**One Big Test**

**The biggest test you ever wrote**

You are a true hero for reaching this lesson! Let's finalize the testing with a big function.

Up until now, we've tested parts of the contract with a focus on checks that should revert in certain conditions. We never fully tested the happy case. We will do that now.

Open your RaffleTest.t.sol and add the following function:

function testFulfillRandomWordsPicksAWinnerResetsAndSendsMoney() public raffleEnteredAndTimePassed {

// Arrange

uint256 additionalEntrants = 3;

uint256 startingIndex = 1;

for (uint256 i = startingIndex; i < startingIndex + additionalEntrants; i++) {

address player = address(uint160(i));

hoax(player, 1 ether);

raffle.enterRaffle{value: entranceFee}();

}

}

This is not the whole function but we break it to not overwhelm you.

1. We define the function, make it public and apply the `raffleEnteredAndTimePassed` modifier;

2. We define two new variables `additionalEntrants` and `startingIndex`;

3. We assign `1` to `startingIndex` because we don't want to start with `index 0`, you will see why in a moment;

4. We use a for loop that creates 5 addresses. If we started from `index 0` then the first created address would have been `address(0)`. `address(0)` has a special status in the Ethereum ecosystem, you shouldn't use it in tests because different systems check against sending to it or against configuring it. Your tests would fail and not necessarily because your smart contract is broken;

5. Inside the loop, we use `hoax` which acts as `deal + prank` to call `raffle.enterRaffle` using each of the newly created addresses. Read more about `hoax` [here](<https://book.getfoundry.sh/reference/forge-std/hoax?highlight=hoax#hoax>).

Ok, now we need to pretend to be Chainlink VRF and call `fulfillRandomWords`. We will need the `requestId` and the `consumer`. The consumer is simple, it's the address of the `Raffle` contract. How do we get the `requestId`? We did this in the previous lesson!

function testFulfillRandomWordsPicksAWinnerResetsAndSendsMoney() public raffleEnteredAndTimePassed {

// Arrange

uint256 additionalEntrants = 333;

uint256 startingIndex = 1;

for (uint256 i = startingIndex; i < startingIndex + additionalEntrants; i++) {

address player = address(uint160(i));

hoax(player, 1 ether);

raffle.enterRaffle{value: entranceFee}();

}

vm.recordLogs();

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

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

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

// Pretend to be Chainlink VRF

VRFCoordinatorV2\_5Mock(vrfCoordinator).fulfillRandomWords(

uint256(requestId),

address(raffle)

);

}

1. We copy what we did in the previous lesson to get the requestId emitted by the performUpkeep;
2. We then use the VRFCoordinatorV2\_5Mock to call the fulfillRandomWords function. This is usually called by the Chainlink nodes but given that we are on our local Anvil chain we need to do that action.
3. fulfillRandomWords expects a uint256 requestId, so we use uint256(requestId) to cast it from bytes32 to the expected type.

With this last call we've finished the Arrange stage of our test. Let's continue with the Assert stage.

Before that, we need a couple of view functions in Raffle.sol:

function getRecentWinner() public view returns (address) {

return s\_recentWinner;

}

function getNumberOfPlayers() public view returns (uint256) {

return s\_players.length;

}

function getLastTimeStamp() public view returns (uint256) {

return s\_lastTimeStamp;

}

We'll use this one in testing the recent winner.

function testFulfillRandomWordsPicksAWinnerResetsAndSendsMoney() public raffleEnteredAndTimePassed {

// Arrange

uint256 additionalEntrants = 3;

uint256 startingIndex = 1;

for (uint256 i = startingIndex; i < startingIndex + additionalEntrants; i++) {

address player = address(uint160(i));

hoax(player, STARTING\_USER\_BALANCE);

raffle.enterRaffle{value: entranceFee}();

}

uint256 prize = entranceFee \* (additionalEntrants + 1);

uint256 winnerStartingBalance = expectedWinner.balance;

// Act

vm.recordLogs();

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

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

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

// Pretend to be Chainlink VRF

VRFCoordinatorV2\_5Mock(vrfCoordinator).fulfillRandomWords(

uint256(requestId),

address(raffle)

);

// Assert

address recentWinner = raffle.getRecentWinner();

Raffle.RaffleState raffleState = raffle.getRaffleState();

uint256 winnerBalance = recentWinner.balance;

uint256 endingTimeStamp = raffle.getLastTimeStamp();

uint256 prize = entranceFee \* (additionalEntrants + 1);

assert(expectedWinner == recentWinner);

assert(uint256(raffleState) == 0);

assert(winnerBalance == winnerStartingBalance + prize);

assert(endingTimeStamp > startingTimeStamp);

}

1. We've made some changes. Whenever you test things make sure to be consistent and try to avoid using magic numbers. We hoaxed the newly created addresses with 1 ether for no obvious reason. We should use the STARTING\_USER\_BALANCE for consistency.
2. We created a new variable previousTimeStamp to record the previous time stamp, the one before the actual winner picking happened.

Now we are ready to start our assertions.

1. We assert that the raffle state is OPEN because that's how our raffle should be after the winner is drawn and the prize is sent;
2. We assert that we have chosen a winner;
3. We assert that the s\_players array has been properly reset, so players from the previous raffle don't get to participate in the next one without paying;
4. We assert that the fulfillRandomWords updates the s\_lastTimeStamp variable;
5. We assert that the winner receives their ETH prize;

Amazing work, let's try it out with forge test --mt testFulfillRandomWordsPicksAWinnerResetsAndSendsMoney --vv.

Aaaaand it failed:

Ran 1 test suite in 2.35s (11.16ms CPU time): 0 tests passed, 1 failed, 0 skipped (1 total tests)

Failing tests:

Encountered 1 failing test in test/unit/RaffleTest.t.sol:RaffleTest

[FAIL. Reason: InvalidRequest()] testFulfillRandomWordsPicksAWinnerResetsAndSendsMoney() (gas: 362400)

Failing is part of this game, whatever you do, at some point you will fail. The trick is not staying in that situation, let's see why this failure happened and turn it into success.

Can you figure out what is wrong? If we trace the InvalidRequest() error, we see that it is emitted from the \_chargePayment function in the VRFCoordinatorV2\_5Mock contract. It looks like we have not funded our subscription with enough LINK tokens. Let's fund it with more by updating our Interactions.s.sol where for now, I'll times our FUND\_AMOUNT by 100.

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

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

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

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

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

vm.startBroadcast(account);

VRFCoordinatorV2\_5Mock(vrfCoordinator).fundSubscription(subscriptionId, FUND\_AMOUNT \* 100);

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(account);

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

vm.stopBroadcast();

}

}

Let's run the test again using forge test --mt testFulfillRandomWordsPicksAWinnerRestesAndSendsMoney -vvv.

Run the test with forge test --mt testFulfillRandomWordsPicksAWinnerResetsAndSendsMoney.

[⠢] Compiling...

No files changed, compilation skipped

Ran 1 test for test/unit/RaffleTest.t.sol:RaffleTest

[PASS] testFulfillRandomWordsPicksAWinnerResetsAndSendsMoney() (gas: 287531)

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 2.93ms (250.30µs CPU time)

Amazing work! Let's try some forked tests in the next lesson.