Apr 11, 2023 12 min read

Foundry Unit Tests

Updated: May 18, 2023

This article will describe how to create unit tests in Solidity using Foundry. We will cover how to test all the state transitions that can occur in a smart contract, plus some additional useful features Foundry provides. Foundry has very extensive testing capabilities, so rather than rehashing the documentation, we will focus on the parts you will use most of the time.

This article assumes you are already comfortable with Solidity. If not, see our free learn solidity tutorial.

Install foundry

If you don't already have Foundry installed, follow the instructions here: https://book.getfoundry.sh/getting-started/installation

Authorship

This article was co-authored by Aymeric Taylor (LinkedIn, Twitter), a research intern at RareSkills.

Foundry hello world

Just run the following commands and it will set up the environment, create tests, and run them for you. (This assumes you have Foundry installed of course).

```
forge init
forge test
```

Solidity testing best practices

Regardless of the framework, the quality of a solidity unit tests depends on three factors:

- **line coverage,**
- **branch coverage**, and
- **completely defined state transitions.**

By understanding each of these, we can motivate why we focus on certain aspects of the Foundry API.

It is of course, not possible to document every range of input to every possible output. However, test quality will generally be correlated with the line coverage, the branch coverage, and defining state transitions. In our other article, we've documented how to measure line and branch coverage with Foundry. We'll explain the significance of the three metrics here:

1. Line coverage

Line coverage is what it sounds like. If a line of code was not executed during the tests, then the line coverage is not 100%. If a line was never executed, you can't be sure if it works as expected or will revert. There is no good reason to not have 100% line coverage in a smart contract. If you are writing code, it means you expect it to be executed at some point in the future, so why not test it?

2. Branch coverage

Even if every line is executed, it doesn't mean every variation in smart contract business logic is tested.

Consider the following function

```
function changeOwner(address newOwner) external {
    require(msg.sender == owner, "onlyOwner");
    owner = newOwner;
}
```

If you test this address by calling it with the owner, you will get 100% line coverage but not 100% branch coverage. That's because both the require statement and the owner assignment executed, but the case where the require reverted did not get tested.

Here is a more subtle example.

```
// @notice anyone can pay off someone else's loan
// @param debtor the person who's loan the sender is making a payment
for
function payDownLoan(address debtor) external payable {
    uint256 loanAmount = loanAmounts[debtor];
    require(loanAmount > 0, "no such loan");

    if (msg.value >= debtAmount {
        loanAmounts[debtor] = 0;
        emit LoanFullyRepaid(debtor);
    } else {
        emit LoanPayment(debtor, debtAmount, msg.value);
        loanAmount -= msg.value;
    }

    if (msg.value > loanAmount) {
        msg.sender.call{value: msg.value - loanAmount}("");
    }
}
```

How many branches are there to test in this case?

1. The case where the loan is zero
2. The case where someone pays less than the loan size
3. The case where someone pays exactly the loan size
4. The case where someone pays more than the loan size

It is possible to get 100% line coverage on this test by sending more ether than the loan size and less ether than the loan size. This would execute both branches of the if else, and the final if statement at the end. But this would not test the else statement where the loan is paid off exactly to zero.

The more branches your functions has, the exponentially harder it gets to unit test them. The technical word for this is cylomatic complexity.

3. Fully defined state transitions

Quality unit tests in solidity document state transitions as thoroughly as possible. State transitions include:

- a change in storage variables
- contracts getting deployed or self-destructed
- ether balances changing
- events getting emitted, with certain messages
- transactions reverting, with certain error messages

If a function does any of these things, the exact way it modifies the state should be captured in the unit tests and any deviation should cause a revert. This way, any accidental modifications, no matter how minor, will be caught automatically.
Going to the early example, what state transitions should be measured?

- the Ether in the contract increases by the same amount the borrower pays back the loan
- the storage variable tracking the loan size is reduced by the expected amount
- the revert happens with the expected error message when the sender pays for a non-existent loan
- the corresponding events and associated messages are emitted

If the business logic in your smart contract changes, the tests should fail. Normally, this is considered a “fragile” unit test in other domains. It can hurt the speed of iterating on the source code. But Solidity code is meant to be written once and never changed, so this is not a problem for smart contract testing.

4. Unit testing best practice conclusion

Why are we covering all of this before documenting how Foundry unit testing works? Because this will help us isolate the high impact testing utilities you will use most of the time. Foundry’s capabilities are vast, but only a small subset will be used in the majority of test cases.

Foundry Asserts

To ensure a state transition actually happened, you will need asserts.
Let’s start with the default test file that Foundry provides after you call forge init.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import "forge-std/Test.sol";
import "../src/Counter.sol";

contract CounterTest is Test {
    Counter public counter;

    function setUp() public {
        counter = new Counter();
        counter.setNumber(0);
    }

    function testIncrement() public {
        counter.increment();
        assertEq(counter.number(), 1);
    }

    function testSetNumber(uint256 x) public {
        counter.setNumber(x);
        assertEq(counter.number(), x);
    }
}
```

the setUp() function deploys the contract you are testing (as well as any other contract you want in the ecosystem).
Any function that starts with the word test will be executed as a unit test. Functions that do not start with "test" will not be executed unless a test or setUp function calls them.

Here are the asserts you have at your disposal.
The ones you would use most frequently are

- assertEq, assert equal
- assertLt, assert less than
- assertLe, assert less than or equal to
- assertGt, assert greater than
- assertGe, assert greater than or equal to
- assertTrue, assert to be true

The first two arguments to the assert are the comparison, but you can also add a helpful error message as a third argument, which you should always do (despite the default example not showing it). Here is the suggested way of writing assertions:

```
function testIncrement() public {
        counter.increment();
        assertEq(counter.number(), 1, "expect x to equal to 1");
}

function testSetNumber(uint256 x) public {
        counter.setNumber(x);
        assertEq(counter.number(), x, "x should be setNumber");
}
```

Changing msg.sender with foundry vm.prank

Foundry's rather humorous method to change the sender (account or wallet) is the vm.prank API (what foundry calls a cheatcode).
Here is a minimal example

```
function testChangeOwner() public {
         vm.prank(owner);
    contractToTest.changeOwner(newOwner);
    assertEq(contractToTest.owner(), newOwner);
}
```

The vm.prank only works for the transaction that happens immediately after. If you want a sequence of transactions to use the same address, use vm.startPrank and end them vm.stopPrank

```
function testMultipleTransactions() public {
     vm.startPrank(owner);
     // behave as owner
     vm.stopPrank();
}
```

Defining accounts and addresses in Foundry

The "owner" variable above can be defined a few ways:

```
// an address created by casting a decimal to an address
address owner = address(1234);

// vitalik's addresss
address owner = 0x0xd8dA6BF26964aF9D7eEd9e03E53415D37aA96045;

// create an address from a known private key;
address owner = vm.addr(privateKey);
```

```
// create an attacker
address hacker = 0x00baddad
```

msg.sender and tx.origin prank

In the above examples, msg.sender is altered. If you specifically want control over both tx.origin and msg.sender, both vm.prank and vm.startPrank optionally take two arguments where the second argument is tx.origin.

```
vm.prank(msgSender, txOrigin);
```

Relying on tx.origin is generally a bad practice, so you will rarely need to use the two-argument version of vm.prank.

Checking balances

When you transfer ether, you should measure that the balances changed as expected. Thankfully, checking balances in foundry is easy, since it is written in Solidity.
Consider this contract

```
contract Deposit {

    event Deposited(address indexed);

    function buyerDeposit() external payable {
        require(msg.value == 1 ether, "incorrect amount");
        emit Deposited(msg.sender);
    }

    // rest of the logic
}
```

The test function would look like this.

```
function testBuyerDeposit() public {
    uint256 balanceBefore = address(depositContract).balance;
    depositContract.buyerDeposit{value: 1 ether}();
    uint256 balanceAfter = address(depositContract).balance;

    assertEq(balanceAfter - balanceBefore, 1 ether, "expect increase of
1 ether");
}
```

Note that we haven't tested the cases where the buyer sent an amount other than 1 ether, which would cause a revert. We'll discuss testing reverts in the next section.

Expecting reverts with vm.expectRevert

The problem with the test above in its current form is that you could delete the require statement and the test would still pass. Let’s improve the test so that deleting the require statement causes a test to fail.

```
function testBuyerDepositWrongPrice() public {
    vm.expectRevert("incorrect amount");
    depositContract.deposit{value: 1 ether + 1 wei}();

    vm.expectRevert("incorrect amount");
    depositContract.deposit{value: 1 ether - 1 wei}();
}
```

Note that vm.expectRevert must be called right before doing the function that we expect to revert. Now if we delete the require statement, it will revert, so we have better modeled the intended functionality of the smart contract.

Testing Custom Errors

If we use custom errors instead of require statements, the way to test the revert would be as follows

```
contract CustomErrorContract {
    error SomeError(uint256);

    function revertError(uint256 x) public pure {
        revert SomeError(x);
    }
}
```

And the test file would be like this

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.13;

import "forge-std/Test.sol";
import "../src/RevertCustomError.sol";

contract CounterTest is Test {
    CustomErrorContract public customErrorContract;
    error SomeError(uint256);

    function setUp() public {
        customErrorContract = new CustomErrorContract();
    }
```

```
        function testRevert() public {
                    // 5 is an arbitrary example
            vm.expectRevert(abi.encodeWithSelector(SomeError.selector, 5));
            customErrorContract.revertError(5);
        }
    }
```

In our example, we've created a parameterized custom error. For the test to pass, the parameter needs to be equal the one actually used during the revert.

Testing logs and events with vm.expectEvent

Although solidity events don't alter the functionality of a smart contract, incorrectly implementing them can break client applications that read the state of a smart contract. To ensure our events function as expected, we can use the vm.expectEmit. This API behaves rather counterintuitively because you must emit the event in the test to ensure it worked in the smart contract.

Here is a minimal example.

```
function testBuyerDepositEvent() public {
    vm.expectEmit();
  emit Deposited(buyer);

    depositContract.deposit{value: 1 ether}();
}
```

Adjusting block.timestamp with vm.warp

Now let's consider a time locked withdrawal. The seller can withdraw the payment after 3 days.

```
contract Deposit {

    address public seller;
    mapping(address => uint256) public depositTime;

    event Deposited(address indexed);
    event SellerWithdraw(address indexed, uint256 indexed);

    constructor(address _seller) {
        seller = _seller;
    }

    function buyerDeposit() external payable {
        require(msg.value == 1 ether, "incorrect amount");
        uint256 _depositTime = depositTime[msg.sender];
        require(_depositTime == 0, "already deposited");
        depositTime[msg.sender] = block.timestamp;

        emit Deposited(msg.sender);
    }

    function sellerWithdraw(address buyer) external {
        require(msg.sender == seller, "not the seller");
        uint256 _depositTime = depositTime[buyer];
        require(_depositTime != 0, "buyer did not deposit");
        require(block.timestamp - _depositTime > 3 days, "refund
period not passed");
        delete depositTime[buyer];

        emit SellerWithdraw(buyer, block.timestamp);
        (bool ok, ) = msg.sender.call{value: 1 ether}("");
        require(ok, "seller did not withdraw");
    }
}
```

We've added a lot of functionality that needs to be tested, but let's focus on the time aspect for now.

We want to test that the seller cannot withdraw the money until 3 days since the deposit. (There is obviously a missing function for the buyer to withdraw before that window, but we'll get to that later).

Note that block.timestamp starts at 1 by default. This is not a realistic number to test against, so we should warp to the present day first.

This can be done with vm.warp(x), but let's be fancy and use a modifier.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import "forge-std/Test.sol";
import "../src/Deposit.sol";

contract DepositTest is Test {
    Deposit public deposit;
    Deposit public faildeposit;
    address constant SELLER = address(0x5E11E7);
    //address constant Rejector = address(RejectTransaction);
    RejectTransaction private rejector;

    event Deposited(address indexed);
    event SellerWithdraw(address indexed, uint256 indexed);

    function setUp() public {
        deposit = new Deposit(SELLER);
        rejector = new RejectTransaction();
        faildeposit = new Deposit(address(rejector));
    }

    modifier startAtPresentDay() {
        vm.warp(1680616584);
        _;
    }

    address public buyer = address(this); // the DepositTest contract
is the "buyer"
    address public buyer2 = address(0x5E11E1); // random address
    address public FakeSELLER = address(0x5E1222); // random address

    function testDepositAmount() public startAtPresentDay {
        // this test checks that the buyer can only deposit 1 ether
        vm.startPrank(buyer);
        vm.expectRevert();
        deposit.buyerDeposit{value: 1.5 ether}();
        vm.expectRevert();
        deposit.buyerDeposit{value: 2.5 ether}();
        vm.stopPrank();
    }
}
```

Adjusting block.number with vm.roll

If you want to adjust the block number (block.number) in Foundry, use

```
vm.roll(blockNumber)
```

To change the block number. To move forward a certain number of blocks, do the following

```
vm.roll(block.number() + numberOfBlocks)
```

Adding the extra tests

For the sake of completeness, let’s write the unit tests for the rest of the functions.

Some additional features need to be tested for the deposit function:

- the public variable depositTime matches the time of the transaction
- a user cannot deposit twice

And for the seller function:

- the seller cannot withdraw for non-existent addresses
- the entry for the buyer is deleted (this allows the buyer to buy again)
- the SellerWithdraw event is emitted
- the contract’s balance is decreased by 1 ether
- an address that is not the seller calling sellerWithdraw gets reverted

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import "forge-std/Test.sol";
import "../src/Deposit.sol";

contract DepositTest is Test {
    Deposit public deposit;
    Deposit public faildeposit;
    address constant SELLER = address(0x5E11E7);
    //address constant Rejector = address(RejectTransaction);
    RejectTransaction private rejector;

    event Deposited(address indexed);
    event SellerWithdraw(address indexed, uint256 indexed);

    function setUp() public {
        deposit = new Deposit(SELLER);
        rejector = new RejectTransaction();
        faildeposit = new Deposit(address(rejector));
    }

    modifier startAtPresentDay() {
        vm.warp(1680616584);
        _;
    }

    address public buyer = address(this); // the DepositTest contract
```

```
is the "buyer"
    address public buyer2 = address(0x5E11E1); // random address
    address public FakeSELLER = address(0x5E1222); // random address

    function testDepositAmount() public startAtPresentDay {
        // this test checks that the buyer can only deposit 1 ether
        vm.startPrank(buyer);
        vm.expectRevert();
        deposit.buyerDeposit{value: 1.5 ether}();
        vm.expectRevert();
        deposit.buyerDeposit{value: 2.5 ether}();
        vm.stopPrank();
    }

    function testBuyerDepositSellerWithdrawAfter3days() public
startAtPresentDay {
        // This test checks that the seller is able to withdraw 3 days
after the buyer deposits

        // buyer deposits 1 ether
        vm.startPrank(buyer); // msg.sender == buyer
        deposit.buyerDeposit{value: 1 ether}();
        assertEq(address(deposit).balance, 1 ether, "Contract balance
did not increase"); // checks to see if the contract balance increases
        vm.stopPrank();

        // after three days the seller withdraws
        vm.startPrank(SELLER); // msg.sender == SELLER
        vm.warp(1680616584 + 3 days + 1 seconds);
        deposit.sellerWithdraw(address(this));
        assertEq(address(deposit).balance, 0 ether, "Contract balance
did not decrease"); // checks to see if the contract balance decreases
    }

    function testBuyerDepositSellerWithdrawBefore3days() public
startAtPresentDay {
        // This test checks that the seller is able to withdraw 3 days
after the buyer deposits

        // buyer deposits 1 ether
        vm.startPrank(buyer); // msg.sender == buyer
        deposit.buyerDeposit{value: 1 ether}();
        assertEq(address(deposit).balance, 1 ether, "Contract balance
did not increase"); // checks to see if the contract balance increases
        vm.stopPrank();

        // before three days the seller withdraws
        vm.startPrank(SELLER); // msg.sender == SELLER
        vm.warp(1680616584 + 2 days);
        vm.expectRevert(); // expects a revert
        deposit.sellerWithdraw(address(this));
    }

    function testdepositTimeMatchesTimeofTransaction() public
startAtPresentDay {
        // This test checks that the public variable depositTime
matches the time of the transaction

        vm.startPrank(buyer); // msg.sender == buyer
        deposit.buyerDeposit{value: 1 ether}();
        // check that it deposits at the right time
        assertEq(
            deposit.depositTime(buyer),
            1680616584, // time of startAtPresentDay
            "Time of Deposit Doesnt Match"
        );
        vm.stopPrank();
    }

    function testUserDepositTwice() public startAtPresentDay {
        // This test checks that a user cannot deposit twice

        vm.startPrank(buyer); // msg.sender == buyer
        deposit.buyerDeposit{value: 1 ether}();

        vm.warp(1680616584 + 1 days); // one day later...
        vm.expectRevert();
        deposit.buyerDeposit{value: 1 ether}(); // should revert since
it hasn't been 3 days
    }
    function testNonExistantContract() public startAtPresentDay {
        // This test checks that the seller cannot withdraw for non-
existent addresses

        vm.startPrank(SELLER); // msg.sender == SELLER
        vm.expectRevert();
        deposit.sellerWithdraw(buyer);
    }

    function testBuyerBuysAgain() public startAtPresentDay {
        // This test checks that the entry for the buyer is deleted
(this allows the buyer to buy again)

        vm.startPrank(buyer); // msg.sender == buyer
        deposit.buyerDeposit{value: 1 ether}();
        vm.stopPrank();

        // seller withdraws
        vm.warp(1680616584 + 3 days + 1 seconds);
        vm.startPrank(SELLER); // msg.sender == SELLER
        deposit.sellerWithdraw(buyer);
        vm.stopPrank();

        // checks depostitime[buyer] == 0
        assertEq(deposit.depositTime(buyer), 0, "entry for buyer is not
deleted");

        // buyer deposits again
        vm.startPrank(buyer); // msg.sender == buyer
        vm.expectEmit();
        emit Deposited(buyer);
        deposit.buyerDeposit{value: 1 ether}();
        vm.stopPrank();
    }

    function testSellerWithdrawEmitted() public startAtPresentDay {
        // this test checks that the SellerWithdraw event is emitted

        //buyer2 deposits
        vm.deal(buyer2, 1 ether); // msg.sender == buyer2
        vm.startPrank(buyer2);
        vm.expectEmit(); // Deposited Emitter checked
        emit Deposited(buyer2);
        deposit.buyerDeposit{value: 1 ether}();
        vm.stopPrank();

        vm.warp(1680616584 + 3 days + 1 seconds);// 3 day and 1 second
later...

        // seller withdraws + checks SellerWithdraw event emmited or
not
        vm.startPrank(SELLER); // msg.sender == SELLER
        vm.expectEmit(); // expects SellerWithdraw Emitterd
        emit SellerWithdraw(buyer2, block.timestamp);
        deposit.sellerWithdraw(buyer2);
        vm.stopPrank();
    }

         function testFakeSeller2Withdraw() public startAtPresentDay {
        // buyer deposits
        vm.startPrank(buyer);
        vm.deal(buyer, 2 ether); // this contract's address is the
buyer
        deposit.buyerDeposit{value: 1 ether}();
        vm.stopPrank();
        assertEq(address(deposit).balance, 1 ether, "Ether deposited
somehow failed");

        vm.warp(1680616584 + 3 days + 1 seconds); // 3 day and 1 second
later...

        vm.startPrank(FakeSELLER); // msg.sender == FakeSELLER
        vm.expectRevert();
        deposit.sellerWithdraw(buyer);
        vm.stopPrank();
    }

    function testRejectedWithdrawl() public startAtPresentDay {
        // This test checks that the entry for the buyer is deleted
```

```
(this allows the buyer to buy again)

                    vm.startPrank(buyer); // msg.sender == buyer
        faildeposit.buyerDeposit{value: 1 ether}();
        vm.stopPrank();
        assertEq(address(faildeposit).balance, 1 ether, "assertion
failed");

        vm.warp(1680616584 + 3 days + 1 seconds); // 3 days and 1
second later...

        vm.startPrank(address(rejector)); // msg.sender == rejector
        vm.expectRevert();
        faildeposit.sellerWithdraw(buyer);
        vm.stopPrank();
    }
}
```

Testing failed ether transfers

Testing the buyer withdraw requires an extra trick to get full line coverage. Here is the snippet we are testing, and we will explain the Rejector contract in the code above.

```
function buyerWithdraw() external {
    uint256 _depositTime = depositTime[msg.sender];
    require(_depositTime != 0, "sender did not deposit");
    require(block.timestamp - _depositTime <= 3 days);

    emit BuyerRefunded(msg.sender, block.timestamp);

    // this is the branch we are testing
    (bool ok,) = msg.sender.call{value: 1 ether}("");
    require(ok, "Failed to withdraw");
}
```

To test the fail condition of “require(ok…)” we need the Ether transfer to fail. The way the test accomplishes this is by creating a smart contract that calls the buyerWithdraw function, but has its receive function set to revert.

Foundry Fuzzing

Although we can specify an arbitrary address that isn't the seller to test the revert of an unauthorized address withdrawing, it's more mentally reassuring to try a lot of different values.

If we supply an argument to the test functions, foundry will try a bunch of different values for the arguments. To prevent it from using arguments that don't apply to the test case (such as when the address is authorized), we would use vm.assume. Here's how we can test the seller withdraw for an unauthorized seller.

```
// notSeller will be chosen randomly
function testInvalidSellerAddress(address notSeller) public {
    vm.assume(notSeller != seller);

    vm.expectRevert("not the seller");
    depositContract.sellerWithdraw(notSeller);
}
```

Here are all the state transitions

- **The contract's balance decreases by 1 ether**
- **The BuyerRefunded event was emitted**
- **The buyer can refund before three days**

Here are the branches that need to be tested

- **the buyer cannot withdraw after 3 days**
- **The buyer cannot withdraw if they never deposited**

Console.log Foundry

To console.log in foundry, import the following

```
import "forge-std/console.sol";
```

And run the test with

```
forge test -vv
```

Testing signatures

Refer to our tutorial on solidity signature verification with foundry, so we refer you to that.

Solidity test internal functions

Refer to our tutorial on testing internal functions in solidity.

Setting address balances with vm.deal and vm.hoax

The cheat code vm.hoax allows you to prank an address and set it's balance simultaneously.

```
vm.hoax(addressToPrank, balanceToGive);
// next call is a prank for addressToPrank

vm.deal(alice, balanceToGive);
```

Some common mistakes with Foundry

Not having a fallback function when receiving Ether

If you are testing withdrawing Ether from the contract, it will be sent to the contract which is running the tests. Foundry tests themselves are a smart contract, and if you send Ether to a smart contract that doesn't have a fallback or receive function, then the transaction will fail. Be sure to have a fallback or receive function in the contract.

Not having an onERC...Received when receiving tokens

By the same token (pun intended), ERC721 safeTransferFrom and ERC1155 transferFrom revert when sending tokens to a smart contract that doesn't have the appropriate transfer hook function. You'll need to add that to your tests if you want to test transferring NFTs (or ERC777-like tokens) to yourself.

Summary

- **aim for 100% line and branch coverage**
- **fully define the expected state transitions**
- **use error messages in your asserts**

Learn More Testing

To learn advanced solidity testing beyond unit tests and basic fuzzing, please see our advanced solidity training course.