

,

,

) = helperConfig.activeNetworkConfig();

return createSubscription(vrfCoordinator);

}

function createSubscription(

address vrfCoordinator

) public returns (uint64) {}

function run() external returns (uint64) {

return createSubscriptionUsingConfig();

}

}

As we said above, we created a run function that calls createSubscriptionUsingConfig. This function deploys the HelperConfig to grab the vrfCoordinator and inside the return statement, we call the createSubscription function. For that to work, we need to define the createSubscription function, which takes the vrfCoordinator address as an input. This is where we create the actual subscription.

Amazing! Let's work on the createSubscription function. We need to import some things to make it work. First, let's update the contract in order to import console, to log a message every time we create a subscription. Second, let's import the VRFCoordinatorV2Mock to be able to call the functions we specified above.

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

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

import {VRFCoordinatorV2Mock} from "chainlink/src/v0.8/vrf/mocks/VRFCoordinatorV2Mock.sol";

Perfect, not let's finish the createSubscription:

function createSubscription(

address vrfCoordinator

) public returns (uint64) {

console.log("Creating subscription on ChainID: ", block.chainid);

vm.startBroadcast();

uint64 subId = VRFCoordinatorV2Mock(vrfCoordinator).createSubscription();

vm.stopBroadcast();

console.log("Your sub Id is: ", subId);

console.log("Please update subscriptionId in HelperConfig!");

return subId;

}

First, we log the Creating subscription message. Then, we encapsulate the VRFCoordinatorV2Mock(vrfCoordinator).createSubscription(); call inside the vm.startBroadcast and vm.stopBroadcast block. We assign the return of the VRFCoordinatorV2Mock(vrfCoordinator).createSubscription call to uint64 subId variable. Then we log the subId and return it to end the function.

Amazing work! Coming back to DeployRaffle.s.sol, we should create a subscription if we don't have one, like this:

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

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

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

import {CreateSubscription} from "./Interactions.s.sol";

contract DeployRaffle is Script {

function run() external returns (Raffle, HelperConfig) {

HelperConfig helperConfig = new HelperConfig(); // This comes with our mocks!

(

uint256 entranceFee,

uint256 interval,

address vrfCoordinator,

bytes32 gasLane,

uint256 subscriptionId,

uint32 callbackGasLimit

) = helperConfig.activeNetworkConfig();

if (subscriptionId == 0) {

CreateSubscription createSubscription = new CreateSubscription();

subscriptionId = createSubscription.createSubscription(vrfCoordinator);

}

vm.startBroadcast();

Raffle raffle = new Raffle(

entranceFee,

interval,

vrfCoordinator,

gasLane,

subscriptionId,

callbackGasLimit

);

vm.stopBroadcast();

return (raffle, helperConfig);

}

}

We import the newly created CreateSubscription contract from Interactions.s.sol. After the helperConfig definition, we check if our subscriptionId is 0. If that yields true then we don't have a subscriptionId and we need to create one. We use the new functions inside the CreateSubscription to get an appropriate subscriptionId.

Amazing work!