**Setup the tests**

**Continuing our testing journey**

Let's jump straight into testing! But where do we start?

Easy! Let's call forge coverage:

Analysing contracts...

Running tests...

| File | % Lines | % Statements | % Branches | % Funcs |

| ------------------------- | ------------- | -------------- | ------------- | ------------- |

| script/DeployRaffle.s.sol | 100.00% (7/7) | 100.00% (9/9) | 100.00% (0/0) | 100.00% (1/1) |

| script/HelperConfig.s.sol | 0.00% (0/9) | 0.00% (0/13) | 0.00% (0/2) | 0.00% (0/2) |

| src/Raffle.sol | 2.94% (1/34) | 2.33% (1/43) | 0.00% (0/8) | 7.69% (1/13) |

| Total | 16.00% (8/50) | 15.38% (10/65) | 0.00% (0/10) | 12.50% (2/16) |

These numbers are weak! Let's improve them!

Open the RaffleTest.t.sol inside the test/unit folder.

In my opinion, when one needs to decide where to start testing there are two sensible approaches one could take:

1. Easy to Complex - start with view functions, then with smaller functions and advance to the more complex functions;
2. From the main entry point(s) to the periphery - what is the main functionality that the external user needs to call in order to interact with your contract;

Patrick chose number 2. So what is the main entry point of our Raffle contract? The enterRaffle function.

Let's look closely at it:

function enterRaffle() external payable {

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

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

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

emit EnteredRaffle(msg.sender);

}

1. We check if the msg.value is high enough;
2. We check if the RaffleState is OPEN;
3. If all of the above are true then the msg.sender should be pushed in the s\_players array;
4. Our function emits the EnteredRaffle event.

Let's test point 1:

function testRaffleRevertsWHenYouDontPayEnough() public {

// Arrange

vm.prank(PLAYER);

// Act / Assert

vm.expectRevert(Raffle.Raffle\_\_NotEnoughEthSent.selector);

raffle.enterRaffle();

}

We call vm.prank(PLAYER) to configure the fact that the next transaction will be called by the PLAYER. [Refresher](https://book.getfoundry.sh/cheatcodes/prank?highlight=prank#prank)

After that we use the vm.expectRevert [cheatcode](https://book.getfoundry.sh/cheatcodes/expect-revert?highlight=expectRevert#expectrevert) to test if the next call will revert. We also have the option to specify the error message. You can do that by calling the errorName.selector as input of the vm.expectRevert cheatcode. Following that we call the enterRaffle without specifying the value of the transaction.

Run the test using forge test --mt testRaffleRevertsWHenYouDontPayEnough.

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

[PASS] testRaffleRevertsWHenYouDontPayEnough() (gas: 10865)

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.99ms (161.70µs CPU time)

We will skip point 2 for now, let's go straight to point 3:

But before being able to test if a player is properly recorded in the s\_players array we first need a view function to access the players in the s\_players:

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

return s\_players[index];

}

Now that we have all the tools we need:

function testRaffleRecordsPlayerWhenTheyEnter() public {

// Arrange

vm.prank(PLAYER);

// Act

raffle.enterRaffle{value: entranceFee}();

// Assert

address playerRecorded = raffle.getPlayer(0);

assert(playerRecorded == PLAYER);

}

We start by pranking the PLAYER, then properly calling the enterRaffle function specifying the correct value. We call the new getPLayer function to copy the player recorded at index 0 in memory. Then we compare that value to the PLAYER address to ensure they match.

Test it with the following command: forge test --mt testRaffleRecordsPlayerWhenTheyEnter.

Amazing work! Let's continue in the next lesson! We are going to learn how to test events in Foundry.