Principles of Distributed Ledgers: Tutorial 2

In this tutorial, the goals are the following:

1. Get a working local environment to develop smart contracts
2. Learn the basics of using a wallet (MetaMask)
3. Develop a simple fungible token (ERC20) contract
4. Deploy our contract on Optimism (low transaction fees blockchain)
5. Develop a simple non-fungible token contract (optional)

These tasks are described in detail below.

0 Initial setup

0.1 Setting up coding environment

In this course, we will use Foundry to develop smart contracts. The simplest way to install Foundry is to use their official installation script for `foundryup` (a tool to manage Foundry's installation):

```
curl -L https://foundry.paradigm.xyz | bash
```

and then run

```
foundryup
```

For those who are (rightfully) uncomfortable piping a shell script directly from the Internet, the content of the official installation script can be inspected and the installation steps reproduced very easily.

If the installation is successful, the `forge` command should become available. It can be checked using

```
forge --version
```

which returns `forge 0.2.0` at the time of writing.

0.2 Using the skeleton

A skeleton project is provided here: `https://gitlab.doc.ic.ac.uk/podl/2024/tutorial-2-skeleton`. The skeleton contains contract files as well as a set of tests.

The following commands will fetch and build the projects:

```
git clone https://gitlab.doc.ic.ac.uk/podl/2024/tutorial-2-skeleton
cd tutorial-2-skeleton
forge build
forge test
```

The above run of `forge test` will fail, as the contracts are not implemented yet. While implementing the contracts, the tests can be run using `forge test` to check that the implementation is correct. Below are some useful flags for `forge test`:

```
forge test --mc ERC20Test   # run all tests in the ERC20Test contract
forge test --mc ERC20Test --mt testName  # run test called testName in ERC20Test
forge test --nmc Bonus  # run all tests apart from the bonus ones
forge test --mc ERC20Test -vvv  # run tests in ERC20Test with a verbose output
```

More information can be found in the Foundry documentation.

0.3 Installing MetaMask

https://metamask.io/ is one of the most popular Ethereum wallets. It lets you easily interact with smart contracts via a browser extension or mobile app.

1. Go to `https://metamask.io/download/` and download MetaMask for your browser. Alternatively, you can also install it directly as an extension through the browser's web store (e.g., Chrome).
2. Once installed, open MetaMask and click on "Create a new wallet".

3. Create a password. Note: this is **not** your private key, but simply a password that allows you to access MetaMask (which stores your actual private key).
4. Select "Secure my wallet" and take note of your twelve-word Secret Recovery Phrase. This phrase is used to generate your private key. More details on why this is common can be found here. **If you lose access to your seed phrase, you will lose access to all associated funds!**
5. Once you have MetaMask successfully set up, open MetaMask and click on the top left to "Select a network". In this tutorial, you will deploy smart contracts on Optimism (a blockchain compatible with Ethereum; it runs EVM code). Hence, click on "Add a network" and select "OP Mainnet".
6. On Optimism, transaction fees are paid in the native unit of account, which is also ETH (however, this version of ETH lives on Optimism, not on Ethereum mainnet; we shall call it Optimism ETH). When you set up your wallet, you do not hold any funds at first (i.e., no Optimism ETH to pay for transactions). Hence, you must first have Optimism ETH sent to your address (see Section 0.4). Your address is shown when you open MetaMask. It starts with `0x`. The workings of Optimism and other so-called "Layer 2s" will be explained in the upcoming lecture. For now, you can think of Optimism as an Ethereum extension that relies on the security of Ethereum while offering low-fee transactions.

To check the history of transactions and balances associated with your address, you can use Optimistic Etherscan or simply use `https://optimistic.etherscan.io/address/<your_address>`.

0.4 Getting funds

To deploy a contract on Optimism, the wallet must pay associated transaction fees, which require Optimism ETH. The typical process is to buy crypto on a centralised exchange (e.g., Coinbase) and to send the funds to the wallet address used for deployment.

For the purpose of this tutorial, we will distribute 0.0001 ETH (∼$0.25) per student, which should be enough to cover deployment costs for multiple contracts.

Visit the website to request the funds: `https://faucet.pdl.wiki`, and do the following:

1. Click on "Connect wallet" and connect MetaMask
2. If MetaMask is not using Optimism yet, accept the network switch
3. Enter your Imperial username and password (same as to login into Scientia)
4. This will send the funds to your account. It might take a minute or so until it shows up in MetaMask

0.5 Exporting your private key

In this tutorial, you deploy a smart contract using the private key that you have generated via MetaMask. In order to use it for deployment (in your coding environment), you need to export the private key from MetaMask.

To do so:

1. Open MetaMask and click on the top right (three dots)
2. Click on "Account details"
3. Click on "Show private key". Generally, the private key is something that should never be shared with anyone, and you should be cautious when you export (or import) it somewhere.

1 Implementing an ERC-20 token (required)

The first task is to implement your own ERC20 token. ERC-20 is a token standard often used to implement tokens representing a currency. These tokens are fungible, which means that each token has the same value as any other token.

The ERC-20 token standard defines the following functions:

`totalSupply` Retrieve the total number of tokens in existence

`balanceOf` Retrieve the balance of a user

`transfer` Transfer tokens to another user

`approve` Approve another user to transfer tokens on the user's behalf `allowance` Retrieve the number of tokens that a user is allowed to transfer on another's half

`transferFrom` Transfer tokens on another user's behalf

Although not mandatory in the standard, most tokens (including ours) also allow querying the following metadata:

`name` The name of the token

`symbol` The symbol (ticker) of the token, e.g., DAI or USDC

`decimals` The number of decimals by which the amounts are scaled, e.g., 18 if the amounts are scaled by 10^{18} (note: 18 decimals is the norm for most tokens deployed on Ethereum)

More information about each of these functionalities can directly be found in the ERC-20 standard.

In this tutorial, the ERC-20 token that you should first implement should have a few particularities:

- The name and symbol are passed into the constructor
- The total supply is initially set to 0
- The contract creator (and only the contract creator) is allowed to mint an arbitrary number of tokens to any user. The function should have the following signature: `mint(address to, uint256 amount)`
- The number of decimals is fixed to 18 (this is returned by the `decimals` function but not used in the contract itself)

Note that the different error messages that should be used when reverting can be found in the test files of the skeleton provided.

2 Deploying our contract (required)

To deploy the contract, we will need an account with Optimism ETH, and the private key for that account, so make sure that you followed all the setup steps.

After the above steps are completed and that you have the private key, the contract can be deployed using the following command. All capitalized constants need to be set correctly.

```
forge create \
    --constructor-args TOKEN_NAME TOKEN_SYMBOL \
    --private-key PRIVATE_KEY \
    --rpc-url RPC \
    --verify \
    --chain optimism \
    --etherscan-api-key ETHERSCAN_API_KEY \
    src/ERC20.sol:ERC20
```

The above command tells `forge` to deploy the contract `ERC20` located in the file `src/ERC20.sol`. It does so by communicating with the node exposing an RPC endpoint at `RPC` and signing the transaction using `PRIVATE_KEY`. The contract's constructor is passed in the token name and symbols as arguments. Finally, this also sends a request to verify the contract on Optimistic Etherscan, i.e., publish the Solidity source code of the contract's bytecode.

For this tutorial, we will use the following variables:

`TOKEN_NAME`: Any name you would like for your token. It needs to be quoted if it contains spaces

`TOKEN_SYMBOL`: Any symbol for your token. It is typically a short capitalized identifier, e.g., ETH

`PRIVATE_KEY`: The private key exported from MetaMask

`RPC`: You get this URL by creating a new project on Infura or Alchemy

`ETHERSCAN_API_KEY`: `634WMXSJSWC7DEW1WD613Z7RKN71NG5N7E`

Note that the Optimistic Etherscan key has been created for the purposes of this course, feel free to create a new one on `https://optimistic.etherscan.io` if you prefer.

Once the above command has succeeded, it will print out the address of your contract, as well as an Optimistic Etherscan URL. You can now use it to interact with your contract. For example, in the "Contract" tab, there is a "Write Contract" section where you can call `mint` to create your first tokens. Note that you will need to click on "Connect to Web3" before, and that the `amount` is scaled using 18 decimals, so to mint 1 token, `amount` should be set to `1000000000000000000` (which is $1e18$). Alternatively, you can use `eclair` to interact with your deployed contracts from the command line. Please refer to the documentation at `https://docs.eclair.so/overview.html`.

3 Implementing an ERC-721 token (optional)

ERC-721 is a token standard used to implement a non-fungible token (NFT). NFTs are often used to represent digital assets, where each token represents a different asset. For example, in the case of a collection of images, each token would represent a single image in the collection.

In this tutorial, the goal is to implement the following functions defined by the ERC-721 standard:

`balanceOf` Retrieve the number of tokens that a user owns

`ownerOf` Retrieve the owner of a given token

`transferFrom` Transfer a token to another address

`approve` Approve another user to transfer a given token on the user's behalf

`getApproved` Retrieve the user approved to transfer a given token

Like the ERC-20 standard, the ERC-721 standard also has some extensions. In this tutorial, we will also implement the metadata extension that defines the following other functions:

`name` The name of the token

`symbol` The symbol of the token

`tokenURI` The URI of the given token. The URI typically points to a JSON file containing metadata about the token

More information can be found in the ERC-721 standard.

In addition to the ERC-721 standard, our token will have the following extra specs:

- The constructor takes 5 arguments:
  1. The name of the token (type: `string memory`)
  2. The symbol of the token (type: `string memory`).
  3. The base URI of the token (type: `string memory`). Typically, the base URI is an URL such as `https://example.com/nft/`
  4. The ERC-20 token used pay for the ERC-721 token (type: `IERC20`)
  5. The initial price (in terms of ERC-20 token) of a token (type: `uint256`)
- To form the `tokenURI`, the base URI (passed in the constructor) should be concatenated with the token ID (using `string.concat`). A function to convert an integer to a string is provided in the `src/libraries/StringUtils.sol` of the skeleton.
- No tokens exist initially
- A new token can be minted by anyone by "burning" the amount of ERC-20 tokens corresponding to the current price. The price starts at the initial price passed in the constructor. To burn the ERC-20 tokens, this contract transfers the ERC-20 tokens from the user and sends them to the zero address. Note that this will revert if the user has not approved this contract to spend his ERC-20 tokens beforehand.
- The first token has an id of 1 and the id is incremented each time a token is minted
- Each time a token is minted, the price increases by 10%

4 Bonus

In the bonus section, the goal is to finish the implementation of the ERC-721 token so that it is completely compliant with the standard. To do so, the following extra functionalities should be implemented:

- The ERC-721 standard specifies that the ERC-165 standard must be implemented. This standard stipulates that a contract must have a function `supportsInterface(bytes4 interfaceID)` that returns true if the given `interfaceID` is supported. For this tutorial `supportsInterface` should return true if and only if the `interfaceID` equals `0x80ac58cd`, which is the interface ID of the ERC-721 interface.
- The ERC-721 standard also allows accounts to have control over all the tokens of a user. This is handled by the two following functions

  `setApprovalForAll` Approve another user to manage (transfer or approve) all tokens owned by the user

  `isApprovedForAll` Retrieve whether a user is approved to manage all tokens owned by another user
- To avoid losing tokens, the ERC-721 has a function named `safeTransferFrom`. It does the same thing as `transferFrom` but performs some extra checks when transferring tokens to a contract address.