Creating a ERC20 Compilant Token

In this exercise, you will write a ERC20-protocol compilant token using the OpenZeppelin library, and test your smart contracts to ensure that they work correctly.

I assume that you have installed truffle, ganache, solidity, and NodeJS.

Setting up an empty project

Setting up the network

Make your way to your newly-created projects folder with the CLI, and type in 'npm update truffle', followed by 'truffle init'. You should now see 3 empty folders and a truffle-config.js file.

First, open the truffle-config.js file, where you'll specify what network to connect to. In our case, the Ganache app simulates a local network. Within the 'networks' field, un-commentthe 'development' field:

```
development: {
    host: "127.0.0.1",     // Localhost (default: none)
    port: 7545,            // Standard Ethereum port (default: none)
    network_id: "*",       // Any network (default: none)
},
```

and make sure that the hostname and port fields matches w/ the 'RPC Server' field of your Ganache App.

Setting up truffle's framework

'truffle init' only gives us the bare minimum of a framework, so we have to populate it some more before start coding our first smart contract.

In the ./contracts folder, create a file named 'Migrations.sol' and populate it with this:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Migrations {
  address public owner = msg.sender;
  uint public last_completed_migration;

  modifier restricted() {
    require(
      msg.sender == owner, "This function is restricted to the contract's owner"
    );
    _;
  }

  function setCompleted(uint completed) public restricted {
    last_completed_migration = completed;
  }
}
```

Now, in the ./migrations folder, create a file named '1_initial_migration.js' and populate it with this:

```
const Migrations = artifacts.require("Migrations");

module.exports = function (deployer) {
  deployer.deploy(Migrations);
};
```

Also, scroll down and take note of the value in this field

```
compilers: {
    solc: {
      version: "0.8.17"
```

If your version number < 0.8.0, your should probably update your solidity compiler by running 'npm update -g truffle'.

Once you are done with this, run 'truffle migrate' to deploy your contracts and your should end up with a successful deployment.

The three folders

./contracts stores all of your .sol solidity contracts. ./migrations stores .js files that help deploy your contracts ./test stores .js files that you use to test your contracts' functions with

---

An Example: storageNum.sol

Writing and Deploying it

Once you successfully set up your environment, clone 'Solidity_Practices' repo from our github, and copy 'storageNum.sol' from ./contracts and '2_deploy_contracts.js' from ./migrations to your local projects file. These are completed files.

Look at the 2_deploy_contracts.js to get a gist of how the deployment file's syntax works. Its nothing too complicated. Additionally, see if you can understand fully what is happening in this contract. Here is a good explaniation for a largely similar contract - https://www.grizzlypeaksoftware.com/articles?id=5vWBWo4Zpi02FSVCmQunxk

Note: In the set() function there is a msg.sender field. This is implicitly referring to the address of the user who initiated the function call. Instead, we could ask the user to explicitly pass in an address, as we did in the get() function. Both ways work.

Testing it

Copy 'storageNumTest.js' from ./test to your local folder, and familiarize yourself with the syntax and overall structure.

The file starts with a header declaring the usage of 'chai', which is an assertion library. It then declares your contract as an artifact, deploys it in an async function 'before' that only runs ONCE, and runs tests in an async function named 'it'.

Note that since the get() function requires an explicit address input while the set() function does not, we simulate an address imput with different syntax during test. Take note of that.

To run the test, you need to install some packages. Run these within the CLI:

```
npm install chai
npm install chai-as-promised
```

Once installed, run 'truffle test'. If everything goes well, you should see an output indicating that the test has passed. Remember to have Ganache running!

A Bigger Example: testToken.sol

Instaling Depedencies

Type in these commands

```
npm install @openzeppelin/contracts
npm install
```

This should create a @openzeppelin/contracts folder within our ./node_modules folder. This is the framework that we are going to use (inherit from) to create our ERC20-compliant token.

Writing the smart contract

Copy 'testToken.sol' from ./contracts and you should see a completed header and contract & inheirtance declaration. We inherit OpenZeppelin's ERC20Capped.sol file, which is a standard implementation of a ERC20-protocol compilant token that has a supply cap. Feel free to research more if you are not familiar with any of these terms.

Familiarize yourself with our framework and the functions existing in the ERC20Capped contract that we inherited. https://docs.openzeppelin.com/contracts/4.x/api/token/erc20#ERC20Capped

See which functions you have inherited, look at the code to see some examples of how to use them.

Try to implement a token contract with AT LEAST the ability to: - Transfer tokens between addresses - Burn tokens of a given address - Return the token balance of a given address - Return the token's name, symbol, and supply cap

This is also a great resource https://ethereum.org/en/developers/tutorials/understand-the-erc-20-token-smart-contract/

Writing the deployer

Like with our first contract, you'd have to write some deployer code in the '2_deploy_contracts.js' file. First declare it using 'artifacts.require()', and deploy it using deployer.deploy().

Remember that the syntax for constructor args is deployer.deploy(contractName, input1, input2, ...). In our case, we have 3 constructor arguments - name, symbol, cap

Writing Tests

Copy 'tokenTest.js' from ./test and you should see the framework and one example test case. Feel free to write any tests you think is necessary. You could test for minting, transfering, burning tokens, etc. Write at least 3 different passing tests.

One more thing

Check out a token's actual deployed smart contract here!
https://etherscan.io/token/0xdac17f958d2ee523a2206206994597c13d831ec7#code

---

Questions

Stack overflow or the OpenZeppelin docs in general are also very good resources. If you can't figure it out, ask on discord!