**Creating an ERC20**

**ERC20 Manual Creation**

Welcome back! Having covered the basics, let's look at how we can manually create our own ERC20 token. This is going to be one of our fastest lessons yet!

**Setting Up Your Environment**

Open a terminal in Visual Studio Code and run the following:

mkdir foundry-erc20

cd foundry-erc20

code .

The above commands will create a new directory for our project, navigate into it, and open the directory in a new Visual Studio Code window.

Once we have Visual Studio Code running, we need to initialize a blank Foundry project. Open up the terminal and execute the following command:

forge init

Completing these steps sets up a development environment with some convenient boilerplate layouts to work with.

Go ahead and delete our 3 Counter examples so we can start with a clean slate.

I'm going to show you 2 different ways to create our own token, first the hard way and then a much easier way!

You can begin by creating a new token our src directory named ManualToken.sol. We can start this contract the same way we've been practicing this whole time (you'll get really familiar with this bit).

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.18;

contract ManualToken {}

Now, as we covered in the last lesson, all we need to do to make our token compatible is follow the [**ERC20 Token Standard**](https://eips.ethereum.org/EIPS/eip-20). Essentially this means we need to include the required functions/methods for our deployment to follow this standard. Let's add thing functionality then!

Let's start with name, decimals and totalSupply

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.18;

contract ManualToken {

function name() public pure returns(string memory) {

return "Manual Token";

}

function totalSupply() public pure returns (uint256) {

return 100 ether; // 100000000000000000000

}

function decimals() public pure returns (uint8) {

return 18;

}

}

❗ **NOTE** Despite being an optional method, we're including decimals here as a point of clarification since we're declaring our total supply as 100 ether. 100 ether = 100 + 18 decimals places.

The next functions required by the ERC20 standard are balanceOf and transfer, so let's apply those now.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.18;

contract ManualToken {

function name() public pure returns(string memory) {

return "Manual Token";

}

function totalSupply() public pure returns (uint256){

return 100 ether; // 100000000000000000000

}

function decimals() public pure returns (uint8) {

return 18;

}

function balanceOf(address \_owner) public pure returns (uint256){

return // ???

}

}

Hmm, what is this function meant to return exactly? We're probably going to need a mapping to track the balances of each address...

mapping(address => uint256) private s\_balances;

So now our balanceOf function can return this mapped value based on the address parameter being passed.

function balanceOf(address \_owner) public view returns (uint256) {

return s\_balances[\_owner];

}

An interesting thing that comes to light from this function is - someone's balance of a token is really just some mapping on a smart contract that says this number is associated with this address That's it. All swaps, transfers and trades are represented as an updating to the balance of this mapping.

❗ **PROTIP** Our name function could also be represented by a public declaration such as string public name = "ManualToken";. This is because Solidity creates public getter functions when compiled for any publicly accessible storage variables!

Our next required function is transfer:

function transfer(address \_to, uint256 \_amount) public {

uint256 previousBalance = balanceOf(msg.sender) + balanceOf(\_to);

s\_balances[msg.sender] -= \_amount;

s\_balances[\_to] += \_amount;

require(balanceOf(msg.sender) + balanceOf(\_to) == previousBalance);

}

So, a basic transfer function could look something like the above, a simple adjustment of the balances mapped to both the sender and receiver addresses in our contract.

**Wrap Up**

We could absolutely continue going through each of the required functions outlined in the [**ERC20 Token Standard**](https://eips.ethereum.org/EIPS/eip-20) and eventually come out the other side with a compatible contract, but there's an easier way.

In the next lesson, we'll investigate an even easier method to spin up a standard ERC20 protocol, with the help of OpenZeppelin.

See you there!