**Foundry tests cheatcodes**

**Foundry magic: Cheatcodes**

Now that we fixed our deployment script, and our tests have become blockchain agnostic, let's get back to increasing that coverage we were talking about some lessons ago.

**Reminder:** Call forge coverage in your terminal. We need to bring that total coverage percentage as close to 100% as we can! Not all things require 100% coverage, or maybe achieving 100% coverage is too time expensive, but ... 12-13%? That is a joke, we can do way better than that.

Let's take a moment and look at the fund function from FundMe.sol. What should it do?

1. require(msg.value.getConversionRate(s\_priceFeed) >= MINIMUM\_USD this means that fund should revert if our msg.value converted in USDC is lower than the MINIMUM\_USD;
2. addressToAmountFunded[msg.sender] += msg.value; The addressToAmountFunded mapping should be updated appropriately to show the funded value;
3. The funders array should be updated with msg.sender;

To test all these we will employ some of Foundry's main features ... it's Cheatcodes. As Foundry states in the Foundry Book: "Cheatcodes give you powerful assertions, the ability to alter the state of the EVM, mock data, and more.". Read more about them [here](https://book.getfoundry.sh/cheatcodes/).

To test point 1 we will use one of the most important cheatcodes: expectRevert (read more about it [here](https://book.getfoundry.sh/cheatcodes/expect-revert)).

Open FundMe.t.sol and add the following function:

function testFundFailsWIthoutEnoughETH() public {

vm.expectRevert(); // <- The next line after this one should revert! If not test fails.

fundMe.fund(); // <- We send 0 value

}

We are attempting to fund the contract with 0 value, it reverts and our test passes.

Before jumping on points 2 and 3, let's refactor our code a little bit. As we've discussed before storage variables should start with s\_. Change all the addressToAmountFunded mentions to s\_addressToAmountFunded and all the funders to s\_funders. Another quick refactoring we need to do is to change the visibility of s\_addressToAmountFunded and s\_funders to private. Private variables are more gas-efficient than public ones.

Call a quick forge test to make sure nothing broke anywhere.

Now that we made those two variables private, we need to write some getters for them, i.e. view functions that we will use to query the state of our smart contract.

Please add the following at the end of FundMe.sol:

/\*\* Getter Functions \*/

function getAddressToAmountFunded(address fundingAddress) public view returns (uint256) {

return s\_addressToAmountFunded[fundingAddress];

}

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

return s\_funders[index];

}

Pfeww! Great now we can test points 2 and 3 indicated above:

Add the following test in FundMe.t.sol:

function testFundUpdatesFundDataStructure() public {

fundMe.fund{value: 10 ether}();

uint256 amountFunded = fundMe.getAddressToAmountFunded(msg.sender);

assertEq(amountFunded, 10 ether);

}

Run forge test --mt testFundUpdatesFundDataStructure in your terminal.

Aaaand it fails! Why does it fail? Let's try it again, but this time put address(this) instead of msg.sender. Now it passed, but we still don't quite get why.

User management is a very important aspect you need to take care of when writing tests. Imagine you are writing a more complex contract, where you have different user roles, maybe the owner has some privileges, different from an admin who has different privileges from a minter, who, as you've guessed, has different privileges from the end user. How can we differentiate all of them in our testing? We need to make sure we can write tests about who can do what.

As always, Foundry can help us with that. Please remember the cheatcodes below, you are going to use them thousands of times.

1. [prank](https://book.getfoundry.sh/cheatcodes/prank) "Sets msg.sender to the specified address for the next call. “The next call” includes static calls as well, but not calls to the cheat code address."
2. [startPrank](https://book.getfoundry.sh/cheatcodes/start-prank) and [stopPrank](https://book.getfoundry.sh/cheatcodes/stop-prank) startPrank Sets msg.sender for all subsequent calls until stopPrank is called. It works a bit like vm.startBroadcast and vm.stopBroadcast we used to write our deployment script. Everything between the vm.startPrank and vm.stopPrank is signed by the address you provide inside the ().

Ok, cool, but who is the actual user that we are going to use in one of the cheatcodes above? We have another cheatcode for this. To create a new user address we can use the makeAddr cheatcode. Read more about it [here](https://book.getfoundry.sh/reference/forge-std/make-addr?highlight=mak#makeaddr).

Add the following line at the start of your FundMeTest contract:

address alice = makeAddr("alice");

Now whenever we need a user to call a function we can use prank and alice to run our tests.

To further increase the readability of our contract, let's avoid using a magic number for the funded amount. Create a constant variable called SEND\_VALUE and give it the value of 0.1 ether (don't be scared by the floating number which technically doesn't work with Solidity - 0.1 ether means 10 \*\* 17 ether).

Back to our test, add the following test in FundMe.t.sol:

function testFundUpdatesFundDataStructure() public {

vm.prank(alice);

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

uint256 amountFunded = fundMe.getAddressToAmountFunded(alice);

assertEq(amountFunded, SEND\_VALUE);

}

Finally, now let's run forge test --mt testFundUpdatesFundDataStructure again.

It fails ... again!

But why? Let's call forge test --mt testFundUpdatesFundDataStructure -vvv to get more information about where and why it fails.

Ran 1 test for test/FundMe.t.sol:FundMeTest

[FAIL. Reason: EvmError: Revert] testFundUpdatesFundDataStructure() (gas: 16879)

Traces:

[16879] FundMeTest::testFundUpdatesFundDataStructure()

├─ [0] VM::prank(alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6])

│ └─ ← [Return]

├─ [0] FundMe::fund{value: 100000000000000000}()

│ └─ ← [OutOfFunds] EvmError: OutOfFunds

└─ ← [Revert] EvmError: Revert

Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 696.30µs (25.10µs CPU time)

How can someone fund our FundMe contract when they have 0 balance? They can't.

We need a way to give alice some ether to be able to use her address for testing purposes.

Foundry to the rescue! There's always a cheatcode to help you overcome your hurdles.

deal allows us to set the ETH balance of a user. Read more about it [here](https://book.getfoundry.sh/cheatcodes/deal).

Add the following line at the end of the setup.

vm.deal(alice, STARTING\_BALANCE);

Declare the STARTING\_BALANCE as a constant variable up top:

uint256 constant STARTING\_BALANCE = 10 ether;

Let's run forge test --mt testFundUpdatesFundDataStructure again.

And now it passes. Congratulations!

I know a lot of new cheatcodes were introduced in this lesson. Keep in mind that these are the most important cheatcodes there are, and you are going to use them over and over again. Regardless if you are developing or auditing a project, that project will always have at least an owner and a user. These two would always have different access to different functionalities. Most of the time the user needs some kind of balance, be it ETH or some other tokens. So, making a new address, giving it some balance, and pranking it to act as a caller for a tx will 100% be part of your every test file.