**Create integration tests**

**Writing a README.md**

A README is often the first item a visitor will see when visiting your repository. It serves as an introduction to your project, explaining what it does, why it is useful, and how users can get started with it. This initial impression can significantly impact whether someone decides to explore your project further.

There are multiple templates available on the internet, but generally, yours should include at least a Title, a Project Overview, a Getting Started Guide and maybe some Contribution Guidelines (if you are building an open-source project).

A README is your project's face to the world, and investing time in making it clear, comprehensive, and engaging can significantly impact your project's success and community engagement.

**Integration tests**

To seamlessly interact with our contract, we need to create a programmatic for using it's functions.

Please create a new file called Interactions.s.sol in the script folder.

In this file, we will create two scripts, one for funding and one for withdrawing.

Each contract will contain one script, and for it to work each needs to inherit from the Script contract. Each contract will have a run function which shall be called by forge script when we run it.

In order to properly interact with our fundMe contract we would want to interact only with the most recent deployment we made. This task is easily achieved using the foundry-devops library. Please install it using the following command:

forge install Cyfrin/foundry-devops --no-commit

Ok, now with that out of the way, let's work on our scripts.

Put the following code in Interactions.s.sol:

// SPDX-License-Identifier: UNLICENSED

pragma solidity ^0.8.19;

import {Script, console} from "forge-std/Script.sol";

import {FundMe} from "../src/FundMe.sol";

import {DevOpsTools} from "foundry-devops/src/DevOpsTools.sol";

contract FundFundMe is Script {

uint256 SEND\_VALUE = 0.1 ether;

function fundFundMe(address mostRecentlyDeployed) public {

vm.startBroadcast();

FundMe(payable(mostRecentlyDeployed)).fund{value: SEND\_VALUE}();

vm.stopBroadcast();

console.log("Funded FundMe with %s", SEND\_VALUE);

}

function run() external {

address mostRecentlyDeployed = DevOpsTools.get\_most\_recent\_deployment("FundMe", block.chainid);

fundFundMe(mostRecentlyDeployed);

}

}

contract WithdrawFundMe is Script {

function withdrawFundMe(address mostRecentlyDeployed) public {

vm.startBroadcast();

FundMe(payable(mostRecentlyDeployed)).withdraw();

vm.stopBroadcast();

console.log("Withdraw FundMe balance!");

}

function run() external {

address mostRecentlyDeployed = DevOpsTools.get\_most\_recent\_deployment("FundMe", block.chainid);

withdrawFundMe(mostRecentlyDeployed);

}

}

We've created a new function called fundFundMe which takes an address corresponding to the most recently deployed FundMe contract. Inside we start and stop a broadcast which sends a transaction calling the fund function from the FundMe contract. We've imported console to be able to log the amount that we funded as a confirmation. Inside the run function, we call get\_most\_recent\_deployment from the DevOpsTools to get the address of the most recently deployed FundMe contract. We then use the newly acquired address as input for the fundFundMe function.

The same thing is done for WithdrawFundMe.

We could run this using the standard forge script script/Interactions.s.sol:FundFundMe --rpc-url xyz --private-key etc ... command, but writing that over and over again is not cool. We could test how this behaves using integration tests.

Integration tests are crucial for verifying how your smart contract interacts with other contracts, external APIs, or decentralized oracles that provide data feeds. These tests help ensure your contract can properly receive and process data, send transactions to other contracts, and function as intended within the wider ecosystem.

Before starting with the integration tests let's organize our tests into folders. Let's separate unit tests from integration tests by creating separate folders inside the test folder.

Create two new folders called integration and unit inside the test folder. Move FundMe.t.sol inside the unit folder. Make sure to update FundMe.t.sol to accommodate this change.

Run a quick forge test to ensure that everything builds and all tests pass.

Inside the integration folder create a new file called FundMeTestIntegration.t.sol.

Paste the following code inside it:

// SPDX-License-Identifier: MIT

pragma solidity 0.8.19;

import {DeployFundMe} from "../../script/DeployFundMe.s.sol";

import {FundFundMe, WithdrawFundMe} from "../../script/Interactions.s.sol";

import {FundMe} from "../../src/FundMe.sol";

import {Test, console} from "forge-std/Test.sol";

contract InteractionsTest is Test {

FundMe public fundMe;

DeployFundMe deployFundMe;

uint256 public constant SEND\_VALUE = 0.1 ether;

uint256 public constant STARTING\_USER\_BALANCE = 10 ether;

address alice = makeAddr("alice");

function setUp() external {

deployFundMe = new DeployFundMe();

fundMe = deployFundMe.run();

vm.deal(alice, STARTING\_USER\_BALANCE);

}

function testUserCanFundAndOwnerWithdraw() public {

uint256 preUserBalance = address(alice).balance;

uint256 preOwnerBalance = address(fundMe.getOwner()).balance;

// Using vm.prank to simulate funding from the USER address

vm.prank(alice);

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

WithdrawFundMe withdrawFundMe = new WithdrawFundMe();

withdrawFundMe.withdrawFundMe(address(fundMe));

uint256 afterUserBalance = address(alice).balance;

uint256 afterOwnerBalance = address(fundMe.getOwner()).balance;

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

assertEq(afterUserBalance + SEND\_VALUE, preUserBalance);

assertEq(preOwnerBalance + SEND\_VALUE, afterOwnerBalance);

}

}

You will see that the first half, including the setUp is similar to what we did in FundMe.t.sol. The test testUserCanFundAndOwnerWithdraw has a similar structure to testWithdrawFromASingleFunder from FundMe.t.sol. We record the starting balances, we use alice to fund the contract then the WithdrawFundMe script to call withdraw. The next step is recording the ending balances and running the same assertions we did in FundMe.t.sol.

Run the integration test using the following command:

forge test --mt testUserCanFundAndOwnerWithdraw -vv

Ran 1 test for test/integration/InteractionsTest.t.sol:InteractionsTest

[PASS] testUserCanFundAndOwnerWithdraw() (gas: 330965)

Logs:

Withdraw FundMe balance!

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 7.78ms (1.01ms CPU time)

Ran 1 test suite in 427.38ms (7.78ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)

Pfew! I know this was a lot. You are a true champion for reaching this point!

**Note 1:** Depending on when you go through this lesson there is a small chance that foundry-devops library has a problem that prevents you from building. The reason this happening is vm.keyExists used at foundry-devops/src/DevOpsTools.sol:119 is deprecated. Please replace vm.keyExists with vm.keyExistsJson in the place indicated. Next, we need to make sure that the Vm.sol contract in your forge-std library contains the vm.keyExistsJson. If you can't find it in your Vm.sol then please run the following command in your terminal: forge update --force. If you still can't forge build the project the please come ask questions in the Updraft section of Cyfrin's discord.

**Note 2:**

Inside the video lesson, Patrick touched on the subject of ffi. We didn't present it at length in the body of this lesson because foundry-devops doesn't need it anymore. But in short:

Forge FFI, which stands for Foreign Function Interface, is a cheatcode within the Forge testing framework for Solidity. It allows you to execute arbitrary shell commands directly from your Solidity test code.

* FFI enables you to call external programs or scripts from within your Solidity tests.
* You provide the command or script name along with any arguments as an array of strings.
* The Forge testing framework then executes the command in the underlying system environment and captures the output.

Read more about it [here](https://book.getfoundry.sh/cheatcodes/ffi?highlight=ffi#ffi).

A word of caution: FFI bypasses the normal security checks and limitations of Solidity. By running external commands, you introduce potential security risks if not used carefully. Malicious code within the commands you execute could compromise your setup. Whenever you clone repos or download other projects please make sure they don't have ffi = true in their foundry.toml file. If they do, we advise you not to run anything before you thoroughly examine where ffi is used and what commands is it calling. Stay safe!