**Refactoring events data**

**Refactoring events data**

In this lesson, we will learn how to access event data inside our tests. Let's create a new event and emit it in performUpkeep to test something. Inside Raffle.sol in the events section create a new event: event RequestedRaffleWinner(uint256 indexed requestId); Emit the event at the end of the performUpkeep function:

function performUpkeep(bytes calldata /\* performData \*/) external override {

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

// require(upkeepNeeded, "Upkeep not needed");

if (!upkeepNeeded) {

revert Raffle\_\_UpkeepNotNeeded(

address(this).balance,

s\_players.length,

uint256(s\_raffleState)

);

}

s\_raffleState = RaffleState.CALCULATING;

VRFV2PlusClient.RandomWordsRequest memory request = VRFV2PlusClient.RandomWordsRequest({

keyHash: i\_keyHash,

subId: i\_subscriptionId,

requestConfirmations: REQUEST\_CONFIRMATIONS,

callbackGasLimit: i\_callbackGasLimit,

numWords: NUM\_WORDS,

extraArgs: VRFV2PlusClient.\_argsToBytes(

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

VRFV2PlusClient.ExtraArgsV1({nativePayment: false})

)

});

uint256 requestId = s\_vrfCoordinator.requestRandomWords(request);

emit RequestedRaffleWinner(requestId);

}

At this point in the video, Patrick asks the audience if this event is redundant. This is an amazing question to ask yourself every time you do something in Solidity because as you know, everything costs gas. Another absolute truth about this environment is that no one wants to pay gas. So we, as developers, need to write efficient code.

To answer Patrick's question: Yes it's redundant, inside the VRFCoordinatorV2Mock you'll find that the requestRandomWords emits a giant event called RandomWordsRequested that contains the requestId we are also emitting in our new event. You'll see this a lot in smart contracts that involve transfers. But more on that in future sections.

We will keep the event for now for testing purposes.

It is important to test events! You might see them as a nice feature to examine what happened more easily using etherscan, but that's not all they are for. For example, the request for randomness is 100% reliant on events, because when requestRandomWords emits the RandomWordsRequested event, that gets picked up by the Chainlink nodes and the nodes use the information to provide the randomness service to you by calling back your fulfillRandomWords. **In the absence of the event, they wouldn't know where and what to send.**

Let's write a test that checks if performUpkeep updates the raffle state and emit the event we created:

Add import {Vm} from "forge-std/Vm.sol"; inside the import sections of RaffleTest.t.sol.

We decided to include the PLAYER entering the raffle and setting block.timestamp into the future inside a modifier. That way we can easily use that everywhere, without typing the same 4 rows of code over and over again.

modifier raffleEntredAndTimePassed() {

vm.prank(PLAYER);

raffle.enterRaffle{value: entranceFee}();

vm.warp(block.timestamp + interval + 1);

vm.roll(block.number + 1);

\_

}

function testPerformUpkeepUpdatesRaffleStateAndEmitsRequestId() public raffleEntredAndTimePassed {

// Act

vm.recordLogs();

raffle.performUpkeep(""); // emits requestId

Vm.Log[] memory entries = vm.getRecordedLogs();

bytes32 requestId = entries[1].topics[1];

// Assert

Raffle.RaffleState raffleState = raffle.getRaffleState();

// requestId = raffle.getLastRequestId();

assert(uint256(requestId) > 0);

assert(uint(raffleState) == 1); // 0 = open, 1 = calculating

}

Let's analyze the test line by line. We start by calling vm.recordLogs(). You can read more about this one [here](https://book.getfoundry.sh/cheatcodes/record-logs). This cheatcode starts recording all emitted events inside an array. After that, we call performUpkeep which emits both the events we talked earlier about. We can access the array where all the emitted events were stored by using vm.getRecordedLogs(). It usually takes some trial and error, or forge debug to know where the event that interests us is stored. But we can cheat a little bit. We know that the big event from the vrfCoordinator is emitted first, so our event is second, i.e. entries[1] (because the index starts from 0). Looking further in the examples provided [here](entries%5b1%5d), we see that the first topic, stored at index 0, is the name and output of the event. Given that our event only emits one parameter, the requestId, then we are aiming for entries[1].topics[1].

Moving on, we get the raffle state using the getRaffleState view function. We assert the requestId is higher than 0, meaning it exists, we also assert that raffleState is equal to 1, i.e. CALCULATING.

Run the test using forge test --mt testPerformUpkeepUpdatesRaffleStateAndEmitsRequestId.

It passes, great job!