**Forked test environment and dynamic private keys**

**Testing on a forked chain**

❗ **IMPORTANT** Up until now we've followed Patrick's video in 99.9% of the cases. But between Patrick filming the video lessons and the writing of this lesson Chainlink VRF got upgraded to v2.5. So with the code we have right now, we won't be able to test it on a fork. One of the main changes is the type of subscriptionId. If you followed the lesson when we created a new subscription using Chainlink's UI you saw that your code looks more like this 56337043680668238338308639953697831315254325227567930909387210179785852470990 while Patrick's looks like this 2924, this is because the new subscriptionId is uint256 and not uint64. Even if this looks like a small change, it has a ton of ramifications throughout the Chainlink contracts. You can read more about the migration from v2.0 to v2.5 [here](https://docs.chain.link/vrf/v2-5/migration-from-v2).

That being said:

1. Please rename your current Raffle.sol into Raffle\_Old.sol. Create a new file called Raffle.sol and paste the following contract inside:

// SPDX-License-Identifier: UNLICENSED

pragma solidity ^0.8.19;

import {IVRFCoordinatorV2Plus} from "chainlink/src/v0.8/vrf/dev/interfaces/IVRFCoordinatorV2Plus.sol";

import {VRFV2PlusClient} from "chainlink/src/v0.8/vrf/dev/libraries/VRFV2PlusClient.sol";

import {VRFCoordinatorV2\_5} from "chainlink/src/v0.8/vrf/dev/VRFCoordinatorV2\_5.sol";

import {VRFConsumerBaseV2Plus} from "chainlink/src/v0.8/vrf/dev/VRFConsumerBaseV2Plus.sol";

import {AutomationCompatibleInterface} from "chainlink/src/v0.8/automation/interfaces/AutomationCompatibleInterface.sol";

/\*\*

\* @title Raffle

\* @author EngrPips

\* @notice This contract if Particpating in a Raffle and standing the chance to win.

\* @dev This contract heavily implements the chainlink VRF and Automation

\*/

// 0x9ddfaca8183c41ad55329bdeed9f6a8d53168b1b => VRFCoordinator

// 54528670710849503547892655734386820566589065322714869834102560641565075666367 => subId

contract Raffle is VRFConsumerBaseV2Plus, AutomationCompatibleInterface {

/\* ERRORS \*/

error Raffle\_\_NotEnoughEthSent();

error Raffle\_\_TransferFailed();

error Raffle\_\_RaffleNotOpen();

error Raffle\_\_UpkeepNotNeeded(

uint256 currentBalance,

uint256 numPlayers,

uint256 raffleState

);

error Raffle\_\_LotteryIsCalculatingWinner();

/\* USER DEFINED TYPES \*/

enum RaffleState {

OPEN,

CALCULATING

}

/\* EVENTS \*/

event EnteredRaffle(address indexed player);

event PickedWinner(address winner);

event RequestedRaffleWinner(uint256 indexed requestId);

/\* STATE VARIABLES \*/

RaffleState private s\_raffleState;

IVRFCoordinatorV2Plus private immutable i\_vrfCoordinatorContract;

uint16 private constant REQUEST\_CONFIRMATIONS = 3;

uint32 private constant NUM\_WORDS = 1;

uint256 private immutable i\_entranceFee;

uint256 private immutable i\_interval;

uint256 private s\_lastTimeStamp;

uint256 private immutable i\_subscriptionId;

bytes32 private immutable i\_gasLane;

uint32 private immutable i\_callbackGasLimit;

address payable private s\_recentWinner;

address payable[] private s\_players;

constructor(uint256 \_entranceFee, uint256 \_interval,address \_vrfCoordinator, bytes32 \_gasLane, uint256 \_subscriptionId, uint32 callbackGasLimit) VRFConsumerBaseV2Plus( \_vrfCoordinator) {

i\_entranceFee = \_entranceFee;

i\_interval = \_interval;

s\_raffleState = RaffleState.OPEN;

s\_lastTimeStamp = block.timestamp;

i\_vrfCoordinatorContract = IVRFCoordinatorV2Plus(\_vrfCoordinator);

i\_gasLane = \_gasLane;

i\_subscriptionId = \_subscriptionId;

i\_callbackGasLimit = callbackGasLimit;

}

function enterRaffle() public payable {

if (s\_raffleState == RaffleState.CALCULATING) revert Raffle\_\_RaffleNotOpen();

if (msg.value < i\_entranceFee) revert Raffle\_\_NotEnoughEthSent();

s\_players.push(payable(msg.sender));

emit EnteredRaffle(msg.sender);

}

function checkUpkeep(bytes memory /\* checkData \*/ )

public

view

returns (bool upkeepNeeded, bytes memory /\* performData \*/ )

{

// We don't use the checkData in this example. The checkData is defined when the Upkeep was registered.

bool timeHasPassed = block.timestamp - s\_lastTimeStamp >= i\_interval;

bool hasPlayer = s\_players.length > 0;

bool hasBalance = address(this).balance > 0;

bool isOpen = s\_raffleState == RaffleState.OPEN;

upkeepNeeded = (timeHasPassed && hasPlayer && hasBalance && isOpen);

return(upkeepNeeded, "0x0");

}

function performUpkeep(bytes calldata /\* performData \*/) public {

(bool upkeepNeeded,) = checkUpkeep("");

if(!upkeepNeeded) revert Raffle\_\_UpkeepNotNeeded(address(this).balance, getNumberOfPlayers(), uint256(getRaffleState()));

s\_raffleState = RaffleState.CALCULATING;

uint256 requestId = i\_vrfCoordinatorContract.requestRandomWords(VRFV2PlusClient.RandomWordsRequest({

keyHash: i\_gasLane,

subId: i\_subscriptionId,

requestConfirmations: REQUEST\_CONFIRMATIONS,

callbackGasLimit: i\_callbackGasLimit,

numWords: NUM\_WORDS,

// Set nativePayment to true to pay for VRF requests with Sepolia ETH instead of LINK

extraArgs: VRFV2PlusClient.\_argsToBytes(VRFV2PlusClient.ExtraArgsV1({nativePayment: false}))

}));

emit RequestedRaffleWinner(requestId);

}

function fulfillRandomWords(uint256 requestId, uint256[] calldata randomWords) internal override {

uint256 indexOfWinner = randomWords[0] % s\_players.length;

address payable addressOfWinner = s\_players[indexOfWinner];

s\_recentWinner = addressOfWinner;

emit PickedWinner(addressOfWinner);

s\_players = new address payable[](0);

s\_lastTimeStamp = block.timestamp;

s\_raffleState = RaffleState.OPEN;

(bool success,) = addressOfWinner.call{value: address(this).balance}("");

if (!success) revert Raffle\_\_TransferFailed();

}

/\*\* GETTERS \*/

function getRaffleState() public view returns (RaffleState) {

return s\_raffleState;

}

function getNumWords() public pure returns (uint256) {

return NUM\_WORDS;

}

function getRequestConfirmations() public pure returns (uint256) {

return REQUEST\_CONFIRMATIONS;

}

function getRecentWinner() public view returns (address) {

return s\_recentWinner;

}

function getPlayer(uint256 index) public view returns (address) {

return s\_players[index];

}

function getLastTimeStamp() public view returns (uint256) {

return s\_lastTimeStamp;

}

function getInterval() public view returns (uint256) {

return i\_interval;

}

function getEntranceFee() public view returns (uint256) {

return i\_entranceFee;

}

function getNumberOfPlayers() public view returns (uint256) {

return s\_players.length;

}

}

You will see that the new contract is very similar. It only has some changes related to the VRF v2.5 integration, other than that the functionality is the same.

1. Please run the following command in your CLI: forge install openzeppelin/openzeppelin-contracts --no-commit
2. Add the following line to your remappings.txt: @openzeppelin/=lib/openzeppelin-contracts/
3. Open lib/chainlink/contracts/src/v0.8/vrf/mocks folder and create a new file called VRFCoordinatorV2PlusMock.sol
4. Copy the following inside the newly created file:

// SPDX-License-Identifier: MIT

// A mock for testing code that relies on VRFCoordinatorV2Plus.

pragma solidity 0.8.23;

import { VRFCoordinatorV2Interface } from "../interfaces/VRFCoordinatorV2Interface.sol";

import { VRFConsumerBaseV2Plus } from "../dev/VRFConsumerBaseV2Plus.sol";

import { ConfirmedOwner } from "../../shared/access/ConfirmedOwner.sol";

import { IVRFCoordinatorV2Plus } from "../dev/interfaces/IVRFCoordinatorV2Plus.sol";

import { VRFV2PlusClient } from "../dev/libraries/VRFV2PlusClient.sol";

// solhint-disable chainlink-solidity/prefix-immutable-variables-with-i

// solhint-disable gas-custom-errors

// solhint-disable avoid-low-level-calls

contract VRFCoordinatorV2PlusMock is ConfirmedOwner, IVRFCoordinatorV2Plus {

uint96 public immutable BASE\_FEE;

uint96 public immutable GAS\_PRICE\_LINK;

uint16 public immutable MAX\_CONSUMERS = 100;

error InvalidSubscription();

error InsufficientBalance();

error MustBeSubOwner(address owner);

error TooManyConsumers();

error InvalidConsumer();

error InvalidRandomWords();

error Reentrant();

event RandomWordsRequested(

bytes32 indexed keyHash,

uint256 requestId,

uint256 preSeed,

uint256 indexed subId,

uint16 minimumRequestConfirmations,

uint32 callbackGasLimit,

uint32 numWords,

address indexed sender

);

event RandomWordsFulfilled(uint256 indexed requestId, uint256 outputSeed, uint96 payment, bool success);

event SubscriptionCreated(uint256 indexed subId, address owner);

event SubscriptionFunded(uint256 indexed subId, uint256 oldBalance, uint256 newBalance);

event SubscriptionFundedWithNative(uint256 indexed subId, uint256 oldNativeBalance, uint256 newNativeBalance);

event SubscriptionCanceled(uint256 indexed subId, address to, uint256 amount);

event ConsumerAdded(uint256 indexed subId, address consumer);

event ConsumerRemoved(uint256 indexed subId, address consumer);

event ConfigSet();

struct Config {

// Reentrancy protection.

bool reentrancyLock;

}

Config private s\_config;

uint256 internal s\_currentSubId;

uint256 internal s\_nextRequestId = 1;

uint256 internal s\_nextPreSeed = 100;

struct Subscription {

address owner;

uint96 nativeBalance;

uint96 balance;

}

mapping(uint256 => Subscription) internal s\_subscriptions; /\* subId \*/ /\* subscription \*/

mapping(uint256 => address[]) internal s\_consumers; /\* subId \*/ /\* consumers \*/

struct Request {

uint256 subId;

uint32 callbackGasLimit;

uint32 numWords;

}

mapping(uint256 => Request) internal s\_requests; /\* requestId \*/ /\* request \*/

constructor(uint96 \_baseFee, uint96 \_gasPriceLink) ConfirmedOwner(msg.sender) {

BASE\_FEE = \_baseFee;

GAS\_PRICE\_LINK = \_gasPriceLink;

setConfig();

}

/\*\*

\* @notice Sets the configuration of the vrfv2 mock coordinator

\*/

function setConfig() public onlyOwner {

s\_config = Config({ reentrancyLock: false });

emit ConfigSet();

}

function consumerIsAdded(uint256 \_subId, address \_consumer) public view returns (bool) {

address[] memory consumers = s\_consumers[\_subId];

for (uint256 i = 0; i < consumers.length; i++) {

if (consumers[i] == \_consumer) {

return true;

}

}

return false;

}

modifier onlyValidConsumer(uint256 \_subId, address \_consumer) {

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

revert InvalidConsumer();

}

\_;

}

/\*\*

\* @notice fulfillRandomWords fulfills the given request, sending the random words to the supplied

\* @notice consumer.

\*

\* @dev This mock uses a simplified formula for calculating payment amount and gas usage, and does

\* @dev not account for all edge cases handled in the real VRF coordinator. When making requests

\* @dev against the real coordinator a small amount of additional LINK is required.

\*

\* @param \_requestId the request to fulfill

\* @param \_consumer the VRF randomness consumer to send the result to

\*/

function fulfillRandomWords(uint256 \_requestId, address \_consumer) external nonReentrant {

fulfillRandomWordsWithOverride(\_requestId, \_consumer, new uint256[](0));

}

/\*\*

\* @notice fulfillRandomWordsWithOverride allows the user to pass in their own random words.

\*

\* @param \_requestId the request to fulfill

\* @param \_consumer the VRF randomness consumer to send the result to

\* @param \_words user-provided random words

\*/

function fulfillRandomWordsWithOverride(uint256 \_requestId, address \_consumer, uint256[] memory \_words) public {

uint256 startGas = gasleft();

if (s\_requests[\_requestId].subId == 0) {

revert("nonexistent request");

}

Request memory req = s\_requests[\_requestId];

if (\_words.length == 0) {

\_words = new uint256[](req.numWords);

for (uint256 i = 0; i < req.numWords; i++) {

\_words[i] = uint256(keccak256(abi.encode(\_requestId, i)));

}

} else if (\_words.length != req.numWords) {

revert InvalidRandomWords();

}

VRFConsumerBaseV2Plus v;

bytes memory callReq = abi.encodeWithSelector(v.rawFulfillRandomWords.selector, \_requestId, \_words);

s\_config.reentrancyLock = true;

(bool success, ) = \_consumer.call{ gas: req.callbackGasLimit }(callReq);

s\_config.reentrancyLock = false;

uint96 payment = uint96(BASE\_FEE + ((startGas - gasleft()) \* GAS\_PRICE\_LINK));

if (s\_subscriptions[req.subId].balance < payment) {

revert InsufficientBalance();

}

s\_subscriptions[req.subId].balance -= payment;

delete (s\_requests[\_requestId]);

emit RandomWordsFulfilled(\_requestId, \_requestId, payment, success);

}

/\*\*

\* @notice fundSubscription allows funding a subscription with an arbitrary amount for testing.

\*

\* @param \_subId the subscription to fund

\* @param \_amount the amount to fund

\*/

function fundSubscription(uint256 \_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);

}

function fundSubscriptionWithNative(uint256 \_subId) external payable override nonReentrant {

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

revert InvalidSubscription();

}

uint96 oldNativeBalance = s\_subscriptions[\_subId].nativeBalance;

s\_subscriptions[\_subId].nativeBalance += uint96(msg.value);

emit SubscriptionFundedWithNative(\_subId, oldNativeBalance, oldNativeBalance + msg.value);

}

function requestRandomWords(

VRFV2PlusClient.RandomWordsRequest calldata req

) external override nonReentrant onlyValidConsumer(req.subId, msg.sender) returns (uint256) {

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

revert InvalidSubscription();

}

uint256 requestId = s\_nextRequestId++;

uint256 preSeed = s\_nextPreSeed++;

s\_requests[requestId] = Request({

subId: req.subId,

callbackGasLimit: req.callbackGasLimit,

numWords: req.numWords

});

emit RandomWordsRequested(

req.keyHash,

requestId,

preSeed,

req.subId,

req.requestConfirmations,

req.callbackGasLimit,

req.numWords,

msg.sender

);

return requestId;

}

function createSubscription() external override returns (uint256) {

s\_currentSubId++;

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

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

return s\_currentSubId;

}

function getSubscription(

uint256 subId

)

external

view

override

returns (uint96 balance, uint96 nativeBalance, uint64 reqCount, address owner, address[] memory consumers)

{

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

revert InvalidSubscription();

}

return (

s\_subscriptions[subId].balance,

s\_subscriptions[subId].nativeBalance,

0,

s\_subscriptions[subId].owner,

s\_consumers[subId]

);

}

function cancelSubscription(uint256 \_subId, address \_to) external override onlySubOwner(\_subId) nonReentrant {

emit SubscriptionCanceled(\_subId, \_to, s\_subscriptions[\_subId].balance);

delete (s\_subscriptions[\_subId]);

}

modifier onlySubOwner(uint256 \_subId) {

address owner = s\_subscriptions[\_subId].owner;

if (owner == address(0)) {

revert InvalidSubscription();

}

if (msg.sender != owner) {

revert MustBeSubOwner(owner);

}

\_;

}

function getRequestConfig() external pure returns (uint16, uint32, bytes32[] memory) {

return (3, 2000000, new bytes32[](0));

}

function addConsumer(uint256 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 removeConsumer(

uint256 \_subId,

address \_consumer

) external override onlySubOwner(\_subId) onlyValidConsumer(\_subId, \_consumer) nonReentrant {

address[] storage consumers = s\_consumers[\_subId];

for (uint256 i = 0; i < consumers.length; i++) {

if (consumers[i] == \_consumer) {

address last = consumers[consumers.length - 1];

consumers[i] = last;

consumers.pop();

break;

}

}

emit ConsumerRemoved(\_subId, \_consumer);

}

function getConfig()

external

pure

returns (

uint16 minimumRequestConfirmations,

uint32 maxGasLimit,

uint32 stalenessSeconds,

uint32 gasAfterPaymentCalculation

)

{

return (4, 2\_500\_000, 2\_700, 33285);

}

function getFeeConfig()

external

pure

returns (

uint32 fulfillmentFlatFeeLinkPPMTier1,

uint32 fulfillmentFlatFeeLinkPPMTier2,

uint32 fulfillmentFlatFeeLinkPPMTier3,

uint32 fulfillmentFlatFeeLinkPPMTier4,

uint32 fulfillmentFlatFeeLinkPPMTier5,

uint24 reqsForTier2,

uint24 reqsForTier3,

uint24 reqsForTier4,

uint24 reqsForTier5

)

{

return (

100000, // 0.1 LINK

100000, // 0.1 LINK

100000, // 0.1 LINK

100000, // 0.1 LINK

100000, // 0.1 LINK

0,

0,

0,

0

);

}

modifier nonReentrant() {

if (s\_config.reentrancyLock) {

revert Reentrant();

}

\_;

}

function getFallbackWeiPerUnitLink() external pure returns (int256) {

return 4000000000000000; // 0.004 Ether

}

function requestSubscriptionOwnerTransfer(uint256 /\*\_subId\*/, address /\*\_newOwner\*/) external pure override {

revert("not implemented");

}

function acceptSubscriptionOwnerTransfer(uint256 /\*\_subId\*/) external pure override {

revert("not implemented");

}

function pendingRequestExists(uint256 /\*subId\*/) public pure override returns (bool) {

revert("not implemented");

}

function getActiveSubscriptionIds(

uint256 /\* startIndex \*/,

uint256 /\* maxCount \*/

) external pure override returns (uint256[] memory) {

revert("not implemented");

}

}

This contract will act as a replacement for VRFCoordinatorMock.

If you try to build the project now you'll most likely face a plethora of errors. Let's make some changes to minimize them.

Please ensure that all the scripts, tests and main contracts use pragma solidity ^0.8.19;. Make sure to change the pragma of the following files lib/chainlink/contracts/src/v0.8/vrf/dev/VRFCoordinatorV2\_5.sol and lib/chainlink/contracts/src/v0.8/vrf/dev/SubscriptionAPI.sol to pragma solidity ^0.8.19;. You shouldn't do this in any other cases. We do it for tutorial purposes. Inside lib/chainlink/contracts/src/v0.8/vendor/@eth-optimism/contracts/v0.8.9/contracts/L2/predeploys/OVM\_GasPriceOracle.sol modify the Ownable() call from the constructor into Ownable(\_owner)

In HelperConfig.s.sol delete the existing import for VRFCoordinatorV2Mock and replace it with the following import {VRFCoordinatorV2PlusMock} from "chainlink/src/v0.8/vrf/mocks/VRFCoordinatorV2PlusMock.sol";. Change the VRF Mock deployment line in getOrCreateAnvilEthConfig from VRFCoordinatorV2Mock vrfCoordinatorV2Mock = new VRFCoordinatorV2Mock(baseFee, gasPriceLink); to VRFCoordinatorV2PlusMock vrfCoordinatorV2Mock = new VRFCoordinatorV2PlusMock(baseFee, gasPriceLink);

In Interactions.s.sol delete the existing import for VRFCoordinatorV2Mock and replace it with the following import {VRFCoordinatorV2PlusMock} from "chainlink/src/v0.8/vrf/mocks/VRFCoordinatorV2PlusMock.sol";. For all 3 existing contracts, create and fund subscriptions and add consumer perform a call to a function from VRFCoordinatorV2Mock, make sure to replace VRFCoordinatorV2Mock with VRFCoordinatorV2PlusMock.

Make sure to add your subscriptionId, which you got by following the tutorial related to obtaining it using the Chainlink UI, inside the HelperConfig::getSepoliaEthConfig function.

I know this was a lot, but it was 100% necessary. Moreover, this is what a dev's life looks like. It's not all about solving complex problems using simple building blocks in a smart way ... sometimes it's fixing broken dependencies, none of us like it, but all of us do it. **You are a true hero for reaching this point!**

Let's finish all with a forge test.

If it looks like this you made it!

/mnt/c/Users/balas/Desktop/lessons/foundry-f23/foundry-smart-contract-lottery-f23$ forge test

[⠢] Compiling...

No files changed, compilation skipped

Ran 12 tests for test/unit/RaffleTest.t.sol:RaffleTest

[PASS] testCheckUpkeepReturnsFalseIfItHasNoBalance() (gas: 19096)

[PASS] testCheckUpkeepReturnsFalseIfRaffleIsntOpen() (gas: 157033)

[PASS] testDontAllowPlayersToEnterWhileRaffleIsCalculating() (gas: 161916)

[PASS] testEmitsEventOnEntrance() (gas: 68752)

[PASS] testFulfillRandomWordsCanOnlyBeCalledAfterPerformUpkeep(uint256) (runs: 257, μ: 80568, ~: 80568)

[PASS] testFulfillRandomWordsPicksAWinnerRestesAndSendsMoney() (gas: 314207)

[PASS] testPerformUpkeepCanOnlyRunIfCheckUpkeepIsTrue() (gas: 153220)

[PASS] testPerformUpkeepRevertsIfCheckUpkeepIsFalse() (gas: 17399)

[PASS] testPerformUpkeepUpdatesRaffleStateAndEmitsRequestId() (gas: 158980)

[PASS] testRaffleInitializesInOpenState() (gas: 7753)

[PASS] testRaffleRecordsPlayerWhenTheyEnter() (gas: 68318)

[PASS] testRaffleRevertsWHenYouDontPayEnough() (gas: 13069)

Suite result: ok. 12 passed; 0 failed; 0 skipped; finished in 33.75ms (23.88ms CPU time)

Ran 1 test suite in 3.75s (33.75ms CPU time): 12 tests passed, 0 failed, 0 skipped (12 total tests)

I wonder if this works on a fork. Will it?

forge test --fork-url $SEPOLIA\_RPC\_URL -vvvv

It failed on setup:

│ │ ├─ [0] VM::startBroadcast()

│ │ │ └─ ← [Return]

│ │ ├─ [236] VRFCoordinatorV2::addConsumer(5000032745829988966686682423284879867102409618787289144283231874950241281744 [5e75], Raffle: [0x90193C961A926261B756D1E5bb255e67ff9498A1])

│ │ │ └─ ← [Revert] EvmError: Revert

│ │ └─ ← [Revert] EvmError: Revert

│ └─ ← [Revert] EvmError: Revert

└─ ← [Revert] EvmError: Revert

It failed on addConsumer. Given that we use our subscriptionId, which we created using our test address using the Chainlink UI, why wouldn't it fail? When we do the setup we don't specify the key we used when we created the subscription, so an external account, provided by default by foundry, tries to add a consumer to our subscriptionId, which should fail, or else everyone could just add random consumers to anyone else's subscriptionIds, and use their funds. That would be a serious security breach.

We need to make sure that we add a consumer using the same private key (account) we used when we created the subscription.

You could say, ok, I get it, let's hardcode the subscriptionId inside the HelperConfig::getSepoliaEthConfig function to 0, so our script create a new subscription. That is a very smart thing to say, but if you do that you'll get this outcome:

│ ├─ [9690] FundSubscription::fundSubscription(VRFCoordinatorV2: [0x8103B0A8A00be2DDC778e6e7eaa21791Cd364625], 11932 [1.193e4], LinkToken: [0x779877A7B0D9E8603169DdbD7836e478b4624789])

│ │ ├─ [0] console::log("Funding subscription: ", 11932 [1.193e4]) [staticcall]

│ │ │ └─ ← [Stop]

│ │ ├─ [0] console::log("Using vrfCoordinator: ", VRFCoordinatorV2: [0x8103B0A8A00be2DDC778e6e7eaa21791Cd364625]) [staticcall]

│ │ │ └─ ← [Stop]

│ │ ├─ [0] console::log("On ChainID: ", 11155111 [1.115e7]) [staticcall]

│ │ │ └─ ← [Stop]

│ │ ├─ [0] VM::startBroadcast()

│ │ │ └─ ← [Return]

│ │ ├─ [3558] LinkToken::transferAndCall(VRFCoordinatorV2: [0x8103B0A8A00be2DDC778e6e7eaa21791Cd364625], 3000000000000000000 [3e18], 0x0000000000000000000000000000000000000000000000000000000000002e9c)

│ │ │ └─ ← [Revert] revert: ERC20: transfer amount exceeds balance

│ │ └─ ← [Revert] revert: ERC20: transfer amount exceeds balance

│ └─ ← [Revert] revert: ERC20: transfer amount exceeds balance

└─ ← [Revert] revert: ERC20: transfer amount exceeds balance

The address used by Foundry doesn't have any test LINK. So even if you create it, you can't finance that subscription.

But no worries, we got the right solution for this.

Let's take these problems one by one. We tried to add a consumer, but it failed because we didn't use the right key, let's fix this:

1. Whenever we use vm.startBroadcast() we could specify the key as indicated [here](https://book.getfoundry.sh/cheatcodes/start-broadcast);
2. Let's do this the smart way, let's define the key inside our HelperConfig and then use it everywhere levraging our deployment script;
3. If we are using Sepolia, we should use our Sepolia test key, if we are on Anvil then we should use one of the keys Anvil provides;
4. Putting our key, even if it's a test key, in plain sight it's something that we agreed we'd never do, so we will use a new cheatcode vm.envUint("SEPOLIA\_PRIVATE\_KEY"), which grabs our private key from our .env. Read more about the cheatcode [here](https://book.getfoundry.sh/cheatcodes/env-uint);

Open your HelperConfig.s.sol and perform the following changes:

function getSepoliaEthConfig()

public

view

returns (NetworkConfig memory)

{

return NetworkConfig({

entranceFee: 0.01 ether,

interval: 30, // 30 seconds

vrfCoordinator: 0x9DdfaCa8183c41ad55329BdeeD9F6A8d53168B1B,

gasLane: 0x787d74caea10b2b357790d5b5247c2f63d1d91572a9846f780606e4d953677ae,

subscriptionId: 5000032745829988966686682423284879867102409618787289144283231874950241281744, // Your own subscriptionId goes here

callbackGasLimit: 500000, // 500,000 gas

link: 0x779877A7B0D9E8603169DdbD7836e478b4624789,

deployerKey: vm.envUint("SEPOLIA\_PRIVATE\_KEY")

});

}

contract HelperConfig is Script {

struct NetworkConfig {

uint256 entranceFee;

uint256 interval;

address vrfCoordinator;

bytes32 gasLane;

uint256 subscriptionId;

uint32 callbackGasLimit;

address link;

uint256 deployerKey;

}

}

Given that we've updated the struct above ... we are going to have to update a lot of places. Let's keep going inside HelperConfig.

We've updated Sepolia, we need to take care of Anvil. Run a quick anvil copy Key 0 from here:

Private Keys

==================

(0) 0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80

(1) 0x59c6995e998f97a5a0044966f0945389dc9e86dae88c7a8412f4603b6b78690d

(2) 0x5de4111afa1a4b94908f83103eb1f1706367c2e68ca870fc3fb9a804cdab365a

[...]

Below the struct NetworkConfig create a new variable:

uint256 public constant DEFAULT\_ANVIL\_KEY = 0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80;

and use it inside the getOrCreateAnvilEthConfig function

function getOrCreateAnvilEthConfig()

public

returns (NetworkConfig memory anvilNetworkConfig)

{

[...]

LinkToken link = new LinkToken();

vm.stopBroadcast();

return NetworkConfig({

entranceFee: 0.01 ether,

interval: 30, // 30 seconds

vrfCoordinator: address(vrfCoordinatorV2Mock),

gasLane: 0x787d74caea10b2b357790d5b5247c2f63d1d91572a9846f780606e4d953677ae,

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

callbackGasLimit: 500000, // 500,000 gas

link: address(link),

deployerKey: DEFAULT\_ANVIL\_KEY

});

}

With this change, we have finished the work in HelperConfig.s.sol. Let's keep going with fixing Interactions.s.sol:

1. AddConsumer contract

contract AddConsumer is Script {

function addConsumer(address raffle, address vrfCoordinator, uint256 subscriptionId, uint256 deployerKey) public {

console.log("Adding consumer contract: ", raffle);

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

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

vm.startBroadcast(deployerKey);

VRFCoordinatorV2PlusMock(vrfCoordinator).addConsumer(subscriptionId, raffle);

vm.stopBroadcast();

}

function addConsumerUsingConfig(address raffle) public {

HelperConfig helperConfig = new HelperConfig();

(

,

,

address vrfCoordinator,

,

uint256 subscriptionId,

,

,

uint256 deployerKey

) = helperConfig.activeNetworkConfig();

addConsumer(raffle, vrfCoordinator, subscriptionId, deployerKey);

}

function run() external {

address raffle = DevOpsTools.get\_most\_recent\_deployment("MyContract", block.chainid);

addConsumerUsingConfig(raffle);

}

}

contract AddConsumer is Script {

function addConsumer(address raffle, address vrfCoordinator, uint256 subscriptionId, uint256 deployerKey) public {

console.log("Adding consumer contract: ", raffle);

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

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

vm.startBroadcast(deployerKey);

VRFCoordinatorV2PlusMock(vrfCoordinator).addConsumer(subscriptionId, raffle);

vm.stopBroadcast();

}

function addConsumerUsingConfig(address raffle) public {

HelperConfig helperConfig = new HelperConfig();

(

,

,

address vrfCoordinator,

,

uint256 subscriptionId,

,

,

uint256 deployerKey

) = helperConfig.activeNetworkConfig();

addConsumer(raffle, vrfCoordinator, subscriptionId, deployerKey);

}

function run() external {

address raffle = DevOpsTools.get\_most\_recent\_deployment("MyContract", block.chainid);

addConsumerUsingConfig(raffle);

}

}

Let's understand the logic behind the change and then change it in the other two contracts FundSubscription and CreateSubscription.

We start everything from helperConfig.activeNetworkConfig because the main change we want to bake in is the NetworkConfig change we did in HelperConfig. We get the deployerKey from the helperConfig.activeNetworkConfig(); call. Then we use it in the addConsumer function, providing it as an input. Scrolling up to the addConsumer function, we need to define the 4th input function addConsumer(address raffle, address vrfCoordinator, uint256 subscriptionId, uint256 deployerKey). Now that we have access to the deployerKey we can provide it in vm.startBroadcast(deployerKey);. So that addConsumer function will be called by the right account.

We will perform the same changes to the other two contracts:

contract FundSubscription is Script {

uint96 public constant FUND\_AMOUNT = 3 ether;

function fundSubscriptionUsingConfig() public {

HelperConfig helperConfig = new HelperConfig();

(

,

,

address vrfCoordinator,

,

uint256 subscriptionId,

,

address link,

uint256 deployerKey

) = helperConfig.activeNetworkConfig();

fundSubscription(vrfCoordinator, subscriptionId, link, deployerKey);

}

function fundSubscription(address vrfCoordinator, uint256 subscriptionId, address link, uint256 deployerKey) public {

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

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

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

if (block.chainid == 31337) {

vm.startBroadcast(deployerKey);

VRFCoordinatorV2PlusMock(vrfCoordinator).fundSubscription(subscriptionId, FUND\_AMOUNT);

vm.stopBroadcast();

} else {

vm.startBroadcast(deployerKey);

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

vm.stopBroadcast();

}

}

function run() external {

fundSubscriptionUsingConfig();

}

}

We start with fundSubscriptionUsingConfig we add the deployerKey variable in the activeNetworkConfig call line. We use that newly acquired deployerKey inside the fundSubscription call, providing it as input. Going to fundSubscription we add the 4th input variable uint256 deployerKey. We use the new input in both vm.startBroadcast places.

contract CreateSubscription is Script {

function createSubscriptionUsingConfig() public returns (uint256) {

HelperConfig helperConfig = new HelperConfig();

(

,

,

address vrfCoordinator,

,

,

,

,

uint256 deployerKey

) = helperConfig.activeNetworkConfig();

return createSubscription(vrfCoordinator, deployerKey);

}

function createSubscription(

address vrfCoordinator,

uint256 deployerKey

) public returns (uint256) {

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

vm.startBroadcast(deployerKey);

uint256 subId = VRFCoordinatorV2PlusMock(vrfCoordinator).createSubscription();

vm.stopBroadcast();

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

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

return subId;

}

function run() external returns (uint256) {

return createSubscriptionUsingConfig();

}

}

The exact same thing for CreateSubscription.

By the way, you should never be afraid of running a forge build regardless of the stage of your project, nothing can happen and you can get free hints regarding what you need to do next. Run forge build:

[⠢] Compiling...

[⠊] Compiling 4 files with Solc 0.8.23

[⠒] Solc 0.8.23 finished in 684.48ms

Error:

Compiler run failed:

Error (6160): Wrong argument count for function call: 1 arguments given but expected 2.

--> script/DeployRaffle.s.sol:25:30:

|

25 | subscriptionId = createSubscription.createSubscription(vrfCoordinator);

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

Error (6160): Wrong argument count for function call: 3 arguments given but expected 4.

--> script/DeployRaffle.s.sol:28:13:

|

28 | fundSubscription.fundSubscription(vrfCoordinator, subscriptionId, link);

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

Error (6160): Wrong argument count for function call: 3 arguments given but expected 4.

--> script/DeployRaffle.s.sol:44:9:

|

44 | addConsumer.addConsumer(address(raffle), vrfCoordinator, subscriptionId);

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

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

--> test/unit/RaffleTest.t.sol:46:13:

|

46 | ) = helperConfig.activeNetworkConfig();

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

Let's read it. We modified the 3 functions createSubscription, fundSubscription and addConsumer, we should reflect these changes inside DeployRaffle.s.sol. Then we are calling helperConfig.activeNetworkConfig in Raffle.t.sol, we should adjust that to reflect the changes of the NetworkConfig struct.

Open the DeployRaffle.s.sol file and modify the 3 functions call to include deployerKey as an input parameter.

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,

address link,

uint256 deployerKey

) = helperConfig.activeNetworkConfig();

if (subscriptionId == 0) {

CreateSubscription createSubscription = new CreateSubscription();

subscriptionId = createSubscription.createSubscription(vrfCoordinator, deployerKey);

FundSubscription fundSubscription = new FundSubscription();

fundSubscription.fundSubscription(vrfCoordinator, subscriptionId, link, deployerKey);

}

vm.startBroadcast();

Raffle raffle = new Raffle(

entranceFee,

interval,

vrfCoordinator,

gasLane,

subscriptionId,

callbackGasLimit

);

vm.stopBroadcast();

AddConsumer addConsumer = new AddConsumer();

addConsumer.addConsumer(address(raffle), vrfCoordinator, subscriptionId, deployerKey);

return (raffle, helperConfig);

}

}

Open the RaffleTest.t.sol and add the deployerKey in the variables section and in the setUp function where helperConfig.activeNetworkConfig() is called:

contract RaffleTest is Test {

event EnteredRaffle(address indexed player);

Raffle public raffle;

HelperConfig public helperConfig;

uint256 entranceFee;

uint256 interval;

address vrfCoordinator;

bytes32 gasLane;

uint256 subscriptionId;

uint32 callbackGasLimit;

address link;

uint256 deployerKey;

address public PLAYER = makeAddr("player");

uint256 public constant STARTING\_USER\_BALANCE = 10 ether;

function setUp() external {

DeployRaffle deployer = new DeployRaffle();

(raffle, helperConfig) = deployer.run();

vm.deal(PLAYER, STARTING\_USER\_BALANCE);

(

entranceFee,

interval,

vrfCoordinator,

gasLane,

subscriptionId,

callbackGasLimit,

link,

deployerKey

) = helperConfig.activeNetworkConfig();

}

}

Let's run forge build again. There are two possible outcomes here:

1. Everything compiles, amazing!
2. You get the following error:

Error:

Compiler run failed:

Error: Compiler error (/solidity/libsolidity/codegen/CompilerUtils.cpp:1429):Stack too deep. Try compiling with `--via-ir` (cli) or the equivalent `viaIR: true` (standard JSON) while enabling the optimizer. Otherwise, try removing local variables.

CompilerError: Stack too deep. Try compiling with `--via-ir` (cli) or the equivalent `viaIR: true` (standard JSON) while enabling the optimizer. Otherwise, try removing local variables.

If you get this error just go and comment out the deployerKey lines in Raffle.t.sol:

contract RaffleTest is Test {

event EnteredRaffle(address indexed player);

Raffle public raffle;

HelperConfig public helperConfig;

uint256 entranceFee;

uint256 interval;

address vrfCoordinator;

bytes32 gasLane;

uint256 subscriptionId;

uint32 callbackGasLimit;

address link;

// uint256 deployerKey;

address public PLAYER = makeAddr("player");

uint256 public constant STARTING\_USER\_BALANCE = 10 ether;

function setUp() external {

DeployRaffle deployer = new DeployRaffle();

(raffle, helperConfig) = deployer.run();

vm.deal(PLAYER, STARTING\_USER\_BALANCE);

(

entranceFee,

interval,

vrfCoordinator,

gasLane,

subscriptionId,

callbackGasLimit,

link,

// deployerKey

) = helperConfig.activeNetworkConfig();

}

}

Run forge build again and everything should compile.

With that out of our way try the forge test --fork-url $SEPOLIA\_RPC\_URL again.

Ran 12 tests for test/unit/RaffleTest.t.sol:RaffleTest

[PASS] testCheckUpkeepReturnsFalseIfItHasNoBalance() (gas: 19096)

[PASS] testCheckUpkeepReturnsFalseIfRaffleIsntOpen() (gas: 130771)

[PASS] testDontAllowPlayersToEnterWhileRaffleIsCalculating() (gas: 135654)

[PASS] testEmitsEventOnEntrance() (gas: 68752)

[FAIL. Reason: call reverted as expected, but without data; counterexample: calldata=0x9a3d2cf80000000000000000000000000000000000000000000000000000000000000b18 args=[2840]] testFulfillRandomWordsCanOnlyBeCalledAfterPerformUpkeep(uint256) (runs: 0, μ: 0, ~: 0)

[FAIL. Reason: EvmError: Revert] testFulfillRandomWordsPicksAWinnerRestesAndSendsMoney() (gas: 297296)

[PASS] testPerformUpkeepCanOnlyRunIfCheckUpkeepIsTrue() (gas: 126958)

[PASS] testPerformUpkeepRevertsIfCheckUpkeepIsFalse() (gas: 17399)

[PASS] testPerformUpkeepUpdatesRaffleStateAndEmitsRequestId() (gas: 133024)

[PASS] testRaffleInitializesInOpenState() (gas: 7753)

[PASS] testRaffleRecordsPlayerWhenTheyEnter() (gas: 68318)

[PASS] testRaffleRevertsWHenYouDontPayEnough() (gas: 13069)

Suite result: FAILED. 10 passed; 2 failed; 0 skipped; finished in 4.45s (1.70s CPU time)

Ok, some failed, but it feels super good not failing at setUp level.

The first failing test is testFulfillRandomWordsCanOnlyBeCalledAfterPerformUpkeep, the fuzz test. The fail reason is call reverted as expected, but without data. This means that it reverts, but not with the message we specified, and this is expected, given that we are using a mock contract, and the real version of the contract is most likely different. Mocks are usually simplified to facilitate ease of testing.

We should not run this test on Sepolia.

Add the following modifier in Raffle.t.sol:

modifier skipFork() {

if (block.chainid != 31337){

return;

}

\_;

}

This will check the block.chainid to see if we are on Sepolia. If we are on Sepolia then it returns, skipping the test.

Add it next to the raffleEnteredAndTimePassed modifier:

function testFulfillRandomWordsCanOnlyBeCalledAfterPerformUpkeep(uint256 requestId)

public

raffleEntredAndTimePassed

skipFork

{

// Arrange

// Act / Assert

vm.expectRevert("nonexistent request");

// vm.mockCall could be used here...

VRFCoordinatorV2PlusMock(vrfCoordinator).fulfillRandomWords(

requestId,

address(raffle)

);

}

The other failing test is testFulfillRandomWordsPicksAWinnerRestesAndSendsMoney. Looking through its code we can see why it fails:

// Pretend to be Chainlink VRF

VRFCoordinatorV2PlusMock(vrfCoordinator).fulfillRandomWords(

uint256(requestId),

address(raffle)

);

We simply can't do this on Sepolia, because you can't pretend you are a Chainlink VRF node on a live testnet, there are already real Chainlink VRF nodes working on the live testnet and they are the ones that should call fulfillRandomWords.

Add the skipFork modifier to this test as well.

Run forge test --fork-url $SEPOLIA\_RPC\_URL again.

Everything passes! Amazing!

This was a lot! Take a break, touch some grass and come back to finish this section!