**Test your ERC20 using AI**

**AI Tests and Recap**

Almost done, you're doing great! The last thing we want to assure is that we're always testing any code we write before it's deployed to a production environment. Fortunately, AI is becoming more and more capable each day at being able to assist us with some basic tests.

❗ **IMPORTANT** I want you to use AI to jumpstart your learning, don't use it to substitute learning. It can be really easy to fall back on having answers provided to us. I strongly encourage you to use AI consciously and with the intent of supporting your efforts, not doing the work for you.

We're going to write a couple tests together, then see if an AI can help us with some others. Go ahead and create a new file within our test folder named OurTokenTest.t.sol.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.18;

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

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

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

contract OurTokenTest is Test {

function setUp() public {}

}

With this boiler plate set up in OurTokenTest.t.sol we can begin by declaring OurToken and Deployer variables and deploying these. We'll also need to create some accounts/addresses for our tests.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.18;

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

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

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

contract OurTokenTest is Test {

OurToken public ourToken;

DeployOurToken public deployer;

address bob = makeAddr("bob");

address alice = makeAddr("alice");

function setUp() public {

deployer = new DeployOurToken();

ourToken = deployer.run();

}

}

The last thing we need in our setUp function is to assure one of our accounts is given some OurToken to play with. We wrote OurToken to mint it's INITIAL\_SUPPLY to the msg.sender. Let's have our msg.sender contract transfer 100 ether worth of OurToken to Bob.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.18;

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

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

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

contract OurTokenTest is Test {

OurToken public ourToken;

DeployOurToken public deployer;

address bob = makeAddr("bob");

address alice = makeAddr("alice");

uint256 public constant STARTING\_BALANCE = 100 ether;

function setUp() public {

deployer = new DeployOurToken();

ourToken = deployer.run();

vm.prank(msg.sender);

ourToken.transfer(bob, STARTING\_BALANCE);

}

}

Bam! With this we're ready to start writing our first test. We'll start with a simple one, let's assure that the STARTING\_BALANCE was in fact sent to Bob.

function testBobBalance() public view {

assertEq(STARTING\_BALANCE, ourToken.balanceOf(bob));

}

Let's run it!

forge test --mt testBobBalance

Easy pass. Let's write a couple more tests and then we'll see what AI can do to help us.

**Approvals and transferFrom**

Next, let's test some approvals. The ERC20 standard contains an important function, transferFrom. It is often the case that a smart contract protocol may need to transfer tokens *on behalf* of a user the way this access is controlled is through the transferFrom function.

In summary, an address needs to be approved by another in order to transfer tokens on their behalf, otherwise the transaction should revert with an error.

Approvals, naturally, are handled through the approve and allowance functionality within the ERC20 standard.

Through these methods a user is able to approve another address to spend, or otherwise interact with, a limited (or often unlimited) number of tokens.

The security risks associated with this are pretty clear, which is why we've seen services like Etherscan's Token Approval Checker pop up. These allow you to see at a glance which addresses possess approvals for tokens in your wallet.

While it costs a little gas, it's good practice to regularly assess your approvals and revoke them when no longer applicable or appropriate.

With all this context in mind, let's look at what a test for OurToken allowances looks like.

function testAllowancesWork() public {

uint256 initialAllowance = 1000;

// Bob approves Alice to spend 1000 tokens.

vm.prank(bob);

ourToken.approve(alice, initialAllowance);

uint256 transferAmount = 500;

vm.prank(alice);

ourToken.transferFrom(bob, alice, transferAmount);

}

Here, we're declaring an initial balance and pranking Bob to call approve on OurToken. This is allowing Alice to transfer up to 1000 OurTokens.

We then declare a transfer amount, and prank Alice as we call transferFrom, transferring tokens from Bob's address to Alice's.

❗ **NOTE** The transfer function won't work here as the from address defaults to msg.sender!

All we need now is our assert statements.

assertEq(ourToken.balanceOf(alice), transferAmount);

assertEq(ourToken.balanceOf(bob), STARTING\_BALANCE - transferAmount);

Let's run the test!

forge test --mt testAllowancesWork

Nice, another pass! However, if we run forge coverage ...

Abysmal. We have a long way to go!

**AI Help**

AI can be a little hit and miss when it comes to accuracy and reliability, but one thing they're pretty good at are tests. I've prepared a prompt you can use in the GitHub repo (and below) as an example of good formatting of a question for AI models. AI models are essentially sophisticated prediction engines, the higher quality your prompt, the higher quality a response.

OpenAI has a very capable free tier for ChatGPT available [**here**](https://chatgpt.com). Navigate to the platform and create an account if needed.

**AI Prompt**

Here is my solidity ERC20 token.

```

// contracts/OurToken.sol

// SPDX-License-Identifier: MIT

pragma solidity 0.8.19;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract OurToken is ERC20 {

constructor(uint256 initialSupply) ERC20("OurToken", "OT") {

\\_mint(msg.sender, initialSupply);

}

}

```

And here our my first couple of tests written in solidity.

```

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.19;

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

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

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

import {StdCheats} from "forge-std/StdCheats.sol";

interface MintableToken {

function mint(address, uint256) external;

}

contract OurTokenTest is StdCheats, Test {

OurToken public ourToken;

DeployOurToken public deployer;

function setUp() public {

deployer = new DeployOurToken();

ourToken = deployer.run();

}

function testInitialSupply() public {

assertEq(ourToken.totalSupply(), deployer.INITIAL\_SUPPLY());

}

function testUsersCantMint() public {

vm.expectRevert();

MintableToken(address(ourToken)).mint(address(this), 1);

}

}

```

Can you write the rest of the tests? Please include tests for:

- Allowances

- transfers

- anything else that might be important

❗ **NOTE** Your specific response will differ from mine, I encourage you to try using an AI tool like ChatGPT to generate a few tests of your own. I've included my AI response below, generated by GPT4o May 31, 2024.

**Example AI Response**

Sure, here are the rest of the tests for your ERC20 token, including tests for allowances, transfers, and a few other important functionalities.

```solidity

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.19;

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

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

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

import {StdCheats} from "forge-std/StdCheats.sol";

interface MintableToken {

function mint(address, uint256) external;

}

contract OurTokenTest is StdCheats, Test {

OurToken public ourToken;

DeployOurToken public deployer;

address public user1 = address(0x123);

address public user2 = address(0x456);

function setUp() public {

deployer = new DeployOurToken();

ourToken = deployer.run();

vm.deal(user1, 1 ether); // Giving some initial balance to user1 for transactions

vm.deal(user2, 1 ether); // Giving some initial balance to user2 for transactions

}

function testInitialSupply() public {

assertEq(ourToken.totalSupply(), deployer.INITIAL\_SUPPLY());

}

function testUsersCantMint() public {

vm.expectRevert();

MintableToken(address(ourToken)).mint(address(this), 1);

}

function testAllowance() public {

uint256 amount = 1000 \* 10 \*\* 18; // Example amount

vm.prank(msg.sender);

ourToken.approve(user1, amount);

assertEq(ourToken.allowance(msg.sender, user1), amount);

}

function testTransfer() public {

uint256 amount = 1000 \* 10 \*\* 18; // Example amount

vm.prank(msg.sender);

ourToken.transfer(user1, amount);

assertEq(ourToken.balanceOf(user1), amount);

assertEq(ourToken.balanceOf(msg.sender), deployer.INITIAL\_SUPPLY() - amount);

}

function testTransferFrom() public {

uint256 amount = 500 \* 10 \*\* 18; // Example amount

vm.prank(msg.sender);

ourToken.approve(user1, amount);

vm.prank(user1);

ourToken.transferFrom(msg.sender, user2, amount);

assertEq(ourToken.balanceOf(user2), amount);

assertEq(ourToken.allowance(msg.sender, user1), 0);

}

function testFailTransferExceedsBalance() public {

uint256 amount = deployer.INITIAL\_SUPPLY() + 1;

vm.prank(msg.sender);

ourToken.transfer(user1, amount); // This should fail

}

function testFailApproveExceedsBalance() public {

uint256 amount = deployer.INITIAL\_SUPPLY() + 1;

vm.prank(msg.sender);

ourToken.approve(user1, amount); // This should fail

}

function testTransferEvent() public {

uint256 amount = 1000 \* 10 \*\* 18; // Example amount

vm.prank(msg.sender);

vm.expectEmit(true, true, false, true);

emit Transfer(msg.sender, user1, amount);

ourToken.transfer(user1, amount);

}

function testApprovalEvent() public {

uint256 amount = 1000 \* 10 \*\* 18; // Example amount

vm.prank(msg.sender);

vm.expectEmit(true, true, false, true);

emit Approval(msg.sender, user1, amount);

ourToken.approve(user1, amount);

}

}

```

### Explanation

- \*\*testAllowance\*\*: Tests the `approve` function by setting an allowance and checking if it is recorded correctly.

- \*\*testTransfer\*\*: Tests the `transfer` function by transferring tokens from the deployer to `user1` and verifying the balances.

- \*\*testTransferFrom\*\*: Tests the `transferFrom` function by approving `user1` to transfer tokens on behalf of the deployer and verifying the balances and allowance.

- \*\*testFailTransferExceedsBalance\*\*: Ensures that transferring more than the balance fails.

- \*\*testFailApproveExceedsBalance\*\*: Ensures that approving more than the balance fails.

- \*\*testTransferEvent\*\*: Verifies that the `Transfer` event is emitted correctly during a transfer.

- \*\*testApprovalEvent\*\*: Verifies that the `Approval` event is emitted correctly during an approval.

I'll caution you against blindly copying and pasting AI responses and to use what's generated intelligently. Some things will make more sense for your situation that others.

When you've implemented your AI suggestion, run forge test -vvvv and see what happens!

**Wrap Up**

It's very likely your test was hit with a bunch of errors, that's ok. AI isn't perfect. By looking through the tests provided you should be able to discern where the AI's suggestion fell down. Worst case scenario, I've curated a few additional tests my AI provided and included them below.

That's all their is to the creation and deployment of your very own ERC20. I know this was a quick lesson, but we're getting more advanced and with that comes less hand holding.

Great work! We're diving into NFTs next!

See you soon!

**AI Tests**

function testTransfer() public {

uint256 amount = 1000 \* 10 \*\* 18; // Example amount

vm.prank(msg.sender);

ourToken.transfer(user1, amount);

assertEq(ourToken.balanceOf(user1), amount);

assertEq(ourToken.balanceOf(msg.sender), deployer.INITIAL\_SUPPLY() - amount);

}

function testTransferFrom() public {

uint256 amount = 500 \* 10 \*\* 18; // Example amount

vm.prank(msg.sender);

ourToken.approve(user1, amount);

vm.prank(user1);

ourToken.transferFrom(msg.sender, user2, amount);

assertEq(ourToken.balanceOf(user2), amount);

assertEq(ourToken.allowance(msg.sender, user1), 0);

}

function testFailTransferExceedsBalance() public {

uint256 amount = deployer.INITIAL\_SUPPLY() + 1;

vm.prank(msg.sender);

ourToken.transfer(user1, amount); // This should fail

}

function testFailApproveExceedsBalance() public {

uint256 amount = deployer.INITIAL\_SUPPLY() + 1;

vm.expectRevert();

vm.prank(msg.sender);

ourToken.approve(user1, amount); // This should fail

}