**Adding more coverage to the tests**

**Let's keep testing**

In the previous lesson, we tested if the s\_addressToAmountFunded is updated correctly. Continuing from there we need to test that funders array is updated with msg.sender.

Add the following test to your FundMe.t.sol:

function testAddsFunderToArrayOfFunders() public {

vm.startPrank(alice);

fundMe.fund{value: SEND\_VALUE}();

vm.stopPrank();

address funder = fundMe.getFunder(0);

assertEq(funder, alice);

}

What's happening here? We start with our user alice who calls fundMe.fund in order to fund the contract. Then we use the getter function we created in the previous lesson to query what is registered inside the funders array at index 0. We then use the assertEq cheatcode to compare the address we queried against alice.

Run the test using forge test --mt testAddsFunderToArrayOfFunders. It passed, perfect!

Each of our tests uses a fresh setUp, so if we run all of them and testFundUpdatesFundDataStrucutre calls fund, that won't be persistent for testAddsFunderToArrayOfFunders.

Moving on, we should test the withdraw function. Let's check that only the owner can withdraw.

Add the following test to your FundMe.t.sol:

function testOnlyOwnerCanWithdraw() public {

vm.prank(alice);

fundMe.fund{value: SEND\_VALUE}();

vm.expectRevert();

vm.prank(alice);

fundMe.withdraw(); }

What's happening here? We start with our user alice who calls fundMe.fund in order to fund the contract. We then use Alice's address to try and withdraw. Given that Alice is not the owner of the contract, it should fail. That's why we are using the vm.expectRevert cheatcode.

**REMEMBER:** Whenever you have a situation where two or more vm cheatcodes come one after the other keep in mind that these would ignore one another. In other words, when we call vm.expectRevert(); that won't apply to vm.prank(alice);, it will apply to the withdraw call instead. The same would have worked if these had been reversed. Cheatcodes affect transactions, not other cheatcodes.

Run the test using forge test --mt testOnlyOwnerCanWithdraw. It passed, amazing!

As you can see, in both testAddsFunderToArrayOfFunders and testOnlyOwnerCanWithdraw we used alice to fund the contract. Copy-pasting the same snippet of code over and over again, if we end up writing hundreds of tests, is not necessarily the best approach. We can see each line of code/block of lines of code as a building block. Multiple tests will share some of these building blocks. We can define these building blocks using modifiers to dramatically increase our efficiency in writing tests.

Add the following modifier to your FundMe.t.sol:

modifier funded() {

vm.prank(alice);

fundMe.fund{value: SEND\_VALUE}();

assert(address(fundMe).balance > 0);

\_;

}

We first use the vm.prank cheatcode to signal the fact that the next transaction will be called by alice. We call fund and then we assert that the balance of the fundMe contract is higher than 0, if true, it means that Alice's transaction was successful. Every single time we need our contract funded we can use this modifier to do it.

Refactor the previous test as follows:

function testOnlyOwnerCanWithdraw() public funded {

vm.expectRevert();

fundMe.withdraw();

}

Slim and efficient!

Ok, we've tested that a non-owner cannot withdraw. But can the owner withdraw?

To test this we will need a new getter function. Add the following to the FundMe.sol file next to the other getter functions:

function getOwner() public view returns (address) {

return i\_owner;

}

Make sure to make i\_owner private.

Cool!

Let's discuss more about structuring our tests.

The arrange-act-assert (AAA) methodology is one of the simplest and most universally accepted ways to write tests. As the name suggests, it comprises three parts:

* **Arrange:** Set up the test by initializing variables, and objects and prepping preconditions.
* **Act:** Perform the action to be tested like a function invocation.
* **Assert:** Compare the received output with the expected output.

We will start our test as usual:

function testWithdrawFromASingleFunder() public funded {

}

Now we are in the first stage of the AAA methodology: Arrange

We first need to check the initial balance of the owner and the initial balance of the contract.

uint256 startingFundMeBalance = address(fundMe).balance;

uint256 startingOwnerBalance = fundMe.getOwner().balance;

We have what we need to continue with the Act stage.

vm.startPrank(fundMe.getOwner());

fundMe.withdraw();

vm.stopPrank();

Our action stage is comprised of pranking the owner and then calling withdraw.

We have reached our final testing part, the Assert stage.

We need to find out the new balances, both for the contract and the owner. We need to check if these match the expected numbers:

uint256 endingFundMeBalance = address(fundMe).balance;

uint256 endingOwnerBalance = fundMe.getOwner().balance;

assertEq(endingFundMeBalance, 0);

assertEq(

startingFundMeBalance + startingOwnerBalance,

endingOwnerBalance

);

The endingFundMeBalance should be 0, because we just withdrew everything from it. The owner's balance should be the startingFundMeBalance + startingOwnerBalance because we withdrew the fundMe starting balance.

Let's run the test using the following command: forge test --mt testWithdrawFromASingleFunder

It passed, amazing!

**Remember to call forge test from time to time to ensure that any changes to the main contract or to testing modifiers or setup didn't break any existing tests. If it did, go back and see how the changes affected the test and modify them to accustom it.**

Ok, we've tested that the owner can indeed withdraw when the fundMe contract is funded by a single user. Can they withdraw when the contract is funded by multiple users?

Put the following test in your FundMe.t.sol:

function testWithdrawFromMultipleFunders() public funded {

uint160 numberOfFunders = 10;

uint160 startingFunderIndex = 1;

for (uint160 i = startingFunderIndex; i < numberOfFunders + startingFunderIndex; i++) {

// we get hoax from stdcheats

// prank + deal

hoax(address(i), SEND\_VALUE);

fundMe.fund{value: SEND\_VALUE}();

}

uint256 startingFundMeBalance = address(fundMe).balance;

uint256 startingOwnerBalance = fundMe.getOwner().balance;

vm.startPrank(fundMe.getOwner());

fundMe.withdraw();

vm.stopPrank();

assert(address(fundMe).balance == 0);

assert(startingFundMeBalance + startingOwnerBalance == fundMe.getOwner().balance);

assert((numberOfFunders + 1) \* SEND\_VALUE == fundMe.getOwner().balance - startingOwnerBalance);

}

That seems like a lot! Let's go through it.

We start by declaring the total number of funders. Then we declare that the startingFunderIndex is 1. You see that both these variables are defined as uint160 and not our usual uint256. Down the road, we will use the startingFunderIndex as an address. If we look at the definition of an [address](https://docs.soliditylang.org/en/latest/types.html#address) we see that it holds a 20 byte value and that explicit conversions to and from address are allowed for uint160, integer literals, bytes20 and contract types. Having the index already in uint160 will save us from casting it when we need to convert it into an address.

We start a loop. Inside this loop we need to deal and prank an address and then call fundMe.fund. Foundry has a better way: [hoax](https://book.getfoundry.sh/reference/forge-std/hoax?highlight=hoax#hoax). This works like deal + prank. It pranks the indicated address while providing some specified ether.

hoax(address(i), SEND\_VALUE);

As we've talked about above, we use the uint160 index to obtain an address. We start our index from 1 because it's not advised to user address(0) in this way. address(0) has a special regime and should not be pranked.

The SEND\_VALUE specified in hoax represents the ether value that will be provided to address(i).

Good, now that we have pranked an address and it has some balance we call fundeMe.fund.

After the loop ends we repeat what we did in the testWithdrawFromASingleFunder. We record the contract and owner's starting balances. This concludes our Arrange stage.

The next logical step is pranking the owner and withdrawing. This starts the Act stage.

In the Assert part of our test, we compare the final situation against what we expected.

assert(address(fundMe).balance == 0);

After withdrawal, fundMe's balance should be 0.

assert(startingFundMeBalance + startingOwnerBalance == fundMe.getOwner().balance);

The owner's balance should be equal to the sum of startingOwnerBalance and the amount the owner withdrew (which is the startingFundMeBalance).

assert((numberOfFunders + 1) \* SEND\_VALUE == fundMe.getOwner().balance - startingOwnerBalance);

We compare the product between the total number of funders and SEND\_VALUE to the total shift in the owner's balance. We added 1 to the numberOfFunders because we used the funded modifier which also adds alice as one of the funders.

Run the test using forge test --mt testWithdrawFromMultipleFunders. Run all tests using forge test.

Let's run forge coverage and see if our coverage table got better.

Congratulations, everything works way better!